CHAPTER 1

Introduction

In distributed algorithms, processes must be able to communicate and synchronize in order to work jointly towards a desired solution. These activities can take the form of mutual exclusion, mutual exclusion with conditional synchronization, asynchronous (i.e., buffered) message passing, or synchronous message passing (i.e., a rendezvous).

1.1. Research Goals

This research examines several known synchronization mechanisms, compares their relative expressiveness, and implements them using the language SR [Andr88]. The synchronization mechanisms we explore are semaphores [Dijk68], conditional critical regions (CCRs) [Brin72], monitors [Brin73], and input/output commands (I/O) in Communicating Sequential Processes (CSP) [Hoar78]. The comparisons include exploring how "equivalent" the above mechanisms are, in the sense that Lauer and Needham [Laue78] equate "message-oriented systems" to "procedure-oriented systems". Also, we judge the above mechanisms and SR by the conclusion reached in Scott's work [Scot83] that a concurrent language should contain "... several similar constructs for interprocess communication."

1.2. Background

Lauer and Needham's work addresses how the choice of synchronization mechanisms affects the structure of operating systems. They go so far as to say that except for some syntactic differences, message- and procedure-oriented systems code can be made to look the same, if the code adheres to their models. In their models, processes in a message-oriented system use no shared data and pass information through send and receive operations, and processes in a procedure-oriented system share data through a monitor-type interface. According to Scott [Scot83], people have been inappropriately applying the labels of message- and procedure-oriented to concurrent languages. He points out that there are actually four alternative types of communication based on the method of sending and receiving of messages, not just message- and procedure-based. Looking at a language from the point of view of the mechanisms it provides for sending data might place it in the message category while looking at the same language in terms of the mechanisms it provides for receiving data can place it in the procedure category. This possible contradictory categorization of a language means that the message/procedure dichotomy is too coarse and any partition should be based on the four send/receive combinations where the sending and receiving sides can be considered independently.

The synchronization mechanisms to be explored in this thesis—semaphores, CCRs, monitors, I/O in CSP—were chosen because they are the ones that are most commonly discussed in the literature and most commonly used in practice. The synchronization mechanisms differ in a number of ways. Semaphores, CCRs, and monitors generally make use of shared memory, while CSP and message passing languages do not. Some of the synchronization mechanisms are easier to use because, for instance, they provide mutual exclusion and information hiding implicitly. Information hiding refers to allowing access to variables only within the encapsulation mechanism in which they are defined.

This thesis focuses on two different types of implementations of each of the mechanisms. The first is a centralized approach. Each process has its own address space (i.e., distributed) and a single process is used as a central arbiter for performing the protocol of the mechanism. The other is a completely decentralized approach. Each process along with its own server process equally shares the responsibility of enforcing the mechanism's protocols with all the other processes. Again all the processes, including servers, have their own address spaces. A third possibility is a simplified version of the centralized approach. All the processes, including the single arbiter process, can be defined within one address space. Thus processes can share data, thereby reducing communication overhead. This approach can also simplify initialization code, since it is not necessary to pass communication links to the processes. The first centralized method was chosen over the simpler version for a number of reasons. First, the first approach is closer in structure to the decentralized approach than the simpler one. Comparisons can therefore be based on the centralized versus decentralized protocol rather than on the structure of the code. Also, in the environment we were working in it was not possible to distribute the processes in the simplified version across multiple machines.

This thesis builds on previous work related to the four synchronization mechanisms under discussion. Schneider presents an implementation of distributed semaphores [Schn80]. A binary semaphore, one whose value is restricted to zero and one, can be used to insure mutual exclusion. Algorithms for implementing mutual exclusion in a distributed environment are discussed by Ricart and Agrawala [Rica81,Rica83] and Suzuki and Kasami [Suzu85]. Andrews discusses how to implement CCRs [Andr89]. Silberschatz and Peterson describe an implementation of monitors [Silb88]. Bagrodia [Bagr86], Ramesh [Rame87], and Buckley and Silberschatz [Buck83] all give algorithms for implementing communication statements in CSP.

SR was chosen as the language to implement these mechanisms. SR supports communication commands and process definition that are suited to implementing the synchronization mechanisms. Other languages could have been used but most would have required considerable low-level work to obtain the mechanisms that SR already provides. After choosing SR, methodologies were developed for converting the synchronization mechanisms into SR code.

1.3. The Cheap Roller Coaster Problem

A single problem, the "cheap roller coaster problem", is used as the basis of comparison for the mechanisms. The problem revolves around a great roller coaster that a group of people want to ride multiple times. Unfortunately, the owner is a miser and will only allow one "car" on the tracks. Also, he will only allow the car to go around the tracks when it is full. This is a rendezvous problem. Customers and the car must communicate their availability to each other. This problem is at least as difficult as the standard synchronization problems, such as bounded buffer, readers/writers, etc. It also allows comparisons of the mechanisms due to a slight modification to the problem, having multiple cars on the tracks. This change is simple conceptually but actually changes the algorithms to solve it a great deal.

Pseudocode was written using each mechanism to solve the roller coaster problem. These four programs were then converted into SR code using the methods developed for each of the mechanisms. The pseudocode was used to assess how easy each of the mechanisms was to use. It also allowed assessment of how the idea of duality applied to concurrent languages. We also can see what difference centralized versus distributed implementations made in general and in each of the languages. The issue of efficiency of the implementations is examined, including tradeoffs such as the amount of concurrency versus the amount of message passing.

1.4. New Ideas

This work also examines some "new" ideas with synchronization mechanisms. We implemented both CCRs and monitors in a completely distributed fashion. We also developed and implemented an anticipatory execution of CCRs. Before executing a region statement, a process must normally obtain exclusive access to the region. In the anticipatory implementation, mutual exclusion is requested but the region code is executed without waiting for the response. At the end of this execution, it is then necessary to check if mutual exclusion was received and to undo the execution if it was not received. We also examine nested processes in CSP, which execute similarly to a nested **cobegin** statement. The "parent" process is suspended and all its "children" processes execute. When all its children have completed executing, the parent resumes. One strategy for implementing nested processes is to transform the nested to non-nested code. We also study the handling of nested processes via modifications to the protocol for implementing I/O commands.

Originally, this work was going to include a fifth mechanism, asynchronous message passing. However, as will be seen in later chapters, all of the decentralized protocols must make use of asynchronous message passing in order to implement their mechanisms. Therefore, it was deemed unnecessary to discuss message passing separately.

1.5. Summary of Results

This thesis succeeded in implementing each of the synchronization mechanisms in both a centralized and decentralized fashion. The observed results are that the distributed code produced is very expensive in terms of message passing. Each request for mutual exclusion in CCRs and monitors typically requires at least as many messages as there are application processes. Further, the need for keeping the data on all the nodes consistent also adds to the message overhead. In contrast, our results indicate that a centralized implementation avoids this overhead because it requires no messages for updating data and just a single request message for mutual exclusion on a local queue. Even the CSP algorithms, which attempt to minimize the message traffic by involving only those nodes doing the communicating, still have cases where they require more queries than in the centralized implementation.

SR proved to be an appropriate choice of languages for the implementations. It provided all four communication alternatives to which Scott refers. SR allowed the exploitation of concurrency much more easily than a language such as Ada, because Ada provides only rendezvous-type interactions. Ada could have been used but the code would have been messier and more expensive to execute.

1.6. Organization of Thesis

The next chapter gives background on each of the synchronization mechanisms. It also discusses the SR language, which is the environment in which the mechanisms were programmed. The subsequent four chapters discuss the implementations of each of the synchronization mechanisms: Semaphores, CCRs, monitors, and CSP are discussed in chapter 3 through 6 respectively. Chapter 7 compares the four mechanisms. Chapter 8 looks at possible extensions to this work.

CHAPTER 2

Background

The first part of this chapter gives an overview of each of the synchronization mechanisms. The second part of the chapter gives background on SR, the language in which the mechanisms are programmed for the experiments in this thesis. More complete descriptions of SR can be found in [Andr88] on which the second half of this chapter is based.

2.1. Overview of Synchronization Mechanisms

Semaphores are the lowest-level synchronization mechanism considered in this work. A semaphore is an integer variable, whose value is always greater than or equal to zero, with the operations **P** and **V** defined on it [Dijk68]. **P** is a request that decrements the variable, while **V** is a signal that increments the integer. A process performing a **P** operation does not proceed unless the integer is positive before the decrement or until it is matched with a **V** operation by another process. A **V** operation is non-blocking. No information hiding is implicit in code using semaphores. Code must be written so that the access to shared variables only occurs between a matching **P** and **V**, insuring exclusive access to the variables.

CCRs give mutual exclusion and conditional synchronization [Brin72]. With a CCR, the user declares a resource containing shared variables. Only one process at a time can be executing a region statement associated with a particular resource, implying exclusive access to the resource variables. A process trying to execute a region statement not only requires exclusive access but also requires the region's guard (conditional expression) be satisfied, giving conditional synchronization. The shared variables are forced into region statements, giving information hiding. However, the operations on the shared variables are spread across the processes that contain the region statements.

A monitor declares shared variables and defines the operations on these variables [Brin73]. Processes use procedure-like calls to these operations within the monitor. Thus, a monitor not only encapsulates the shared data but also the operations on that data. Monitors also contain condition variables, which have the operations **signal** and **wait** defined on them. These operations allow a process in the monitor to notify other processes or wait for another process to notify it that a particular condition has been met. Only one process can be executing within the monitor at a time. A process gives up control of the monitor upon executing a **wait** on a condition variable and is placed on a queue for that variable. The process will be removed from the queue when another process executes a **signal** on that variable. It is important to note that signals are not queued, i.e., if no processes are waiting on a condition variable when that condition variable is signaled, the **signal** has no effect.

There are a number of different protocols for the scheduling of processes when a **signal** occurs. This work will look at signal and continue (SC), signal and wait (SW), signal and urgent wait (SU). SC is used in Mesa [Lamp80]. A form of SW is used in Concurrent Pascal [Brin75]. SU is used in Pascal-Plus [Wels79]. In SC, when a process performs a **signal** it continues to execute inside the monitor. The first process, if any, waiting on the signaled condition variable is removed from the queue of that variable and waits for exclusive access to the monitor before executing. Under SW, when a process executes a **signal** on a condition variable that has processes waiting on it, the signaling process immediately gives up access to the monitor. The signaled process immediately starts to execute in the monitor. Protocol SU differs from SW only in what happens to the signaling process. In SW, the signaling process must wait for exclusive access with no preferential treatment over any of the other waiting processes. In SU, when a process performs a **signal** and gives up exclusive access to the monitor, it is placed on an urgent queue. When a process gives up mutual exclusion by methods other than signaling, the next process to execute will come from the urgent queue, if the queue is not empty.

A standard extension of the **wait** operation allows processes to wait according to some priority scheme. A process executing a prioritized wait supplies an integer rank parameter along with the name of the condition variable on which it wishes to wait. When a signal is made to a condition variable, the waiting process with the smallest rank is the one that is awakened.

Two additional functions are usually provided along with condition variables. The first is an **empty** function, which returns a boolean value indicating whether or not the condition variable has any processes waiting on it. The other is a **minrank** function, which only makes sense when used with prioritized condition variables. It returns the value of the smallest rank of all the processes waiting on the condition variable.

CSP contains synchronized interprocess communication, called input/output commands, where both the sender and receiver must be ready to execute the "matching" I/O command before it is actually executed (i.e., like an Ada rendezvous) [Hoar78]. I/O commands explicitly state the name of the process with which to communicate. In addition to the two processes having to name each other in order to be matching, the target variable of the input command must also be of the same type as the value of the expression of the output command. The following is an example of a pair of matching I/O commands assuming that both processes are at the I/O statements and a and b are both integer variables.

[Y:: ... X!a+3 ...|| X:: ... Y?b ...]

When the I/O statements are matching, the value of a+3 is assigned to b, and then the two processes continue on in parallel. The exclamation point signifies an output command and the question mark signifies an input command. CSP also allows guards in conditional and iterative statements to contain a single I/O command, which has a boolean value associated with the preparedness of the matching I/O command. This thesis, as in many other works on CSP, allows guards to contain output commands, even though the original CSP paper [Hoar78] did not. The I/O portion of the guard is true if a corresponding I/O command in the named process is ready to execute. As discussed below, the guard is false if the process named in the I/O command has terminated. The I/O guard has an indeterminate value if the corresponding process is not ready to communicate and has also not terminated. If none of the guards is immediately true, then the indeterminate value of an I/O guard can become determinate if the named process either terminates or reaches a matching I/O command. After a CSP process terminates, it is possible that other processes will try to reference it in an I/O command. Any such I/O command, either in a guard or in a simple I/O statment, fails. This condition is known as automatic or distributed termination. If a guard fails due to distributed termination, the guard is false. If a simple I/O statement fails due to distributed termination, the whole program fails.

2.2. The SR Distributed Programming Language

An SR program is composed from three kinds of separately-compiled *components*: resource specifications, resource bodies, and globals. Resources are the main building block; they are the unit of abstraction and encapsulation comparable to modules in Modula-2 or packages in Ada. Globals contain declarations of constants and types shared by resources; they have the form

global identifier
  declarations of constants and types
end identifier

A resource is a parameterized pattern, instances of which are created dynamically. Resources define operations and are implemented by one or more processes that execute in the same address space. Processes interact by means of operations; processes in the same resource may also share variables.

A resource has a *specification* and a *body*. The specification identifies other components the resource uses and declares operations, constants, and types exported by the resource. The basic form of a resource specification is:

resource identifier
  import identifier list
  declarations of operations, constants, and types
end identifier

All objects declared in the specification are exported from the resource; they may be used in other resources that import this resource.

The body of a resource contains the processes that implement the resource, declarations of objects shared by those processes, and initialization and finalization code. The general form of a body is:

body identifier(formal parameters)
  import identifier list
  declarations of shared objects
  processes
  initial block **end**
  final block **end**
end identifier

A body inherits all objects declared in or imported into the correspondingly named specification. As indicated, the body of a resource is parameterized; actual parameters are assigned during resource creation (see below). The shared objects may be constants, types, variables, and additional, local operations; none of these objects are visible outside the body. A *block* is a sequence of declarations and statements. The specification and body of a resource may be combined when it is not necessary to compile them separately.

Instances of resources are created dynamically by means of the **create** statement. Execution of

variable := **create** *resource_identifier*(arguments)

causes a new instance of the named resource to be created. Arguments are passed by value to the new instance and then the resource's initialization code, if any, is executed (as a process). Execution of **create** terminates when the initialization code terminates. A *resource capability* is returned by **create** and assigned to *variable.* This capability can be used to invoke the operations exported by the resource, can be copied and passed to other resources, and can be used in a **destroy** statement to destroy the instance. The **destroy** statement terminates after the resource's finalization code (if any) terminates and space allocated to the resource has been freed. An example illustrating resources and the use of the **create** statement appears later in this section.

By default, execution of **create** places a new resource in the same virtual machine (address space) as that of the resource on which **create** is executed. It is also possible to cause an instance to be placed in a different virtual machine by appending ‘‘**on** *vmcap*’’ to the **create** statement, where *vmcap* is a capability for a virtual machine. New virtual machines are generated by creating instances of the *vm* pseudo-resource in a manner analogous to creating a resource. A virtual machine can be placed on a specific physical machine (network node) by appending ‘‘**on** machine_id’’ to the **create** statement.

In addition to capabilities for resources, which are generated by **create** statements, SR provides capabilities for individual operations, such as one declared in a resource’s specification or local to a process (see below). Capabilities for individual operations provide a mechanism similar to procedural parameters. They are used to support finer-grained control over the communication paths between resources.

Operations are patterns for actions on objects. They are declared in **op** declarations, which can appear in resource specifications, within resources, or within processes. Operations are invoked by **call** or **send** statements; they are serviced by procedures (**proc**) or input (**in**) statements. In the four possible combinations, these primitives have the following effects:

Invocation	*Service*	*Effect*
call	**proc**	procedure call (possibly remote or recursive)
call	**in**	rendezvous
send	**proc**	dynamic process creation
send	**in**	message passing

These are the four alternatives to which Scott was referring.

Invocation statements have the forms

call operation(arguments)
send operation(arguments)

where operation is a field of a resource capability, an operation capability, or the name of an operation declared in the scope of the invocation statement. A **call** statement terminates when the operation has been serviced and results have been returned. A **send** statement terminates when the arguments have been stored on the machine on which the resource that services the operation resides. Thus, **call** is *synchronous* whereas **send** is *semi-synchronous* [Bern87].

An operation is serviced either by a **proc** or by one or more **in** statements. A **proc** is a generalization of a procedure: it is declared like a procedure and may be called like a procedure but has the semantics of a process. Its general form is

proc operation_identifier(formal_identifiers) **returns** result_identifier
block
end operation_identifier

where the operation identifier is the same as that in an **op** declaration in the resource containing the **proc.** Whenever that operation is invoked, an instance of the **proc** is created. This instance executes as a process and uses the formal and result identifiers (if any) to access the arguments of the invocation and to construct the result. If the operation was called, the caller waits for the instance to terminate. However, if the operation was invoked by **send**, the sender and instance of the **proc** execute concurrently; the effect in this case is like forking a process.

The other way to service operations is to employ **in** statements, which have the general form

in operation_command [] ... [] operation_command **ni**

Each operation command is structurally like a **proc** except it may also contain a synchronization and scheduling expression:

operation_identifier(formal_identifiers) **returns** result_identifier
  and synchronization_expression **by** scheduling_expression → block

An **in** statement delays the executing process until some invocation is selectable; then the corresponding block is executed. An invocation is *selectable* if the boolean-valued synchronization expression in the corresponding operation command is true; the synchronization expression is optional and is implicitly true if omitted. If more than one operation command has selectable invocations, the oldest selectable invocation is serviced. This default selection can be overridden by the use of **by**, which causes selectable invocations of the associated operation to be serviced in ascending order of the arithmetic scheduling expression following **by.**

To illustrate how resources and operations fit together, consider the bounded buffer resource shown in Figure 2.1. This resource provides two operations: *insert* and *remove*; *insert* places a value in the next available slot if one exists, while *remove* returns the oldest value from the buffer. A calling process is delayed should the buffer be full when an *insert* is attempted or empty when a *remove* is attempted. Note that the resource has a parameter *size*; its value determines the number of slots in the buffer. The use of resource parameters in this way allows instances to be created from the same pattern, yet still vary to a certain degree. Note also that executions of *insert* and *remove* are mutually exclusive since a single **in** statement services both. Finally, the **send** statement in the initialization code creates one instance of the main **proc** *bb.* Resources often contain

```
resource BoundedBuffer
    op insert(item : int)
    op remove() returns item : int
body BoundedBuffer(size : int)
    op bb()
    # create an instance of bb when BoundedBuffer is created.
    # see text for a better way to achieve the same effect.
    initial  send bb()  end
    proc bb()
        var store[0:size–1] : int
        var front, rear, count : int := 0, 0, 0
        do true →
            in insert(item) and count<size →
                    store[rear] := item;  rear := (rear+1)%size;  count++
             [] remove() returns item and count>0 →
                    item := store[front];  front := (front+1)%size; count−−
            ni
        od
    end
end BoundedBuffer
```

Figure 2.1 — A bounded buffer resource.

‘‘worker’’ processes such as *bb*. SR provides a **process** abbreviation to simplify programming such processes. For example, the above resource could be coded more compactly by deleting the declaration of the *bb* operation, deleting the initialization code, and replacing the line ‘‘**proc** *bb*()’’ by ‘‘**process** *bb*’’.

Figure 2.2 outlines how another resource, *User*, might create and use an instance of the *BoundedBuffer* resource. As shown, *User* imports *BoundedBuffer*. Its *Main* (background) process declares a resource capability variable, *buf*, that can be used to refer to an instance of *BoundedBuffer*. *Main* then creates an instance of *BoundedBuffer* with 20 slots and calls the *insert* operation in that instance to place a 33 in the buffer. The interface part of *BoundedBuffer* defines the two operations *insert* and *remove*.

```
spec User
    import BoundedBuffer
body User()
    process Main
        var buf: cap BoundedBuffer
         ...
        buf := create BoundedBuffer(20)
        buf.insert(33)
         ...
    end
end
```

Figure 2.2 — A user of *BoundedBuffer*.

The **receive** statement has the general form:

receive *operation*(*v1*, ..., *vN*)

It is a useful abbreviation for an **in** statement that waits for an invocation of *operation* and then assigns the values of the formal parameters to variables *v1*, ..., *vN*. Together with the **send** form of invocation, **receive** supports asynchronous message passing in a familiar way. Synchronous message passing is supported by **receive** together with the **call** form of invocation. Semaphores can also be simulated using **send** and **receive.** For example, given the declaration

op *s*()

the following statements simulate the semaphore **P** and **V** operations:

```
receive s()     # P operation
send s()        # V operation
```

This simulation is often used within a resource to synchronize access to shared resource variables.

SR provides a variety of data types and sequential statements similar to those found in other Pascal-class languages. However, their form and many of their details are different. Largely this is to facilitate the integration of the sequential and concurrent components of SR. It also results from our desire to make it easy to program commonly occurring algorithmic patterns.

In addition to the usual kinds of basic and structured types, SR provides capabilities and what is called an **optype** (operation type). An **optype** defines a pattern for an operation, i.e., the types of the parameters and return value. Such a type is used when the same operation pattern is used more than once, for example when several resources implement the same type of operation such as file read. What makes capabilities and operation types especially useful is that type-checking is based on structural equivalence; hence an operation capability can be bound to any operation that has the specified pattern.

SR's **if** and **do** statements are based on Dijkstra's guarded commands [Dijk76] mainly so their structure is similar to that of the **in** statement. However, if no boolean expression is true, **if** has no effect (unlike Dijkstra's **if**, which aborts in this situation). We also allow the last guard of **if**, **do**, and **in** to be ''**else**'', which is interpreted as the conjunction of the negations of the other boolean expressions.

SR also provides a novel iterative statement, **fa** (for all), that employs quantifiers. The form of **fa** is

fa quantifier , ..., quantifier → block **af**

where quantifiers have the general form

bound_variable := initial_value direction final_value **st** boolean_expression

For example, the single statement

fa *i* := lb(*a*) **to** ub(*a*)−1,
    j := *i*+1 **to** ub(*a*) **st** *a*[*i*]>*a*[*j*] →
        a[*i*] :=: *a*[*j*]   # swap statement
af

sorts array *a* into ascending order. This statement employs the builtin array lower-bound ('lb') and upper-bound ('ub') functions. Note how the range of values for the second bound variable *j* depends on *i*. Also note the use of **st** (such that), which is used to limit execution of the body of **fa** to those values of *i* and *j* for which *a*[*i*]>*a*[*j*].

Two final sequential statements are **exit** and **next.** The **exit** statement is used to force early termination of the smallest enclosing iterative statement. The **next** statement is used to force return to the loop control of the smallest enclosing iterative statement.

The current implementation of scheduling in SR contains no time slicing. The only times when a process is swapped out is if it becomes blocked waiting on an invocation or due to I/O operations or appears to be executing in an infinite loop. This non-preemptive scheduling is used to explains the behavior of certain executions in later chapters.

CHAPTER 3

Semaphores

This chapter discusses the centralized and distributed implementations of semaphores. To illustrate the implementations, it presents the semaphore solution to the roller coaster problem. It also discusses more efficient algorithms for the implementation of a binary semaphore for use in mutual exclusion.

3.1. Implementations

As discussed in Chapter 2, a **P** operation on a semaphore does not proceed unless the integer associated with the semaphore is positive or until a matching **V** operation is performed. The case where the integer is positive is equivalent to having some positive number of **V** operations that are unmatched by **P** operations. The centralized implementation uses this idea of unmatched operations rather than an explicit integer variable to keep track of the value of the semaphore.

The following is the essence of the server code.

```
do true →
  receive V_operation()
  receive P_operation()
od
```

Since a **V** is non-blocking, a process performing a **V** uses a **send** to V_operation. On the other hand, a process performing a **P** must use a **call** to P_operation, since **P** is blocking. If no **P** requests are pending, the value of the semaphore is the number of V messages waiting to be serviced plus one. The extra one reflects that the server has serviced the first invocation and is now waiting for an invocation of P_operation. If a semaphore is to have a positive initial value n, then initialization code must send to V_operation n times.

Schneider's distributed semaphore algorithm is based upon Lamport's work [Lamp78], which had two main components: (1) a distributed queue and (2) a logical clock. The distributed queue is just an extension of a standard queue to a distributed environment. Rather than sending requests for a resource to a single queue, the entries for the queue are broadcast to all nodes. Each node maintains its own copy of the queue. To be able to order the entries at the nodes, each request contains a timestamp, based on logical clock time (see below), of when it was sent. The order is then simply based upon the timestamp, with the oldest request having the smallest timestamp. Processes are given a unique process number so that if two or more requests have the same timestamp, the request from the process with the smallest process number is deemed to be the oldest. The timestamp of a message is set using Lamport's logical clocks, which follow two rules. The first rule is that the clock at a node will increase by at least one between each event at the node. An event, in this case, is defined as the sending of or receipt of a message. The second rule states that when a process receives a message from another process the clock is set to the maximum of the timestamp of the message received and its own clock plus one.

This algorithm makes the assumption that messages sent by a node are received at other nodes in the same order they were sent. This assumption allows a node to know that when a message from node j is received with timestamp x, that all succeeding messages from node j will have a timestamp greater than x. It does not, however, say anything about the order in which messages from different nodes will be received.

When a request is received, a timestamped acknowledgement is sent back to the requester. This acknowledgement is not a direct granting of the request. Rather it notifies the requester that no more messages from the acknowledging process will be received with a timestamp less than or equal to the original request. The requester knows this because of the message ordering assumption and the rules for updating the clock. The second clock rule ensures that the clock at all acknowledging processes will be at least equal to the timestamp of the request. The first rule makes it necessary for any future request by an acknowledging process to be at least one greater than the message being acknowledged. Therefore, no future request can have a timestamp of less than or equal value than that of the acknowledged message. When a request becomes fully acknowledged, all older requests in the queue are considered fully acknowledged.

When a process has finished using a resource, it broadcasts a release message. Release messages do not require acknowledgements since the relative order of release messages to each other is unimportant. When a release message is received from node *j*, the oldest request message from node *j* can be removed from the queue. A process request is granted when its request is the oldest fully acknowledged unsatisfied request in the queue and there have been more releases than satisfied requests.

Schneider's semaphore algorithm is Lamport's algorithm with a **P** being the resource requests and a **V** being the release message. This algorithm requires that acknowledgements be broadcast to all nodes. This requirement, however, is unnecessary and expensive as will be discussed below. The algorithm makes use of two counters, one for the number of V messages received and one for the number of fully acknowledged P requests. As stated above, when a P message becomes fully acknowledged, all older messages in the queue are now considered fully acknowledged. The P counter is incremented by one for each fully acked[1] P message and the message is removed from the queue. If one of the fully acked messages at a site is that site's request, the value of the P counter is saved after that request message is removed from the queue. The P request is granted when the value of the V counter is equal to or greater than the saved value of the P counter.

Schneider claims "The capacity of that queue need not exceed the number of sites in the system." When the algorithm is being used for mutual exclusion, as it was in his paper, this bound holds. In the case of mutual exclusion, a site must send a release message before sending another request. By the message ordering assumption, the release message will be received at a site before the next request is received. The satisfied request is deleted upon receipt of the release, ensuring the maximum size of the queue to be the number of sites in the system. In more general uses of semaphores, however, this bound on the size of the queue is too strong. As will be seen in the solution to the roller coaster problem, a process can perform **P**'s on a semaphore without ever having to perform **V**'s on the semaphore. This lack of release messages makes it possible for multiple request from the same site to be in another site's queue at one time. The earlier requests will already have been satisfied at the requesting site but have not yet been completely acknowledged at other sites. When queried, Dr. Schneider verified that the bound was too strong [Schn89].

Our first implementation of distributed semaphores is a slight modification of Schneider's algorithm. The basic algorithm for this implementation is shown in Figure 3.1. Acknowledgement messages are only sent to the sender of the request. Also, the

---

[1] Henceforth, *acked* will be used interchangeably with *acknowledged*, and *ack* will be used interchangeably with *acknowledgement.*

queue of requests has been replaced by two integer variables. Note that a process that sends out a **P** request only needs to know how many requests need to be serviced before its own request can be serviced. The process need not know which processes' requests are the older ones, just how many. Also note if acknowledgements are not received for other processes' requests, when a local request message becomes fully acknowledged, all older request messages become fully acknowledged. Therefore, all that needs to be stored when a **P** request message arrives is whether the request is before or after any current local request. This information is stored in the variables *earlier* and *later*.

The above algorithm in a system with n sites would require $2(n - 1)$ messages for a node to have its **P** request message fully acknowledged. If acknowledgements were sent to all nodes rather than just the requesting node, the algorithm would require $(n - 1)^2 + (n - 1)$ messages. These extra messages serve little purpose except possibly to aid in speeding up detection of error conditions. When a node fails, it will stop being able to send acknowledgements. If the acks are broadcasted, then all nodes would fail to receive acks upon the first request after failure.

Both Lamport and Schneider point out that it is possible to use request messages as a form of implicit acknowledgements. This improvement is again based on the fact that an ack just notifies a requester that no message with timestamp less than or equal to the timestamp of the acknowledgement will be received. The timestamp on a request message can serve the same purpose. For example, suppose node j receives a request with timestamp x from node i and node j has already sent a request with a timestamp y, which is greater than or equal to x. Then, no acknowledgement message need be sent. When node i receives the request from node j with timestamp y, it will consider this message as an ack to its own request. Our second implementation uses such implicit

```
process semaphore_handler
  do true →
    in local_request( request_type) →
          # receiving request from the process we are serving
          clck++
      fa i := 1 to N st i ≠ me →
        send sem_handlers[ i].request( request_type, clck, me)
      af
          if request_type = Poper →
            # P request, put message on Q to be fully acked
            # and save time of our request
            ts_my_req := clck
           [] else →
            # V request, just update Vcount
            Vcount++
          fi
        [] request( type_req, ts, who_req) →
          clck := max( ts, clck + 1)
          if type_req = Poper →
            # need to ack a P message
            send sem_handlers[ who_req].ack_op( clck)
            # determine if the message is before or after our msg
            if ( ts_my_req > 0) and
              (( ts > ts_my_req) or
               ( ts = ts_my_req and who_req > me)) →
                  later++
             [] else →
                  earlier++
            fi
           [] else →
            # this is a V message, increment V count
            Vcount++
        # check if a P that me requested is fully acked.
            if (myPcount > 0) and (myPcount ≤ Vcount) →
              send request_serviced()
              myPcount := 0
            fi
          fi
```

Figure 3.1a — Distributed Semaphore Handler

```
            [] ack_op( tsack) →
              # receiving an ack from another process
              clck := max( tsack, clck + 1)
              ack_count++
              if ack_count = acks_needed →
                # our request is fully acknowledged
                Pcount := Pcount + earlier + 1
                myPcount := Pcount
                earlier := later
                later := 0
                ts_my_req := 0
                if (myPcount > 0) and (myPcount ≤ Vcount) →
                  send request_serviced()
                  myPcount := 0
                fi
              fi
            ni
      od
   end
```

Figure 3.1b — Distributed Semaphore Handler (continued)

acknowledgements. Figure 3.2 shows the modifications required to use implicit acks. All this entails is that when a request is received, the receiver checks if this request acknowledges a local request and whether it is necessary to send an acknowledgement message to the request. An acknowledgement array is also needed; it is used to ensure that messages from a site do not get counted as multiple acks to a single request.

SR makes it easy to satisfy the message ordering assumption. The use of **send**, with its semi-synchronous nature, ensures that requests are stored in the order that a node sent them. By not using a scheduling expression on the **in** statement, requests will be removed in a FIFO manner. Together, these two facts satisfy the assumption.

3.2. Semaphore Solutions to the Roller Coaster Problem

The following solution to the roller coaster problem, shown in Figure 3.3, uses four semaphores. The vehicle uses the semaphore *board_vehicle* to notify passengers of the

```
[] request( type_req, ts, who_req) →
  clck := max( ts, clck + 1)
  if type_req = Poper →
    if ts_my_req < ts →
      # need to ack a P message
      send sem_handlers[ who_req].ack_op( me, clck)
    [] else →
          # our last request is newer than this request
          # so have already implicitly acked request
    fi
    ...
    # check if message received implicitly acks our latest P request
    if Preq_out and not ack_array[ who_req] and ts ≥ ts_my_req →
      ack_array[ who_req] := true
      ack_count+
      if ack_count = acks_needed →
        # duplicate code as in Figure 3.1b within same conditional
      fi
    fi
  fi
[] ack_op( who_ack, tsack) →
  ack_array[ who_ack] := true
  ...
```

Figure 3.2 — Modifications to Use Implicit Acknowledgements

availability of seats. Passengers perform a **P** on this semaphore to wait in line. Passengers use the semaphore *passenger_on_board* to inform the car that a seat has been taken. The car performs **P**'s on this semaphore to wait for all the seats to be filled before leaving the station. The vehicle uses the semaphore *debark* to tell passengers that the ride is over. Passengers wait on this semaphore to know when they may exit the car. Finally, passengers use *passenger_off* to signify their having exited the car. The vehicle waits on this semaphore to ensure all the passengers are out before going to pick up passengers for the next ride.

The constant N is the number of passengers allowed in one vehicle. The constant R is the number of rides each passenger wishes to take. The variable T is the number of trips the vehicle needs to make to satisfy all the passengers. The variable A is the number of passengers who wish to ride.

The following runs were made with N equal to four and the number of times a passenger wishes to ride equal to five. The different implementations were run with the number of passengers wishing to ride as eight and then as six. The code and output of runs using semaphores appear in Appendix I. The output for eight passengers is from the

```
process vehicle();
  fa trips := 1 to T →
        # arrive at station, notify N passengers that they can board
        fa i := 1 to N →
          V ( board_vehicle)
        af
        # wait for N passengers to have boarded
        fa i := 1 to N →
          P ( passenger_on_board)
        af
        # leave station and "do" the ride
        # ride is now over notify passengers to debark
        fa i := 1 to N →
          V ( debark)
        af
        # wait for N passengers to have debarked
        fa i := 1 to N →
          P ( passenger_off)
        af
  af
end vehicle;
process passenger();
  fa rides := 1 to R →
        # wait for notification that it is ok to board
        P ( board_vehicle)
        # notify vehicle that we have boarded
        V ( passenger_on_board)
        # now wait for ride to be over
        P ( debark)
        # notify vehicle that we are off
        V ( passenger_off)
  af
end passenger;
```

Figure 3.3 — Semaphore Solution of Roller Coaster Problem

run using the code with implicit acknowledgements. The output from the other two implementations were identical except that they did not contain the count of acknowledgement messages. The program completed with all eight passengers having ridden five times. However, if the passengers had varying length delays between queuing up for another ride, the solutions could have terminated without all passengers having ridden their allotted five times. Such outcomes would still be deemed correct since the implementations of the semaphores are fair (i.e., free of starvation).

The centralized solution required four instantiations of the resource *cent_Semaphore*, one for each of the semaphores. This implementation requires just one message for a **P** request and only one message for a **V** request. Therefore, the number of messages required by this implementation if the program terminates with all passengers having ridden five times is

$$(R * 4 * A) + N * 4 * T.$$

The passengers make five trips, each requiring two **P**'s and two **V**'s. The car makes two sets of four **P**'s and two sets of four **V**'s per trip. So, the eight passenger case required 320 messages, since the car made ten trips.

Both the distributed algorithms required four instantiations of the *dist_Semaphore* resource per passenger and car resource. It should be noted that instead of each site (i.e., process plus four semaphore servers) having a single clock, each semaphore has its own logical clock. The number of messages required by the algorithm with only explicit acknowledgements can be directly calculated. A + 1 is the total number of passenger and car processes, a **P** request requires 2A messages and a **V** requires A messages. Therefore, the number of messages required by the protocol in this implementation if the program terminates with all passengers having ridden five times is

$$(R * 2 * (2 * A + A) * A) + N * 2 * (2 * A + A) * T.$$

Table 3.1 contains data on the messages required by the three implementations. As was stated, the above equation only counts the messages between server processes. It does not include the messages between the passenger and car processes and their handlers. These application-to-handler messages are included in the first two columns of the table. There are actually even more messages in the implementation that are not being counted. The **proc**s P_oper and V_oper provide a cleaner interface to the application processes. These procs use messages to pass on the requests. The code could have been written in another way that would not have necessitated the use of these extra messages, so they were not included in the counts. The local_request operation could be made visible to application processes. These processes would make calls to local_oper with appropriate parameters, rather than calls P_oper and V_oper. The second distributed algorithm used 212 implicit acks, a savings of about 16.5%. This percentage savings was used to estimate how many messages would be required in the 16 passenger case. The other two entries in the last column of the table are exact. In the centralized case, the number of messages grows linearly as the number of processes increase. However, in the decentralized case, the number of messages grows as a second order polynomial as the number of processes increase.

The solutions in the eight passenger case behaved in a predictable fashion. This is because the passengers have no self-imposed delay between their exiting the ride and requesting a seat on the next ride, and due to current implementation of scheduling in SR. Any solution that is an integer multiple of N will behave in this orderly fashion. There-

Algorithm	local requests/ releases	request fulfilled	global requests/ releases	acks	total messages for 8 passengers	total messages for 16 passengers
Centralized	320	0	0	0	320	640
Distributed	320	160	2560	1280	4320	16320
Distributed with implicit acks	320	160	2560	1068	4108	15472 (estimate)

Table 3.1 — Message Counts for the Semaphore Implementations

fore, runs with six passengers were made to see if the algorithms were still fair under altered conditions. These solutions end with processes being blocked. This behavior is only natural since it was impossible for all the passengers to have completed all five rides. Under the given conditions, the solutions are correct, since they all handle the semaphores in a fair manner. Two apparent differences can be seen between the distributed and centralized solutions. In the cases where the centralized algorithm shows passengers boarding in the order 5, 6, 1 and 2, the distributed algorithms show an order of 1, 2, 5 and 6. The order of boarding is different, but the same four passengers ride. In the centralized algorithm, the **P** requests are satisfied in the order they were received. Since, the requests from 5 and 6 are older, they are satisfied first. In the distributed algorithms, the requests by passengers 5 and 6 are also older, which results in them ending up with the two lowest *myPcount*'s. Passengers 1 and 2 end up with the next two lowest *myPcount*'s. Because the car sends out four **V**'s to *board_vehicle* all four passengers are now allowed to board. The algorithm sends the messages to the processes in the order of their process numbers. Therefore, passengers 1 and 2 receive and handle the four **V**'s before 5 and 6 do. Hence, 1 and 2 board before 5 and 6.

The other difference can be seen to start in the fifth trip by the car in the solution with implicit acks. Instead of 1, 2, 5, and 6 riding as is the case in the centralized and explicit distributed algorithm, 1, 3, 5, and 6 ride. Further tests, which are not shown, determined that this different ordering of passengers was not an artifact of the implicit algorithm but rather of the distributed algorithm without broadcasting acks. When the distributed algorithm with explicit acks was run with passengers riding ten times instead of five, the same kind of behavior was found in later rides. Though this solution is different then those given by the centralized algorithm, it is still fair. The two oldest requests, those from passengers 5 and 6, were serviced. Passengers 1 through 4 have just exited the bus and which of their requests should be considered the oldest is up to the scheduling algorithm and its effect on the timestamps of the requests.

3.3. Semaphores for Mutual Exclusion

The use of semaphores in centralized implementations for mutual exclusion will be discussed in Chapters 4 and 5.

Either of the above distributed algorithms could be used for mutual exclusion, but they are more expensive than necessary. The first algorithm with explicit acks would require $3(n - 1)$ messages. The first $2(n - 1)$ messages are required to get the **P** request fully acknowledged. The other $(n - 1)$ messages are due to having to release mutual exclusion via a **V** request.

Ricart and Agrawala [Rica81] introduced an algorithm that required $2(n - 1)$ messages. Their algorithm also uses Lamport's logical clocks at every site and broadcasts **P** requests in the same fashion as the semaphore algorithm. However, when a node receives a request it does not always immediately send an ack. Rather, it checks to see if the request is older (i.e., has a smaller timestamp) than a local request. If the request is older, then an ack is sent to the requester. Otherwise, the requester's process number is saved on a defer list to be acked later. The processes on this list are not sent acks until the deferring process has received mutual exclusion and is ready to release it. Any process whose request is received while a node has mutual exclusion is also added to the defer list. When mutual exclusion is to be released, all the processes on the defer list are acked. This alleviates the need for the release messages. If no processes are in the defer list, then no messages are sent. Full acknowledgement in this algorithm means the process has mutual exclusion.

Later another algorithm that requires only n messages was introduced separately by two groups, Ricart and Agrawala[Rica83], and Suzuki and Kasami[Suzu85]. These algorithms are discussed in depth by Raynal[Rayn86]. In the above $2(n - 1)$ algorithm, when a process releases exclusion and no processes are on its defer list, this process must still broadcast a request and receive acks before again having mutual exclusion. The new algorithm instead initially gives the privilege of mutual exclusion to one of the processes. This process holds onto this privilege until it is not currently making use of mutex[2] and receives an unsatisfied request. While a process has the privilege it can obtain mutex whenever it likes, as long as it does not receive an unsatisfied request. It can therefore obtain mutex without sending any messages. When a node without the privilege requires mutual exclusion, it again broadcasts a **P** request to all nodes. In this algorithm, instead of timestamping requests, the request contains a request number. This request number is just a count of the number of times the process has requested mutex. All nodes maintain an array containing the request numbers of the most recent **P** request by each of the other nodes. The node currently holding the privilege maintains an additional array containing the request numbers of the most recently satisfied request of all nodes. It also has a

[2] Henceforth, *mutex* will be used interchangeably with *mutual exclusion.*

queue of processes numbers who have unsatisfied **P** requests. When the process holding the privilege is not currently requesting mutex and the queue is not empty, the privilege is passed to the process at the head of the queue. Along with the privilege is passed the array of satisfied requests and the queue of requests. Therefore, in general, a request will require n messages: n - 1 request messages, and one message when being passed the privilege. A straightforward implementation of this algorithm is included in Appendix III as part of the implementation of monitors.

Note that neither the Ricart and Agrawala algorithms nor the Suzuki and Kasami algorithm require the message ordering assumption of the Schneider algorithm.

3.4. Summary

This chapter discussed centralized and decentralized implementations for semaphores. The distributed implementation make use of distributed queues and logical clocks. The distributed algorithm was modified to make use of implicit acknowledgements to global request messages. The implicit acks resulted in a savings of about 16% in the number of acks required. This chapter also describes optimizations to the distributed algorithm for binary semaphores. The most efficient algorithm examined is used for obtaining mutex in the distributed implementations of CCRs and monitors in the following chapters.

CHAPTER 4

Conditional Critical Regions

This chapter discusses the centralized and distributed implementations of CCRs. Then, it discusses anticipatory execution of CCRs. Finally, it examines the CCR solution to the roller coaster problem.

The general form for the declaration of a CCR is:

> **var** *r* : **resource**
>   v1: type; v2: type; ... vN: type
> **end**

where v1, v2, ... vN are resource variables. These variables are only allowed to be referenced within a region statement of the resource *r*.

The general form for a region statement is:

> **region** *r* **when** B $\rightarrow$ S **end**

where B is a boolean expression containing variables local to the process containing the region statement and/or resource variables; and S is a statement list again containing local and/or resource variables. CCRs force mutual exclusion by allowing only one region statement (of each resource defined) to be executing at any one time. Also, conditional synchronization is given by the guard B, such that S will only be executed if B is true.

4.1. Implementations

4.1.1. Centralized Solution

SR allows a neat and concise way to implement a centralized solution to CCRs. Each CCR resource maps to an SR resource. It contains declarations for all of the resource variables and declarations of operations for each unique region statement. This SR resource has a single process containing an **in** statement enclosed in an infinite loop. Consider, for example, the CCR shown in Figure 4.1. The resource *R* is transformed into the SR code shown in Figure 4.2. r1 and r2 are arbitrary names given to the operations for the two unique region statements. If local variables of a process including a region statement are used in the guard and/or statement list, they too would need to be included as parameters to the operation. The **in** statement within the the **do** loop provides mutual exclusion by the sequential nature of the code. Each time an **in** statement is executed, only one invocation is serviced. One region statement is executed per pass through the loop, hence mutual exclusion. Since the **in** statements do not contain scheduling

---

```
var R : resource
   v1, v2, v3 : int
end
process P1
  ...
  region R when B1 → S1 end
  ...
end P1
process P2
  ...
  region R when B2 → S2 end
  ...
end P2
```

Figure 4.1 — Example of a CCR Resource *R*

```
process server
   var v1, v2, v3 : int
   do true →
     in r1() and B1 → S1
      [] r2() and B2 → S2
     ni
   od
end server
process P1
  ...
  call r1()
  ...
end P1
process P2
  ...
  call r2()
  ...
end P2
```

Figure 4.2 — SR Coding of Resource *R*

expressions, each pass through the loop will service the oldest invocation whose synchronization expression is true. This ensures a fair implementation.

Andrews [Andr89] outlines a few other implementations for CCRs. One of these solutions is semaphore-based, which makes use of three semaphores plus two counters. The semaphores are used to break processes into three groups: processes waiting to try to execute a region statement; processes whose guards were false and are waiting for updated data; and processes waiting to re-evaluate their guards. The counters are used to keep track of the number of processes in the latter two groups. Another solution given by Andrews is a kernel-based implementation. This approach defines primitive operations, **Create_resource**, **Enter**, **Delay**, and **Exit**, which take the place of the semaphore and counter interactions of the semaphore solution.

4.1.2. Distributed Solution

When working in a distributed environment, satisfying the requirements of CCRs is more complicated. In the centralized solution, it is possible to obtain mutex and have the guard test to true in one statement. In the distributed case, the mutex and conditional synchronization must be obtained separately. The centralized solution has one server process. The distributed solution allocates one handler process per node. Shared variables also complicate the distributed solution. In particular, when a region is executed and it modifies resource variables, this updated data must be passed to all the other handler processes for this resource.

4.1.2.1. Basic Handler Scheme

When writing algorithms using CCRs, a region statement is just another statement in the code. The region statements were implemented not as in-line code but as operations in a separate SR resource. This was done mainly for ease of debugging, but it also "hides" the CCR’s variables from application code. Each application process has an instantiation of the CCR handler. Each CCR server has its own instantiation of the mutex server, which is called to get mutual exclusion. Figure 4.3 shows the general code for implementing one of the region statements. The code is again an **in** statement within an infinite loop. The basic idea is to get mutex and then get the current values of the resource variables. The guard is then tested. If the guard is true, then the statement list is executed, the resource variables are broadcast, and mutex is released. However, if the guard is false, it is necessary to give up mutex and try again. To avoid busy waiting, the conditional to wait for an update message is included. This ensures that a region statement that failed does not get mutex without having received an update.

```
in ...
[] r1() →
  do true →
    if (need_new_data) →
      receive Pass_globals( resource variables)
      need_new_data := false
    fi
    # get mutual exclusion
    call mutex_server.get_exclusion()
    # get the most up to date resource variable values
    do (?Pass_globals ≠ 0) →
      receive Pass_globals( resource variables)
    od
    need_new_data := false
    if B1 →
      # guard is true exit loop
      exit
     [] else →
      # guard not true must give up mutex, wait for new data
      need_new_data := true
      mutex_server.release()
    fi
  od
  # do S
  S1
  # notify all other CCR servers of updated variables
  fa i := 1 to total_processes st i ≠ me →
    send others[ i].Pass_globals( resource variables)
  af
  mutex_server.release()
ni
```

Figure 4.3 — Distributed Region Statement Handler

4.1.2.2. The Starvation Problem

Unfortunately, the code in Figure 4.3 can allow the starvation of processes. The following is an example of this starvation. Processes 1 and 2 execute a loop containing a region statement, r1. Process 3 executes a loop containing a region statement, r2. After the execution of r2, one process can execute r1 and r2 cannot be executed again until r1 is executed. With the above implementation, processes 3 and 1 can alternate executing their region statements and process 2 will never get to execute r2.

The causes of starvation are the distributed mutex algorithms and the method for sending resource variable updates. The mutex algorithms have a bias to lower numbered processes when processes make requests at the same time. In many of the algorithms, this bias is explicitly spelled out: when multiple processes have made a request with the same timestamp, the request with the smallest process number is given priority. In the Suzuki and Kasami [Suzu85] algorithm originally used, the bias is subtle. This algorithm uses no timestamp only a request number. The relative values of request numbers from different nodes is not a consideration in passing on the privilege for mutex. Therefore, the "simultaneity" of requests is based upon when the requests are received and not upon the value of the request numbers. If the process holding the privilege is currently making use of mutual exclusion, the mutex handler process is waiting to receive a release message. While the mutex handler is waiting, it can receive requests. These requests just have their request number stored. It is these messages that are considered to have been made at the same time. After the process holding the privilege releases mutex, the mutex server then checks to see if any requests that have been received are unsatisfied. The unsatisfied requests are added to a queue. The bias comes from this check for new requests. This check is done in order from lowest to highest process number, meaning that the handling of requests made at "the same time" is biased in favor of lower numbered processes.

The method of "broadcasting" resource variables introduces this same bias. The "broadcasting" actually involves looping and sending the messages one at a time to each of the processes. This will wake up the lower number processes before the higher numbered ones because that is the order in which the messages are sent. These two biases can combine to cause starvation.

Starvation can occur in the following manner. Multiple processes are attempting to execute the same region statement, call it r1, and only a few processes are allowed to succeed per execution of another region statement *x* by some process. Let *L* denote the set of processes that succeed in executing r1 (these will be the requesting processes with the lowest process numbers as discussed above) and let *K* be the processes failed in their attempt. Starvation can occur if the processes in *L* are able to return to the point where they again wish to execute r1 before the execution of *x* occurs again. The processes in *K* will be blocked at the **receive** statement before the request for mutex. These processes will remain blocked until another process successfully executes a region statement and sends updated values. But until the process that succeeds in executing *x*, the processes in *K* will return to being blocked. The processes in *L* will become blocked too if *x* has yet to occur. When *x* does occur, all the processes in *L* and *K* will be unblocked and will request mutex. The processes in *L* will again win over the processes in *K* because their requests will be made at the "same time". This sequence can continually repeat, hence, starvation.

4.1.2.3. Eliminating Starvation

The starvation problem was solved by the addition of a failed list. This list contains the processes whose guards were false the last time they attempted to execute a region statement. The members of set *K* would be in this list. Whenever mutex is released after a successful execution of a region statement, mutex will be given to all the processes on the failed list before being given to any process with a new request. If one of the processes from the failed list succeeds when given mutex, the process will be removed from the failed list and mutex will again be granted to the process at the head of the failed list. The failed list gives older requests priority over new requests in a manner similar to the centralized solution.

The only change required to the CCR handler is that when releasing mutex it now returns a boolean telling if it executed the region statement (true) or will have to try again after receiving new data (false).

The mutex handler must also change in a number of ways. The process holding the privilege now maintains the failed list. When a handler receives a release with a parameter of false, the process adds itself to the end of the failed list and marks the fact that its attempt failed. When a process that has failed on its last attempt requests mutex again, it does not broadcast its request, it instead just waits to receive the privilege. Thus, on a retry, the cost of gaining mutex is a single message not *n*.

4.1.3. Anticipatory Implementation

Jefferson in his paper Virtual Time [Jeff85] and Reed in his paper Implementing Atomic Actions on Decentralized Data [Reed83] discuss working with shared data in a distributed environment. One of the goals of both of these works is to exploit concurrency as much as possible. To this end, they allow processes to work on shared data without initially ensuring their exclusive access to the data. Jefferson allows the modified data to be transmitted and used at other sites. If later on, it is found that a synchronization conflict has occurred, "anti-messages" must be sent to rollback the data to a correct state. Reed also allows modified data to be transmitted but other processes cannot use this data until the data is "committed". If a conflict is found, the transmitted data is just erased at all sites.

The anticipatory handler for CCRs, outlined in Figure 4.4, is similar in overall structure to the distributed handler (Figure 4.3). In applying this anticipatory idea to CCRs, update messages are not allowed to be sent until synchronization is achieved, so that rollback has only to be done locally. Instead of requesting mutual exclusion before trying to test the guard, as was done above, the code tests the guard first. It also checks to see if any update data has come from another CCR server. Once there is no known update data and the guard is true, the CCR server requests its mutex server to try to gain mutual exclusion. The CCR server, however, does not block waiting for a response to its request. It saves any local variables that might be changed by the execution of the statement list S and then executes S! It then waits to receive a response to its mutex request. This execution might need to be rolled back for a couple of reasons. It might not get mutual exclusion (as discussed below). Thus, another CCR server has or is about to obtain exclusive access to the region. Another possibility is that between the time mutex was requested and when it was granted, another CCR server could complete and send updated data. Either way, the server must undo its modifications to local variables passed in, release mutex if received, and totally restart the attempt to execute the region. However, if mutex was obtained and no other server finished and sent updated data, then all that remains is for updates to be sent to all the other CCR servers and mutex to be released.

This anticipatory implementation can also have problems with starvation. Therefore, the modified mutex code with the addition of the failed list, is the starting point for this implementation. When an update message was sent in the distributed algorithm, it is known that all processes on the failed list will request mutex again. However, in the anticipatory case, new data might make the guard false of a process on the failed list. So, the

```
in r1() →
  do true →
    do true →
          if B1 and (?Pass_globals = 0) →
            exit
          [] else →
            # need to wait for updated data
            in Pass_globals( resource variables) →
            [] check( need_priv) and ?Pass_globals = 0 →
              need_priv := false
            ni
          fi
    od
    temp_resource_vars := resource_vars # save resource variables
    # guard is true, request mutual exclusion
    send mutex_server.get_exclusion( got_excl)
    # do S as if we have exclusion
    S1
    in got_excl( temp_have_excl) →
          have_excl := temp_have_excl
    [] check( need_priv) →
      need_priv := true
          reply
          receive got_excl( have_excl)
    ni
    if not have_excl or ?Pass_globals ≠ 0 →
          # a CCR server has exited region since we
          # tested guard or we didn't get exclusion
          # Therefore we must rollback
          if have_excl →
            # need to release exclusion
            mutex_server.release( false)
          fi
          # restore resource variable to values prior to executing S
          resource_vars := temp_resource_vars
          next
    fi
    # notify all other CCR servers of updated variables
    fa i := 1 to total_processes st i ≠ me →
          send others[ i].Pass_globals( resource variables)
    af
    mutex_server.release( true)
    exit
  od
```

Figure 4.4 — Anticipatory Region Statement Handler

mutex server, when being passed the privilege while being on the failed list and not having currently received a request from its customer, must check with the customer to see if it wants mutex. Another modification is that a deny message is sent in response to a request when a mutex server either has mutex or is waiting for a release message. This change was not necessary but was seen as an optimization since if a process receives a delayed reply, it will find that an update message will have been received invalidating its execution of S. Therefore, the time a CCR server has to wait to find that it needs to try again is shortened.

4.2. CCR Solutions to the Roller Coaster Problem

The following solution to the roller coaster problem, shown in Figure 4.5, uses a single CCR resource, *roller_coaster*. This resource contains five resource variables. The number of times the vehicle has arrived at the station is maintained in *vin.* The car notifies passengers of the availability of seats by incrementing *vin.* The total number of passengers that have taken a seat in a car is contained in *pin.* When a passenger notices that there is an empty seat, it takes the seat and signals this action by incrementing *pin.* The passenger knows that a seat is empty if the total number of seats that have been available is greater than *pin.* The number of trips the vehicle made is maintained in *vout.* The vehicle waits until it is full before leaving. Once it is done with the ride, the vehicle notifies the passengers on board that they can debark by incrementing *vout.* The total number of passengers that have completed the ride and have debarked from the car is held in *pout.* When a passenger sees that the ride is over, it leaves the car and marks the event by incrementing *pout.* The number of times that the vehicle has completed the ride and had all passengers get off is maintained in *vdest.* When the car sees that all passengers have left their seats, it increments *vdest.*

```
var roller_coaster : resource
    pin : int := 0;  pout : int := 0;  vout : int := 0;
    vin : int := 0;  vdest : int := 0
end
Vehicle ::
   fa trips := 1 to T →
    # arrive at station
    region roller_coaster →
              # notify passengers of arrival
              vin++
    end
    # wait for N passengers to board
    region roller_coaster when ( vout + 1) * N ≤ pin →
              # do ride and notify passengers that ride is over
              vout++
    end
    # wait for the passengers to debark
    region roller_coaster when ( vdest + 1) * N ≤ pout →
              # mark that car has finished a circuit
              vdest++
    end
   af
end
Passenger ::
   fa rides := 1 to R →
    # wait for a vehicle with room to board
    region roller_coaster when ( pin + 1) ≤ vin * N →
              # notify car that we have boarded
              pin++
    end
    # wait for ride to be over
    region roller_coaster when ( pout + 1) ≤ vout * N →
              # notify car that we are off
              pout++
    end
   af
end
```

Figure 4.5 — CCR Solution of Roller Coaster Problem

It should be noted that it was not necessary for running totals to be kept in the above solution. Instead, there could have been just two resource variables. One to hold the number of seats available for boarding and one for the number that can now be vacated. The above solution was used for purposes of comparison to the more general problem with multiple cars on the track.

Runs were again made with the number of passengers per car, N, being four and the number of rides per passenger, R, being five. The centralized, distributed and anticipatory implementations were run with the number of passengers wishing to ride, A, set to eight and then six. The number of trips the car needs to make, T, is ten for eight passenger and eight for six passengers. The code and output of these CCR implementations appears in Appendix II.

The centralized implementation required a single instance of the resource *roller_server_cent*, which included code for all five region statements. Each region statement needs only a single message to execute it. Therefore, the total number of messages to implement a solution that ran to completion with all passengers riding R times is

$$R * 2 * A + 3 * T$$

The passenger has two rendezvouses per ride and the vehicle has three per trip. Hence, the eight passenger run required 110 messages.

The distributed and anticipatory algorithms require many more processes. Each passenger and vehicle process requires an instance of the *CCR_server* resource and each of these handler processes require an instance of the *Mutex_server* resource. In general, with these two algorithms it is not possible to know what the interactions will be, so it is not possible to give an equation for the amount of message passing. The passenger and vehicle processes will make the same number of requests to execute region statments as in the centralized algorithm. However, where the centralized algorithm requires no additional messages, the distributed and anticipatory algorithms require messages for requesting mutex and for updating resource variables. Table 4.1 contains counts of messages required by each of the implementations. It should be noted that the centralized solution

Algorithm	local requests	messages to gain mutex	messages to release mutex	check messages	update messages	total messages	retries/ rollbacks
Centralized	110	0	0	0	0	110	0
Distributed	110	1675	402	0	880	3067	292
Anticipatory	110	2416	374	324	880	4104	422

Table 4.1 — Message Counts for CCR, A = 8, N = 4

incurs zero rollbacks/retries, which makes this code look better than it is. This solution contains "hidden" cpu costs. Each time through the loop in the centralized code, the invocations' synchronization expressions are re-evaluated until one is found to be true. So, even though there are no visible retries, they are occurring in a manner similar to the explicit retries in the distributed algorithm. However, the cpu cost of executing the synchronization expressions is cheaper then the passing of messages.

Table 4.1 shows the anticipatory implementation requiring 34% more messages than the distributed implementation. The roller coaster problem in its present form is not conducive to good results for the anticipatory algorithm. Every time the car executes a region statement, there are then four or more passenger processes attempting to execute a region statement. This heavy contention for exclusive access to the CCR leads to numerous rollbacks. The roller coaster problem was modified in the following manner to show how the anticipatory algorithm performs in a more conducive environment. N was set to one and delays were added to the passenger and vehicle processes. The results of runs for the anticipatory and the distributed implementations are shown in Table 4.2. As

Algorithm	local requests	messages to gain mutex	messages to release mutex	check messages	update messages	total messages	retries/ rollbacks
Distributed	100	1042	326	0	400	1868	226
Anticipatory	100	934	138	114	400	1686	78

Table 4.2 — Message Counts for CCR, A = 4, N = 1

shown, the anticipatory case had fewer rollbacks/retries and required 10% fewer messages in the anticipatory case. This single passenger roller coaster problem is not considered important on its own, but rather an example showing that the anticipatory algorithm is beneficial at least in some cases. Beyond just the reduced number of messages, there is also the added concurrency of executing the regions statement list at the same time as obtaining mutex. In the roller coaster problem, the actual concurrency is fairly minimal, since the statement lists are each just single arithmetic operations. A case where more was being done within a region statement such as scientific calculations would yield even more concurrency.

4.3. Summary

This chapter has shown centralized and distributed implementations for CCRs. The problem of starvation was encountered in the distributed implementation. This problem was handled by the addition of a failed list to ensure that older requests have higher priority. This chapter then looked at an anticipatory implementation of CCRs. Again the problem of starvation arose and was overcome in a similar manner to the distributed case. In some cases the anticipatory implementation was shown to be better than the distributed implementation.

CHAPTER 5

Monitors

This chapter discusses the centralized and distributed implementations of monitors. This discussion examines three standard signaling protocols: signal and continue (SC), signal and wait (SW), and signal and urgent wait (SU). It also examines prioritized waits. Then it shows solutions to the roller coaster problem. Finally, it examines the solution to a disk scheduling problem using prioritized waits.

The general form of a monitor is:

```
monitor m
        var v1: type
        ...
        var vN: type

        proc P1
        ...
        end P1
        ...
        proc PM
        ...
        end PM
  end m
```

where v1 ... vN are monitor variables. Along with the usual **integer**, **real**, **boolean**, etc. types, variables can be declared as type **condition.** These condition variables have **signal** and **wait** operations defined on them. Monitor variables can only be accessed by the monitor routines P1 ... PM. These procedures are callable from processes external to the monitor. Only a single process can be executing in the monitor at a time.

5.1. Implementations

When a monitor procedure is called, it must gain mutex at the beginning and release mutex at the end of the routine. The beginning and end of routines are not the only place where mutex must be obtained and released. **wait** and **signal** operations on condition variables can force a procedure to release mutex and wait to regain it. A procedure performing a **wait** will always be blocked by being placed on the queue of the particular condition variable; the effect of a **signal** is determined by the signaling protocol being used.

5.1.1. Centralized Solutions

In a centralized environment, a single SR operation suffices for gaining and relinquishing mutex as described in Chapter 2. A **receive** on the operation is performed when entering a routine and a **send** to the operation is executed at the conclusion of a routine.

The effects of **signal** and **wait** operations are handled in two parts. The obtaining and releasing (or passing) of mutex is dealt with by the code making the operation call and is discussed in the sections on the signaling protocols. The queuing and dequeuing of processes on a condition variable's queue is serviced by a separate process, one per variable. The code for this handler is shown in Figure 5.1. A **signal** is performed by using a call statement. The synchronization expression in the arm of the **in** for the signal operation ensures that the **signal** will be "dropped" if and only if no processes are waiting on the condition variable queue. If the queue is empty, then this arm is taken and the value of false is returned to the signaler to show that no processes were waiting. Because the process performing the call to the signal operation has mutex, this dropping prevents the call from signaling a process that waits on this variable later. A process performs a

```
proc cond_var()
  do true →
    in wait( return_to_caller) and ?signal ≠ 0 →
      # execute wait only if there is a signal
      in signal() returns proc_waiting →
        proc_waiting := true
      ni
      send return_to_caller()
    [] signal() returns proc_waiting and ?wait = 0 →
      # drop a sent signal if there are no waits
          proc_waiting := false
    [] empty() returns results →
      results :=  ( ?wait = 0)
    ni
  od
end
```

Figure 5.1 — Centralized Condition Variable Handler

wait by executing a **send.** Again a synchronization expression is used to ensure that the process is removed from the queue when a matching signal has been performed. When a signal is made, the oldest wait invocation is serviced by the wait arm. The signal invocation is serviced within this arm so that only one wait is serviced for a given signal. The process returns a value of true to the signaler. Then a message is sent to awaken the process whose wait operation was just signaled.

5.1.1.1. Signal and Continue

In SC, a process continues to execute after performing a **signal.** Therefore, it does not release mutex. So, a **signal** is coded as a call to the signal operation. Because the signaling process does not care whether or not a process was on the condition queue, the code that returns the boolean in the condition variable handler can be removed. A process performing a **wait** on condition variable *x* has code:

```
send x.wait( return_signal)
send mutex()
receive return_signal()
receive mutex()
```

After the send to wait, the process must release mutex. Then it waits to receive the message that it is no longer blocked on the queue. Finally, it must again obtain mutex.

5.1.1.2. Signal and Wait

In SW, a process that performs a **signal** must know if a process is on the queue. If no processes are on the queue, then the signaler continues as in SC. But if a process is on the queue, exclusive access to the monitor must be passed to the awakened process. It does so by performing a **receive** on mutex. A **signal** on condition variable *x* has code:

```
if x.signal() →
  receive mutex()
fi
```

The code for the process performing the **wait** is as shown for SC, except that it does not perform the **receive** on mutex. Mutex is passed by the **signal**er performing a **receive** on mutex. In this way, the signaler becomes blocked and the signaled process begins executing in the monitor again.

5.1.1.3. Signal and Urgent Wait

In SU, as in SW, a process that **signal**s a non-empty condition variable must pass access to the monitor to the signaled process. Unlike SW, the signaling process is given priority over processes starting to enter the monitor (i.e., a monitor routine that has just been called). The signaler is said to be put on the urgent queue. The signaler does this by making a **call** to UrgentQ after having sent a **signal** with a process waiting. The code is the same as for a **signal** in SW except the call replaces the receive on mutex. A procedure releasing mutex, when not signaling, has code:

```
if ?UrgentQ > 0 →
  receive UrgentQ()
[] else →
  send mutex()
fi
```

It first checks to see if any processes are on the UrgentQ. If a process is waiting, then it executes a **receive** allowing the first process on the UrgentQ to have exclusive access. If no processes are on the UrgentQ, then it **send**s to mutex, allowing a new process to enter the monitor.

5.1.1.4. Prioritized Wait

The addition of prioritized wait allows for the selecting of the waiting process to be awakened on a basis other than FIFO. A process when performing a wait specifies an additional integer parameter, its rank. When multiple processes are on a queue of a condition variable that is signaled, the process whose rank is minimum is awakened. In the case of a tie (i.e., multiple processes with the same minimum rank), the tied processes are awakened on a FIFO basis. The first steps in adding prioritized wait are to include a rank parameter with the wait operation and append a scheduling expression to the wait arm in the condition variable handler.

If the **minrank** function did not exist, the above steps would be sufficient to handle prioritization. The **minrank** function returns the value of the smallest rank currently waiting on the conditional variable being checked. Figure 5.2 shows an *incorrect* solution to prioritized wait. It violates the fairness constraint. In order to determine the

```
in wait( return_to_caller, priority) and ?signal ≠ 0 by priority →
  receive signal()
  send return_to_caller()
[] ...
[] minrank( success) returns rank →
  if ?wait ≠ 0 →
    in wait( return_to_caller, priority) by priority →
      send wait( return_to_caller, priority)
      rank := priority
      success := true
    ni
  [] else →
    success := false
  fi
ni
```

Figure 5.2 — Incorrect Centralized Handler for Prioritized Wait

minimum rank on the condition variable's queue, the first step of the minrank code performs an **in** on the wait operation. This **in** uses a scheduling expression to get the waiting process with minimum rank. After determining the rank, the minrank code resubmits the invocation of the wait operation of the process with minimum rank. Suppose that processes P1 and P2 have performed **wait**s with the same rank, P1's request is older, and their rank is currently minimal. If a process invokes the **minrank** function, P1's invocation will be removed and replaced on the wait operation. Now, if a process executes a **signal** on the condition variable that P1 and P2 are **wait**ing on, P2 will be awakened, because its request is now the older invocation of **wait.** Potentially process P1 will starve, which violates the required fairness.

To ensure fairness, an additional operation, *wait_ranks*, is used. When a process performs a **wait**, in addition to sending to *wait* parameterized by rank, it now also **send**s to *wait_ranks* with the single parameter rank. The order that these two **send**s are made is unimportant since the process executing the **wait** has exclusive access to the monitor.

The **minrank** code services the *wait_ranks* operation instead of the *wait* to determine what the minimum rank is. Thus, the temporal order of invocations to *wait* is preserved. The modified and new parts of the condition variable handler are shown in Figure 5.3. This version assumes that SC protocol is being used. So, signal does not return a boolean indicating whether any processes were on the wait queue and a **receive** on signal is used instead of an **in** in the *wait* arm. The *wait* code must also include an **in** to remove an invocation of *wait_ranks* with minimum rank. Because it is considered an error to call **minrank** when the condition variable's queue is empty, the minrank code first checks to see if any invocations are pending. If no processes are waiting, then a value of false is returned in success to denote this error. Otherwise, the wait_ranks operation is serviced and the invocation is immediately resent.

```
in wait( return_to_caller, priority) and ?signal ≠ 0 by priority →
  in wait_ranks( priority) by priority → ni
  receive signal()
  send return_to_caller()
[] ...
[] minrank( success) returns rank →
  if ?wait_ranks ≠ 0 →
    in wait_ranks( priority) by priority →
      send wait_ranks( priority)
      rank := priority
      success := true
    ni
  [] else →
    success := false
  fi
ni
```

Figure 5.3 — (Correct) Centralized Handler for Prioritized Wait

5.1.2. Distributed Solutions

In the distributed environment, versions of the Suzuki and Kasami algorithm are used to implement mutual exclusion. One monitor server and one mutex handler process are allocated per node. As for the distributed CCR implementations in Chapter 4, it is necessary to broadcast updated values of monitor variables. This updating must be done more frequently in the monitor case due to the releasing of mutex by **wait**s and **signal**s, not just at the end of the routine.

The obtaining and releasing of mutex due to **signal** and **wait** calls are handled by the code making the call and the mutex servers. The operations on the condition variables' queues are handled by server processes, which are allocated one per variable per node. The wait requests are handled by a distributed queue, similar to the semaphore **P** requests in Chapter 3. This distributed condition variable handler is shown in Figure 5.4. When a process performs a **wait**, it calls the wait_request operation. This operation broadcasts the request to the operation wait_update to all the nodes, including itself. Note that timestamps and acks are not required for the waits as was the case with semaphores. Semaphore requests are made asynchronously, which allows different nodes to receive requests in different orders. The timestamps and acks are needed to ensure that the different nodes remove entries from the distributed queue in the same order. Monitor routines have mutex when performing a **wait** and use **send** statements, which are semi-synchronous. Therefore, the wait messages are queued in the order they are sent, without timestamps or acks. When a process performs a **signal**, it calls the signal_request operation. This operation checks its local copy of the queue to see if any processes are waiting. If no processes are waiting, then the signal is dropped. Otherwise, all the other handlers are notified to remove the oldest wait from the wait queue by a **send** to the

```
do true →
  in wait_request( return_from_wait) →
  # this handlers process is requesting a wait on the condition
  # variable, notify all cond_var_handlers to put request on wait queue
    fa i := 1 to N →
      send cond_var_handlers[ i].wait_update( return_from_wait, me)
    af
  [] signal_request( waiting_proc_num) returns proc_waiting →
    # this handlers process is signaling the condition variable
    # check if there are any waiting processes
    if ?wait_update ≠ 0 →
      # there is a wait request pending, notify other handlers of signal
      fa i := 1 to N st i ≠ me →
        send cond_var_handlers[ i].signal_update()
      af
      # get operation from local wait_queue and send to it
      in wait_update( return_oper, from) →
        send return_oper()
        waiting_proc_num := from
      ni
      proc_waiting := true
    fi
  [] empty() returns result →
    result := ( ?wait_update = 0)
  [] signal_update() →
    # a process has signaled the condition variable
    # remove the first element off of wait queue
    if ?wait_update > 0 →
      in wait_update( temp, from) → ni
    fi
  ni
od
```

Figure 5.4 — Distributed Condition Variable Handler

signal_update operation. After broadcasting the signal_update the handler removes the operation from its local copy of the queue and sends to it to awaken the signaled process. As in the centralized case, the distributed version returns to the signaler whether or not a process was waiting. The distributed case also returns the process number, which is required for the passing of mutex to the signaled process.

5.1.2.1. Signal and Continue

For SC, the structure of **signal** and **wait** calls are the same in the distributed implementation as in the centralized. The **signal** call is to *signal_request* instead of the *signal* operation. As in the centralized case, the code to return a boolean indicating whether any processes were waiting can be omitted. Since the **signal**er in SC does not pass mutex after signaling, the code to return the process number of the process that was signaled can be omitted, too. The **wait** call is modified from a **send** to the *wait* operation to a **call** to *wait_request*. The obtaining of and releasing of mutex are changed from **send** and **receive** on the local mutex operation to **call**s to the *get_exclusion* and *release* operations of the mutex server. So, a **wait** on *x* has code:

call *x*.wait_request(return_signal)
call mutex_server.release()
receive return_signal()
call mutex_server.get_exclusion()

A straightforward implementation of the Suzuki and Kasami mutex algorithm is used for SC.

5.1.2.2. Signal and Wait

The distributed SW **wait** code is the same as the SC code above, except that it does not request mutex after receiving the return_signal. In the centralized SW code, when a process **signal**s and a process is waiting, the signaler blocks itself by receiving on the local mutex operation. The following code for a **signal** on *x* effects the same result in the distributed environment:

```
if x.signal_request( signaled_proc) →
    call mutex_server.pass_mutex( signaled_proc)
    call mutex_server.get_exclusion()
  fi
```

To release mutex in this case, the signaler must notify its mutex handler to pass the mutex privilege to the signaled process's mutex handler. Thus, the signal_request code must return the process number of the awakened process, necessitating the use of a modified version of the mutex algorithm. In the distributed mutex algorithm, as used for SC, the mutex handler broadcasts a request for the privilege; receives it and notifies its customer; then waits for the customer to release mutex. The operation pass_mutex is added as an alternative to a simple release. If pass_mutex is sent to rather than release, the signaled process is passed the privilege instead of the oldest entry on the unsatisfied mutex request queue. Because the awakened process does not request mutex, the mutex code must additionally be able to receive a privilege message without its customer having requested it.

5.1.2.3. Signal and Urgent Wait

The code for **signal** and **wait** operations in SU is identical to that of SW. The placing of **signal**ing processes on the UrgentQ and dealing with their priorities is handled by further modifications to the mutex code. The handler process that has the privilege maintains a queue of unsatisfied requests. The modifications for SU has an UrgentQ that is kept by the process with the privilege, too. This queue is added to the modifications made to the SW mutex code. When a mutex handler receives a pass_mutex, which means its customer is performing a **signal**, the process appends itself to the end of UrgentQ. So, when mutex is being released, the privilege is passed to the signaled process if there was a **signal.** If there was no **signal** and the UrgentQ is not empty, then the process at the head of the UrgentQ is given the privilege. If the UrgentQ is empty and there is an unsatisfied request, then the privilege is passed to the oldest entry on that queue.

5.1.2.4. Prioritized Wait

The distributed solution for prioritized wait uses a distributed queue wait_ranks in a similar manner as the centralized implementation uses the local operation wait_ranks. Figure 5.5 shows the distributed handler for the prioritized wait. Rather than the process performing the **wait send**ing to both queues, it only **send**s to wait_request. The condition variable handler sends to the wait_ranks queue while broadcasting the request to wait_update. When a signal_request or signal_update comes in, the top entry is removed from both of the distributed queues. The **minrank** function uses the local copy of the distributed wait_ranks queue to determine the minimum rank, which maintains the temporal order of the wait_update queue.

5.2. Monitor Solutions to the Roller Coaster Problem

The monitor solutions to the roller coaster problem are similar to the CCR solution. The passenger and vehicle processes for all the monitor solutions are shown in Figure 5.6. The passenger process uses two calls to the monitor to board and exit the car, which resembles the CCR algorithm where two region statements were used for this purpose. The vehicle process makes three calls to the monitor to arrive at the station, leave when the car is full and do the ride, and wait for all the passengers to debark; three region statements were used for this purpose in the CCR solution. The monitor *roller_coaster* uses the same five variables—*vin*, *vout*, *vdest*, *pin*, and *pout*—used in the CCR solution.

```
do true →
  in wait_request( return_from_wait, priority) →
    fa i := 1 to N →
      send cond_var_handlers[ i].wait_update( return_from_wait,
                                       priority)
      send cond_var_handlers[ i].wait_ranks( priority)
    af
  [] signal_request() →
    if ?wait_update ≠ 0 →
      fa i := 1 to N st i ≠ me →
        send cond_var_handlers[ i].signal_update()
      af
      in wait_update( return_oper, priority) by priority →
        send return_oper()
      ni
      in wait_ranks( priority) by priority → ni
    fi
  [] empty() returns result →
    result := ( ?wait_update = 0)
  [] signal_update() →
    if ?wait_update > 0 →
      in wait_update( return_oper, priority) by priority → ni
      in wait_ranks( priority) by priority → ni
    fi
  [] minrank( success) returns rank →
    if ?wait_ranks ≠ 0 →
      in wait_ranks( priority) by priority →
        send wait_ranks( priority)
        rank := priority
        success := true
      ni
    [] else →
      success := false
    fi
  ni
od
```

Figure 5.5 — Distributed Handler for Prioritized Condition Variable

```
process Passenger
 fa i := 1 to R →
    call roller_coaster.board_car()
    call roller_coaster.exit_car()
 af
end Passenger
process Vehicle
 fa i := 1 to T →
    call roller_coaster.car_arriving()
    call roller_coaster.car_leaving()
    call roller_coaster.car_empty()
 af
end Vehicle
```

Figure 5.6 — Passenger and Vehicle Processes for Roller Coaster Problem

The major difference between the CCR and monitor solution is the use of condition variables. Instead of continually retrying to execute a region statement when the guard is false, the monitor code performs a **wait** on a condition variable when conditions do not allow it to continue. Instead of just incrementing a resource variable and causing a flood of retries of region statements, a monitor routine can signal a condition variable, thereby waking just a single process.

The three solutions use four condition variables. A passenger that gets in line and finds no empty seats on a car available performs a **wait** upon *car_arrived*. A car that arrives and sees passengers in line will **signal** *car_arrived*. A car that has yet to fill with passenger performs a **wait** on *passengers_in_car*. A passenger that finds itself filling the last empty seat in a car does a **signal** to *passengers_in_car*. Passengers enjoying the ride perform a **wait** on *wait_to_arrive* before debarking. When the ride is finished, the car performs a **signal** on *wait_to_arrive*. The car waits for all passengers to debark by performing a **wait** on *passengers_off*. The last passenger debarking from a car will **signal** *passengers_off*.

Figure 5.7 shows the monitor for this problem using the SC protocol. When the car arrives at the station in *car_arriving* it increments *vin* to mark this occurrence. It then checks to see if any passengers are in line. For each passenger in line, up to a maximum

```
monitor roller_coaster
  # N is a constant, the number of passengers per car
  var pin, pout, vin, vout, vdest :integer := 0, 0, 0, 0, 0
  var car_arrived       : condition; # car arrives at stop
  var passengers_in_car : condition; # car is full
  var wait_to_arrive    : condition; # car has reached destination
  var passengers_off    : condition; # passengers have disembarked
  proc board_car()
    if pin ≥ vin * N →
      wait( car_arrived)
     [] else  →
      pin++
    fi
    if ( vout + 1) * N ≤ pin and not empty( passengers_in_car) →
      signal( passengers_in_car)
      vout++
    fi
  end
  proc exit_car()
    if pout ≥ vout * N →
      wait( wait_to_arrive)
     [] else →
      pout++
    fi
    if ( vdest + 1) * N ≤ pout and not empty( passengers_off) →
      signal( passengers_off)
      vdest++
    fi
  end
  proc car_arriving()
    # car arrived at stop, notify passengers
    vin++
    count := N
    do count > 0 and not empty( car_arrived) →
      signal( car_arrived)
      pin++
      count––
    od
  end
```

Figure 5.7a — SC Monitor Solution to Roller Coaster Problem

```
proc car_leaving()
  # leave stop when car is full, and arrived at destination
  if ( vout + 1) * N > pin →
    wait( passengers_in_car)
   [] else  →
    vout++
  fi
  count := N
  do count > 0 and not empty( arrived_at_dest)
    signal( wait_to_arrive)
    pout++
    count––
  od
end
proc car_empty()
  # wait for all passengers to disembark
  if ( vdest + 1) * N > pout →
    wait( passengers_off)
   [] else →
    vdest++
  fi
end
end roller_coaster
```

Figure 5.7b — SC Monitor Solution to Roller Coaster Problem

of N, the car not only performs a **signal** on *car_arrived* but also increments *pin.* The incrementing is done to avoid the stealing of a **signal.** If the car just signaled, it would be possible for a passenger to **call** *board_car* and to take a seat, even though the car might have made N **signal**s. This situation can occur because the **signal**ed process has no priority over a process just trying to enter the monitor. So, the car marks a seat taken by incrementing *pin* for each **signal** it performs. A passenger process awakened by this **signal** does not increment *pin*, since that has already been done for it. A passenger getting in line in *board_car* checks if a seat is available. If so, *pin* is incremented. If no seat is open, the passenger **wait**s on *car_arriving.* After taking a seat the passenger checks if he has filled the last seat on the car and the car has performed a **wait** on *passengers_in_car.*

If he has, a **signal** to *passengers_in_car* is performed. Note that *vout* is incremented, as *pin* was. This incrementing was not necessary since there is only one car, but was included for consistency. The car **wait**s on *passengers_in_car* in *car_leaving* if it is not yet full. Once full, the car does the ride, then signals the completion of the ride to any passengers already **wait**ing on *wait_to_arrive.* A passenger checks if the ride is over in *exit_car.* If the ride is not over, it performs a **wait** on *wait_to_arrive.* If it is over, it just increments *pout* and debarks. After exiting the car, the passenger checks to see if he is the last person out of the car and if the car has performed a **wait** on *passengers_off.* If the car is waiting, the condition variable is *signal*ed. The car checks if all the passengers are off in *car_empty*. If they are not all off, it performs a *wait* on *passengers_off*, else it just increments *vdest.*

Figure 5.8 shows the monitor for the roller coaster problem using the SU protocol. The SU code is the same as the SC code except for the removal of the code to avoid the stealing of a **signal.** The extra code is unnecessary due to the nature of the SU protocol. When a process is **signal**ed, it is immediately given exclusive access to the monitor. So, when the car process performs the **signal** on *car_arrived* in *car_arriving*, the awakened passenger can increment *pin.* Also, because the car process is placed on the UrgentQ, it will be given exclusive access to the monitor before any passenger process just calling *board_car* is given access. Therefore, no passenger process can board the car without performing a wait if a processes is already waiting. Note that rather than using the **empty** function in the **do** loops in *car_arriving* and *car_leaving*, calls to **signal** can be made. The loops will terminate if the condition variable's queue is empty.

The first attempt at using the SW protocol for this problem used the same monitor as used in SU. This solution is unfair because it allows passengers on the *car_arrived*

```
monitor roller_coaster
  var pin, pout, vin, vout, vdest :integer := 0, 0, 0, 0, 0
  var car_arrived         : condition; # car arrives at stop
  var passengers_in_car   : condition; # car is full
  var wait_to_arrive      : condition; # car has reached destination
  var passengers_off      : condition; # passengers have disembarked
  proc board_car()
    if pin ≥ vin * N →
          wait( car_arrived)
    fi
    pin++
    if ( vout + 1) * N ≤ pin →
          signal( passengers_on_car)
    fi
  end
  proc exit_car()
    if pout ≥ ( vout * N) →
          wait( wait_to_arrive)
    fi
    pout++
    if ( vdest + 1) * N ≤ pout →
          signal( passengers_off)
    fi
  end
  proc car_arriving( )
    vin++
    count := N
    do count > 0 and signal( car_arrived) →
          count––
    od
  end
  proc car_leaving( )
    if ( vout + 1) * N > pin →
          wait( passengers_on_car)
    fi
    vout++
    count := N
    do count > 0 and signal( wait_to_arrive) →
          count––
    od
  end
  proc car_empty( )
    if (vdest + 1) * N > pout →
          wait( passengers_off)
    fi
    vdest++
  end
end
```

Figure 5.8 — SU Monitor Solution to Roller Coaster Problem

queue to be passed by passengers just entering the line. The problem is that after the car process performs the first **signal**, it must contend for exclusive access with all the other processes. So, passenger processes can come into the monitor and board the car while other processes are waiting on *car_arrived.* This situation can be avoided by moving the incrementing of *vin* to after the **do** instead of before it. In this way, a passenger that just entered the monitor will find the conditional false in *board_car* unless no processes are waiting on the queue. In the same manner, the incrementing of *vout* is moved to after the **do** in *car_leaving.*

Runs were again made with the number of passengers per car, N, being four and the number of rides per passenger, R, being five. The centralized and distributed implementations were run for all three protocols with the number of passengers wishing to ride, A, set to eight and then six. The number of trips the car needs to make, T, is ten for eight passenger and eight for six passengers. The code and output of these monitor implementations appear in Appendix III.

The three centralized implementations create a single instance of their particular *roller_server_cent* server. The server starts four copies of the cond_var procedure to act as servers for each of the four condition variables used by the solution. One passenger process is instantiated for each of the N passengers and one vehicle process is instantiated for the car.

The distributed solutions require many more processes. There are again one instantiation for each passenger and vehicle. Each of these processes requires an instance of the particular *Monitor_server* resource and each of these servers has an instance of the *Cond_var* resource for each condition variable. Each *Monitor_server* requires an instance of the particular *Mutex_server*, too.

The message requirements of the six solutions for eight passengers are shown in Table 5.1. All but the centralized SC solution give the exact same ordering of passenger boarding, i.e., passengers 1 through 4 riding then 5 through 8 riding. This pattern is repeated five times. The centralized SC solution actually gives the same passenger ordering. However, this solution has the strange occurrence of three passengers boarding the car, the car leaving, and then the fourth passenger boarding. The problem arises when N or more passengers are waiting on *car_arrived.* The car will **signal** the first N passengers and increment *pin* N times. This puts the N passengers in competition for access to the monitor. The vehicle process then gives up access when leaving *car_arriving* but immediately tries to reenter the monitor by calling *car_leaving.* In a solution that was "absolutely" fair, i.e., FIFO, the car would not get access until the N passengers did. However, the semantics of monitors does not require this strict fairness. So, the car in this case gets in before the fourth passenger does. Because *vout* has already been incremented four times, the car appears full and leaves. A question that arises is "When does a passenger actually enter the car?" Is it when *pin* is incremented, or when the passenger has exclusive access to the monitor after being **signal**ed? For this particular problem, it does not seem too important, but in other cases an SC solution might need to be modified

Algorithm	local requests	msgs to gain mutex	msgs to release mutex	wait msgs	signal msgs	update msgs	total msgs
Centralized SC	110	160	0	50	100	0	420
Centralized SW	110	190	0	80	180	0	560
Centralized SU	110	180	0	70	170	0	530
Distributed SC	110	1482	150	400	400	880	3322
Distributed SW	110	1872	270	800	820	880	4752
Distributed SU	110	1872	270	800	820	880	4752

Table 5.1 — Message Counts for Roller Coaster Solutions

to clarify this situation. The distributed SC solution does not have this problem because of the different timing that always allowed the passengers access before the car executed in *car_leaving.*

Further tests were run to determine which processes were performing the **wait**s in the centralized solutions. In the SC case, 40 of the **wait**s were performed by passengers getting blocked in *board_car*. The other 10 were performed by the car being blocked in *car_empty*. In the distributed SC code, the passengers also were blocked but the car never had to **wait.** In the centralized SW case, the passengers had to **wait** in both *board_car* and *exit_car*, accounting for all 80 waits. The distributed SW solution was identical. In the centralized SU case, all the passengers were blocked in *board_car* (40 times) and all but the last passenger in each ride were blocked in *exit_car* (30 times). In the distributed SU case, the last passenger also had to **wait** in *exit_car*, making the total number of **wait**s 80. The centralized case appears to exhibit behavior similar to the SC case. Three of the passengers are getting to the **wait** in *exit_car* but the fourth is not. In this case, all that is affected is the scheduling. It does not affect when the passenger boards or debarks.

The six passenger case was again run to check for any hidden starvation. All the different runs were fair since the two passengers that did not ride on the previous trip always were allowed on the next one. The centralized SC solution again exhibited the same unexpected boarding order "problem". The distributed SC solution exhibited a bias toward passengers 1 and 2, which can be attributed to the bias in the mutex algorithm discussed in Chapter 4. The other four solutions had identical orderings, which have the same ordering as the centralized SC.

5.3. Monitor Solutions Using Prioritized Wait

To demonstrate prioritized wait, consider the classic circular scan (CSCAN) disk scheduling algorithm discussed in [Silb88]. In this algorithm, the disk head moves in one direction, from 0 to Maxcylinder. The disk scheduler services the next request whose cylinder is the smallest and greater than or equal to the current cylinder. If no cylinder request is greater than or equal to the current cylinder, then the disk scheduler sets current cylinder to -1 and starts over from there.

A process that wants to access a cylinder makes a request, specifying the desired cylinder. After the process's request has been granted and the process performs the desired operation on the cylinder, it must then release access.

The code for the monitor is shown in Figure 5.9, which is from [Andr89]. The algorithm makes use of four variables. The integer *position* holds the value of the cylinder currently being accessed. The condition array *scan* breaks requests into two sets: the current and the next set. The current set contains the requests that will be serviced in the current pass across the disk. The next set contains delayed requests that will not be serviced until the next pass. The integers c and n are used to delineate which of the elements of the array is the current set and which is the next.

When a request is received, the first check determines if the disk is currently in use. If not, the head is positioned on the requested cylinder and access is immediately granted. If the disk is in use, then the requested cylinder's number is compared to the current cylinder. If the requested cylinder's number is greater than the current position, then the request is delayed on the current set (scan[c]); otherwise, it is delayed on the next set (scan[n]). Note that this is a slight modification from the stated CSCAN algorithm.

```
monitor disk_scheduler
  var scan[ 0: 1] : condition
  var position := -1, c := 0, n:= 1
  proc request( cyl)
    if position = -1 →
      position := cyl
    [] position ≠ -1 and cyl > position →
      wait( scan[ c], cyl)
    [] position ≠ -1 and cyl ≤ position →
      wait( scan[ n], cyl)
    fi
  end
  proc release()
    if not empty( scan[ c]) →
      position := minrank( scan[ c])
    [] empty( scan[ c]) and not empty( scan[ n]) →
      c :=: n  # swap operation
      position := minrank( scan[ c])
    [] empty( scan[ c]) and empty( scan[ n]) →
      position := -1
    fi
    signal( scan[ c])
  end
```

Figure 5.9 — CSCAN Monitor using Prioritized Wait

Requests that are the same cylinder as the current position are put in to the next set instead of the current to eliminate the possibility of starvation. Otherwise a steady stream of requests for the current position could cause the disk head never to move. Thus, any requests for cylinders other than the current one would never be serviced.

The release code first determines if the current set is empty. If not, the position of the next cylinder is obtained using **minrank.** If the current set is empty and the next set is not, then the next set becomes the current set and the new next set becomes the empty set by simply swapping the values of c and n. Then the position is again obtained using the **minrank** function on the current set (scan[c]). If both sets are empty, then the position is set to -1 to show that no cylinder is currently being accessed. Finally, the request whose position was determined above is **signal**ed. Note that the **signal** has no effect if no requests are pending.

The centralized solution creates an instance of *useds*. This process creates an instance of the *disk_scheduler*, which in turn starts up the processes to handle the scan array. The *useds* resource also starts up six simple processes to exercise the disk_scheduler monitor. These processes just make three pairs of request and release **call**s.

The distributed solution differs in that each of the six simple processes have an instance of the *CSCAN_monitor* resource. Each of these instances have an instance of the *Mutex_server* resource (the straightforward version) and two instances of the *Pri_Cond_var* resource for the scan array.

The code and runs for these solutions are in Appendix III. The message requirements of these solutions is shown in Table 5.2. The centralized and distributed solutions had identical runs. The distributed code required many more messages to solve the problem but the number of **wait**, **signal**, and **minrank** operations were identical. Each **wait** required only a single message in the centralized implementation. The distributed code used seven messages: one local request plus six messages to broadcast to the distributed queues. Of the eighteen **signal**s performed, seventeen had matching **wait**s and one was dropped. The successful **signal**s required two messages each and the dropped **signal**

Algorithm	local requests	msgs to gain mutex	msgs to release mutex	wait msgs	signal msgs	wait rank msgs	update msgs	total msgs
Centralized	36	0	53	17	35	34	0	175
Distributed	36	335	53	119	120	119	180	962

Table 5.2 — Message Counts for Disk Scheduler Solutions

required only one message in the centralized case. The distributed code required seven messages for each successful **signal** and only one for the dropped **signal.** There were seventeen calls to **minrank**, each requiring two messages in the centralized case and seven messages in the distributed case.

5.4. Summary

This chapter has shown centralized and distributed implementations for monitors. The major difference in how the three **signal**ing protocols were implemented was through modification of when mutex was requested and modifications to the mutex algorithm. The addition of the **minrank** function in prioritized condition variable code required the addition of a *wait_ranks* queue in parallel to the *wait* queue. This allowed the minimum rank to be found without affecting the order that **wait**s were serviced.

CHAPTER 6

Communicating Sequential Processes

This chapter examines centralized and distributed implementations of the language CSP. The major emphasis is on implementing the input and output commands of the language. This chapter also explores ways to handle CSP code with nested processes. It concludes by showing how the techniques developed can be applied to a CSP solution to the roller coaster problem and to a CSP program containing nested processes.

Processes in CSP communicate synchronously via input/output (I/O) commands. An I/O command has the form name$expression, where "$" is replaced by "!" for an output command and by "?" for an input command [Schn82]. The name is that of the process with which the process executing the I/O command wishes to communicate. This form of communication is synchronous in that both processes must be ready to execute corresponding statements before communication can occur. Beyond naming each other and both being ready to execute their respective I/O commands, the expression in the output command must match the type of the target variable in the input command. Each guard of repetitive and alternative statements is allowed to contain a single I/O command, which evaluates to a boolean value that depends on the preparedness of the matching I/O command. The I/O command in a guard only evaluates to false if the named process has completed. It evaluates to true if the other process is ready to perform the matching I/O command. If neither of the two above cases is true, then the I/O portion of the guard has indeterminate value until one of the two cases does occur. The situation where a process terminates causing an I/O command to fail is called automatic or distributed termination. Some implementations of CSP-like languages support distributed termination; some do not. An implementation that supports distributed termination must detect and handle the effects of such terminations.

6.1. Implementations

The solutions being examined make use of a server(s) to handle a process's requests to find a process with a corresponding I/O command. In order for servers to match up corresponding I/O commands, the processes must be able to specify uniquely which commands it is trying to execute. Further, in the case of repetitive and alternative statements, which can contain multiple arms with guards containing I/O commands, a server must be able to receive a list of I/O commands for possible execution. The server in this case attempts to find a command in another process matching one of those in the list of commands it received.

The following is a sample CSP program, which is not intended to be meaningful except to illustrate some of the requirements placed on an implementation. (The variables i, j, and k are integers, while y is a real.)

```
[ A || B || C ]  where
A = [  C?i ]          B = [ C!7.5;          C = *[ A!2 → ...
                            C?j  ]                [] B?y → ...
                                                  [] B!k → ...  ]
```

In order to designate to a server which process an I/O command references, each process is given a unique id number. In the example, processes A, B, and C can be assigned id numbers 1, 2, and 3 respectively. As can be seen in the example, processes B and C have two I/O commands of distinct types that reference each other, i.e., a real being sent from B to C and an integer being sent from C to B. Therefore, each of the types of possible corresponding statements is given a unique request number. For communication between B and C, these processes can use request number 1 for the real being sent from B to C and request number 2 for the integer being sent from C to B. These request numbers only have to be unique for communications between these two processes; process C could also use request number 1 when attempting to communicate with process A to mean sending an integer to A. Thus, a process wishing to execute a simple I/O statement must specify to its handler the number of the target process and the request number of the particular type of I/O command to be executed. Also, a process executing a statement including guards with I/O commands must be able to specify to its server a set of I/O commands.

In order for a process to specify this multitude of possibilities, the following implementations make use of boolean *ready* arrays. Each array is of size N x r, where N is the total number of processes and r is the maximum number of types of requests between any two processes. Elements of *ready* are set to true to indicate a process's willingness to execute particular I/O commands. A process attempting to execute an I/O statement sets the single element of *ready* associated with this statement. When executing an alternative or repetitive statement whose guards contain I/O commands, the executing process evaluates the non-I/O portion of the guard and places the result in the I/O command's entry in *ready*. In either case, after setting the appropriate elements, the process passes the array to its server.

In incorrectly written CSP code, deadlock can occur and all the server process(es) might be able to do is to detect it. However, an incorrectly written server can cause deadlock in a situation where matching I/O commands are ready to execute. Consider, for example, the CSP program

```
[ A || B ]  where
A = [  B?5; ... ]            B = *[ A?i → ...
                                   [] A!j → ... ]
```

where the variables i and j are integers. Process B’s server will receive a *ready* array with elements for a send and a receive of an integer with process A set to true. B’s server can create deadlock by attempting to match up the command to receive from A and ignore all other requests until that match occurs. This deadlock occurs with a pair of matching commands ready to execute. Therefore, a server must not lock itself into a single command out of a list of commands.

While a server must avoid causing deadlock, Hoare [Hoar78] states that a server should attempt to be fair, but it is not a requirement. If a repetitive statement contains a guard without an I/O command that is always true, an implementation might never allow guards with I/O commands a chance to execute. A program must not depend on an implementation’s fairness to execute "correctly".

6.1.1. Centralized Solution

In the centralized case, a single server is used by all the processes. This server maintains a list of processes whose I/O commands do not currently have a matching request. When a new request is made to the server, the server checks to see if any of the waiting processes has an I/O command pending that matches the new request. If a match is found, the two processes with the corresponding commands are notified of the command that they can execute. If no match is found, the new request is just added to the end of the list of waiting processes.

The handler services two operations *find_match* and *proc_done* in the following manner.

> **do true** →
>   **in** find_match(in_ready, who, found) → ...
>   [] proc_done(who) → ...
>   **ni**
> **od**

A process wishing to perform an I/O command **send**s to the operation *find_match.* When a process is about to terminate, the last thing it must do is to notify the server, via a **send** to *proc_done*, that the process is completing.

Figure 6.1 shows the body of the *find_match* operation. The process making the request to *find_match* specifies its *ready* array, an integer *who* indicating what process it is, and an operation *found*, to which the results of the attempt to find a match are sent. The operation *ready_op* acts as the list of waiting process requests. When a new request is received, the invocations of *ready_op* are serviced, one at a time. The corresponding elements of the ready array of the new request are compared to the ready array of the process removed from *ready_op.* If two corresponding elements are both true, then a match has been found and the two processes are notified. If no match is found with the current invocation of *ready_op*, the invocation is returned to the end of *ready_op* and the next invocation is removed and checked for a match. This checking of the invocations on *ready_op* against the new request continues until a match is found or until all the invocations have been tried. If no match is found, then the server determines if any of the processes that are targets of the new request's communication commands has not terminated. If one or more of these processes is still running, then the new request is added to the list by sending it to *ready_op.* Otherwise, the requester is informed that all the

```
var request : int; var match_found : bool := false
count := ?ready_op
do count > 0 and not match_found →
  # check next item on the list to see if any requests match
  in ready_op( next_ready) →
    fa j := 1 to num_requests →
      if next_ready.ready[ who, j] and
                   in_ready[ next_ready.procnum, j] →
        match_found := true
        request := j
        exit
      fi
    af
    if not match_found →
          # no match, return the process’s requests to the list
      send ready_op( next_ready)
     [] else →
          # have a match, notify the two processes
      send next_ready.found( true, who, request)
      send found( true, next_ready.procnum, request)
    fi
    count––
  ni
od
# move unchecked requests to the end of the list
if count ≠ ?ready_op →
  do count > 0 →
    in ready_op( ready_rec) →
      send ready_op( ready_rec)
    ni
    count––
  od
fi
if not match_found →
  count := 0
  fa i := 1 to N st i ≠ who, j := 1 to num_requests
              st in_ready[ i, j] →
    if term_procs[ i] → in_ready[ i, j] := false
     [] else → count++
    fi
  af
  if count > 0 →
    send ready_op( ready_rec ( who, in_ready, count, found))
   [] else →
    send found( false, 0, 0)
  fi
fi
```

Figure 6.1 — Centralized Protocol *find_match* Handler

processes with which it was trying to communicate have terminated. Note that if a match is found in the loop, then the order of requests on *ready_op* is no longer the order in which they were received. To maintain the temporal ordering, a loop is used to move the unchecked requests from the beginning of the list to the end.

The handler must also deal with distributed termination. The body of *proc_done* is shown in Figure 6.2. The code loops through all the pending requests on *ready_op*. It checks each invocation to see if the termination of the process causes all the remaining communication statements of *ready_op* that are still possible to fail. If all the I/O commands for a given process have failed, the process is notified. If processes that are targets of the invocation's commands have not terminated, the invocation is resent to *ready_op*.

```
# mark that process who has terminated
term_procs[ who] := true
count := ?ready_op
do count > 0 →
  in ready_op( next_ready) →
    fa j := 1 to num_requests st next_ready.ready[ who, j] →
      next_ready.ready[ who, j] := false
      next_ready.ready_count––
    af
    if next_ready.ready_count = 0 →
          # all processes in this process's ready array have terminated
          # notify it of distributed termination
      send next_ready.found( false, 0, 0)
    [] else →
          # some processes in ready array are still active, return
          # request to the list
      send ready_op( next_ready)
    fi
  ni
  count––
od
```

Figure 6.2 — Centralized Protocol *proc_done* Handler

6.1.2. Distributed Solutions

In the centralized case, the server was a separate process equally accessible by all the processes. In the distributed case, the server code could be included with the application processes or they could be separate processes. In the papers describing the two algorithms examined below, the discussion was from the former point of view. This has the advantage that the state of the CSP process and variables such as the *ready* array can be visible to the server code without having to pass them as parameters. However, the server code must be able to respond to communication requests even when the CSP process is executing non-I/O code and does not currently wish to communicate. Therefore, the server must have a separate process that can respond to requests or an interrupt handler that can deal with requests no matter what the CSP process's state. For this thesis, each of the servers was placed in a separate process. Since the implementations of this thesis differ in point of view from the original papers, at times this thesis will refer to the server's state where the paper will refer to the process's state.

6.1.2.1. Bagrodia's Algorithm

Bagrodia [Bagr86] designed an algorithm that makes use of tokens, three queues, and three server states. Bagrodia defines an interaction as a pair of matched I/O commands and a set of interactions that a process currently wishes to execute. In this algorithm, a process sets the elements of its *ready* array for the members of its current interaction set. The algorithm seems to make the assumption that only a single interaction can occur between two processes, so the *ready* array is only one dimensional. Since, in general, some pair of processes is likely to have more than one interaction (e.g., process C in the first example of section 6.1), Bagrodia's algorithm is extended in this thesis.

See Appendix IV for the code implementing Bagrodia's algorithm.

The first addition to the extended algorithm is to add a second dimension to the *ready* array to make it look like the one in the centralized solution. The algorithm makes use of tokens that servers pass to one another in an attempt to find matching communication statements that are ready to be executed. A token contains the process id of the process requesting the interaction and an array of length two that contains the ids of the processes involved in the command. The information contained in a token is extended by adding the request number to specify which of the interactions between the two processes is being requested. There is a single token per interaction.

Each process's server maintains three token queues: *token_q*, *not_ready_q*, and *delay_q*. If the server's process is currently attempting to execute I/O commands, the *token_q* holds tokens that this process currently has that are members of the current interaction_set. The *not_ready_q* holds the tokens that this process currently has but that are not a part of the current interaction set. As in the centralized solution, when a process wants to execute I/O commands, it passes its *ready* array to its server. The server, when it receives this request, moves all the tokens that it currently holds onto the *not_ready_q*. It then goes through that queue. If the token is in the interaction set, it is moved to the *token_q*, otherwise it is returned to the *not_ready_q*.

The server maintains a tri-state variable, color, to keep track of its current state (black, white, or grey). If its process is currently executing and does not wish to perform an I/O command, the server's state is black. The server remains in state black until it receives a request from its process. It then initializes the *token_q* as discussed above and changes to state white. While white, the server can remove the first element, if it exists, from the *token_q*. It sends this token in a token message to the other server specified in the token and changes to state grey. If the server receiving this message is in state black or its process does not currently wish to execute the interaction specified by the token, it sends a cancel message to the requesting server. It then places the token onto the end of its *not_ready_q*. However, if the server receiving the token is in state white and does wish to perform this communication it just returns the token to the sender. It then notifies its process of the agreed upon communication and changes to black. The original sender of the token upon receiving its token back also notifies its process of the agreed upon communication and returns to state black. When a server is in state grey, it can receive token messages from servers other than the one to which it sent its token. As in state black or white, if the server's process is not ready to execute this command, a cancel message is sent. However, if its process is ready to execute this command, the server must be do something with this token while waiting for a reply to its request.

This situation is where the *delay_q* is used. A token received from process with an id less than the id of the server's process is placed on the *delay_q*. If later the server receives a cancel message to its own request, then it can remove the first element from the *delay_q* and send it to the original sender, thereby agreeing upon the communication. If the process id of the sender of the token is greater than that of the server, a cancel message is sent and the token is placed on the *token_q*. This action prevents deadlock. If all acceptable tokens received while in state grey were indiscriminately placed on the *delay_q*, then deadlock could occur. It is possible for all the servers to send out a token message and change to state grey. If all these requests were ready to be executed by the receiving processes, the tokens would all be placed on the *delay_q*'s. Then, all the servers would wait forever for a response to their token that will never arrive, hence deadlock. By having higher numbered processes send cancel messages while in state grey to requests from lower numbered processes, this ensures that a lower numbered process will receive a cancel message and accept one of the requests on its *delay_q*. Therefore, deadlock will not occur in this situation. When a server with a non-empty *delay_q* agrees upon an interaction with another server, it must send cancel messages to all the servers with tokens on the *delay_q*. These tokens are then placed into *not_ready_q*.

The implementation must give each token to one of the servers of the two processes involved in the communication. To facilitate this, two operations were added to the implementation of Bagrodia's code. For each token a process initially has, it calls *init_token* with the process id of the other process and the request number. This operation builds a token and adds it to *not_ready_q*. The other operation, *token_init_done*(), is called when all the tokens this process has have been sent. In this work, the tokens were always initially given to the process performing the output command.

The algorithm as given by Bagrodia has a problem that can be considered a matter of interpretation of semantics. Bagrodia states that only elements of the ready array that are set to true are the ones corresponding to members of its interaction set. This requirement seems to mean that elements of process i's ready array ready[i, j] would be false for all j, which is a problem. When a process is in state grey and it receives a token it checks its *ready* array using the requester's process id as the first index. Unfortunately, if this message is its token being returned to it by the other process to agree to the I/O command the server originally requested, it finds this element of *ready* to be false. The server that returned the token has already notified its process of the "agreed" upon communication. Since the element of ready is false for the token it received, the server that originally sent the token sends a cancel message to itself! Therefore, no I/O commands can ever be executed. Rather than just have all the processes set their own portion of their *ready* arrays to true, the code was modified. Upon receiving a token in state grey, the server now first checks if the requesting process in the token is itself. If the process number is the server's, it notifies its process of the agreed upon I/O command.

The first problem leads into an issue of fairness. Even though fairness of implementation is not a necessity, the following simple fix improves the fairness of Bagrodia's algorithm. When a server in state grey receives its token back, it places this token on the *not_ready_q*, which can lead to starvation. Consider the following process C.

[A || B || C]
C = [*[A?j → ... [] B?k → ...]]

C just executes a loop in which it receives an integer from either A or B, does some processing, and then repeats. If the server for process C has both the tokens for these interactions and they are on its token_q in the order of A then B, then B's request might never be serviced. C's server will send the first token to A. A will accept by returning the token. According to the algorithm, C now puts this token on the *not_ready_q*. On the next pass of C's loop, it will send its server its *ready* array showing both interactions as being ready. The server moves all the tokens from the *token_q* to the *not_ready_q*. Then it moves the tokens that are ready onto the *token_q*. Since both interactions are ready, the two tokens will end up on the *token_q* in the order of A then B again. If process A is always ready to communicate with C, C will always make the communication with A, starving B. This starvation can be avoided by having the token placed on the end of the token_q rather than on the *not_ready_q* when an interaction is agreed upon. In this way, the next time C executes the loop, the order of tokens on the *token_q* will be B then A. Thus, the communication with B has the opportunity to execute.

6.1.2.2. Ramesh's Algorithm

Ramesh [Rame87] designed an algorithm that uses a ready array that differs from the one discussed above. This algorithm calls the array *guard*, whose elements can take on one of three values. One value is *closed*, which means the process does not wish to execute that I/O command. *closed* is equivalent to a process setting an element to false in the ready array before sending the array to its server. Another value is *untried*, which means that the process wishes to execute that I/O command but the server has yet to attempt to match this command. The final value is *rejected*, which means that a request to match up that command was refused by the other process.

In Bagrodia's algorithm, when a server initiates a request by sending a token to another server, it has locked itself into agreeing to this communication. It delays or sends a cancel message to any new requests until a response is received to the original request. Ramesh's algorithm uses the notion of capturing the processes involved in an I/O command. A server must not only capture the server of the other process involved but it must also capture itself. A server initiates a request by determining the two processes involved in the command. It then attempts to capture the process with the lower valued process id first by sending a request message. If the server successfully captures the first process, it then must capture the second. A server that is attempting to capture a second server and is not yet captured, may be captured by a third server. The point is that a server sending a request message is not locked into a command as a Bagrodia server sending a token message is. The server does not become locked in until it is captured. A server that is captured has stated its willingness to perform an I/O command but does not notify its process until it is notified that both servers have been captured.

The server can be in one of four possible states (E, Q1, Q2, W). In the E-state, the server's process does not currently wish to execute any I/O command, which is equivalent to Bagrodia's color black. When the server's process wishes to execute an I/O command, the server changes to the Q1-state. If an element of *guard* that is set to *untried* exists, the array *list* is set to the process ids in increasing order of the two processes involved in the communication. Then a *req* (request) message is sent to the first element of list and the server changes to state Q2. If a server finds itself in state Q1 with no elements of *guard* set to *untried*, it changes to state W. While in state W, a server must wait since it is unable to send any new requests. A server in state Q2 means the server is waiting for a response to a pending request. Servers in states Q1, Q2, and W all react similarly to messages from other servers, with the exception that servers in Q2 must deal with replies to their requests.

A server that receives a *req* message for which the element of the *guard* array is closed sends a *no* message in response. The server that receives the *no* message in response sets that element of *guard* to *rejected*, and returns to Q1-state. A server that receives a *req* message for which the *guard* is not closed sends a *yes* message to the requester and sets a boolean variable *captured* to true. When a server that is already captured receives a request whose *guard* is not *closed*, the request is placed on a delay queue. If both servers for an I/O command are captured, then *success* messages are sent to both servers by the server that initially made the request. These servers' processes are then notified of the agreed upon communication. Any requests on the delay queues of these servers are then sent *no* messages to signify their unwillingness to execute the I/O command. These servers then return to the E-state. A server that has captured one of the two servers in *list* but is rejected in its attempt to capture the second must send an *abort* message to the captured process. When a server receives an *abort* message, if its delay queue is non-empty, the oldest request on the queue is removed and a *yes* message is sent to the appropriate server. The server remains captured in this case. If the delay queue is empty then the server resets *captured* to false.

Ramesh gives a state transition table for the actions that servers should perform. In implementing this table, care must be taken to handle "unexpected" messages that are not explicitly dealt with in the table. For example, Figure 6.3 shows timelines for servers 1, 2, and 3. Processes 1 and 2 have a matching I/O command and so do 2 and 3. Server 2 is the first to enter state Q1 and sends a *req* message to server 1. While waiting for a response from 1, server 2 receives a *req* message from server 3. Server 2's guard is not *closed*, so it sends a *yes* message to 3. Server 3 then captures itself and sends a *success*

Figure 6.3 — Timing Problem with Ramesh's Algorithm

message to 2. Server 2 has still not received a response from 1 when it changes to state E. Sometime in the future, server 1 will send either a *yes* or a *no* message. If the message is a *no* message, nothing need be done in response. However, if it is a *yes* message, an abort message must be sent to 1. Nothing in the table explicitly deals with this possibility. There are other cases where "unexpected" messages can be received. However, it appears that in most, if not all, of these other cases, the server would just need to do nothing with the message.

6.2. Nested Processes

In the BNF that Hoare uses to define CSP, one of the commands that a process can execute is a parallel command. Thus, nested processes are introduced into the language. The parallel command does not complete until all its processes successfully terminate. Apt and Francez [Apt84] claim that scoping rules allow nested processes to refer directly (in an I/O command) to their siblings and siblings of their ancestors but a non-nested process can only refer to the ancestors of the nested processes. This claim seems to be an assumption based on the way most programming languages deal with nested procedures: nested procedures cannot be called by outside routines but a nested procedure can call a non-nested procedure. Since Hoare does not define the relationship of I/O commands between nested processes and non-nested processes, the above assumption is used in this work. The following example helps to illustrate what this assumption means:

```
[ X1 :: X2?k [ X11 :: X2?i; X12!i*5 || X12 :: X11?m ]
|| X2 :: X1!5; X1!8 ]
```

Both X1 and X11 contain an input statement to receive an integer from X2. The first of X2’s output statements will match up with X1’s input command and X2’s second output will match up with X11's command. Since X11 and X12 are siblings, they directly name each other in the passing of a value from X11 to X12.

Hoare also states that the processes in a parallel command must not use variable that are assigned values in other processes in the command. He does not say, however, if and how variables defined in a parent process can be used in the children processes. In Occam 2 [Burn88], a language with a communication mechanism similar to CSP's I/O commands, children processes are allowed to reference and even modify variables defined in their parent process. If a nested process does contain code to modify a variable defined in its parent, none of its siblings can refer to this variable. For the purposes of this work, nested processes will be allowed to reference their parent's variables but will not be allowed to modify them.

6.2.1. Transformation

Apt and Francez [Apt84] describe a methodology for removing the use of distributed termination in CSP code. They transform CSP code that uses distributed termination to code that sends explicit termination messages. This code obviates the protocol server code dealing with termination, *proc_done*, that was shown in section 6.1.1. This section describes a methodology, which is similar in spirit to the above transformation, for transforming CSP code that contains nested processes to code with all the processes at the same level. This transformed code can then be run without modifying the above communication protocol handlers.

The children processes are modified to start at the same time as all the top-level processes. The first thing these children processes do is to wait for a message from their parent process to begin executing. The process Y1 with nested process Y11 and Y12 that have code segments C1 and C2 respectively

```
[ Y1 :: ... [ Y11 :: C1 || Y12 :: C2] ... ]
```

will be transformed to

```
[ Y1 :: ... done1 := true; done2 := true;
        *[ done1 & Y11!start() → done1 := false
          [] done2 & Y12!start() → done2 := false
        ]
        *[ not done1 & Y11?end() → done1 := true
          [] not done2 & Y12?end() → done2 := true
        ]
       ...
|| Y11 ::  *[ Y1?start() → C1 ; Y1!end() ]
|| Y12 ::  *[ Y1?start() → C2 ; Y1!end() ]
]
```

The parent process will send start messages to all the children at the point in the original code where the nested processes would have been "created". This start message will also contain the values of any variables from the parent process that the children might need to access. The parent process after sending all the start messages will wait for end() messages from all its children. The nested processes will terminate via distributed termination when their parent process finishes.

The next major modification is to make all references that indirectly name a nested process as its target to directly name the formerly nested process. In the case of a simple I/O statement, the code is modified to an alternative statement with the guards referencing the process named in the original statement and the formerly nested processes with a corresponding I/O command. For instance, the code for X2 in the example in Section 6.2 would become

[[X1!5 → skip [] X11!5 → skip] [X1!8 → skip [] X11!8 → skip]]

Since it is obvious that the first output statement will correspond with the input statement in X1 and the second output command will correspond with the input statement in X11, the alternative statements are unnecessary. However, in general, when nested processes are involved, it will not be possible apriori to determine with which process an I/O command will communicate. Therefore, code like the alternative statements shown will be necessary. If an alternative or repetitive statement contains guards that indirectly reference a nested process, the arm of the statement is duplicated with the nested process's name being substituted for the parent's name.

Another area requiring modification is if a nested process makes use of distributed termination. If a guard in a nested process references a non-nested process, then no change needs to be made. However, if a guard in a nested process depends on distributed termination and the process referred to is a nested process, additional code is required. The problem is that when a nested process finishes executing its user code, it does not terminate, but instead is waiting for another start message. So, distributed termination will not occur. Therefore, a nested process that is used by its siblings (and/or its sibling's descendants) for distributed termination must send explicit termination messages to these processes.

In general, code with nesting deeper than one level can be transformed by recursively applying the above method. However, an additional parameter must be passed in start messages if a nested process depends on a sibling of one of its ancestors, which is also a nested process, for distributed termination. The terminating process will only send the explicit termination messages once per incarnation. If the dependent process is created multiple times per incarnation of its parent, it needs to remember if it has received the termination message during its parent's current incarnation. The additional parameter in the start message will tell the child if this is its first or a subsequent creation during the parent's current incarnation. If this is the child's first incarnation, then all the flags for the reception of termination messages are initialized to false. Subsequent incarnations will not modify these flags during initialization.

6.2.2. Centralized Solution

The centralized solution also starts the nested processes at the same time as their parent. Instead of using I/O commands to start the children, an SR message is used. Actually, the same SR message is used in the transformed code, but in the centralized case, no server need be involved. Because the start message is not a CSP message, the children processes cannot use distributed termination to exit the start loop. Instead, another SR message is used to notify the children to terminate completely. The end messages from the children processes to the parent are now just SR messages, too.

In this centralized implementation, the CSP processes only set those elements of the *ready* array they directly name. The modifications shown in Figure 6.4 handle the possible matching of an I/O command with a nested process. The centralized server is passed at initialization a *parent* array, which has an entry for every process. If the process is not nested, then it has no parent and its entry is zero. If the process is nested, then its entry has the process id of its parent. It is important in this implementation that the same request number be used by a parent and its children for the same type I/O command with another process. When checking a new request against the old requests on *ready_op*, it is necessary to check more than just the corresponding elements. A nested process will not

```
in find_match(in_ready, who, found) →
  var request : int
  var match_found : bool := false
  count := ?ready_op
  do count > 0 and not match_found →
    in ready_op( next_ready) →
    fa j := 1 to num_requests →
      # check to see if any processes are
      # children that match
      var ancestor_who := who
      var ancestor_next : int
      do ancestor_who ≠ 0 and not match_found →
        if next_ready.ready[ ancestor_who, j] →
          ancestor_next := next_ready.procnum
          do ancestor_next ≠ 0 and not match_found →
            if in_ready[ ancestor_next, j] →
              match_found := true
              request := j
            [] else →
              ancestor_next := parent[ ancestor_next]
            fi
          od
        fi
        ancestor_who := parent[ ancestor_who]
      od
    af
    if not match_found →
      send ready_op( next_ready)
    [] else →
      send next_ready.found( true, who, request)
      send found( true, next_ready.procnum, request)
      found_sends := found_sends + 2
    fi
[] ...
[] awaken( whose_children) →
    fa i := 1 to N st parent[ i] = whose_children →
      term_procs[ i] := false
    af
ni
```

Figure 6.4 — Modifications to Centralized Handler for Nesting

be directly named in a non-nested processes ready array (i.e., its element will not be set to true). Therefore, the other process's *ready* array must be checked to see if it is ready to perform the requested I/O operation with the nested process's parent. The *parent* array is used to get the id of the parent process. If both processes are nested, then both *ready* arrays will be checked with the id of the other process's parent. Note that if a match is found, it is still the processes with ids *who* and *next_ready.procnum* that have the matching I/O command and that are notified.

A new operation, *awaken*, is called by a process about to send start messages to its children. *awaken* sets elements of *term_procs* for the children processes of the calling process to **false** to show that these processes are not terminated. The operation is necessary because "normal" distributed termination is used in this solution. A nested process when ending its current incarnation **send**s a *proc_done* message to the server before sending an *end* message to its parent. This message allows the server to detect distributed termination for a process directly referencing a nested process. The call to *awaken* will not affect any process's *ready* array because the only processes that can directly reference the awakened process are its siblings (or their siblings), which cannot currently have a request pending.

6.2.3. Distributed Solution

The distributed solution uses the same method as the centralized implementation to start and end nested processes, i.e., just SR messages. As in the centralized code, the CSP processes only set those elements of the *ready* which they directly name. This includes the initial distribution of tokens via *init_token.* The distributed servers deal with nesting in a manner similar to the method used in the centralized solution.

The distributed servers deal with nested processes via use of a *parent* array. While placing tokens on *not_ready_q* due to calls to *init_token*, a token for each descendant of the named process is also created and placed on the queue. This can mean that tokens will be created for communications that do not actually exist. These erroneous tokens might be passed once but will never cause an error. When the server receives a *find_match* request, it scans the *not_ready_q* for tokens to be placed in the *token_q*. It not only places those tokens in the queue for which the element of *ready* is **true**, but also those tokens whose target process has an ancestor whose element is **true.** When a process receives a token message, it must check the *ready* array indexed by the id of the process sending the token and any ancestors of the sending process. If any of these elements is set to true, then the token is returned to the sending server and the communication is agreed upon.

An *awaken* operation plus an *awakened* operation were added to the server code. As in the centralized solution, the *awaken* operation is again called by a process about to "create" nested processes. The server, when receiving the **call** to *awaken*, **send**s *awakened* messages to all the servers with its id. A process upon receipt of an *awakened* message loops through the *parent* array and sets the elements of *term_procs* to false for the children of the process whose id was passed in. These messages are necessary for dealing with distributed termination. The discussion about termination in the centralized case applies to the distributed solution.

6.3. CSP Solutions to the Roller Coaster Problem

The CSP solution to the roller coaster problem, shown in Figure 6.5, uses two different I/O commands. The vehicle process upon arriving at the station attempts to output the message *board*() to any passenger process waiting to board. The passenger processes get in line to board the car by performing an input command on the *board*() message from the car. When this rendezvous occurs, the passenger enters the car and the car marks the fact that a seat has been taken. The car process uses an output command to

```
[ vehicle::
  *[ trips →
     # notify N passengers that they can board
     i := 0
     *[ i < N & (j : 1..num_pass) passenger(j)!board() →
             i++
     ]
     # leave station and "do" the ride
     # ride is now over notify passengers to debark
     *[ i > 0 & (j : 1..num_pass) passenger(j)!debark() →
             i−−
     ]
  ]
||
 passenger(i:1..num_pass)::
  *[ rides →
     # wait for notification that it is ok to board
     vehicle?board()
     # now wait for ride to be over
     vehicle?debark()
  ]
]
```

Figure 6.5 — CSP Solution to Roller Coaster Problem

send message *debark*() to notify passengers that the ride is over. Passengers perform an input command waiting to receive the *debark*() message from the car to know when they can exit. Note that when a rendezvous for either of these messages occurs, no data is passed. (Unless one counts the fact that the car process knows which passenger process, *j*, with which it communicated.) This lack of data means that either or both of these messages could have been coded having the passenger performing the output command and the vehicle performing the input command.

As was done for the other synchronization mechanisms, runs were made with the number of passengers per car, N, being four and the number of rides per passenger, R, being five. The implementations were run with the number of passengers wishing to ride, A, set to eight and then six. The number of trips the car needs to make, T, is ten for eight passenger and eight for six passengers. The code and output of these implementations appear in Appendix IV.

The centralized implementations uses a single instance of the server *central_csp_server* to service all I/O requests. An instance of the passenger process is used for each of the A passengers and one vehicle process is created for the car. Each request by a process to execute an I/O command requires two messages, *find_match* and *found*. *find_match* is sent by a process to the server to make a request. *found* is sent by the server to a process to indicate the I/O command that the process can execute or the failure to find a matching command. If the request succeeds and the I/O command is an output command, then a third message, the output command, is sent. Therefore, the total number of messages required by this implementation with all the passengers having ridden R times is

$$R * 4 * A + N * 6 * T + (A + 1).$$

The first term comes from A passengers taking R rides with each ride requiring two input commands. The second term comes from the car making T trips which require two sets of N output commands. Each process also sends a *proc_done* message to the server when it terminates, which gives the the last term of the expression.

The distributed solution instantiates the same number of vehicle and passenger processes as the centralized. An instance of Bagrodia's distributed handler *csp_server* is created per process. During initialization all the tokens are arbitrarily given to the server of the vehicle process.

The message counts for the eight passenger runs are shown in Table 6.1. Both the eight and six passenger runs gave the typical non-starvation patterns of the other

Algorithm	local requests	found msgs	output msgs	token msgs	cancel msgs	termination messages	token init	total msgs
Centralized	160	160	80	0	0	9	0	409
Distributed	160	160	80	225	65	61	25	776

Table 6.1 — Message Counts for Roller Coaster Solutions

mechanisms. However, this lack of starvation came from the order in which the servers used found matching I/O commands rather than the CSP code being starvation free. The centralized server avoided starvation because it maintained the order in which requests were made by the passengers. The distributed solution avoided starvation because of the order in which tokens happened to be passed (i.e., lucky happenstance, not foresight). The non-determinism of CSP could allow a different server algorithm to pick interactions such that passenger(s) starved.

6.4. Nested Process Problem

Figure 6.6 contains CSP code with nested processes to calculate the determinant of a 3 x 3 matrix using cofactor expansion. Process X1 executes a loop to receive a 3 x 3 matrix into the array A. It then executes a parallel command creating four processes. The three processes, X2(1), X2(2), and X2(3), each calculate a term in the cofactor expansion of A along the first row and pass the value onto the fourth process. The fourth process, X2(4), sums these values to determine the value of the determinant, which is then passed to process Y. For simplicity, a dummy process INPUT with two predefined matrices is used to simulate user input. Process X2(4) uses distributed termination to know when it has received all the partial values. X1 and Y also make use of distributed termination to know when to quit.

Figure 6.7 shows the code from Figure 6.6 after it has been transformed to eliminate nesting. The major modifications being the addition of start and startA messages from

```
[ X1 :: A[3,3] : int; *[ INPUT?A → [ X2(i:1..3)::CALC || X2(4)::TOTAL] ]
|| INPUT :: M1[3,3], M2[3,3]:int; X1!M1; X1!M2
|| Y :: detvalue, count : int; count := 1;
    *[ X1?detvalue →
              write("The determinant of Matrix",count,"is",detvalue);
              count++ ]
]
where
CALC =
  First_col, Second_col, Cofactor, value : int;
  First_col := (i mod 3) + 1
  Second_col := (First_col mod 3) + 1
  Cofactor := A[ 2, First_col] * A [ 3, Second_col] -
           A[ 3, First_col] * A [ 2, Second_col]
  value := Cofactor * A[ 1, i]
  X2( 4)!value
TOTAL =
   total_value, partial_value : int;
   total_value := 0;
   *[(i:1..3) X2(i)?partial_value →
                   total_value := total_value + partial_value
    ]
   Y!total_value
```

Figure 6.6 — Nested CSP code to Calculate a Determinant

```
[ X1 :: A[3,3] : int; done[4] : bool;
  *[ INPUT?A → send start messages to X2’s
     i := 0; *[ i < 3 → i++; X2(i)!startA( A); done[i]:= false]
     X2(4)!start(); done[4] := false;
     # wait for end messages from X2’s
     *[(i:1..4)not done[i] & X2(i)?end → done[i]:= true]
  ]
|| X2(i:1..3)::CALC
|| X2(4)::TOTAL
|| INPUT :: M1[3,3], M2[3,3]:int; X1!M1; X1!M2
|| Y :: detvalue, count : int; count := 1;
   *[ X1?detvalue →
              write("The determinant of Matrix",count,"is",detvalue);
              count++
    [] X2(4)?detvalue →
              write("The determinant of Matrix"count"is"detvalue);
              count++
   ]
]
where
CALC =
   *[ X1?startA( A) →
      First_col, Second_col, Cofactor, value, A[ 3,3] : int;
      First_col := (i mod 3) + 1
      Second_col := (First_col mod 3) + 1
      Cofactor := A[ 2, First_col] * A [ 3, Second_col] –
             A[ 3, First_col] * A [ 2, Second_col]
      value := Cofactor * A[ 1, i]
      X2( 4)!value
      X2( 4)!end()
      X1!end()
   ]
TOTAL =
   *[ X1?start() →
      total_value, partial_value, j : int; done[4]:bool;
      total_value := 0;
      j:= 0; *[ j<3 → j++; done[j] := false]
      *[(i:1..3)not done[i] X2(i)?partial_value →
             total_value := total_value + partial_value
        [](i:1..3)not done[i] & X2(i)?end() → done[ i] := true
      ]
      Y!total_value
      X1!end()
   ]
```

Figure 6.7 — Transformed CSP Code to Calculate a Determinant

X1 to the X2s. Also added are the end messages from all the X2s to X1 and the end messages from X2(1)..X2(3) to X2(4). Y is modified to receive an integer from X2(4) as well as from X1. In this case, the receiving of an integer from X1 could have been removed as an optimization.

For ease of programming, the solutions each used a resource, *determinant*, that contained the code for all of the CSP processes. An instance of this resource was created for each of the seven CSP processes in the solution. Each instance, with its own address space, started up a single CSP process based upon its process id. The transformed code was run once with the original centralized server and once with the original distributed server. The non-transformed code was then run using the modified centralized and distributed servers. The four solutions each used a 7 x 7 *ready* array. In the centralized and distributed implementations, the array could have been reduced to 7 x 3 since four of the requests became control messages instead of CSP messages.

Table 6.2 contains the message counts for the four runs. The control messages in the table are start, end, awaken, and awakened messages. The table shows that the transformed solutions do not make use of control messages. This is because the start and end messages in the transformed code are explicit CSP messages and their cost is included in other columns of the table. The transformed code makes no use of awaken or awakened messages. Distributed termination occurred six times in the transformed code and four times in both the centralized and distributed code. The transformed code had

Algorithm	local requests	found msgs	output msgs	token msgs	cancel msgs	term msgs	control msgs	token init	total msgs
Transformed Centralized	72	72	32	0	0	7	0	0	183
Transformed Distributed	72	72	32	85	21	34	0	19	335
Centralized	24	24	10	0	0	11	22	0	91
Distributed	24	24	10	30	10	53	36	12	199

Table 6.2 — Message Counts for Determinant Solution

eight cases where the server returned not found. However, two of these cases were when X2(4) made requests with the entire *ready* array set to **false** due to the *done* array being all **true.** So, these cases are not distributed termination.

6.5. Summary

This chapter examined centralized and distributed implementations of CSP communication constructs. Along with communication statements, distributed termination was also implemented. A method was developed to transform a CSP program that contains nested processes to a program with no nesting. This method involved additional CSP messages to simulate the nesting. A program modified by this transform can be run with either the centralized or distributed implementation without change to these servers. Other methods of implementing nesting were developed through modification of the centralized and distributed server code.

CHAPTER 7

Discussion

This chapter compares the synchronization mechanisms implemented in this research. It examines the usefulness of anticipatory execution and its possible application to synchronization mechanisms other than CCRs. This chapter also discusses the choice of SR as the language to implement the work done in this thesis.

7.1. Comparison of Mechanisms

Pseudocode was written for the cheap roller coaster problem using the four synchronization mechanisms. The same basic algorithm is used in all four programs. A car arrives in the station and signals this occurrence to allow N passengers to board. As passengers board, they notify the car so that it knows when it can leave. Similarly, the car marks the end of the ride and passengers mark their exiting from the car.

The semaphore solution did not make use of shared memory; instead it used four semaphores to signal/wait for the events to occur. Because the V's performed on a semaphore are asynchronous, pairs of semaphore operations need to be performed to notify a passenger that (s)he can board and notify the car that the passenger has taken a seat. In contrast, in the CSP code, which uses synchronous communication, required only two distinct messages. When a board message is sent between the car and a passenger, both processes execute the I/O command synchronously, so the two operations of being allowed to board and actually boarding can be performed in one step. In these two solutions, the car must perform N **P** operations or perform N output commands to notify passengers of arriving at the station or ending the ride. On the other hand, because the CCR and monitor code use shared variables, by incrementing a resource or monitor variable once, the car process is able to notify all passengers of arrival or ending. These two solutions are very similar. The conditionals in the monitor code are simply the negation of the guards in the CCR code. The major difference between the two programs being the use of **signal** and **wait** operations on condition variables in the monitor code.

The CSP and semaphore solutions could have been written in the same manner as the CCR and monitor code. The CSP program could have used an additional process, the conductor, that served the role of a monitor. The car and passenger processes would communicate with the conductor process rather than directly with each other. The conductor would use variables identical to the resource or monitor variables. The conductor code would look like the centralized CCR server code, *roller_server_cent*. Rather than contain an **in** statement in an infinite loop using SR operations and scheduling expressions, the CSP code would contain a repetitive statement containing guards with I/O commands. The I/O commands would take the place of the operations and the rest of the guards would be identical to the synchronization expression. The semaphore code using shared variables and a semaphore for mutex could mimic the CCR code. This coding should be readily apparent, since the distributed implementation of the CCR code used a distributed mutex algorithm (i.e., a binary semaphore) to obtain exclusive access to the shared resource variables.

The semaphore, CCR, and monitor mechanisms are fair in the sense that they service the oldest invocations first. CSP, on the other hand, is non-deterministic and as such has no required fairness. The CSP code gave a fair solution because the servers happened to find matching I/O commands in an order that avoided starvation. The fairness properties of the other three mechanisms ensured the starvation free execution of the other solutions.

7.2. Generalizing the Cheap Roller Coaster Problem

The cheap roller coaster problem allows only a single car to be on the tracks. This section first discusses the effect on the pseudocode when the problem is extended to allow multiple cars on the track.

The CSP code is simple to modify. The code for the car process remains the same, the program uses an array of car processes instead of a single one. The passenger code is modified to use an alternative statement whose guards are to receive a *board* message from any car process. The number of the car process that sent the *board* message is stored. Then the input statement waiting for a *debark* message is modified to receive the message from the car process whose id was stored. Thus, a *debark* message can only be received from the car that the passenger boarded.

The CCR and monitor solutions can be extended to handle multiple cars by having a car save the value of *vin* (the number of cars that have arrived at the station) and passengers saving the value of *pin* (the total number of passengers that have taken a seat in a car). In the case of CCRs, these values are saved in local variables and for monitors the values are passed to the calling process in a parameter. The guards or conditional are then modified to use these saved values. For instance, in the cheap roller coaster code, after the car process arrives (and increments *vin*), it checks if it is full by using an expression based on *vout*, the number of cars that have departed. The extended code makes use of the saved arrival number. This change is necessary because multiple cars can be in the station at one time. Similarly, a passenger uses the saved value of *pin* in the guard to know when it can exit the car. If the original guard were used, a passenger in a car that was in the middle of a ride might think (s)he could exit. The problem is that the original guard just checks if fewer passengers have exited a car than the number of seats that were available in the cars that have completed the trip. By using the saved value of *pin*, the passengers can only exit when their particular seat is available for exiting.

It seems impossible to extend the semaphore algorithm from the way it was originally written to accommodate multiple cars. Since no shared memory is used, passengers have no knowledge of which car they entered. They just wait on the *debark* semaphore. It is not possible to ensure that a passenger that has boarded a later car has not performed a **P** on *debark* before a passenger in an earlier car. Therefore, it is possible for a passenger in a car that is not in the station to receive a *debark* signal. Thus, the semaphore code would have to be extensively rewritten to use shared memory and mimic the CCR code.

If the roller coaster problem is further extended so that cars need not finish in the same order they started, the CCR and monitor code must be modified further while the modified CSP code given above could be used in its entirety. The CCR and monitor code would have to be modified to look more like the CSP solution. Passengers would need to know the number of the car that they boarded. Cars would need arrays of variables to keep track of their current status and the number of passengers on board.

7.3. Duality of Synchronization Mechanisms

The fact that the four mechanisms were able to be used to solve the cheap roller coaster problem using the same basic algorithm makes them "equivalent" to a certain extent. However, to consider the mechanisms to be duals of each other, i.e., to be able use them completely interchangeably, seems to be unrealistic. The generalized roller coaster problem was much easier to solve using the CSP code than with the other mechanisms. The roller coaster problems are rendezvous problems. Because CSP communication is synchronous (i.e., rendezvous), it is easier to implement these problems in CSP. On the other hand, the disk scheduler problem solved using monitors with prioritized condition variables was much easier to code than it would have been using the other mechanisms. Monitors have operations that support synchronized scheduling, which none of the other mechanisms support. Hence, the disk scheduling problem, which requires prioritized scheduling is easier to implement with monitors. The point is that problems exist where it is advantageous to use a particular synchronization mechanism over the others. As Scott points out [Scot83], there exist four send/receive combinations. A language having all four of these combinations allows a programmer to choose the preferred method rather than being forced in one direction.

7.4. Distributed and Anticipatory Implementations

Though CCRs and especially monitors are considered centralized mechanisms, this thesis also implemented them in a distributed fashion. In the case of CCRs, the implementation required the use of a distributed mutex algorithm and the broadcasting of update messages for modified resource variable. The monitor implementation additionally required a distributed condition variable handler beyond the distributed mutex code and update messages for monitor variables. The distributed implementations did work, but their message costs were prohibitively more expensive than the centralized solutions.

The distributed CCR solution was extended to make use of anticipatory execution. By allowing the CCR server to execute the region statement code while the associated mutex server attempts to obtain exclusive access to the CCR, additional concurrency can be achieved. This solution also attempts to cut down on contention for mutex by requesting mutex only after the guard tests true. Update messages are only sent after it is determined that the region statement was executed without interference by another server. Given favorable conditions, i.e., light contention for access to the CCR, the anticipatory implementation was able to outperform the distributed solution in regards to message costs. However, the cost was still many times more expensive than a centralized approach.

Could anticipatory execution be applied to monitors? The answer is yes, however, it seems that it should not be. In the CCR case, the mutex is requested only when entering the region statement and released when exiting. Monitors not only deal with mutex at the top and bottom of monitor procedure but also when performing **signal** and **wait** operations on condition variables. In keeping with update messages only being sent after executing without interference, we assume that a monitor implementation would not perform a **signal** or a **wait** until a similar non-interference condition held. After executing a **signal** in SW or SU, or performing a **wait** and having been **signal**ed in SC, immediate anticipatory execution is sure to fail. In the SW and SU case, this is true because the process that was **signal**ed will immediately have access to the monitor, invalidating the anticipatory execution. A similar situation holds for the SC case, since the process that performed the **signal** will continue to execute in the monitor. Thus, it would appear that many attempts at anticipatory execution would occur without ever having the possibility of succeeding, making it unlikely that such an implementation would outperform a decentralized one.

By distributing processes of a program, the processes can concurrently move forward toward the program's goal. Processes executing in non-critical section of code, i.e., not currently making use of a synchronization mechanism, can execute in parallel with other processes. The synchronization mechanisms are necessary to allow the processes to communicate. It is an aesthetically pleasing notion to be able to use decentralized servers for the mechanisms. Unfortunately, as the implementations of the mechanism in this work have shown, the cost of the distributed implementations is much greater than the centralized versions. The distributed implementation of generalized semaphores was 15 times more expensive than the centralized algorithm. Furthermore, as more processes are added, the distributed message costs increase quadratically as opposed to the linear growth of the centralized. The distributed CCR and monitor code was from 8 to 40 times more expensive than their centralized counterparts. This was due to the cost of the distributed mutex algorithm and update messages. The distributed CSP server is twice as expensive as the centralized solution, even though it attempts to minimize the number of processes involved in finding a pair of matching I/O commands.

The distributed implementations of the critical sections of semaphores, CCRs, and monitors did not increase the overall concurrency of the solutions given by just distributing the non-critical code and using a centralized server. The serialized nature of these mechanisms means the distributed implementations of the critical sections adds message overhead without increasing concurrency. However, the distributed servers for CSP do allow more concurrency than the centralized server. The centralized handler is only able to find one pair of matching I/O commands at a time. With distributed servers, multiple pairs of corresponding I/O commands can be found at the same time.

Another reason for using decentralized server processes, which was not discussed in this work, is to aid in fault tolerance. However, the additional code required to ensure fault tolerance will further increase the already high message overhead of these implementations.

7.5. Using SR

As can be seen by the successful implementations of the four synchronization mechanisms that were solved using SR, SR is a robust language and was an appropriate choice for this research. SR provided all the low-level details so that all efforts could go into programming the mechanisms. Its communication commands support procedure calls, rendezvous, process creation, and message passing that Scott feels should all be available to a programmer.

A language such as Ada could have been used to implement the mechanisms, but would have required additional work and a loss in concurrency. For instance, the distributed mutex algorithm requires a server to broadcast its request for mutex. Both SR and Ada would use a loop to pass these requests onto its fellow servers. The SR code uses the **send** statement, which is *semi-synchronous* and does not require the recipient to be ready to service the *send* immediately. Thus, as soon as the data in the message is stored on the machine of the recipient process, the next **send** can be performed. Because Ada uses synchronous communication, a process must receive the message sent before the next message can be sent. So, concurrency is lost waiting for the non-buffered send to be serviced. The mutex server process could be given a server process of its own to receive request messages from the other servers. While this improves the concurrency for the sending process, this adds more processes and can still get backlogged by requests from multiple servers at the same time.

A couple of tools could have assisted in debugging and understanding the execution interleavings of SR code. When trying to determine the order of code execution in programs, **write** statements were added. Unfortunately, in the implementation we used, this output code can alter the scheduling of the processes. A debugger that was able to allow stepping through code without modifying the execution sequence would be of great value. Also, finalization code was used, when the program successfully completed, to print out statistics that were kept by the program. However, when the run did not complete successfully, as in the six passenger runs of the cheap roller coaster problem, this data was not available. It would be useful for the SR run-time support to maintain information about the use of operations (e.g., the number of invocations of each of the operations) and make this information available at the end of a run.

CHAPTER 8

Conclusions and Future Research

This thesis has examined four synchronization mechanisms: semaphores, CCRs, monitors, and the communication commands in CSP. This chapter summarizes the work done in this thesis and discusses possible future work.

8.1. Summary

We implemented the four mechanisms in centralized and decentralized manners. The work with semaphores not only implemented general semaphores but also implemented an optimized distributed algorithm for binary semaphores. This optimized algorithm was used for obtaining exclusive access in the distributed CCR and monitor implementations. The distributed CCR algorithm was extended to use anticipatory execution. The anticipatory algorithm was shown to outperform the distributed algorithm under favorable conditions. Monitors were implemented for three different signaling protocols: signal and continue, signal and wait, and signal and urgent wait. Also implemented was prioritized wait on condition variables. Beyond implementing I/O commands and distributed termination in CSP, methods of implementing nested processes in CSP were examined. These methods included the development of a transformation that maps code with nested processes into code with no nesting. Nesting was also implemented by changes to the server processes.

The distributed implementations of the synchronization code were more expensive in message costs to implement than the centralized versions. For the semaphore, CCR, and monitor solutions to the cheap roller coaster problem, the distributed solutions required 8 to 40 times more messages than the centralized versions. Programs with distributed user processes with one of these three distributed mechanism servers had no more concurrency than the solutions that had distributed user processes and a centralized server process. The distributed CSP code only required twice as many messages as the centralized algorithm. The distributed version also can increase concurrency by finding multiple pairs of matching I/O commands simultaneously.

8.2. Future Research

In this thesis, the conversion of pseudocode containing the synchronization mechanisms into SR code using the methodologies developed was all done by hand. One extension to this work would be to automate this conversion process. A standardized syntax would need to be defined that does not require the user to know more than the basic sequential statements of a language like SR or Pascal and the syntax of the mechanism to be used. This syntax should allow the code for problems such as dining philosophers, bounded buffer, etc., found in textbooks to be used with only minor modification.

Some of the implementations in this thesis have been very costly in terms of message passing. Optimizations not only to decrease the number of messages but also to decrease the size of messages can be found. In the case of CCRs and monitors, a server in the distributed implementations has to send update messages to the other servers when resource or monitor variables are modified. These update messages contain all the shared variables even if only one of the variables is modified. By using a set of update messages that send subsets of the shared variables, the size of messages could be decreased without increasing the amount of message traffic. These extra messages would require that there be more SR operations defined, which could increase the size of the programs and lengthen compile time.

The sending of only the variables that have changed could also be used to reduce contention for mutual exclusion in the distributed implementation of CCRs. If the guard of a region statement fails, the server releases mutex and waits to receive updated data. When the new values arrive, the server obtains mutex and retests the guard. If the variables used in the guard were not changed by the update message(s), the obtaining of mutex was a waste of time. With update messages that only send the changed variables, wasted requests for mutex and retests of guards can be avoided. A server whose guard was false on its last try can check if the update message(s) received have modified those variables used in its guard. If the guard variables have not changed, it is not necessary to get mutex and retest the guard. The server can just wait for more update messages.

In CSP, certain cases might allow the server process to be bypassed. For instance, a process that wants to execute a simple output statement might be allowed to just perform the SR **call** to the other process's operation. When the process that is the target of this output command is ready to execute the corresponding input command, the code for this process would check if the other process is ready by using the SR '?' operator. If the check shows that the other process has already performed the **call**, the process can execute the matching **receive** statement. The corresponding I/O commands would execute without the servers being contacted. Of course, precautions must be taken against the case where the process performing the input command is prepared to execute the command before the outputting process gets to its command.

The distributed server algorithm for CSP code with nested processes used a rather naive approach. It is expensive in terms of having to search through the *parent* and *ready* arrays when receiving tokens from nested processes. It is hoped that further research can develop a more efficient algorithm.

APPENDIX I

Semaphore Code and Runs

The following are listings and runs for the roller coaster problem using semaphores. First comes all the code for the centralized solution and a run with this code for six passengers. Next comes the startup resource for the distributed solution, the distributed semaphore resource with only explicit acks, Passenger and Vehicle resources. Roller_globals is identical so it is omitted. A run with six passengers is shown. Finally, comes the distributed semaphore resource with implicit acks. All the rest of the code is identical to the code used for the other distributed solution. Runs for eight and six passengers are shown. The runs for eight passengers for the centralized and distributed with only explicit acks were omitted since they are identical except for lacking the message counts at the end.

```
This is the start up process for the roller coaster problem
using centralized semaphores
resource roller_main_cent
    import Roller_globals
    import cent_Semaphore
    import Vehicle
    import Passenger

body roller_main_cent()
    op done()
    var num_of_pass : int
    var num_of_trips : int
    var tot_procs : int

    process start
        writes("Welcome to the roller coaster!")
        write("  Each passenger will get to ride 5 times.")
        write("Enter the total number of passengers who will wish to ride.")
        read( num_of_pass)
        num_of_trips := ( num_of_pass * 5 + ( N - 1))/ N
        tot_procs := num_of_pass + 1
        var passengers[ 1 : num_of_pass] : cap Passenger
        var car : cap Vehicle
        var Board_vehicle : cap cent_Semaphore
        var Passenger_on_board : cap cent_Semaphore
        var Debark : cap cent_Semaphore
        var Passenger_off : cap cent_Semaphore

        create centralized semaphore handlers
        Board_vehicle := create cent_Semaphore( "Board_vehicle")
        Passenger_on_board := create cent_Semaphore( "Passenger_on_board")
        Debark := create cent_Semaphore( "Debark")
        Passenger_off := create cent_Semaphore( "Passenger_off")

        create passenger processes
        fa i := 1 to num_of_pass →
           passengers[ i] := create Passenger( i, Board_vehicle,
                                 Passenger_on_board,
                                 Debark, Passenger_off,
```

```
                        done)
        af

        create the car process
        car := create Vehicle( num_of_trips, Board_vehicle,
                      Passenger_on_board, Debark,
                      Passenger_off, done)


        wait for the passenger and car processes to complete
        fa i := 1 to tot_procs →
           receive done()
        af

        destroy the instances of all the processes created
        destroy car

        fa i := 1 to num_of_pass →
           destroy passengers[ i]
        af

        destroy Board_vehicle
        destroy Passenger_on_board
        destroy Debark
        destroy Passenger_off
    end
end roller_main_cent


global Roller_globals
        const N := 4    number passengers per car
end Roller_globals
```

This resource implements the passenger process of the roller coaster *problem* using centralized semaphore servers. The passenger rides five times.

```
resource Passenger
   import Roller_globals
   import cent_Semaphore

body Passenger( Pnum : int; Board_vehicle, Passenger_on_board,
            Debark, Passenger_off : cap cent_Semaphore;
            done : cap())

   process Passenger_proc
      fa i := 1 to 5 →
         wait for notification that is ok to board
         call Board_vehicle.P_oper()
         write("Passenger ",Pnum," boarding car")
         notify vehicle that we have boarded
         send Passenger_on_board.V_oper()
         wait for ride to be over
         call Debark.P_oper()
         write("Passenger ",Pnum," leaving car")
         notify vehicle that we are off
         send Passenger_off.V_oper()
      af
      write("Passenger ",Pnum," done")
      send done()
   end
end
```

This resource implements the car process of the roller coaster
problem using centralized semaphore servers.

```
resource Vehicle
   import Roller_globals
   import cent_Semaphore

body Vehicle( num_of_trips : int;
          Board_vehicle, Passenger_on_board, Debark,
          Passenger_off : cap cent_Semaphore;
          done : cap())

   process Vehicle_proc
      fa i := 1 to num_of_trips →
         arrive at station, notify N passengers that they can board
         write("Car arrived at station")
         fa j := 1 to N →
            send Board_vehicle.V_oper()
         af
         wait for N passengers to have boarded
         fa j := 1 to N →
            call Passenger_on_board.P_oper()
         af
         leave station and do ride
         ride is no over, notify passengers to debark
         write("Car left station, done ride, and returned to station")
         fa j := 1 to N →
            send Debark.V_oper()
         af
         wait for all passengers to debark
         fa j := 1 to N →
            call Passenger_off.P_oper()
         af
         write("Car now empty")
      af
      write("Car done")
      send done()
   end
end
```

resource to handle in a centralized manner the P and V *operations* of a semaphore

```
resource cent_Semaphore
  op P_oper()                    P request
  op V_oper()                    V request

body cent_Semaphore( name:string(*))

  process semaphore_handler
    do true →
      receive V_oper()
      receive P_oper()
    od
  end
end cent_Semaphore
```

```
Script started on Wed May 31 10:33:53 1989
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times.
Enter the total number of passengers who will wish to ride.
6
Car arrived at station
Passenger  1  boarding car
Passenger  2  boarding car
Passenger  3  boarding car
Passenger  4  boarding car
Car left station, done ride, and returned to station
Passenger  1  leaving car
Passenger  2  leaving car
Passenger  3  leaving car
Passenger  4  leaving car
Car now empty
Car arrived at station
Passenger  5  boarding car
Passenger  6  boarding car
Passenger  1  boarding car
Passenger  2  boarding car
Car left station, done ride, and returned to station
Passenger  5  leaving car
Passenger  6  leaving car
Passenger  1  leaving car
Passenger  2  leaving car
Car now empty
Car arrived at station
Passenger  3  boarding car
Passenger  4  boarding car
Passenger  5  boarding car
Passenger  6  boarding car
Car left station, done ride, and returned to station
Passenger  3  leaving car
Passenger  4  leaving car
Passenger  5  leaving car
Passenger  6  leaving car
Car now empty
Car arrived at station
```

Passenger 1 boarding car
Passenger 2 boarding car
Passenger 3 boarding car
Passenger 4 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car
Passenger 2 leaving car
Passenger 3 leaving car
Passenger 4 leaving car
Car now empty
Car arrived at station
Passenger 5 boarding car
Passenger 6 boarding car
Passenger 1 boarding car
Passenger 2 boarding car
Car left station, done ride, and returned to station
Passenger 5 leaving car
Passenger 6 leaving car
Passenger 1 leaving car
Passenger 2 leaving car
Car now empty
Car arrived at station
Passenger 3 boarding car
Passenger 4 boarding car
Passenger 5 boarding car
Passenger 6 boarding car
Car left station, done ride, and returned to station
Passenger 3 leaving car
Passenger 4 leaving car
Passenger 5 leaving car
Passenger 6 leaving car
Car now empty
Car arrived at station
Passenger 1 boarding car
Passenger 2 boarding car
Passenger 3 boarding car
Passenger 4 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car

Passenger 2 leaving car
Passenger 3 leaving car
Passenger 4 leaving car
Passenger 1 done
Passenger 2 done
Passenger 3 done
Passenger 4 done
Car now empty
Car arrived at station
Passenger 5 boarding car
Passenger 6 boarding car
RTS warning: 8 blocked processes during termination
ivy% exit
ivy%
script done on Wed May 31 10:34:11 1989

Centralized run with 8 passengers was omitted, see run for 8 passengers using distributed algorithm explicit acks for an identical run at the bottom of Appendix I.

```
This is the start up process for the roller coaster problem
using "distributed" semaphores
resource roller_main_dist
    import Roller_globals
    import dist_Semaphore
    import Vehicle
    import Passenger

body roller_main_dist()
    op done()
    var num_of_pass : int
    var num_of_trips : int
    var tot_procs : int

    process start
        writes("Welcome to the roller coaster!")
        write("  Each passenger will get to ride 5 times.")
        write("Enter the total number of passengers who will wish to ride.")
        read( num_of_pass)
        num_of_trips := ( num_of_pass * 5 + ( N - 1))/ N
        tot_procs := num_of_pass + 1
        var passengers[ 1 : num_of_pass] : cap Passenger
        var car : cap Vehicle
        var Board_vehicle[ 1: tot_procs] : cap dist_Semaphore
        var Passenger_on_board[ 1: tot_procs] : cap dist_Semaphore
        var Debark[ 1: tot_procs] : cap dist_Semaphore
        var Passenger_off[ 1: tot_procs] : cap dist_Semaphore

        create passenger processes with their semaphore handlers
        fa i := 1 to num_of_pass →
           Board_vehicle[ i] := create dist_Semaphore( i, tot_procs,
                                 0, "Board_vehicle")
           Passenger_on_board[ i] := create dist_Semaphore( i,
                                 tot_procs,
                                 0, "Passenger_on_board")
           Debark[ i] := create dist_Semaphore( i,
                                 tot_procs,
                                 0, "Debark")
           Passenger_off[ i] := create dist_Semaphore( i,
```

```
                    tot_procs,
                    0, "Passenger_off")
  passengers[ i] := create Passenger( i, Board_vehicle[ i],
                  Passenger_on_board[ i],
                  Debark[ i], Passenger_off[ i],
                  done)
af

create car process with its semaphore handlers
var sem_number : int := num_of_pass + 1
Board_vehicle[ sem_number] := create dist_Semaphore( sem_number,
                    tot_procs, 0, "Board_vehicle")
Passenger_on_board[ sem_number] := create dist_Semaphore( sem_number,
                    tot_procs, 0, "Passenger_on_board")
Debark[ sem_number] := create dist_Semaphore( sem_number,
                    tot_procs, 0, "Debark")
Passenger_off[ sem_number] := create dist_Semaphore( sem_number,
                    tot_procs, 0, "Passenger_off")
car := create Vehicle( num_of_trips, Board_vehicle[ sem_number],
              Passenger_on_board[ sem_number],
              Debark[ sem_number],
              Passenger_off[ sem_number],
              done)

send communication links to semaphore handlers
fa i := 1 to tot_procs →
  send Board_vehicle[i].links( Board_vehicle)
  send Passenger_on_board[i].links( Passenger_on_board)
  send Debark[i].links( Debark)
  send Passenger_off[i].links( Passenger_off)
af

wait for the passenger and car processes to complete
fa i := 1 to tot_procs →
  receive done()
af

destroy the instances of all the processes created
destroy car
```

```
        fa i := 1 to num_of_pass →
          destroy passengers[ i]
        af

        fa i := 1 to tot_procs →
          destroy Board_vehicle[ i]
          destroy Passenger_on_board[ i]
          destroy Debark[ i]
          destroy Passenger_off[ i]
        af
    end
end roller_main_dist
```

This resource implements the passenger process of the roller coaster *problem* using distributed semaphore servers. The passenger rides five times.

```
resource Passenger
   import Roller_globals
   import dist_Semaphore

body Passenger( Pnum : int; Board_vehicle, Passenger_on_board,
            Debark, Passenger_off : cap dist_Semaphore;
            done : cap())

   process Passenger_proc
      fa i := 1 to 5 →
         wait for notification that is ok to board
         call Board_vehicle.P_oper()
         write("Passenger ",Pnum," boarding car")
         notify vehicle that we have boarded
         call Passenger_on_board.V_oper()
         wait for ride to be over
         call Debark.P_oper()
         write("Passenger ",Pnum," leaving car")
         notify vehicle that we are off
         call Passenger_off.V_oper()
      af
      write("Passenger ",Pnum," done")
      send done()
   end
end
```

```
This resource implements the car process of the roller coaster
problem using distributed semaphore servers.
resource Vehicle
   import Roller_globals
   import dist_Semaphore

body Vehicle( num_of_trips : int;
        Board_vehicle, Passenger_on_board, Debark,
        Passenger_off : cap dist_Semaphore;
        done : cap())

   process Vehicle_proc
      fa i := 1 to num_of_trips →
         arrive at station, notify N passengers that they can board
         write("Car arrived at station")
         fa j := 1 to N →
            call Board_vehicle.V_oper()
         af
         wait for N passengers to have boarded
         fa j := 1 to N →
            call Passenger_on_board.P_oper()
         af
         leave station and do ride
         ride is no over, notify passengers to debark
         write("Car left station, done ride, and returned to station")
         fa j := 1 to N →
            call Debark.V_oper()
         af
         wait for all passengers to debark
         fa j := 1 to N →
            call Passenger_off.P_oper()
         af
         write("Car now empty")
      af
      write("Car done")
      send done()
   end
end
```

```
resource to handle in a distributed manner the P and V
operations of a semaphore - using "Lamport/Schneider Alg"
this version stores only P requests and only sends
acks to the requester.  It uses only explicit acks.
resource dist_Semaphore
  type PorV = enum( Poper, Voper)
  op P_oper()                          local P
                                    from "our" process
  op V_oper()                          local V
                                    from "our" process
  op request( type_req : PorV; timestamp, who : int)
  op ack_op( tsack : int )
  op links( sem_handlers[ 1: *] : cap dist_Semaphore)

body dist_Semaphore( me, N, sem_initial_val : int; name:string(*))

  var Pcount := 0, Vcount := sem_initial_val
  var clck : int := 0
  var ack_count := 0
  const acks_needed := N - 1
  op local_request( request_type : PorV)
  op request_serviced()
  op Qupdate()

  proc P_oper()
    send local_request( Poper)
    receive request_serviced()
  end

  proc V_oper()
    send local_request( Voper)
  end

  var ts_my_req : int
  var myPcount := 0
  var earlier := 0, later := 0

  process semaphore_handler
    get the capabilities of the handlers of this semaphore
```

```
var sem_handlers[ 1 : N] : cap dist_Semaphore
receive links( sem_handlers)

do true →
  in local_request( request_type) →
    clck++
    fa i := 1 to N st i ≠ me →
      send sem_handlers[ i].request( request_type, clck, me)
    af
    if request_type = Poper →
      P request, put message on Q to be fully acked
      and save time of our request
      ts_my_req := clck
    [] else →
      V request, just update Vcount
      Vcount++
    fi
  [] request( type_req, ts, who_req) →
    clck := max( ts, clck + 1)
    if type_req = Poper →
      need to ack a P message
      send sem_handlers[ who_req].ack_op( clck)
      determine if the message is before or after our msg
      if ( ts_my_req > 0) and
        (( ts > ts_my_req) or
         ( ts = ts_my_req and who_req > me)) →
        later++
      [] else →
        earlier++
      fi
    [] else →
      this is a V message, increment V count
      Vcount++
      check if a P that me requested is fully acked.
      if (myPcount > 0) and (myPcount ≤ Vcount) →
        send request_serviced()
        myPcount := 0
      fi
    fi
```

```
        [] ack_op( tsack) →
          clck := max( tsack, clck + 1)
          ack_count++
          if ack_count = acks_needed →
             Qupdate()
          fi
       ni
    od
  end

  proc Qupdate()
    message is fully acked, remove everything from Q with "lesser" timestamp
    Pcount := Pcount + earlier
    earlier := later
    later := 0
    now increment Pcount for our message being fully acked
    and save this Pcount as ours
    Pcount++
    myPcount := Pcount
    ts_my_req := 0
    ack_count := 0
    now count messages that were fully acked but of lower precedence
    check if a P that me requested is fully acked.
    if (myPcount > 0) and (myPcount ≤ Vcount) →
       send request_serviced()
       myPcount := 0
    fi
  end
end dist_Semaphore
```

```
Script started on Tue May 16 18:39:09 1989
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times.
Enter the total number of passngers who will wish to ride.
6
Car arrived at station
Passenger  1  boarding car
Passenger  2  boarding car
Passenger  3  boarding car
Passenger  4  boarding car
Car left station, done ride, and returned to station
Passenger  1  leaving car
Passenger  2  leaving car
Passenger  3  leaving car
Passenger  4  leaving car
Car now empty
Car arrived at station
Passenger  1  boarding car
Passenger  2  boarding car
Passenger  5  boarding car
Passenger  6  boarding car
Car left station, done ride, and returned to station
Passenger  1  leaving car
Passenger  2  leaving car
Passenger  5  leaving car
Passenger  6  leaving car
Car now empty
Car arrived at station
Passenger  3  boarding car
Passenger  4  boarding car
Passenger  5  boarding car
Passenger  6  boarding car
Car left station, done ride, and returned to station
Passenger  3  leaving car
Passenger  4  leaving car
Passenger  5  leaving car
Passenger  6  leaving car
Car now empty
Car arrived at station
```

Passenger 1 boarding car
Passenger 2 boarding car
Passenger 3 boarding car
Passenger 4 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car
Passenger 2 leaving car
Passenger 3 leaving car
Passenger 4 leaving car
Car now empty
Car arrived at station
Passenger 1 boarding car
Passenger 2 boarding car
Passenger 5 boarding car
Passenger 6 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car
Passenger 2 leaving car
Passenger 5 leaving car
Passenger 6 leaving car
Car now empty
Car arrived at station
Passenger 3 boarding car
Passenger 4 boarding car
Passenger 5 boarding car
Passenger 6 boarding car
Car left station, done ride, and returned to station
Passenger 3 leaving car
Passenger 4 leaving car
Passenger 5 leaving car
Passenger 6 leaving car
Car now empty
Car arrived at station
Passenger 1 boarding car
Passenger 2 boarding car
Passenger 3 boarding car
Passenger 4 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car

Passenger 2 leaving car
Passenger 3 leaving car
Passenger 4 leaving car
Passenger 1 done
Passenger 2 done
Passenger 3 done
Passenger 4 done
Car now empty
Car arrived at station
Passenger 5 boarding car
Passenger 6 boarding car
RTS warning: 32 blocked processes during termination
ivy% exit
ivy%
script done on Tue May 16 18:39:38 1989

Distributed run with 8 passengers was omitted, see run for 8 passengers using distributed algorithm explicit acks for an identical run at the bottom of Appendix I.

The following is the distributed semaphore handler that uses implicit acks. The code for the resources roller_main_dist, Passenger and Vehicle are omitted, since the ones used for this version are identical to the distributed version with only explicit acks shown above.

resource to handle in a distributed manner the P and V
operations of a semaphore - using "Lamport/Schneider Alg"
this version stores only P requests and only sends
acks to the requester. It makes use of implicit acks.

```
resource dist_Semaphore
   type PorV = enum( Poper, Voper)
   op P_oper()                    local P
                              from "our" process
   op V_oper()                    local V
                              from "our" process
   op request( type_req : PorV; timestamp, who : int)
   op ack_op( who_ack, tsack : int )
   op links( sem_handlers[ 1: *] : cap dist_Semaphore)

body dist_Semaphore( me, N, sem_initial_val : int; name:string(*))

   op local_request( request_type : PorV)
   op request_serviced()

   proc P_oper()
      send local_request( Poper)
      receive request_serviced()
   end

   proc V_oper()
      send local_request( Voper)
   end

   var Preq_out : bool := false
   var ts_my_req : int
   var myPcount := 0
   var earlier := 0, later := 0
   var ack_array[ N] : bool := ([N]true)
   var explicit_ack := 0, implicit_ack := 0, Total_requests := 0
   var Pcount := 0, Vcount := sem_initial_val
```

```
var clck : int := 0
var ack_count := 0
const acks_needed := N - 1
op Qupdate()

process semaphore_handler
   get the capabilities of the handlers of this semaphore
   var sem_handlers[ 1 : N] : cap dist_Semaphore
   receive links( sem_handlers)

   do true →
      in local_request( request_type) →
         clck++
         fa i := 1 to N st i ≠ me →
            send sem_handlers[ i].request( request_type, clck, me)
            ack_array[ i] := false
         af
         ts_my_req := clck
         Total_requests++
         if request_type = Poper →
            P request, put message on Q to be fully acked
            and save time of our request
            Preq_out := true
          [] else →
            V request, just update Vcount
            Vcount++
         fi
      [] request( type_req, ts, who_req) →
         clck := max( ts, clck + 1)
         if type_req = Poper →
            if ts_my_req < ts →
               need to ack a P message
               send sem_handlers[ who_req].ack_op( me, clck)
            fi
            determine if the message is before or after our msg
            if Preq_out and
               (( ts > ts_my_req) or
                ( ts = ts_my_req and who_req > me)) →
               later++
```

```
          [] else →
            earlier++
          fi
        [] else →
          this is a V message, increment V count
          Vcount++
          check if a P that me requested is fully acked.
          if (myPcount > 0) and (myPcount ≤ Vcount) →
            send request_serviced()
            myPcount := 0
          fi
        fi
        check if message received implicitly acks our latest P request
        if Preq_out and not ack_array[ who_req] and ts ≥ ts_my_req →
          ack_array[ who_req] := true
          ack_count++
          implicit_ack++
          if ack_count = acks_needed →
            Qupdate()
          fi
        fi
      [] ack_op( who_ack, tsack) →
        clck := max( tsack, clck + 1)
          ack_array[ who_ack] := true
          ack_count++
          explicit_ack++
          if ack_count = acks_needed →
            Qupdate()
          fi
    ni
  od
end

proc Qupdate()
  message is fully acked, remove everything from Q with "lesser" timestamp
  Pcount := Pcount + earlier
  earlier := later
  later := 0
  now increment Pcount for our message being fully acked
```

```
    and save this Pcount as ours
    Pcount++
    myPcount := Pcount
    ack_count := 0
    Preq_out := false
    now count messages that were fully acked but of lower precedence
    check if a P that me requested is fully acked.
    if (myPcount > 0) and (myPcount ≤ Vcount) →
      send request_serviced()
      myPcount := 0
    fi
  end

  final
    write("For process",me,"handler of ",name,"there were")
    writes(explicit_ack," explicit acks and ",implicit_ack," implicit acks")
    write(" on",Total_requests,"requests.")
  end

end dist_Semaphore
```

```
Script started on Tue May 16 18:40:33 1989
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times.
Enter the total number of passengers who will wish to ride.
8
Car arrived at station
Passenger  1  boarding car
Passenger  2  boarding car
Passenger  3  boarding car
Passenger  4  boarding car
Car left station, done ride, and returned to station
Passenger  1  leaving car
Passenger  2  leaving car
Passenger  3  leaving car
Passenger  4  leaving car
Car now empty
Car arrived at station
Passenger  5  boarding car
Passenger  6  boarding car
Passenger  7  boarding car
Passenger  8  boarding car
Car left station, done ride, and returned to station
Passenger  5  leaving car
Passenger  6  leaving car
Passenger  7  leaving car
Passenger  8  leaving car
Car now empty
Car arrived at station
Passenger  1  boarding car
Passenger  2  boarding car
Passenger  3  boarding car
Passenger  4  boarding car
Car left station, done ride, and returned to station
Passenger  1  leaving car
Passenger  2  leaving car
Passenger  3  leaving car
Passenger  4  leaving car
Car now empty
Car arrived at station
```

Passenger 5 boarding car
Passenger 6 boarding car
Passenger 7 boarding car
Passenger 8 boarding car
Car left station, done ride, and returned to station
Passenger 5 leaving car
Passenger 6 leaving car
Passenger 7 leaving car
Passenger 8 leaving car
Car now empty
Car arrived at station
Passenger 1 boarding car
Passenger 2 boarding car
Passenger 3 boarding car
Passenger 4 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car
Passenger 2 leaving car
Passenger 3 leaving car
Passenger 4 leaving car
Car now empty
Car arrived at station
Passenger 5 boarding car
Passenger 6 boarding car
Passenger 7 boarding car
Passenger 8 boarding car
Car left station, done ride, and returned to station
Passenger 5 leaving car
Passenger 6 leaving car
Passenger 7 leaving car
Passenger 8 leaving car
Car now empty
Car arrived at station
Passenger 1 boarding car
Passenger 2 boarding car
Passenger 3 boarding car
Passenger 4 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car

Passenger 2 leaving car
Passenger 3 leaving car
Passenger 4 leaving car
Car now empty
Car arrived at station
Passenger 5 boarding car
Passenger 6 boarding car
Passenger 7 boarding car
Passenger 8 boarding car
Car left station, done ride, and returned to station
Passenger 5 leaving car
Passenger 6 leaving car
Passenger 7 leaving car
Passenger 8 leaving car
Car now empty
Car arrived at station
Passenger 1 boarding car
Passenger 2 boarding car
Passenger 3 boarding car
Passenger 4 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car
Passenger 2 leaving car
Passenger 3 leaving car
Passenger 4 leaving car
Passenger 1 done
Passenger 2 done
Passenger 3 done
Passenger 4 done
Car now empty
Car arrived at station
Passenger 5 boarding car
Passenger 6 boarding car
Passenger 7 boarding car
Passenger 8 boarding car
Car left station, done ride, and returned to station
Passenger 5 leaving car
Passenger 6 leaving car
Passenger 7 leaving car

```
Passenger  8  leaving car
Passenger  5  done
Passenger  6  done
Passenger  7  done
Passenger  8  done
Car now empty
Car done
For process 1 handler of  Board_vehicle  there were
21 explicit acks and 19 implicit acks.
For process 1 handler of  Passenger_on_board  there were
0 explicit acks and 0 implicit acks.
For process 1 handler of  Debark  there were
25 explicit acks and 15 implicit acks.
For process 1 handler of  Passenger_off  there were
0 explicit acks and 0 implicit acks.
For process 2 handler of  Board_vehicle  there were
22 explicit acks and 18 implicit acks.
For process 2 handler of  Passenger_on_board  there were
0 explicit acks and 0 implicit acks.
For process 2 handler of  Debark  there were
27 explicit acks and 13 implicit acks.
For process 2 handler of  Passenger_off  there were
0 explicit acks and 0 implicit acks.
For process 3 handler of  Board_vehicle  there were
23 explicit acks and 17 implicit acks.
For process 3 handler of  Passenger_on_board  there were
0 explicit acks and 0 implicit acks.
For process 3 handler of  Debark  there were
29 explicit acks and 11 implicit acks.
For process 3 handler of  Passenger_off  there were
0 explicit acks and 0 implicit acks.
For process 4 handler of  Board_vehicle  there were
24 explicit acks and 16 implicit acks.
For process 4 handler of  Passenger_on_board  there were
0 explicit acks and 0 implicit acks.
For process 4 handler of  Debark  there were
31 explicit acks and 9 implicit acks.
For process 4 handler of  Passenger_off  there were
0 explicit acks and 0 implicit acks.
```

For process 5 handler of Board_vehicle there were
21 explicit acks and 19 implicit acks.
For process 5 handler of Passenger_on_board there were
0 explicit acks and 0 implicit acks.
For process 5 handler of Debark there were
25 explicit acks and 15 implicit acks.
For process 5 handler of Passenger_off there were
0 explicit acks and 0 implicit acks.
For process 6 handler of Board_vehicle there were
24 explicit acks and 16 implicit acks.
For process 6 handler of Passenger_on_board there were
0 explicit acks and 0 implicit acks.
For process 6 handler of Debark there were
29 explicit acks and 11 implicit acks.
For process 6 handler of Passenger_off there were
0 explicit acks and 0 implicit acks.
For process 7 handler of Board_vehicle there were
27 explicit acks and 13 implicit acks.
For process 7 handler of Passenger_on_board there were
0 explicit acks and 0 implicit acks.
For process 7 handler of Debark there were
33 explicit acks and 7 implicit acks.
For process 7 handler of Passenger_off there were
0 explicit acks and 0 implicit acks.
For process 8 handler of Board_vehicle there were
30 explicit acks and 10 implicit acks.
For process 8 handler of Passenger_on_board there were
0 explicit acks and 0 implicit acks.
For process 8 handler of Debark there were
37 explicit acks and 3 implicit acks.
For process 8 handler of Passenger_off there were
0 explicit acks and 0 implicit acks.
For process 9 handler of Board_vehicle there were
0 explicit acks and 0 implicit acks.
For process 9 handler of Passenger_on_board there were
320 explicit acks and 0 implicit acks.
For process 9 handler of Debark there were
0 explicit acks and 0 implicit acks.
For process 9 handler of Passenger_off there were

```
320 explicit acks and 0 implicit acks.
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times.
Enter the total number of passengers who will wish to ride.
6
Car arrived at station
Passenger  1  boarding car
Passenger  2  boarding car
Passenger  3  boarding car
Passenger  4  boarding car
Car left station, done ride, and returned to station
Passenger  1  leaving car
Passenger  2  leaving car
Passenger  3  leaving car
Passenger  4  leaving car
Car now empty
Car arrived at station
Passenger  1  boarding car
Passenger  2  boarding car
Passenger  5  boarding car
Passenger  6  boarding car
Car left station, done ride, and returned to station
Passenger  1  leaving car
Passenger  2  leaving car
Passenger  5  leaving car
Passenger  6  leaving car
Car now empty
Car arrived at station
Passenger  3  boarding car
Passenger  4  boarding car
Passenger  5  boarding car
Passenger  6  boarding car
Car left station, done ride, and returned to station
Passenger  3  leaving car
Passenger  4  leaving car
Passenger  5  leaving car
Passenger  6  leaving car
Car now empty
Car arrived at station
```

Passenger 1 boarding car
Passenger 2 boarding car
Passenger 3 boarding car
Passenger 4 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car
Passenger 2 leaving car
Passenger 3 leaving car
Passenger 4 leaving car
Car now empty
Car arrived at station
Passenger 1 boarding car
Passenger 3 boarding car
Passenger 5 boarding car
Passenger 6 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car
Passenger 3 leaving car
Passenger 5 leaving car
Passenger 6 leaving car
Car now empty
Car arrived at station
Passenger 1 boarding car
Passenger 2 boarding car
Passenger 4 boarding car
Passenger 5 boarding car
Car left station, done ride, and returned to station
Passenger 1 leaving car
Passenger 2 leaving car
Passenger 4 leaving car
Passenger 5 leaving car
Passenger 1 done
Car now empty
Car arrived at station
Passenger 2 boarding car
Passenger 3 boarding car
Passenger 4 boarding car
Passenger 6 boarding car
Car left station, done ride, and returned to station

```
Passenger  2  leaving car
Passenger  3  leaving car
Passenger  4  leaving car
Passenger  6  leaving car
Passenger  2  done
Passenger  3  done
Passenger  4  done
Car now empty
Car arrived at station
Passenger  5  boarding car
Passenger  6  boarding car
RTS warning: 32 blocked processes during termination
ivy% exit
ivy%
script done on Tue May 16 18:41:01 1989
```

APPENDIX II

CCR Code and Runs

The following are listings and runs for the roller coaster problem using CCRs. First comes all the code for the centralized solution and runs with this code for eight and six passengers. Next comes all the code for the distributed solution and runs for eight and six passengers. Next comes the CCR_server and Mutex_server for the anticipatory solution. All the other code is omitted, since it is identical to that of the distributed version. Runs for eight and six passengers are shown. Finally comes the modified Passenger and Vehicle code for the runs with N=1 (i.e., a single passenger per car). This code again is identical for both the distributed and anticipatory versions. Runs for four passengers follow for both versions.

```
this is a CCR solution to the cheap roller coaster problem
using a centralized CCR handler.
resource main_roller
   import Roller_globals
   import roller_server_cent
   import Passenger
   import Vehicle

body main_roller()
   op done()
   var num_of_pass : int
   var num_of_trips : int
   var tot_procs : int

   process start
      writes("Welcome to the roller coaster!")
      write("  Each passenger will get to ride 5 times")
      write("Enter the total number of passengers who will wish to ride.")
      read( num_of_pass)
      num_of_trips := ( num_of_pass * 5 + ( N - 1))/N
      tot_procs := num_of_pass + 1
      var car : cap Vehicle
      var passengers[ 1: num_of_pass] : cap Passenger
      var server : cap roller_server_cent

      create centralized CCR handler
      server := create roller_server_cent()

      create passenger processes
      fa i := 1 to num_of_pass →
         passengers[ i] := create Passenger( server, i, done)
      af

      create the car process
      car := create Vehicle( server, num_of_trips, done)

      wait for the passenger and car processes to complete
      fa i := 1 to tot_procs →
         receive done()
```

```
        af

        destroy the instances of all the processes created
        destroy car

        fa i := 1 to num_of_pass →
            destroy passengers[ i]
        af

        destroy server
    end
end

global Roller_globals
        const N := 4  number passengers per car
end Roller_globals
```

```
This resource implements the car process of the roller coaster
problem using centralized CCRs.
resource Vehicle
   import Roller_globals
   import roller_server_cent

body Vehicle( server: cap roller_server_cent; number_of_trips : int;
           done: cap())

   process car
      fa i := 1 to number_of_trips →
         call server.car_arrived()
         write("Car has arrived at station")
         call server.car_leaving()
         write("Car has filled, doing ride, now unloading")
         call server.car_empty()
      af
      write("car done")
      send done()
   end
end
```

This resource implements the passenger process of the roller coaster *problem* using centralized CCRs. The passenger rides five times.

```
resource Passenger
   import Roller_globals
   import roller_server_cent

body Passenger( server: cap roller_server_cent; process_number : int;
            done: cap() )

   process passenger
      fa i := 1 to 5 →
         call to wait in line and board the car
         call server.get_on()
         write("Passenger ",process_number," is on board")
         call to exit the car
         call server.get_off()
         write("Passenger ",process_number," is off")
      af
      write("passenger ",process_number,"done")
      send done()
   end
end
```

```
This resource implements the centralized CCR server
for the roller coaster problem.
resource roller_server_cent
   import Roller_globals
   op get_on()
   op get_off()
   op car_arrived()
   op car_leaving()
   op car_empty()

body roller_server_cent()

   process worker
      declare vars only usable in CCR
      var pin := 0, pout := 0, vin := 0, vout := 0, vdest := 0
      do true →
         in get_on() and ( pin + 1) ≤ vin * N →
            pin++
         [] get_off() and ( pout + 1) ≤  vout * N →
            do S
            pout++
         [] car_arrived() →
           no guard, execute S
           vin++
         [] car_leaving() and (vout + 1) * N ≤ pin  →
            do S
            vout++
         [] car_empty() and (vdest + 1) * N ≤ pout →
            guard is true, execute S
            vdest++
         ni
      od
   end
end
```

```
Script started on Mon Jun 12 15:35:37 1989
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Car has arrived at station
Passenger  1  is on board
Passenger  2  is on board
Passenger  3  is on board
Passenger  4  is on board
Car has filled, doing ride, now unloading
Passenger  1  is off
Passenger  2  is off
Passenger  3  is off
Passenger  4  is off
Car has arrived at station
Passenger  5  is on board
Passenger  6  is on board
Passenger  7  is on board
Passenger  8  is on board
Car has filled, doing ride, now unloading
Passenger  5  is off
Passenger  6  is off
Passenger  7  is off
Passenger  8  is off
Car has arrived at station
Passenger  1  is on board
Passenger  2  is on board
Passenger  3  is on board
Passenger  4  is on board
Car has filled, doing ride, now unloading
Passenger  1  is off
Passenger  2  is off
Passenger  3  is off
Passenger  4  is off
Car has arrived at station
Passenger  5  is on board
Passenger  6  is on board
Passenger  7  is on board
```

Passenger 8 is on board
Car has filled, doing ride, now unloading
Passenger 5 is off
Passenger 6 is off
Passenger 7 is off
Passenger 8 is off
Car has arrived at station
Passenger 1 is on board
Passenger 2 is on board
Passenger 3 is on board
Passenger 4 is on board
Car has filled, doing ride, now unloading
Passenger 1 is off
Passenger 2 is off
Passenger 3 is off
Passenger 4 is off
Car has arrived at station
Passenger 5 is on board
Passenger 6 is on board
Passenger 7 is on board
Passenger 8 is on board
Car has filled, doing ride, now unloading
Passenger 5 is off
Passenger 6 is off
Passenger 7 is off
Passenger 8 is off
Car has arrived at station
Passenger 1 is on board
Passenger 2 is on board
Passenger 3 is on board
Passenger 4 is on board
Car has filled, doing ride, now unloading
Passenger 1 is off
Passenger 2 is off
Passenger 3 is off
Passenger 4 is off
Car has arrived at station
Passenger 5 is on board
Passenger 6 is on board

```
Passenger  7  is on board
Passenger  8  is on board
Car has filled, doing ride, now unloading
Passenger  5  is off
Passenger  6  is off
Passenger  7  is off
Passenger  8  is off
Car has arrived at station
Passenger  1  is on board
Passenger  2  is on board
Passenger  3  is on board
Passenger  4  is on board
Car has filled, doing ride, now unloading
Passenger  1  is off
Passenger  2  is off
Passenger  3  is off
Passenger  4  is off
passenger  1 done
passenger  2 done
passenger  3 done
passenger  4 done
Car has arrived at station
Passenger  5  is on board
Passenger  6  is on board
Passenger  7  is on board
Passenger  8  is on board
Car has filled, doing ride, now unloading
Passenger  5  is off
Passenger  6  is off
Passenger  7  is off
Passenger  8  is off
passenger  5 done
passenger  6 done
passenger  7 done
passenger  8 done
car done
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
```

6
Car has arrived at station
Passenger 1 is on board
Passenger 2 is on board
Passenger 3 is on board
Passenger 4 is on board
Car has filled, doing ride, now unloading
Passenger 1 is off
Passenger 2 is off
Passenger 3 is off
Passenger 4 is off
Car has arrived at station
Passenger 5 is on board
Passenger 6 is on board
Passenger 1 is on board
Passenger 2 is on board
Car has filled, doing ride, now unloading
Passenger 5 is off
Passenger 6 is off
Passenger 1 is off
Passenger 2 is off
Car has arrived at station
Passenger 3 is on board
Passenger 4 is on board
Passenger 5 is on board
Passenger 6 is on board
Car has filled, doing ride, now unloading
Passenger 3 is off
Passenger 4 is off
Passenger 5 is off
Passenger 6 is off
Car has arrived at station
Passenger 1 is on board
Passenger 2 is on board
Passenger 3 is on board
Passenger 4 is on board
Car has filled, doing ride, now unloading
Passenger 1 is off
Passenger 2 is off

Passenger 3 is off
Passenger 4 is off
Car has arrived at station
Passenger 5 is on board
Passenger 6 is on board
Passenger 1 is on board
Passenger 2 is on board
Car has filled, doing ride, now unloading
Passenger 5 is off
Passenger 6 is off
Passenger 1 is off
Passenger 2 is off
Car has arrived at station
Passenger 3 is on board
Passenger 4 is on board
Passenger 5 is on board
Passenger 6 is on board
Car has filled, doing ride, now unloading
Passenger 3 is off
Passenger 4 is off
Passenger 5 is off
Passenger 6 is off
Car has arrived at station
Passenger 1 is on board
Passenger 2 is on board
Passenger 3 is on board
Passenger 4 is on board
Car has filled, doing ride, now unloading
Passenger 1 is off
Passenger 2 is off
Passenger 3 is off
Passenger 4 is off
passenger 1 done
passenger 2 done
passenger 3 done
passenger 4 done
Car has arrived at station
Passenger 5 is on board
Passenger 6 is on board

RTS warning: 5 blocked processes during termination
ivy% exit
ivy%
script done on Mon Jun 12 15:35:50 1989

```
this is a CCR solution to the cheap roller coaster problem
using a decentralized CCR server.
resource main_roller
   import Roller_globals
   import CCR_server
   import Mutex_server
   import Passenger
   import Vehicle

body main_roller()
   op done()
   var num_of_pass : int
   var num_of_trips : int
   var tot_procs : int

   process start
      writes("Welcome to the roller coaster!")
      write("  Each passenger will get to ride 5 times")
      write("Enter the total number of passengers who will wish to ride.")
      read( num_of_pass)
      num_of_trips := ( num_of_pass * 5 + ( N-1)) / N
      tot_procs := num_of_pass + 1
      var car : cap Vehicle
      var passengers[ 1: num_of_pass] : cap Passenger
      var CCR_servers[ 1 : tot_procs ] : cap CCR_server
      var mutex_servers [ 1 : tot_procs] : cap Mutex_server

      create passenger processes with associated server processes
      fa i := 1 to num_of_pass →
         mutex_servers[ i] := create Mutex_server( i, tot_procs)
         CCR_servers[ i] := create CCR_server( mutex_servers[ i],
                                    i, tot_procs)
         passengers[ i] := create Passenger( CCR_servers[ i], i, done)
      af

      create the car process with associated server processes
      mutex_servers[ tot_procs] := create Mutex_server( tot_procs, tot_procs)
      CCR_servers[ tot_procs] := create CCR_server(
                              mutex_servers[ tot_procs], tot_procs,
```

```
                              tot_procs)
    car := create Vehicle( CCR_servers[ tot_procs], num_of_trips, done)

    send links to server processes
    fa i := 1 to tot_procs →
        send mutex_servers[ i].links( mutex_servers)
        send CCR_servers[ i].links( CCR_servers)
    af

    wait for the passenger and car processes to complete
    fa i := 1 to tot_procs →
        receive done()
    af

    destroy the instances of all the processes created
    destroy car

    fa i := 1 to num_of_pass →
        destroy mutex_servers[ i]
        destroy CCR_servers[ i]
        destroy passengers[ i]
    af

    destroy mutex_servers[ tot_procs]
    destroy CCR_servers[ tot_procs]
  end
end


global Roller_globals
    op type for CCRs to share the CCR variables
    optype CCR_Pass_globals = ( pin, pout, bin, bout, bdest : int)
    const N := 4  number passengers per car
end Roller_globals
```

```
This resource implements the car process of the roller coaster
problem using CCRs.
resource Vehicle
   import CCR_server

body Vehicle( server: cap CCR_server; number_of_trips : int; done: cap())

   process car
      fa i := 1 to number_of_trips →
         call server.car_arrived()
         call server.car_leaving()
         call server.car_empty()
      af
      write("Car done")
      send done()
   end
end



This resource implements the passenger process of the roller coaster
problem using distributed CCRs.  The passenger rides five times.
resource Passenger
   import CCR_server

body Passenger( server: cap CCR_server; CCR_server_number : int; done: cap())

   process passenger
      fa i := 1 to 5 →
         call to get in car
         call server.get_on()
         call to debark
         call server.get_off()
      af
      write("passenger ",CCR_server_number,"done")
      send done()
   end
end
```

```
This resource implements the region statement handler for the
cheap roller coaster problem using distributed CCRs.
resource CCR_server
  import Mutex_server
  import Roller_globals
  op links( others[ 1:* ] : cap CCR_server)
  op Pass_globals CCR_Pass_globals
  op get_on()
  op get_off()
  op car_arrived()
  op car_leaving()
  op car_empty()

body CCR_server( server: cap Mutex_server; CCR_server_number,
          total_processes: int)

  var others[ 1: total_processes] : cap CCR_server
  var num_retries := 0
  var num_successful := 0

  process worker
    get capabilities for all the other CCR servers
    receive links( others)
    declare vars only usable in CCR
    var pin := 0, pout := 0, vin := 0, vout := 0, vdest := 0
    do true →
      var need_new_data : bool := false
      in get_on() →
        do true →
          if (need_new_data) →
            receive Pass_globals( pin, pout, vin, vout, vdest)
            need_new_data := false
          fi
          get mutual exclusion
          call server.get_exclusion()
          get the most up to date region variable values
          do (?Pass_globals ≠ 0) →
            receive Pass_globals( pin, pout, vin, vout, vdest)
          od
```

```
        need_new_data := false
        if ( pin + 1) ≤ vin * N →
          guard is true exit loop
          exit
        [] else →
          guard not true must give up mutex
          need_new_data := true
          num_retries++
          server.release( false)
        fi
      od
      do S
      pin++
      write("passenger #",CCR_server_number," is on the car")
      num_successful++
      notify all other CCR servers of updated variables
      fa i := 1 to total_processes st i ≠ CCR_server_number →
      fa i := total_processes downto 1 st i ≠ CCR_server_number →
        send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      af
      server.release( true)
    [] get_off() →
      do true →
        if (need_new_data) →
          receive Pass_globals( pin, pout, vin, vout, vdest)
          need_new_data := false
        fi
        get mutual exclusion
        call server.get_exclusion()
        get the must up to date region variable values
        do (?Pass_globals ≠ 0)  →
          receive Pass_globals( pin, pout, vin, vout, vdest)
        od
        need_new_data := false
        if ( pout + 1 ≤ vout * N) →
          # guard is true exit loop
          exit
        [] else →
          guard not true must give up mutex
```

```
        need_new_data := true
        num_retries++
        server.release( false)
      fi
  od
  do S
  pout++
  num_successful++
  notify all other CCR servers of updated variables
  fa i := 1 to total_processes st i ≠ CCR_server_number →
  fa i := total_processes downto 1 st i ≠ CCR_server_number →
    send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
  af
  write("passenger #",CCR_server_number,"exiting car")
  server.release( true)
[] car_arrived() →
  get mutual exclusion
  call server.get_exclusion()
  get must update values of region variables
  do ( ?Pass_globals ≠ 0) →
    receive Pass_globals( pin, pout, vin, vout, vdest)
  od
  no guard, execute S
  vin++
  num_successful++
  notify all other CCR servers of updated variables
  fa i := 1 to total_processes st i ≠ CCR_server_number →
   fa i := total_processes downto 1 st i ≠ CCR_server_number →
    send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
  af
  write("Car has arrived, try to leave")
  call server.release( true)
[] car_leaving() →
  do true →
    if (need_new_data) →
      receive Pass_globals( pin, pout, vin, vout, vdest)
      need_new_data := false
    fi
    get mutual exclusion
```

```
    call server.get_exclusion()
    get must update values of region variables
    do ( ?Pass_globals ≠ 0)  →
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
    need_new_data := false
    if (vout + 1) * N ≤ pin  →
      # guard is true exit loop
      exit
    [] else →
      guard not true, must give up mutex
      need_new_data := true
      num_retries++
      call server.release( false)
    fi
  od
  guard is true, execute S
  vout++
  num_successful++
  notify all other CCR servers of updated variables
  fa i := 1 to total_processes st i ≠ CCR_server_number →
  fa i := total_processes downto 1 st i ≠ CCR_server_number →
    send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
  af
  write("Car is leaving")
  server.release( true)
[] car_empty() →
  do true →
    if (need_new_data) →
      receive Pass_globals( pin, pout, vin, vout, vdest)
      need_new_data := false
    fi
    get mutual exclusion
    call server.get_exclusion()
    get must update values of region variables
    do ( ?Pass_globals ≠ 0) →
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
    need_new_data := false
```

```
            if (vdest + 1) * N ≤ pout →
              # guard is true exit loop
              exit
            [] else →
              guard not true, must give up mutex
              need_new_data := true
              num_retries++
              call server.release( false)
            fi
          od
          guard is true, execute S
          vdest++
          num_successful++
          notify all other CCR servers of updated variables
          fa i := total_processes downto 1 st i ≠ CCR_server_number →
            send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
          af
          write("Car is now empty")
          call server.release( true)
        ni
      od
    end

    final
      write("For process",CCR_server_number,"CCR handler there were")
      writes(num_successful," successful invocations")
      write(" and ",num_retries," retries")
    end
  end
```

```
This resource is a server which is associated with a node and
implements the Suzuki and Kasami distributed mutual exclusion
algorithm from the 85 ACM paper.
resource Mutex_server
   op invoke()
   op release( success : bool)
   op privilege( Q[0:*], front, rear, count, LN[ 1 : *],
                 failed_list[ 1 : *], failed_list_length,
                 failed_list_ptr : int)
   op request( from, its_seq_num:int)
   op links( others[1:*]: cap Mutex_server)
   op get_exclusion() routine called by node to get mut. excl.

body Mutex_server( me, total_processes : int)
   var RN[ 1 : total_processes] :int := ([total_processes]0)
   var LN[ 1 : total_processes] :int := ([total_processes]0)
   var HavePrivilege : bool := false
   var Q [ 0 : total_processes-1] : int
   var front := 0, rear := 0, count := 0
   var failed_list[ 0 : total_processes-1] : int
   var failed_list_length := 0, failed_list_ptr := 0
   var failed_last_try : bool := false

   op have_crit()

   var obtain_msgs := 0, num_releases := 0

   proc get_exclusion()
      send invoke()
      receive have_crit()
      obtain_msgs++
   end

   process server
      var other_servers[ 1 : total_processes]: cap Mutex_server
      receive links( other_servers)
      if me = 1 →
         if we are node 1's server, then we initially have the privilege
         HavePrivilege := true
```

```
fi
do true →
   in invoke() →
      if not HavePrivilege →
        if not failed_last_try →
          new request, need to broadcast
          RN[ me]++
          fa j := 1 to total_processes st  j ≠ me →
             send other_servers[ j].request( me, RN[ me])
             obtain_msgs++
          af
        fi
        wait for privilege
        receive privilege( Q, front, rear, count, LN, failed_list,
                      failed_list_length, failed_list_ptr)
        HavePrivilege := true
        obtain_msgs++
      fi
      send have_crit()
      do true →
        in request( from, its_seq_num) →
          update request number for requesting node
          RN[ from] := max( RN[ from], its_seq_num)
        [] release( success) →
          node is releasing mutual exclusion
          num_releases++
          if success →
            if failed_last_try →
              remove self from failed_list
              fa i := failed_list_ptr to failed_list_length - 2 →
                failed_list[ i] := failed_list[ i + 1]
              af
              failed_list_length--
              failed_last_try := false
            fi
            failed_list_ptr := 0
          [] not success →
            if failed_last_try →
              increment failed list pointer
```

```
            failed_list_ptr := min( failed_list_ptr + 1,
                                    failed_list_length)
          [] else →
            add self to failed_list
            failed_list[ failed_list_length] := me
            failed_list_length++
            failed_list_ptr := failed_list_length
            failed_last_try := true
          fi
        fi
        exit
      ni
    od
    mark our request as having been satisfied
    LN[ me] := RN[ me]
    add request to the unsatisfied request queue if it is
    new and is not already on the queue
    fa j := 1 to total_processes st j ≠ me →
      if RN[ j] = (LN[j] +1) →
        var found : bool := false
        var search_count : int := 0
        var search_ptr := front
        do search_count < count →
          if Q[ search_ptr] = j →
            found := true
            exit
          fi
          search_count++
          search_ptr := (search_ptr + 1) % total_processes
        od
        if not found →
          Q[ rear] := j
          rear := (rear + 1) % total_processes
          count++
        fi
      fi
    af
    if failed_list_length > 0 and
      failed_list_ptr < failed_list_length →
```

```
            give privilege to next entry on failed list
            HavePrivilege := false
            send other_servers[ failed_list[ failed_list_ptr.privilege(
                        Q, front, rear, count, LN, failed_list,
                        failed_list_length, failed_list_ptr)
          [] else →
            no failed process currently waiting for privilege
            pass on to new request, if there is one
            if count ≠ 0 →
              HavePrivilege := false
              send other_servers[ Q[ front.privilege( Q,
                            (front+1)total_processes,
                            rear, count-1, LN, failed_list,
                            failed_list_length,
                            failed_list_ptr)
            fi
          fi
        [] request( from, its_seq_num) →
          pass on privilege to this request if it is new and
          we currently hold the privilege
          RN[ from] := max( RN[ from], its_seq_num)
          if HavePrivilege and ( RN[ from] = LN[ from] + 1) →
            HavePrivilege := false
            send other_servers[ from].privilege( Q, front, rear, count,
                                LN, failed_list,
                                failed_list_length,
                                failed_list_ptr)
          fi
      ni
    od
  end

  final
    write("For process",me," mutex handler there were ")
    writes(obtain_msgs," messages to get mutex and ")
    write(num_releases," messages to release mutex")
  end
end
```

```
Script started on Thu Jun 29 08:45:51 1989
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 7  is on the car
passenger # 8  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 7 exiting car
passenger # 8 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
```

```
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 7  is on the car
passenger # 8  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 7 exiting car
passenger # 8 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 7  is on the car
passenger # 8  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 7 exiting car
passenger # 8 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
```

passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 5 is on the car
passenger # 6 is on the car
passenger # 7 is on the car
passenger # 8 is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 7 exiting car
passenger # 8 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1 is on the car
passenger # 2 is on the car
passenger # 3 is on the car
passenger # 4 is on the car
Car is leaving
passenger # 1 exiting car
passenger 1 done
passenger # 2 exiting car
passenger 2 done
passenger # 3 exiting car
passenger 3 done
passenger # 4 exiting car
passenger 4 done
Car is now empty
Car has arrived, try to leave
passenger # 5 is on the car
passenger # 6 is on the car
passenger # 7 is on the car
passenger # 8 is on the car
Car is leaving
passenger # 5 exiting car
passenger 5 done
passenger # 6 exiting car

passenger 6 done
passenger # 7 exiting car
passenger 7 done
passenger # 8 exiting car
passenger 8 done
Car is now empty
Car done
For process 1 mutex handler there were
157 messages to get mutex and 43 messages to release mutex
For process 1 CCR handler there were
10 successful invocations and 33 retries
For process 2 mutex handler there were
166 messages to get mutex and 43 messages to release mutex
For process 2 CCR handler there were
10 successful invocations and 33 retries
For process 3 mutex handler there were
166 messages to get mutex and 43 messages to release mutex
For process 3 CCR handler there were
10 successful invocations and 33 retries
For process 4 mutex handler there were
166 messages to get mutex and 43 messages to release mutex
For process 4 CCR handler there were
10 successful invocations and 33 retries
For process 5 mutex handler there were
180 messages to get mutex and 50 messages to release mutex
For process 5 CCR handler there were
10 successful invocations and 40 retries
For process 6 mutex handler there were
180 messages to get mutex and 50 messages to release mutex
For process 6 CCR handler there were
10 successful invocations and 40 retries
For process 7 mutex handler there were
180 messages to get mutex and 50 messages to release mutex
For process 7 CCR handler there were
10 successful invocations and 40 retries
For process 8 mutex handler there were
180 messages to get mutex and 50 messages to release mutex
For process 8 CCR handler there were
10 successful invocations and 40 retries

```
For process 9  mutex handler there were
300 messages to get mutex and 30  messages to release mutex
For process 9 CCR handler there were
30 successful invocations and  0  retries
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
6
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 1  is on the car
passenger # 2  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 1 exiting car
passenger # 2 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 3  is on the car
passenger # 4  is on the car
passenger # 5  is on the car
passenger # 6  is on the car
Car is leaving
passenger # 3 exiting car
passenger # 4 exiting car
passenger # 5 exiting car
```

```
passenger # 6 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 1  is on the car
passenger # 2  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 1 exiting car
passenger # 2 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 3  is on the car
passenger # 4  is on the car
passenger # 5  is on the car
passenger # 6  is on the car
Car is leaving
passenger # 3 exiting car
passenger # 4 exiting car
passenger # 5 exiting car
passenger # 6 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
```

```
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger  1 done
passenger # 2 exiting car
passenger  2 done
passenger # 3 exiting car
passenger  3 done
passenger # 4 exiting car
passenger  4 done
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
RTS warning: 18 blocked processes during termination
ivy% exit
ivy%
script done on Thu Jun 29 08:46:20 1989
```

The following is the distributed CCR handler with anticipatory execution and modified distributed mutex algorithm.
The code for the resources main_roller, Passenger and Vehicle are omitted, since the ones used for this version are identical to the distributed version shown above.
This resource implements the region statement handler for the *cheap* roller coaster problem using distributed CCRs with anticipatory *execution.*

```
resource CCR_server
   import Mutex_server
   import Roller_globals
   op links( others[ 1:* ] : cap CCR_server)
   op Pass_globals CCR_Pass_globals
   op get_on( )
   op get_off( )
   op car_arrived()
   op car_leaving()
   op car_empty()

body CCR_server( server: cap Mutex_server; CCR_server_number,
            total_processes: int)

   var others[ 1 : total_processes] : cap CCR_server
   var num_roll_backs := 0, check_cnt := 0
   var num_successful := 0

   process worker
      receive links( others)
      op got_excl( success : bool)
      op check( var need_priv : bool)
      send server.servee_ops( check, got_excl)
      declare vars only usable in CCR
      var pin := 0, pout := 0, vin := 0, vout := 0, vdest := 0
      do true →
         in get_on() →
            do true →
               do true →
                  if ( pin + 1) ≤ vin * N and (?Pass_globals = 0) →
                     guard is true exit loop
```

```
    exit
  [] else →
    need to wait for updated data
    in Pass_globals( temp_pin, temp_pout, temp_vin,
                    temp_vout, temp_vdest) →
      pin := temp_pin
      pout := temp_pout
      vin := temp_vin
      vout := temp_vout
      vdest := temp_vdest
     [] check( need_priv) and ?Pass_globals = 0 →
      need_priv := false
      check_cnt++
    ni
  fi
od
guard is true, request mutual exclusion
var have_excl : bool
var save_pin := pin
send server.get_exclusion()
do S as if we have exclusion
pin++
in got_excl( temp_have_excl) →
  have_excl := temp_have_excl
 [] check( need_priv) →
  need_priv := true
  reply
  receive got_excl( have_excl)
  check_cnt++
ni
if not have_excl or ?Pass_globals ≠ 0 →
   a CCR server has exited region since we
   tested guard or we didn't get exclusion
   if have_excl →
     need to release exclusion
     server.release( false)
   fi
   need to rollback to pre-execution values
   pin := save_pin
```

```
        num_roll_backs++
        next
      fi
      num_successful++
      notify all other CCR servers of updated variables
      fa i := 1 to total_processes st i ≠ CCR_server_number →
        send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      af
      write("passenger #",CCR_server_number," is on the car")
      server.release( true)
      exit
    od
[] get_off( ) →
    do true →
      do true →
        if ( pout + 1 ≤ vout * N) and ( ?Pass_globals = 0) →
          # guard is true exit loop
          exit
        [] else →
          # need to wait for updated data
          in Pass_globals( temp_pin, temp_pout, temp_vin,
                           temp_vout, temp_vdest) →
            pin := temp_pin
            pout := temp_pout
            vin := temp_vin
            vout := temp_vout
            vdest := temp_vdest
          [] check( need_priv) and ?Pass_globals = 0 →
            need_priv := false
            check_cnt++
          ni
        fi
      od
      guard is true, request mutual exclusion
      var have_excl : bool
      var save_pout := pout
      send server.get_exclusion()
      execute S as if we have exclusion
      pout++
```

```
      in got_excl( temp_have_excl) →
        have_excl := temp_have_excl
      [] check( need_priv) →
        need_priv := true
        reply
        receive got_excl( have_excl)
        check_cnt++
      ni
      if not have_excl or ?Pass_globals ≠ 0 →
        a CCR server has exited region since we
        tested guard or didn't get mutual exclusion
        if have_excl →
          need to release mutual exclusion
          server.release( false)
        fi
        need to rollback to pre-execution values
        pout := save_pout
        num_roll_backs++
        next
      fi
      num_successful++
      notify all other CCR servers of updated variables
      fa i := 1 to total_processes st i ≠ CCR_server_number →
        send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      af
      write("passenger #",CCR_server_number,"exiting car")
      server.release( true)
      exit
    od
  [] car_arrived() →
    do true →
      do true →
        if ( ?Pass_globals = 0) →
          # guard is true exit loop
          exit
         [] else →
          # need to wait for updated data
          in Pass_globals( temp_pin, temp_pout, temp_vin,
                      temp_vout, temp_vdest) →
```

```
          pin := temp_pin
          pout := temp_pout
          vin := temp_vin
          vout := temp_vout
          vdest := temp_vdest
        [] check( need_priv) and ?Pass_globals = 0 →
          need_priv := false
          check_cnt++
        ni
    fi
od
guard is true, request mutual exclusion
var have_excl : bool
var save_vin := vin
send server.get_exclusion()
execute S as if we have exclusion
vin++
in got_excl( temp_have_excl) →
   have_excl := temp_have_excl
[] check( need_priv) →
   need_priv := true
   reply
   receive got_excl( have_excl)
   check_cnt++
ni
if not have_excl or ?Pass_globals ≠ 0 →
    a CCR server has exited region since we
    tested guard or didn't get exclusion
    if have_excl →
        server.release( false)
    fi
    need to rollback to pre-execution values
    vin := save_vin
    num_roll_backs++
    next
fi
num_successful++
notify all other CCR servers of updated variables
fa i := 1 to total_processes st i ≠ CCR_server_number →
```

```
        send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      af
      write("car has arrived, try to leave")
      server.release( true)
      exit
    od
[] car_leaving() →
    do true →
      do true →
        if ( (vout + 1) * N ≤ pin ) and ( ?Pass_globals = 0) →
          # guard is true exit loop
          exit
        [] else →
          # need to wait for updated data
          in Pass_globals( temp_pin, temp_pout, temp_vin,
                          temp_vout, temp_vdest) →
            pin := temp_pin
            pout := temp_pout
            vin := temp_vin
            vout := temp_vout
            vdest := temp_vdest
          [] check( need_priv) and ?Pass_globals = 0 →
            need_priv := false
            check_cnt++
          ni
        fi
      od
      guard is true, request mutual exclusion
      var have_excl : bool
      var save_vout := vout
      send server.get_exclusion()
      execute S as if we have exclusion
      vout++
      in got_excl( temp_have_excl) →
        have_excl := temp_have_excl
      [] check( need_priv) →
        need_priv := true
        reply
        receive got_excl( have_excl)
```

```
            check_cnt++
        ni
        if not have_excl or ?Pass_globals ≠ 0 →
            a CCR server has exited region since we
            tested guard or didn’t get exclusion
            if have_excl →
                server.release( false)
            fi
            need to rollback to pre-execution values
            vout := save_vout
            num_roll_backs++
            next
        fi
        num_successful++
        notify all other CCR servers of updated variables
        fa i := 1 to total_processes st i ≠ CCR_server_number →
            send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
        af
        write("car is leaving")
        server.release( true)
        exit
    od
[] car_empty() →
    do true →
        do true →
            if ( (vdest + 1) * N ≤ pout ) and ( ?Pass_globals = 0) →
                # guard is true exit loop
                exit
            [] else →
              # need to wait for updated data
              in Pass_globals( temp_pin, temp_pout, temp_vin,
                               temp_vout, temp_vdest) →
                pin := temp_pin
                pout := temp_pout
                vin := temp_vin
                vout := temp_vout
                vdest := temp_vdest
              [] check( need_priv) and ?Pass_globals = 0 →
                need_priv := false
```

```
            check_cnt++
          ni
        fi

    od
    guard is true, request mutual exclusion
    var have_excl : bool
    var save_vdest := vdest
    send server.get_exclusion()
    execute S as if we have exclusion
    vdest++
    in got_excl( temp_have_excl) →
      have_excl := temp_have_excl
    [] check( need_priv) →
      need_priv := true
      reply
      receive got_excl( have_excl)
      check_cnt++
    ni
    if not have_excl or ?Pass_globals ≠ 0 →
        a CCR server has exited region since we
        tested guard or didn't get exclusion
        if have_excl →
            server.release( false)
        fi
        need to rollback to pre-execution values
        vdest := save_vdest
        num_roll_backs++
        next
    fi
    num_successful++
    notify all other CCR servers of updated variables
    fa i := 1 to total_processes st i ≠ CCR_server_number →
        send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
    af
    write("car is now empty")
    server.release( true)
    exit
od
```

```
        ni
      od
    end

    final
      write("For process",CCR_server_number,"CCR handler there were ")
      writes(num_successful," successful invocations")
      writes(" and ",num_roll_backs," roll_backs")
      write(" and ",check_cnt," check msgs")
    end
end
```

This resource is a server which is associated with a node and *implements* the Suzuki and Kasami mutual exclusion algorithm *from* the 85 ACM paper, this version explicitly defers delayed *requests.* This version has been modified for use with the *anticipatory* implementation of CCRs.

```
resource Mutex_server
  op invoke()
  op release( success : bool)
  op privilege( LN[ 1 : *], failed_list[ 1 : *], failed_list_length,
            failed_list_ptr : int)
  op priv_denied()
  op request( from, its_seq_num:int)
  op links( others[1:*]: cap Mutex_server)
  op get_exclusion()        routine called by node to get mut. excl.
  op servee_ops( check : cap ( var need_priv : bool);
            return_op : cap (success: bool))
body Mutex_server( me, total_processes : int)
  var my_request_number := 0
  var LN[ 1 : total_processes] :int := ([total_processes]0)
  var HavePrivilege : bool := false
  var failed_list[ 0 : total_processes-1] : int
  var failed_list_length := 0, failed_list_ptr := 0
  var failed_last_try : bool := false
  var deferQ[ 0 : total_processes-1] : int := ([total_processes]0)
  var deferQ_length := 0
  var need_priv : bool := false
  var other_servers[ 1 : total_processes]: cap Mutex_server
  var check : cap ( var success : bool)
  var return_result : cap (success: bool)

  op have_crit()
  op defered()

  var obtain_msgs := 0, num_releases := 0

  initial
    reply
    receive links( other_servers)
    receive servee_ops( check, return_result)
```

```
end

proc get_exclusion()
    var success : bool
    send invoke()
    in have_crit() →
      successfully acquired mutual exclusion
      success := true
    [] defered() →
      did not get mutual exclusion
      success := false
    ni
    send return_result( success)
    obtain_msgs := obtain_msgs + 2
end

process server
    if me = 1 →
      if we are node 1's server, then we initially have the privilege
      HavePrivilege := true
    fi
    do true →
       in invoke() →
          need_priv := false
          if not HavePrivilege →
            if not failed_last_try →
              my_request_number++
              fa j := 1 to total_processes st  j ≠ me →
                send other_servers[ j].request( me, my_request_number)
                obtain_msgs++
              af
            fi
            wait for privilege
            in privilege( temp_LN, temp_failed_list,
                      temp_failed_list_length,
                      temp_failed_list_ptr) →
              LN := temp_LN
              failed_list := temp_failed_list
              failed_list_length := temp_failed_list_length
```

```
        failed_list_ptr := temp_failed_list_ptr
        obtain_msgs++

      [] priv_denied() →
        send defered()
        failed_last_try := true
        obtain_msgs++
        next
    ni
    HavePrivilege := true
fi
var temp_bool : bool := true
if failed_last_try and
    failed_list_length = failed_list_ptr and
    failed_list_length > 1 →
    #
    send defered()
    failed_list_ptr := 0
    temp_bool := false
[] else →
    send have_crit()
fi
do temp_bool →
    in request( from, its_seq_num) →
        check if this is a new request
        if its_seq_num = LN[ from] + 1 →
            new request, deny it
            send other_servers[ from].priv_denied()
            LN[ from] := its_seq_num
            deferQ[ deferQ_length] := from
            deferQ_length++
        fi
    [] release( success) →
        node is releasing mutual exclusion
        num_releases++
        if success →
            if failed_last_try →
            remove self from failed_list
                fa i := failed_list_ptr to failed_list_length - 2 →
```

```
            failed_list[ i] := failed_list[ i + 1]
          af
          failed_list_length–
          failed_last_try := false
        [] else →
        fi
        failed_list_ptr := 0
      [] not success →
        if failed_last_try →
          failed_list_ptr := min( failed_list_ptr + 1,
                                  failed_list_length)
        [] else →
          add self to failed_list
          failed_list[ failed_list_length] := me
          failed_list_length++
          failed_list_ptr := failed_list_length
          failed_last_try := true
        fi
      fi
      exit
    ni
  od
  LN[ me] := my_request_number
  add defered processes to failed_list
  fa i := 1 to deferQ_length →
    failed_list[ failed_list_length] := deferQ[ i-1]
    failed_list_length++
  af
  deferQ_length := 0
  if failed_list_length > 0 and
    failed_list_ptr < failed_list_length →
    HavePrivilege := false
    send other_servers[ failed_list[ failed_list_ptr.privilege(
                 LN, failed_list, failed_list_length,
                 failed_list_ptr)
  fi
[] request( from, its_seq_num) and HavePrivilege and not need_priv →
  check if this is a new request
  if its_seq_num = LN[ from] + 1 →
```

```
            new request, pass privilege on
            HavePrivilege := false
            send other_servers[ from].privilege( LN, failed_list,
                                               failed_list_length,
                                               failed_list_ptr)
           fi
        [] privilege( temp_LN, temp_failed_list, temp_failed_list_length,
                 temp_failed_list_ptr) →
          HavePrivilege := true
          LN := temp_LN
          failed_list := temp_failed_list
          failed_list_length := temp_failed_list_length
          failed_list_ptr := temp_failed_list_ptr
          call check( need_priv)
          if not need_priv →
            failed_list_ptr := min( failed_list_ptr + 1,
                                failed_list_length)
            if failed_list_length > 0 and
              failed_list_ptr < failed_list_length →
              HavePrivilege := false
              send other_servers[ failed_list[failed_list_ptr.privilege(                LN, failed_list, failed_list_length,
                         failed_list_ptr)
            fi
          fi
        ni
      od
    end

    final
      write("For process",me," mutex handler there were ")
      writes(obtain_msgs," messages to get mutex and ")
      write(num_releases," messages to release mutex")
    end
  end
```

```
Script started on Thu Jun 29 08:45:51 1989
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 7  is on the car
passenger # 8  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 7 exiting car
passenger # 8 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
```

```
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 7  is on the car
passenger # 8  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 7 exiting car
passenger # 8 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 7  is on the car
passenger # 8  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 7 exiting car
passenger # 8 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
```

```
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 7  is on the car
passenger # 8  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 7 exiting car
passenger # 8 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger  1 done
passenger # 2 exiting car
passenger  2 done
passenger # 3 exiting car
passenger  3 done
passenger # 4 exiting car
passenger  4 done
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 7  is on the car
passenger # 8  is on the car
Car is leaving
passenger # 5 exiting car
passenger  5 done
passenger # 6 exiting car
```

```
passenger  6 done
passenger # 7 exiting car
passenger  7 done
passenger # 8 exiting car
passenger  8 done
Car is now empty
Car done
For process 1  mutex handler there were
157 messages to get mutex and 43  messages to release mutex
For process 1 CCR handler there were
10 successful invocations and  33  retries
For process 2  mutex handler there were
166 messages to get mutex and 43  messages to release mutex
For process 2 CCR handler there were
10 successful invocations and  33  retries
For process 3  mutex handler there were
166 messages to get mutex and 43  messages to release mutex
For process 3 CCR handler there were
10 successful invocations and  33  retries
For process 4  mutex handler there were
166 messages to get mutex and 43  messages to release mutex
For process 4 CCR handler there were
10 successful invocations and  33  retries
For process 5  mutex handler there were
180 messages to get mutex and 50  messages to release mutex
For process 5 CCR handler there were
10 successful invocations and  40  retries
For process 6  mutex handler there were
180 messages to get mutex and 50  messages to release mutex
For process 6 CCR handler there were
10 successful invocations and  40  retries
For process 7  mutex handler there were
180 messages to get mutex and 50  messages to release mutex
For process 7 CCR handler there were
10 successful invocations and  40  retries
For process 8  mutex handler there were
180 messages to get mutex and 50  messages to release mutex
For process 8 CCR handler there were
10 successful invocations and  40  retries
```

```
For process 9  mutex handler there were
300 messages to get mutex and 30  messages to release mutex
For process 9 CCR handler there were
30 successful invocations and  0  retries
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
6
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 1  is on the car
passenger # 2  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 1 exiting car
passenger # 2 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 3  is on the car
passenger # 4  is on the car
passenger # 5  is on the car
passenger # 6  is on the car
Car is leaving
passenger # 3 exiting car
passenger # 4 exiting car
passenger # 5 exiting car
```

```
passenger # 6 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger # 2 exiting car
passenger # 3 exiting car
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
passenger # 1  is on the car
passenger # 2  is on the car
Car is leaving
passenger # 5 exiting car
passenger # 6 exiting car
passenger # 1 exiting car
passenger # 2 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 3  is on the car
passenger # 4  is on the car
passenger # 5  is on the car
passenger # 6  is on the car
Car is leaving
passenger # 3 exiting car
passenger # 4 exiting car
passenger # 5 exiting car
passenger # 6 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
```

```
passenger # 4  is on the car
Car is leaving
passenger # 1 exiting car
passenger  1 done
passenger # 2 exiting car
passenger  2 done
passenger # 3 exiting car
passenger  3 done
passenger # 4 exiting car
passenger  4 done
Car is now empty
Car has arrived, try to leave
passenger # 5  is on the car
passenger # 6  is on the car
RTS warning: 18 blocked processes during termination
ivy% exit
ivy%
script done on Thu Jun 29 08:46:20 1989
```

The following are the Vehicle and Passenger resources modified to allow only a single passenger per car, N=1 in Roller_globals, and to include delays between calls to execute region statements.

```
This resource implements the car process of the roller coaster
problem using CCRs.
resource Vehicle
   import CCR_server

body Vehicle( server: cap CCR_server; number_of_trips : int; done: cap())

   process car
      fa i := 1 to number_of_trips →
         call server.car_arrived()
         nap(50)
         call server.car_leaving()
         call server.car_empty()
      af
      write("Car done")
      send done()
   end
end
```

```
This resource implements the passenger process of the roller coaster
problem using distributed CCRs.  The passenger rides five times.
resource Passenger
   import CCR_server

body Passenger( server: cap CCR_server; CCR_server_number : int; done: cap())

   process passenger
      fa i := 1 to 5 →
         call to get in car
         call server.get_on()
         nap(  200 )
         call to debark
         call server.get_off()
         nap( 100)
      af
      write("passenger ",CCR_server_number,"done")
      send done()
   end
end
```

The following is the run with the code Passenger and Vehicle modified above using the distributed CCR server.

```
Script started on Thu Jun 29 13:46:34 1989
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
4
Car has arrived, try to leave
passenger # 1  is on the car
Car is leaving
passenger # 1 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 2  is on the car
Car is leaving
passenger # 2 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 3  is on the car
Car is leaving
passenger # 3 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 4  is on the car
Car is leaving
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
Car is leaving
passenger # 1 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 2  is on the car
Car is leaving
passenger # 2 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 3  is on the car
```

```
Car is leaving
passenger # 3 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 4  is on the car
Car is leaving
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
Car is leaving
passenger # 1 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 2  is on the car
Car is leaving
passenger # 2 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 3  is on the car
Car is leaving
passenger # 3 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 4  is on the car
Car is leaving
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1  is on the car
Car is leaving
passenger # 1 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 2  is on the car
Car is leaving
passenger # 2 exiting car
Car is now empty
Car has arrived, try to leave
```

passenger # 3 is on the car
Car is leaving
passenger # 3 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 4 is on the car
Car is leaving
passenger # 4 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 1 is on the car
Car is leaving
passenger # 1 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 2 is on the car
Car is leaving
passenger 1 done
passenger # 2 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 3 is on the car
Car is leaving
passenger 2 done
passenger # 3 exiting car
Car is now empty
Car has arrived, try to leave
passenger # 4 is on the car
Car is leaving
passenger 3 done
passenger # 4 exiting car
Car is now empty
Car done
passenger 4 done
For process 1 mutex handler there were
145 messages to get mutex and 55 messages to release mutex
For process 1 CCR handler there were
10 successful invocations and 45 retries
For process 2 mutex handler there were

160 messages to get mutex and 60 messages to release mutex
For process 2 CCR handler there were
10 successful invocations and 50 retries
For process 3 mutex handler there were
168 messages to get mutex and 64 messages to release mutex
For process 3 CCR handler there were
10 successful invocations and 54 retries
For process 4 mutex handler there were
174 messages to get mutex and 67 messages to release mutex
For process 4 CCR handler there were
10 successful invocations and 57 retries
For process 5 mutex handler there were
395 messages to get mutex and 80 messages to release mutex
For process 5 CCR handler there were
60 successful invocations and 20 retries
ivy% exit
ivy%
script done on Thu Jun 29 13:46:51 1989

The following is the run with the code Passenger and Vehicle modified above using the distributed CCR server with anticipatory execution.

```
Script started on Thu Jun 29 13:47:06 1989
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
4
car has arrived, try to leave
passenger # 1  is on the car
car is leaving
passenger # 1 exiting car
car is now empty
car has arrived, try to leave
passenger # 2  is on the car
car is leaving
passenger # 2 exiting car
car is now empty
car has arrived, try to leave
passenger # 3  is on the car
car is leaving
passenger # 3 exiting car
car is now empty
car has arrived, try to leave
passenger # 4  is on the car
car is leaving
passenger # 4 exiting car
car is now empty
car has arrived, try to leave
passenger # 1  is on the car
car is leaving
passenger # 1 exiting car
car is now empty
car has arrived, try to leave
passenger # 2  is on the car
car is leaving
passenger # 2 exiting car
car is now empty
car has arrived, try to leave
passenger # 3  is on the car
```

```
car is leaving
passenger # 3 exiting car
car is now empty
car has arrived, try to leave
passenger # 4  is on the car
car is leaving
passenger # 4 exiting car
car is now empty
car has arrived, try to leave
passenger # 1  is on the car
car is leaving
passenger # 1 exiting car
car is now empty
car has arrived, try to leave
passenger # 2  is on the car
car is leaving
passenger # 2 exiting car
car is now empty
car has arrived, try to leave
passenger # 3  is on the car
car is leaving
passenger # 3 exiting car
car is now empty
car has arrived, try to leave
passenger # 4  is on the car
car is leaving
passenger # 4 exiting car
car is now empty
car has arrived, try to leave
passenger # 1  is on the car
car is leaving
passenger # 1 exiting car
car is now empty
car has arrived, try to leave
passenger # 2  is on the car
car is leaving
passenger # 2 exiting car
car is now empty
car has arrived, try to leave
```

passenger # 3 is on the car
car is leaving
passenger # 3 exiting car
car is now empty
car has arrived, try to leave
passenger # 4 is on the car
car is leaving
passenger # 4 exiting car
car is now empty
car has arrived, try to leave
passenger # 1 is on the car
car is leaving
passenger # 1 exiting car
car is now empty
car has arrived, try to leave
passenger # 2 is on the car
car is leaving
passenger 1 done
passenger # 2 exiting car
car is now empty
car has arrived, try to leave
passenger # 3 is on the car
passenger 2 done
car is leaving
passenger # 3 exiting car
car is now empty
car has arrived, try to leave
passenger # 4 is on the car
car is leaving
passenger 3 done
passenger # 4 exiting car
car is now empty
Car done
passenger 4 done
For process 1 mutex handler there were
109 messages to get mutex and 18 messages to release mutex
For process 1 CCR handler there were
10 successful invocations and 13 roll_backs and 24 check msgs
For process 2 mutex handler there were

```
112 messages to get mutex and 19  messages to release mutex
For process 2 CCR handler there were
10 successful invocations and 14 roll_backs and  27  check msgs
For process 3  mutex handler there were
115 messages to get mutex and 20  messages to release mutex
For process 3 CCR handler there were
10 successful invocations and 15 roll_backs and  30  check msgs
For process 4  mutex handler there were
118 messages to get mutex and 21  messages to release mutex
For process 4 CCR handler there were
10 successful invocations and 16 roll_backs and  33  check msgs
For process 5  mutex handler there were
480 messages to get mutex and 60  messages to release mutex
For process 5 CCR handler there were
60 successful invocations and 20 roll_backs and  0  check msgs
ivy% exit
ivy%
script done on Thu Jun 29 13:47:19 1989
```

APPENDIX III

Monitor Code and Runs

The following are listings and runs for the roller coaster problem using monitors. Also included is the code and runs for a disk scheduling problem which uses prioritized condition variables. First comes all the code and runs for the centralized SC version of the roller coaster problem. Next comes the server code and runs for the SW version. All the other code is omitted since it is identical to the SC version. Next comes the server code and run for the SU version. All the other code is omitted since it is identical to the SC version. The six passenger run is omitted since it is identical to the SW run. Next comes the code and runs for the distributed SC version. The mutex server code is omitted since it identical to the unmodified version of the Suzuki and Kasami algorithm given in Appendix II. Next comes the monitor server code, condition variable code and modified mutex server code and run for the SW version. The other code is identical to that for the SC version. Next comes the monitor server code, condition variable code and further modified mutex server code and run for the SU version. Next comes the code and runs for the centralized SC version of the disk scheduler problem. Finally comes the code and run for the distributed SC version fo the disk scheduler problem.

```
this is a monitor solution to the cheap roller coaster problem
using a centralized monitor handler.
resource main_roller
   import Roller_globals
   import roller_server_cent
   import Passenger
   import Vehicle

body main_roller()
   op done()
   var num_of_pass : int
   var num_of_trips : int
   var tot_procs : int

   process start
      writes("Welcome to the roller coaster!")
      write("  Each passenger will get to ride 5 times")
      write("Enter the total number of passengers who will wish to ride.")
      read( num_of_pass)
      num_of_trips := ( num_of_pass * 5 + ( N - 1))/N
      tot_procs := num_of_pass + 1
      var car : cap Vehicle
      var passengers[ 1: num_of_pass] : cap Passenger
      var server : cap roller_server_cent

      create centralized monitor handler
      server := create roller_server_cent()

      create passenger processes
      fa i := 1 to num_of_pass →
         passengers[ i] := create Passenger( server, i, done)
      af

      create the car process
      car := create Vehicle( server, num_of_trips, done)

      wait for the passenger and car processes to complete
      fa i := 1 to tot_procs →
         receive done()
```

```
        af

        destroy the instances of all the processes created
        destroy car

        fa i := 1 to num_of_pass →
            destroy passengers[ i]
        af

        destroy server
    end
end


global Roller_globals
        const N := 4  number passengers per car
end Roller_globals


This resource implements the car process of the roller coaster
problem using monitor.
resource Vehicle
    import Roller_globals
    import roller_server_cent

body Vehicle( server: cap roller_server_cent; number_of_trips : int;
              done: cap())

    process car
        fa i := 1 to number_of_trips →
            call server.car_arriving()
            call server.car_leaving()
            call server.car_empty()
        af
        write("car done")
        send done()
    end
end
```

This resource implements the passenger process of the roller coaster *problem* using centralized monitors. The passenger rides five times.

```
resource Passenger
    import Roller_globals
    import roller_server_cent

body Passenger( server: cap roller_server_cent; process_number : int;
                done: cap() )

    process passenger
        fa i := 1 to 5 →
            call to wait in line and board the car
            call server.board_car( process_number)
            call to exit the car
            call server.exit_car( process_number)
        af
        write("passenger ",process_number,"done")
        send done()
    end
end
```

This resource implements the roller coaster problem using
monitors with the signal and continue protocal.

```
resource roller_server_cent
   import Roller_globals
   op board_car( passenger_number : int)
   op exit_car( passenger_number : int)
   op car_arriving()
   op car_leaving()
   op car_empty()

body roller_server_cent()

# N is a constant, the number of passengers per car
   var pin := 0,  pout := 0,  vin := 0,  vout := 0,  vdest  := 0
   op mutex()

   type return_rec = rec ( wait: cap( return_from_wait: cap());
                        signal: cap();
                        empty : cap() returns results : bool)

   var car_arrived, passengers_in_car, wait_to_arrive,
      passengers_off : return_rec

   op cond_var() returns signal_wait : return_rec

   var num_releases := 0, num_wait_msgs := 0, num_signal_msgs := 0

   initial
      create the condition variables
      car_arrived  := cond_var( )
      passengers_in_car  := cond_var( )
      wait_to_arrive  := cond_var( )
      passengers_off  := cond_var( )
      send mutex()
   end

   proc board_car( passenger_number)
      get monitorexclusively by waiting on mutex
      receive mutex()
```

```
    if pin ≥ vin * N →
       implementing wait( car_arrived)
       op return_from_wait1()
       send car_arrived.wait( return_from_wait1)
       send mutex()
       num_releases++
       receive return_from_wait1()
       receive mutex()
    [] else  →
       pin++
    fi
    write("Passenger ",passenger_number," is in the car")
    if ( vout + 1) * N ≤ pin and not passengers_in_car.empty() →
       call passengers_in_car.signal()
       vout++
    fi
    give up exclusive rights to monitor when exiting
    send mutex()
    num_releases++
end

proc exit_car( passenger_number)
    get monitor exclusively by waiting on mutex
    receive mutex()
    if pout ≥ vout * N →
       implementing wait( wait_to_arrive)
       op return_from_wait2()
       send wait_to_arrive.wait( return_from_wait2)
       send mutex()
       num_releases++
       receive return_from_wait2()
       receive mutex()
    [] else →
       pout++
    fi
    write("Passenger ",passenger_number," has exited car")
    if ( vdest + 1) * N ≤ pout and not passengers_off.empty() →
       call passengers_off.signal()
       vdest++
```

```
    fi
    give up exclusive rights to monitor when exiting
    send mutex()
    num_releases++
end

proc car_arriving()
    get monitor exclusively by waiting on mutex
    receive mutex()
    # car arrived at stop, notify passengers
    vin++
    write("Car has arrived at station")
    var count := N
    do count > 0 and not car_arrived.empty() →
      call car_arrived.signal()
      pin++
      count–
    od
    give up exclusive rights to monitor when exiting
    send mutex()
    num_releases++
end

proc car_leaving()
    get monitor exclusively by waiting on mutex
    receive mutex()
    # leave stop when car is full, and arrived at destination
    if ( vout + 1) * N > pin →
      implementing wait( passengers_on_bus)
      op return_from_wait3()
      send passengers_in_car.wait( return_from_wait3)
      send mutex()
      num_releases++
      receive return_from_wait3()
      receive mutex()
    [] else →
      vout++
    fi
    write("Car has filled, doing ride, now unloading")
```

```
    var count := N
    do count > 0 and not wait_to_arrive.empty() →
       call wait_to_arrive.signal()
       pout++
       count–
    od
    give up exclusive rights to monitor when exiting
    send mutex()
    num_releases++
end

proc car_empty()
    get monitor exclusively by waiting on mutex
    receive mutex()
    # wait for all passengers to disembark
    if ( vdest + 1) * N > pout →
       implementing wait( passengers_off)
       op return_from_wait4()
       send passengers_off.wait( return_from_wait4)
       send mutex()
       num_releases++
       receive return_from_wait4()
       receive mutex()
     [] else →
       vdest++
    fi
    write("car is now empty")
    give up exclusive rights to monitor when exiting
    send mutex()
    num_releases++
end

proc cond_var() returns signal_wait
process to implement the wait and signal operations on
# a condition variable
    op wait( return_to_caller:cap())
    op signal()
    op empty() returns results:bool
    signal_wait.wait := wait
```

```
    signal_wait.signal := signal
    signal_wait.empty := empty
    return the wait, signal, empty functions
    reply
    do true →
      in wait( return_to_caller) and ?signal ≠ 0 →
        # execute wait only if there is a signal
        # receive the signal and then get mutual exclusion
        # before returning
        num_wait_msgs++
        receive signal()
        num_signal_msgs++
        send return_to_caller()
        num_signal_msgs++
      [] signal() and ?wait = 0 →
        # drop a sent signal if there are no waits
        num_signal_msgs++
      [] empty() returns results →
        results :=  ( ?wait = 0)
      ni
    od
  end

  final
    write("There were ",num_releases," messages to release mutex and")
    writes( num_wait_msgs," wait messages and ")
    write( num_signal_msgs," signal messages")
  end

end
```

```
Script started on Sat Jul 22 09:39:15 1989
ivy% sc.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Car has filled, doing ride, now unloading
Passenger  4  is in the car
Passenger  1  has exited car
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
Passenger  7  is in the car
Car has filled, doing ride, now unloading
Passenger  8  is in the car
Passenger  5  has exited car
Passenger  6  has exited car
Passenger  7  has exited car
Passenger  8  has exited car
car is now empty
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Car has filled, doing ride, now unloading
Passenger  4  is in the car
Passenger  1  has exited car
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
```

Passenger 5 is in the car
Passenger 6 is in the car
Passenger 7 is in the car
Car has filled, doing ride, now unloading
Passenger 8 is in the car
Passenger 5 has exited car
Passenger 6 has exited car
Passenger 7 has exited car
Passenger 8 has exited car
car is now empty
Car has arrived at station
Passenger 1 is in the car
Passenger 2 is in the car
Passenger 3 is in the car
Car has filled, doing ride, now unloading
Passenger 4 is in the car
Passenger 1 has exited car
Passenger 2 has exited car
Passenger 3 has exited car
Passenger 4 has exited car
car is now empty
Car has arrived at station
Passenger 5 is in the car
Passenger 6 is in the car
Passenger 7 is in the car
Car has filled, doing ride, now unloading
Passenger 8 is in the car
Passenger 5 has exited car
Passenger 6 has exited car
Passenger 7 has exited car
Passenger 8 has exited car
car is now empty
Car has arrived at station
Passenger 1 is in the car
Passenger 2 is in the car
Passenger 3 is in the car
Car has filled, doing ride, now unloading
Passenger 4 is in the car
Passenger 1 has exited car

Passenger 2 has exited car
Passenger 3 has exited car
Passenger 4 has exited car
car is now empty
Car has arrived at station
Passenger 5 is in the car
Passenger 6 is in the car
Passenger 7 is in the car
Car has filled, doing ride, now unloading
Passenger 8 is in the car
Passenger 5 has exited car
Passenger 6 has exited car
Passenger 7 has exited car
Passenger 8 has exited car
car is now empty
Car has arrived at station
Passenger 1 is in the car
Passenger 2 is in the car
Passenger 3 is in the car
Car has filled, doing ride, now unloading
Passenger 4 is in the car
Passenger 1 has exited car
passenger 1 done
Passenger 2 has exited car
passenger 2 done
Passenger 3 has exited car
passenger 3 done
Passenger 4 has exited car
passenger 4 done
car is now empty
Car has arrived at station
Passenger 5 is in the car
Passenger 6 is in the car
Passenger 7 is in the car
Car has filled, doing ride, now unloading
Passenger 8 is in the car
Passenger 5 has exited car
passenger 5 done
Passenger 6 has exited car

```
passenger  6 done
Passenger  7  has exited car
passenger  7 done
Passenger  8  has exited car
passenger  8 done
car is now empty
car done
There were  160  messages to release mutex and
50 wait messages and 100  signal messages
ivy% exit
ivy%
script done on Sat Jul 22 09:39:58 1989
Script started on Thu Aug 31 15:37:01 1989
ivy% sc.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
6
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Car has filled, doing ride, now unloading
Passenger  4  is in the car
Passenger  1  has exited car
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
Passenger  1  is in the car
Car has filled, doing ride, now unloading
Passenger  2  is in the car
Passenger  5  has exited car
Passenger  6  has exited car
Passenger  1  has exited car
Passenger  2  has exited car
car is now empty
```

```
Car has arrived at station
Passenger  3  is in the car
Passenger  4  is in the car
Passenger  5  is in the car
Car has filled, doing ride, now unloading
Passenger  6  is in the car
Passenger  3  has exited car
Passenger  4  has exited car
Passenger  5  has exited car
Passenger  6  has exited car
car is now empty
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Car has filled, doing ride, now unloading
Passenger  4  is in the car
Passenger  1  has exited car
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
Passenger  1  is in the car
Car has filled, doing ride, now unloading
Passenger  2  is in the car
Passenger  5  has exited car
Passenger  6  has exited car
Passenger  1  has exited car
Passenger  2  has exited car
car is now empty
Car has arrived at station
Passenger  3  is in the car
Passenger  4  is in the car
Passenger  5  is in the car
Car has filled, doing ride, now unloading
Passenger  6  is in the car
```

```
Passenger  3  has exited car
Passenger  4  has exited car
Passenger  5  has exited car
Passenger  6  has exited car
car is now empty
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Car has filled, doing ride, now unloading
Passenger  4  is in the car
Passenger  1  has exited car
passenger  1 done
Passenger  2  has exited car
passenger  2 done
Passenger  3  has exited car
passenger  3 done
Passenger  4  has exited car
passenger  4 done
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
RTS warning: 8 blocked processes during termination
ivy% exit
ivy%
script done on Thu Aug 31 15:37:18 1989
```

```
This resource implements the monitor of the cheap roller
coaster problem using the signal and wait protocol.
resource roller_server_cent
   import Roller_globals
   op board_car( passenger_number : int)
   op exit_car( passenger_number : int)
   op car_arriving()
   op car_leaving()
   op car_empty()

body roller_server_cent()

# N is a constant, the number of passengers per car
   var pin := 0,  pout := 0,  vin := 0,  vout := 0,  vdest  := 0
   op mutex()

   type return_rec = rec ( wait: cap( return_from_wait: cap());
                  signal: cap() returns proc_waiting : bool;
                  empty : cap() returns results : bool)

   var car_arrived, passengers_in_car, wait_to_arrive,
      passengers_off : return_rec

   op cond_var() returns signal_wait : return_rec

   var num_releases := 0, num_wait_msgs := 0, num_signal_msgs := 0

   initial
      create the condition variables
      car_arrived  := cond_var()
      passengers_in_car  := cond_var()
      wait_to_arrive  := cond_var()
      passengers_off  := cond_var()
      send mutex()
   end

   proc board_car( passenger_number)
      get monitorexclusively by waiting on mutex
      receive mutex()
```

```
  if pin ≥ vin * N →
    implementing wait( car_arrived)
    op return_from_wait1()
    send car_arrived.wait( return_from_wait1)
    send mutex()
    num_releases++
    receive return_from_wait1()
  fi
  pin++
  write("Passenger ",passenger_number," is in the car")
  if ( vout + 1) * N ≤ pin →
    var b : bool
    this signal and all subsequent signals have code that
    differs from what was given in Chapter 5.  This is
    due to a bug currently in the SR compiler.
    b := passengers_in_car.signal()
    if b →
      receive mutex()
    fi
  fi
  give up exclusive rights to monitor when exiting
  send mutex()
  num_releases++
end

proc exit_car( passenger_number)
  get monitor exclusively by waiting on mutex
  receive mutex()
  if pout ≥ vout * N →
    implementing wait( wait_to_arrive)
    op return_from_wait2()
    send wait_to_arrive.wait( return_from_wait2)
    send mutex()
    num_releases++
    receive return_from_wait2()
  fi
  pout++
  write("Passenger ",passenger_number," has exited car")
  if ( vdest + 1) * N ≤ pout →
```

```
      var b : bool
      b := passengers_off.signal()
      if b →
         receive mutex()
      fi
   fi
   give up exclusive rights to monitor when exiting
   send mutex()
   num_releases++
end

proc car_arriving()
   get monitor exclusively by waiting on mutex
   receive mutex()
   # car arrived at stop, notify passengers
   write("Car has arrived at station")
   var count := N
   var b : bool := true
   do count > 0 and b →
      b := car_arrived.signal()
      if b →
         receive mutex()
         count–
      fi
   od
   vin++
   give up exclusive rights to monitor when exiting
   send mutex()
   num_releases++
end

proc car_leaving()
   get monitor exclusively by waiting on mutex
   receive mutex()
   # leave stop when car is full, and arrived at destination
   if ( vout + 1) * N > pin →
      implementing wait( passengers_in_car)
      op return_from_wait3()
      send passengers_in_car.wait( return_from_wait3)
```

```
      send mutex()
      num_releases++
      receive return_from_wait3()
    fi
    write("Car has filled, doing ride, now unloading")
    var count := N
    var b : bool := true
    do count > 0 and b →
      b := wait_to_arrive.signal()
      if b →
        receive mutex()
        count–
      fi
    od
    vout++
    give up exclusive rights to monitor when exiting
    send mutex()
    num_releases++
end

proc car_empty()
    get monitor exclusively by waiting on mutex
    receive mutex()
    # wait for all passengers to disembark
    if ( vdest + 1) * N > pout →
      implementing wait( passengers_off)
      op return_from_wait4()
      send passengers_off.wait( return_from_wait4)
      send mutex()
      num_releases++
      receive return_from_wait4()
    fi
    vdest++
    write("car is now empty")
    give up exclusive rights to monitor when exiting
    send mutex()
    num_releases++
end
```

```
proc cond_var() returns signal_wait
process to implement the wait and signal operations on
# a condition variable
    op wait( return_to_caller:cap())
    op signal() returns proc_waiting : bool
    op empty() returns results:bool
    signal_wait.wait := wait
    signal_wait.signal := signal
    signal_wait.empty := empty
    return the wait, signal, empty functions
    reply
    do true →
        in wait( return_to_caller) and ?signal ≠ 0 →
            # execute wait only if there is a signal
            # receive the signal and then get mutual exclusion
            # before returning
            num_wait_msgs++
            in signal() returns proc_waiting →
               proc_waiting := true
               num_signal_msgs++
            ni
            send return_to_caller()
            num_signal_msgs++
        [] signal() returns proc_waiting and ?wait = 0 →
            # drop a sent signal if there are no waits
            proc_waiting := false
            num_signal_msgs++
        [] empty() returns results →
            results :=  ( ?wait = 0)
        ni
    od
end

final
   write("There were ",num_releases," messages to release mutex and ")
   writes( num_wait_msgs," wait messages and ")
   write( num_signal_msgs," signal messages")
end
```

end

```
Script started on Sat Jul 22 09:40:05 1989
ivy% sw.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Passenger  4  is in the car
Car has filled, doing ride, now unloading
Passenger  1  has exited car
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
Passenger  7  is in the car
Passenger  8  is in the car
Car has filled, doing ride, now unloading
Passenger  5  has exited car
Passenger  6  has exited car
Passenger  7  has exited car
Passenger  8  has exited car
car is now empty
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Passenger  4  is in the car
Car has filled, doing ride, now unloading
Passenger  1  has exited car
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
```

Passenger 5 is in the car
Passenger 6 is in the car
Passenger 7 is in the car
Passenger 8 is in the car
Car has filled, doing ride, now unloading
Passenger 5 has exited car
Passenger 6 has exited car
Passenger 7 has exited car
Passenger 8 has exited car
car is now empty
Car has arrived at station
Passenger 1 is in the car
Passenger 2 is in the car
Passenger 3 is in the car
Passenger 4 is in the car
Car has filled, doing ride, now unloading
Passenger 1 has exited car
Passenger 2 has exited car
Passenger 3 has exited car
Passenger 4 has exited car
car is now empty
Car has arrived at station
Passenger 5 is in the car
Passenger 6 is in the car
Passenger 7 is in the car
Passenger 8 is in the car
Car has filled, doing ride, now unloading
Passenger 5 has exited car
Passenger 6 has exited car
Passenger 7 has exited car
Passenger 8 has exited car
car is now empty
Car has arrived at station
Passenger 1 is in the car
Passenger 2 is in the car
Passenger 3 is in the car
Passenger 4 is in the car
Car has filled, doing ride, now unloading
Passenger 1 has exited car

```
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
Passenger  7  is in the car
Passenger  8  is in the car
Car has filled, doing ride, now unloading
Passenger  5  has exited car
Passenger  6  has exited car
Passenger  7  has exited car
Passenger  8  has exited car
car is now empty
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Passenger  4  is in the car
Car has filled, doing ride, now unloading
Passenger  1  has exited car
passenger  1 done
Passenger  2  has exited car
passenger  2 done
Passenger  3  has exited car
passenger  3 done
Passenger  4  has exited car
passenger  4 done
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
Passenger  7  is in the car
Passenger  8  is in the car
Car has filled, doing ride, now unloading
Passenger  5  has exited car
passenger  5 done
Passenger  6  has exited car
```

```
passenger  6 done
Passenger  7  has exited car
passenger  7 done
Passenger  8  has exited car
passenger  8 done
car is now empty
car done
There were  190  messages to release mutex and
80 wait messages and 180  signal messages
ivy% exit
ivy%
script done on Sat Jul 22 09:40:20 1989
Script started on Thu Aug 31 15:41:43 1989
ivy% sw.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
6
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Passenger  4  is in the car
Car has filled, doing ride, now unloading
Passenger  1  has exited car
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
Passenger  1  is in the car
Passenger  2  is in the car
Car has filled, doing ride, now unloading
Passenger  5  has exited car
Passenger  6  has exited car
Passenger  1  has exited car
Passenger  2  has exited car
car is now empty
```

```
Car has arrived at station
Passenger  3  is in the car
Passenger  4  is in the car
Passenger  5  is in the car
Passenger  6  is in the car
Car has filled, doing ride, now unloading
Passenger  3  has exited car
Passenger  4  has exited car
Passenger  5  has exited car
Passenger  6  has exited car
car is now empty
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Passenger  4  is in the car
Car has filled, doing ride, now unloading
Passenger  1  has exited car
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
Passenger  1  is in the car
Passenger  2  is in the car
Car has filled, doing ride, now unloading
Passenger  5  has exited car
Passenger  6  has exited car
Passenger  1  has exited car
Passenger  2  has exited car
car is now empty
Car has arrived at station
Passenger  3  is in the car
Passenger  4  is in the car
Passenger  5  is in the car
Passenger  6  is in the car
Car has filled, doing ride, now unloading
```

```
Passenger  3  has exited car
Passenger  4  has exited car
Passenger  5  has exited car
Passenger  6  has exited car
car is now empty
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Passenger  4  is in the car
Car has filled, doing ride, now unloading
Passenger  1  has exited car
passenger  1 done
Passenger  2  has exited car
passenger  2 done
Passenger  3  has exited car
passenger  3 done
Passenger  4  has exited car
passenger  4 done
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
RTS warning: 8 blocked processes during termination
ivy% exit
ivy%
script done on Thu Aug 31 15:41:57 1989
```

```
This resource implements the roller coaster problem monitor
with the signal and urgent wait protocal.
resource roller_server_cent
   import Roller_globals
   op board_car( passenger_number, trip : int)
   op exit_car( passenger_number, trip : int)
   op car_arriving()
   op car_leaving()
   op car_empty()

body roller_server_cent()

# N is a constant, the number of passengers per car
   var pin := 0,  pout := 0,  vin := 0,  vout := 0,  vdest  := 0
   op mutex()
   op UrgentQ()
   type return_rec = rec ( wait: cap( return_from_wait: cap());
                     signal: cap() returns proc_waiting : bool;
                     empty : cap() returns results : bool)

   var car_arrived, passengers_in_car, wait_to_arrive,
      passengers_off : return_rec

   op cond_var() returns signal_wait : return_rec

   var num_releases := 0, num_wait_msgs := 0, num_signal_msgs := 0

   initial
      create the condition variables
      car_arrived  := cond_var( )
      passengers_in_car  := cond_var( )
      wait_to_arrive  := cond_var( )
      passengers_off  := cond_var( )
      send mutex()
   end

   proc board_car( passenger_number, trip)
      get monitorexclusively by waiting on mutex
      receive mutex()
```

```
    if pin ≥ vin * N →
      implementing wait( car_arrived)
      op return_from_wait1()
      send car_arrived.wait( return_from_wait1)
      if ?UrgentQ > 0 →
        receive UrgentQ()
      [] else →
        send mutex()
        num_releases++
      fi
      receive return_from_wait1()
    fi
    pin++
    write("Passenger ",passenger_number," is in the car")
    if ( vout + 1) * N ≤ pin →
      var b : bool
      this signal and all subsequent signals have code that
      differs from what was given in Chapter 5.  This is
      due to a bug currently in the SR compiler.
      b := passengers_in_car.signal()
      if b →
        call UrgentQ()
        num_releases++
      fi
    fi
    give up exclusive rights to monitor when exiting
    if ?UrgentQ > 0 →
      receive UrgentQ()
    [] else →
      send mutex()
      num_releases++
    fi
  end

  proc exit_car( passenger_number, trip)
    get monitor exclusively by waiting on mutex
    receive mutex()
    if pout ≥ vout * N →
      implementing wait( wait_to_arrive)
```

```
        op return_from_wait2()
        send wait_to_arrive.wait( return_from_wait2)
        if ?UrgentQ > 0 →
          receive UrgentQ()
        [] else →
          send mutex()
          num_releases++
        fi
        receive return_from_wait2()
      fi
      pout++
      write("Passenger ",passenger_number," has exited car")
      if ( vdest + 1) * N ≤ pout →
        var b : bool
        b := passengers_off.signal()
        if b →
          call UrgentQ()
          num_releases++
        fi
      fi
      give up exclusive rights to monitor when exiting
      if ?UrgentQ > 0 →
        receive UrgentQ()
      [] else →
        send mutex()
        num_releases++
      fi
    end

    proc car_arriving()
      get monitor exclusively by waiting on mutex
      receive mutex()
      # car arrived at stop, notify passengers
      vin++
      write("Car has arrived at station")
      var count := N
      var b : bool := true
      do count > 0 and b →
        b := car_arrived.signal()
```

```
    if b →
      call UrgentQ()
      num_releases++
      count–
    fi
  od
  give up exclusive rights to monitor when exiting
  if ?UrgentQ > 0 →
    receive UrgentQ()
  [] else →
    send mutex()
    num_releases++
  fi
end

proc car_leaving()
  get monitor exclusively by waiting on mutex
  receive mutex()
  # leave stop when car is full, and arrived at destination
  if ( vout + 1) * N > pin →
    implementing wait( passengers_on_bus)
    op return_from_wait3()
    send passengers_in_car.wait( return_from_wait3)
    if ?UrgentQ > 0 →
      receive UrgentQ()
    [] else →
      send mutex()
      num_releases++
    fi
    receive return_from_wait3()
  fi
  vout++
  write("Car has filled, doing ride, now unloading")
  var count := N
  var b : bool := true
  do count > 0 and b →
    b := wait_to_arrive.signal()
    if b →
      call UrgentQ()
```

```
      num_releases++
      count–
    fi
  od
  give up exclusive rights to monitor when exiting
  if ?UrgentQ > 0 →
    receive UrgentQ()
  [] else →
    send mutex()
    num_releases++
  fi
end

proc car_empty()
  get monitor exclusively by waiting on mutex
  receive mutex()
  # wait for all passengers to disembark
  if ( vdest + 1) * N > pout →
    implementing wait( passengers_off)
    op return_from_wait4()
    send passengers_off.wait( return_from_wait4)
    if ?UrgentQ > 0 →
      receive UrgentQ()
    [] else →
      send mutex()
      num_releases++
    fi
    receive return_from_wait4()
  fi
  vdest++
  write("car is now empty")
  give up exclusive rights to monitor when exiting
  if ?UrgentQ > 0 →
    receive UrgentQ()
  [] else →
    send mutex()
    num_releases++
  fi
end
```

```
proc cond_var() returns signal_wait
process to implement the wait and signal operations on
# a condition variable
    op wait( return_to_caller:cap())
    op signal() returns proc_waiting : bool
    op empty() returns results:bool
    signal_wait.wait := wait
    signal_wait.signal := signal
    signal_wait.empty := empty
    return the wait, signal, empty functions
    reply
    do true →
        in wait( return_to_caller) and ?signal ≠ 0 →
            # execute wait only if there is a signal
            # receive the signal and then get mutual exclusion
            # before returning
            num_wait_msgs++
            in signal() returns proc_waiting →
               proc_waiting := true
               num_signal_msgs++
            ni
            send return_to_caller()
            num_signal_msgs++
        [] signal() returns proc_waiting and ?wait = 0 →
            # drop a sent signal if there are no waits
            proc_waiting := false
            num_signal_msgs++
        [] empty() returns results →
            results :=  ( ?wait = 0)
        ni
    od
end

final
   write("There were ",num_releases," messages to release mutex and ")
   writes( num_wait_msgs," wait messages and ")
   write( num_signal_msgs," signal messages")
end
```

end

```
Script started on Sat Jul 22 09:40:25 1989
ivy% su.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Passenger  4  is in the car
Car has filled, doing ride, now unloading
Passenger  1  has exited car
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
Passenger  5  is in the car
Passenger  6  is in the car
Passenger  7  is in the car
Passenger  8  is in the car
Car has filled, doing ride, now unloading
Passenger  5  has exited car
Passenger  6  has exited car
Passenger  7  has exited car
Passenger  8  has exited car
car is now empty
Car has arrived at station
Passenger  1  is in the car
Passenger  2  is in the car
Passenger  3  is in the car
Passenger  4  is in the car
Car has filled, doing ride, now unloading
Passenger  1  has exited car
Passenger  2  has exited car
Passenger  3  has exited car
Passenger  4  has exited car
car is now empty
Car has arrived at station
```

Passenger 5 is in the car
Passenger 6 is in the car
Passenger 7 is in the car
Passenger 8 is in the car
Car has filled, doing ride, now unloading
Passenger 5 has exited car
Passenger 6 has exited car
Passenger 7 has exited car
Passenger 8 has exited car
car is now empty
Car has arrived at station
Passenger 1 is in the car
Passenger 2 is in the car
Passenger 3 is in the car
Passenger 4 is in the car
Car has filled, doing ride, now unloading
Passenger 1 has exited car
Passenger 2 has exited car
Passenger 3 has exited car
Passenger 4 has exited car
car is now empty
Car has arrived at station
Passenger 5 is in the car
Passenger 6 is in the car
Passenger 7 is in the car
Passenger 8 is in the car
Car has filled, doing ride, now unloading
Passenger 5 has exited car
Passenger 6 has exited car
Passenger 7 has exited car
Passenger 8 has exited car
car is now empty
Car has arrived at station
Passenger 1 is in the car
Passenger 2 is in the car
Passenger 3 is in the car
Passenger 4 is in the car
Car has filled, doing ride, now unloading
Passenger 1 has exited car

Passenger 2 has exited car
Passenger 3 has exited car
Passenger 4 has exited car
car is now empty
Car has arrived at station
Passenger 5 is in the car
Passenger 6 is in the car
Passenger 7 is in the car
Passenger 8 is in the car
Car has filled, doing ride, now unloading
Passenger 5 has exited car
Passenger 6 has exited car
Passenger 7 has exited car
Passenger 8 has exited car
car is now empty
Car has arrived at station
Passenger 1 is in the car
Passenger 2 is in the car
Passenger 3 is in the car
Passenger 4 is in the car
Car has filled, doing ride, now unloading
Passenger 1 has exited car
passenger 1 done
Passenger 2 has exited car
passenger 2 done
Passenger 3 has exited car
passenger 3 done
Passenger 4 has exited car
passenger 4 done
car is now empty
Car has arrived at station
Passenger 5 is in the car
Passenger 6 is in the car
Passenger 7 is in the car
Passenger 8 is in the car
Car has filled, doing ride, now unloading
Passenger 5 has exited car
passenger 5 done
Passenger 6 has exited car

```
passenger  6 done
Passenger  7  has exited car
passenger  7 done
Passenger  8  has exited car
passenger  8 done
car is now empty
car done
There were  180  messages to release mutex and
70 wait messages and 170  signal messages
ivy% exit
ivy%
script done on Sat Jul 22 09:40:37 1989
```

The 6 passenger SU run was omitted since it was identical to the SC run.

This is the start up process for the cheap roller coaster problem
using "distributed" monitors

```
resource main_roller
    import Roller_globals
    import Cond_var
    import Mutex_server
    import Monitor_server
    import Passenger
    import Vehicle

body main_roller()
    op done()
    var num_of_passengers, num_of_trips : int

    process start
        writes("Welcome to the roller coaster!")
        write("  Each passenger will get to ride 5 times")
        write("Enter the total number of passengers who will wish to ride.")
        read( num_of_passengers)
        num_of_trips := ( num_of_passengers * 5 + ( N - 1)) / N
        var total_number_of_processes := num_of_passengers + 1
        var wait_to_arrive[ 1: total_number_of_processes] : cap Cond_var
        var car_arrived[ 1: total_number_of_processes] : cap Cond_var
        var passengers_in_car[ 1: total_number_of_processes] : cap Cond_var
        var passengers_off[ 1: total_number_of_processes] : cap Cond_var
        var car : cap Vehicle
        var passengers[ 1 : num_of_passengers] : cap Passenger
        var mutex_servers[ 1 : total_number_of_processes] : cap Mutex_server
        var monitor_servers[ 1 : total_number_of_processes] : cap Monitor_server

        fa i := 1 to total_number_of_processes →
           wait_to_arrive[ i] := create Cond_var( i,
                                       total_number_of_processes,
                                       "wait_to_arrive")
           car_arrived[ i] := create Cond_var( i,
                                       total_number_of_processes,
                                       "car_arrived")
           passengers_in_car[ i] := create Cond_var( i,
                                       total_number_of_processes,
```

```
                                    "passengers_in_car")
   passengers_off[ i] := create Cond_var( i,
                                    total_number_of_processes,
                                    "passengers_off")
   mutex_servers[ i] := create Mutex_server( i,
                          total_number_of_processes)
   monitor_servers[ i] := create Monitor_server( mutex_servers[ i],
                           i, total_number_of_processes,
                           car_arrived[ i],
                           passengers_in_car[ i],
                           wait_to_arrive[ i],
                           passengers_off[ i])
af
```

create the passengers

```
fa i := 1 to num_of_passengers →
   passengers[ i] := create Passenger( monitor_servers[ i],
                                   i, done)
af
```

create the Vehicle

```
car := create Vehicle( monitor_servers[total_number_of_processes],
                     num_of_trips, done)


fa i := 1 to total_number_of_processes →
   send passengers_in_car[i].links( passengers_in_car)
   send car_arrived[i].links( car_arrived)
   send wait_to_arrive[ i].links( wait_to_arrive)
   send passengers_off[i].links( passengers_off)
   send mutex_servers[ i].links( mutex_servers)
   send monitor_servers[ i].links( monitor_servers)
af

fa i := 1 to total_number_of_processes →
   receive done()
af

fa i := 1 to total_number_of_processes →
```

```
      destroy passengers_in_car[ i]
      destroy car_arrived[ i]
      destroy wait_to_arrive[ i]
      destroy passengers_off[ i]
      destroy mutex_servers[ i]
      destroy monitor_servers[ i]
    af

    destroy car
    fa i := 1 to num_of_passengers →
      destroy passengers[ i]
    af
  end
end

global Roller_globals
    op type for monitor routines to share the monitor variables
    optype Monitor_Pass_globals = ( pin, pout, bin, bout, bdest : int)
    const N := 4 number passengers per car
end Roller_globals
```

```
# This resource implements the passenger process of the roller coaster
# problem using distributed monitors.  The passenger rides five times.
resource Passenger
    import Roller_globals
    import Monitor_server      # distributed monitor server

body Passenger( server: cap Monitor_server;
             process_number : int; done: cap())

    process Passenger_process
        fa i := 1 to 5 ->
           server.board_car()
           server.exit_car()
        af
        send done()
        write("passenger",process_number,"is done")
    end

end
```

This resource implements the car process of the roller coaster *problem* using distributed monitor server.

```
resource Vehicle
   import Roller_globals
   import Monitor_server      distributed monitor server

body Vehicle( server: cap Monitor_server;
            trips : int;
            done: cap())

   process Vehicle_process
      fa i := 1 to trips →
        call server.car_arriving()
        call server.car_leaving()
        call server.car_empty()
      af
      write("car is done")
      send done()
   end

end
```

```
This resource implements the monitor server for the
roller coaster problem using distributed monitors.
This version uses the signal and continue protocol.
resource Monitor_server
   import Mutex_server  distributed mutual exclusion server
   import Roller_globals
   import Cond_var     handles monitor signal and wait calls
                  in a distrubuted fashion
   op links( others[ 1:* ] : cap Monitor_server)
   op board_car()
   op exit_car()
   op car_arriving()
   op car_leaving()
   op car_empty()
   op Pass_globals Monitor_Pass_globals

body Monitor_server( mutex_server: cap Mutex_server;
                process_number, total_processes : int;
                car_arrived, passengers_in_car,
                wait_to_arrive, passengers_off : cap Cond_var)


   var pin := 0, pout := 0, vin := 0, vout := 0, vdest := 0
   var others [ 1 : total_processes] : cap Monitor_server

   var num_update_msgs := 0

   initial
     reply
     receive links( others)
   end

   proc board_car()
      get monitor exclusively by calling exclusion server
      mutex_server.get_exclusion()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
```

```
    if pin ≥ vin * N →
      # wait( car_arrived)
      op return_from_wait1()
      call car_arrived.wait_request( return_from_wait1)
      mutex_server.release()
      receive return_from_wait1()
      mutex_server.get_exclusion()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
    [] else →
      pin++
    fi
    if ( vout + 1) * N ≤ pin and not passengers_in_car.empty() →
      there are enough passengers in line for the car, signal a car
      call passengers_in_car.signal_request()
      vout++
    fi
    update monitor globals
    fa i := 1 to total_processes
              st i ≠ process_number →
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    write("passenger #",process_number," is in the car")
    # giving up exclusive rights when exiting
    mutex_server.release()
end

proc exit_car()
    get monitor exclusively by calling exclusion server
    mutex_server.get_exclusion()
    get any updates before processing
    do ?Pass_globals ≠ 0 →
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
    now try to exit car
    if pout ≥ ( vout * N) →
```

```
      no car with room, wait
      implementing wait( )
      op return_from_wait2()
      call wait_to_arrive.wait_request( return_from_wait2)
      mutex_server.release()
      receive return_from_wait2()
      mutex_server.get_exclusion()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
    [] else →
      pout++
    fi
    if ( vdest + 1) * N ≤ pout and not passengers_off.empty() →
      passengers_off.signal_request()
      vdest++
    fi
    update monitor globals
    fa i := 1 to total_processes
                st i ≠ process_number →
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    write( "passenger #",process_number," has exited car")
    # giving up exclusive rights when exiting
    mutex_server.release()
end

proc car_arriving( )
    get monitor exclusively by calling mutex server
    mutex_server.get_exclusion()
    get any updates before processing
    do ?Pass_globals ≠ 0 →
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
    car arriving
    vin++
    var count := N
```

```
    do count > 0 and not car_arrived.empty() →
      call car_arrived.signal_request()
      pin++
      count–
    od
    update monitor globals
    fa i := 1 to total_processes
                st i ≠ process_number →
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    write("Car has arrived at station")
    give up exclusive rights to monitor when exiting
    mutex_server.release()
end

proc car_leaving( )
    get monitor exclusively by calling mutex server
    mutex_server.get_exclusion()
    get any updates before processing
    do ?Pass_globals ≠ 0 →
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
    if ( vout + 1) * N > pin →
      implementing wait( passengers_in_car)
      op return_from_wait1()
      call passengers_in_car.wait_request( return_from_wait1)
      mutex_server.release()
      receive return_from_wait1()
      mutex_server.get_exclusion()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
    [] else →
      vout++
    fi
    var count := N
    do count > 0 and not wait_to_arrive.empty() →
```

```
      call wait_to_arrive.signal_request()
      pout++
      count–
    od
    update monitor globals
    fa i := 1 to total_processes
                st i ≠ process_number →
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    write("Car has filled, doing ride, now unloading")
    give up exclusive rights to monitor when exiting
    mutex_server.release()
end

proc car_empty( )
    get monitor exclusively by calling mutex server
    mutex_server.get_exclusion()
    get any updates before processing
    do ?Pass_globals ≠ 0 →
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
    if (vdest + 1) * N > pout →
      implementing wait( passengers_off)
      op return_from_wait1()
      call passengers_off.wait_request( return_from_wait1)
      mutex_server.release()
      receive return_from_wait1()
      mutex_server.get_exclusion()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
    [] else →
      vdest++
    fi
    update monitor globals
    fa i := 1 to total_processes
                st i ≠ process_number →
```

```
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    write("car is now empty")
    give up exclusive rights to monitor when exiting
    mutex_server.release()
  end

  final
    writes("Monitor server for process ",process_number)
    write(" sent ",num_update_msgs," update messages")
  end
end
```

```
resource to handle in a distributed manner the signal and wait
operations on a monitors condition variable
resource Cond_var
  op wait_request( return_from_wait : cap()) local wait request
                                             from "our" process
  op signal_request()                        local signal request
                                             from "our" process
  op wait_update( return_from_wait : cap())  global wait queue
  op signal_update()                         global signal message
  op empty() returns result : bool           check if global wait Q is empty
  op links( other_servers[ 1: *] : cap Cond_var)

body Cond_var( me, N : int; name:string(*))

  var local_wait_requests := 0, global_wait_msgs := 0
  var local_signal_requests :=0, global_signal_msgs := 0

  process cond_var_handler
    get the capabilities of the handlers of this condition variable
    var cond_var_handlers[ 1 : N] : cap Cond_var
    receive links( cond_var_handlers)

    do true →
      in wait_request( return_from_wait) →
        this handlers process is requesting a wait on the condition
        variable, notify all cond_var_handlers to put request on
        wait queue
        local_wait_requests++
        fa i := 1 to N →
          send cond_var_handlers[ i].wait_update( return_from_wait)
          global_wait_msgs++
        af
      [] signal_request() →
        this handlers process is signaling the condition variable
        check if there are any waiting processes
        local_signal_requests++
        if ?wait_update ≠ 0 →
          there is a wait request pending
          notify other handlers of signal
```

```
          fa i := 1 to N st i ≠ me →
            send cond_var_handlers[ i].signal_update()
            global_signal_msgs++
          af
          get operation from local wait_queue and send to it
          in wait_update( return_oper) →
            send return_oper()
            global_signal_msgs++
          ni
        [] else →
          no waits pending, drop signal on floor
        fi
      [] empty() returns result →
        result := ( ?wait_update = 0)
      [] signal_update() →
        a process has signaled the condition variable
        remove the first element off of wait queue
        if ?wait_update > 0 →
          in wait_update( temp) → ni
        [] else →
          write("ERROR: erroneous signal_update received")
        fi
      ni
    od
  end

  final
    write("For ",name," handler for process #",me,"there were")
    writes(local_wait_requests," local wait requests and ")
    write(global_wait_msgs," global wait messages")
    writes(local_signal_requests," local signal requests and ")
    write(global_signal_msgs," global signal messages")
  end
end
```

The distributed mutex algorithm for the SC version is omitted since it is identical to the unmodified version of the algorithm in Appendix II.

```
Script started on Mon Jul 24 10:07:19 1989
ivy% sc.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Car has arrived at station
passenger # 1  is in the car
passenger # 2  is in the car
passenger # 3  is in the car
passenger # 4  is in the car
Car has filled, doing ride, now unloading
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
Car has arrived at station
passenger # 5  is in the car
passenger # 6  is in the car
passenger # 7  is in the car
passenger # 8  is in the car
Car has filled, doing ride, now unloading
passenger # 5  has exited car
passenger # 6  has exited car
passenger # 7  has exited car
passenger # 8  has exited car
car is now empty
Car has arrived at station
passenger # 1  is in the car
passenger # 2  is in the car
passenger # 3  is in the car
passenger # 4  is in the car
Car has filled, doing ride, now unloading
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
Car has arrived at station
```

```
passenger # 5  is in the car
passenger # 6  is in the car
passenger # 7  is in the car
passenger # 8  is in the car
Car has filled, doing ride, now unloading
passenger # 5  has exited car
passenger # 6  has exited car
passenger # 7  has exited car
passenger # 8  has exited car
car is now empty
Car has arrived at station
passenger # 1  is in the car
passenger # 2  is in the car
passenger # 3  is in the car
passenger # 4  is in the car
Car has filled, doing ride, now unloading
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
Car has arrived at station
passenger # 5  is in the car
passenger # 6  is in the car
passenger # 7  is in the car
passenger # 8  is in the car
Car has filled, doing ride, now unloading
passenger # 5  has exited car
passenger # 6  has exited car
passenger # 7  has exited car
passenger # 8  has exited car
car is now empty
Car has arrived at station
passenger # 1  is in the car
passenger # 2  is in the car
passenger # 3  is in the car
passenger # 4  is in the car
Car has filled, doing ride, now unloading
passenger # 1  has exited car
```

```
passenger # 2  has exited car
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
Car has arrived at station
passenger # 5  is in the car
passenger # 6  is in the car
passenger # 7  is in the car
passenger # 8  is in the car
Car has filled, doing ride, now unloading
passenger # 5  has exited car
passenger # 6  has exited car
passenger # 7  has exited car
passenger # 8  has exited car
car is now empty
Car has arrived at station
passenger # 1  is in the car
passenger # 2  is in the car
passenger # 3  is in the car
passenger # 4  is in the car
Car has filled, doing ride, now unloading
passenger # 1  has exited car
passenger 1 is done
passenger # 2  has exited car
passenger 2 is done
passenger # 3  has exited car
passenger 3 is done
passenger # 4  has exited car
passenger 4 is done
car is now empty
Car has arrived at station
passenger # 5  is in the car
passenger # 6  is in the car
passenger # 7  is in the car
passenger # 8  is in the car
Car has filled, doing ride, now unloading
passenger # 5  has exited car
passenger 5 is done
passenger # 6  has exited car
```

passenger 6 is done
passenger # 7 has exited car
passenger 7 is done
passenger # 8 has exited car
passenger 8 is done
car is now empty
car is done
For passengers_in_car handler for process # 1 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 1 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 1 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 1 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 1 mutex handler there were
141 messages to get mutex and 15 to release mutex.
Monitor server for process 1 sent 80 update messages
For passengers_in_car handler for process # 2 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 2 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 2 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 2 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 2 mutex handler there were
150 messages to get mutex and 15 to release mutex.
Monitor server for process 2 sent 80 update messages
For passengers_in_car handler for process # 3 there were
0 local wait requests and 0 global wait messages

0 local signal requests and 0 global signal messages
For car_arrived handler for process # 3 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 3 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 3 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 3 mutex handler there were
150 messages to get mutex and 15 to release mutex.
Monitor server for process 3 sent 80 update messages
For passengers_in_car handler for process # 4 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 4 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 4 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 4 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 4 mutex handler there were
150 messages to get mutex and 15 to release mutex.
Monitor server for process 4 sent 80 update messages
For passengers_in_car handler for process # 5 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 5 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 5 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 5 there were
0 local wait requests and 0 global wait messages

0 local signal requests and 0 global signal messages
For process 5 mutex handler there were
150 messages to get mutex and 15 to release mutex.
Monitor server for process 5 sent 80 update messages
For passengers_in_car handler for process # 6 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 6 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 6 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 6 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 6 mutex handler there were
150 messages to get mutex and 15 to release mutex.
Monitor server for process 6 sent 80 update messages
For passengers_in_car handler for process # 7 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 7 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 7 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 7 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 7 mutex handler there were
150 messages to get mutex and 15 to release mutex.
Monitor server for process 7 sent 80 update messages
For passengers_in_car handler for process # 8 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 8 there were
5 local wait requests and 45 global wait messages

0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 8 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 8 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 8 mutex handler there were
150 messages to get mutex and 15 to release mutex.
Monitor server for process 8 sent 80 update messages
For passengers_in_car handler for process # 9 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 9 there were
0 local wait requests and 0 global wait messages
40 local signal requests and 360 global signal messages
For wait_to_arrive handler for process # 9 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 9 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 9 mutex handler there were
291 messages to get mutex and 30 to release mutex.
Monitor server for process 9 sent 240 update messages
ivy% exit
ivy%
script done on Mon Jul 24 10:07:57 1989
Script started on Thu Jul 20 12:53:06 1989
ivy% sc.out
Welcome to the roller coaster! Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
6
car arrived
passenger # 1 is on the car
passenger # 2 is on the car
passenger # 3 is on the car
passenger # 4 is on the car
car is leaving

```
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
car arrived
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 5  is on the car
passenger # 6  is on the car
car is leaving
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 5  has exited car
passenger # 6  has exited car
car is now empty
car arrived
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
car is leaving
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
car arrived
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 5  is on the car
passenger # 6  is on the car
car is leaving
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 5  has exited car
passenger # 6  has exited car
car is now empty
car arrived
```

```
passenger # 1  is on the car
passenger # 2  is on the car
passenger # 3  is on the car
passenger # 4  is on the car
car is leaving
passenger # 1  has exited car
passenger 1 is done
passenger # 2  has exited car
passenger 2 is done
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
car arrived
passenger # 3  is on the car
passenger # 4  is on the car
passenger # 5  is on the car
passenger # 6  is on the car
car is leaving
passenger # 3  has exited car
passenger # 4  has exited car
passenger # 5  has exited car
passenger # 6  has exited car
car is now empty
car arrived
passenger # 3  is on the car
passenger # 4  is on the car
passenger # 5  is on the car
passenger # 6  is on the car
car is leaving
passenger # 3  has exited car
passenger 3 is done
passenger # 4  has exited car
passenger 4 is done
passenger # 5  has exited car
passenger # 6  has exited car
car is now empty
car arrived
passenger # 5  is on the car
passenger # 6  is on the car
```

```
RTS warning: 39 blocked processes during termination
ivy% exit
ivy%
script done on Thu Jul 20 12:53:22 1989
```

```
This resource implements the monitor server for the
roller coaster problem using distributed monitors.
This version uses the signal and wait protocol.
resource Monitor_server
  import Mutex_server  distributed mutual exclusion server
  import Roller_globals
  import Cond_var      handles monitor signal and wait calls
                       in a distrubuted fashion
  op links( others[ 1:* ] : cap Monitor_server)
  op board_car()
  op exit_car()
  op car_arriving()
  op car_leaving()
  op car_empty()
  op Pass_globals Monitor_Pass_globals

body Monitor_server( mutex_server: cap Mutex_server;
            process_number, total_processes : int;
            car_arrived, passengers_on_car,
            wait_to_arrive, passengers_off : cap Cond_var)


  var pin := 0, pout := 0, vin := 0, vout := 0, vdest := 0
  var others [ 1 : total_processes] : cap Monitor_server

  var num_update_msgs := 0

  initial
    reply
    receive links( others)
  end

  proc board_car()
    get monitor exclusively by calling exclusion server
    mutex_server.get_exclusion()
    get any updates before processing
    do ?Pass_globals ≠ 0 →
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
```

```
    if pin ≥ vin * N →
      # wait( car_arrived)
      op return_from_wait1()
      call car_arrived.wait_request( return_from_wait1)
      mutex_server.release()
      receive return_from_wait1()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
    fi
    pin++
    write("passenger #",process_number," is in the car")
    update monitor globals
    fa i := 1 to total_processes
               st i ≠ process_number →
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    if ( vout + 1) * N ≤ pin →
      there are enough passengers in line for the car, signal a car
      var b : bool
      var signaled_proc : int
      this signal and all subsequent signals have code that
      differs from what was given in Chapter 5.  This is
      due to a bug currently in the SR compiler.
      b := passengers_on_car.signal_request( signaled_proc)
      if b →
        mutex_server.pass_mutex( signaled_proc)
        mutex_server.get_exclusion()
      fi
    fi
    # giving up exclusive rights when exiting
    mutex_server.release()
end

proc exit_car()
    get monitor exclusively by calling exclusion server
    mutex_server.get_exclusion()
```

```
    get any updates before processing
    do ?Pass_globals ≠ 0 →
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
    now try to exit car
    if pout ≥ ( vout * N) →
      no car with room, wait
      implementing wait( )
      op return_from_wait2()
      call wait_to_arrive.wait_request( return_from_wait2)
      mutex_server.release()
      receive return_from_wait2()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
    fi
    pout++
    write( "passenger #",process_number," has exited car")
    update monitor globals
    fa i := 1 to total_processes
                st i ≠ process_number →
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    if ( vdest + 1) * N ≤ pout →
      var b : bool
      var signaled_proc : int
      b := passengers_off.signal_request( signaled_proc)
      if b →
        mutex_server.pass_mutex( signaled_proc)
        mutex_server.get_exclusion()
      fi
    fi
    # giving up exclusive rights when exiting
    mutex_server.release()
  end

  proc car_arriving( )
```

```
   get monitor exclusively by calling mutex server
   mutex_server.get_exclusion()
   get any updates before processing
   do ?Pass_globals ≠ 0 →
     receive Pass_globals( pin, pout, vin, vout, vdest)
   od
   car arriving
   write("Car has arrived at station")
   var count := N
   var b : bool := true
     var signaled_proc : int
   do count > 0 and b →
     b := car_arrived.signal_request( signaled_proc)
     if b →
       mutex_server.pass_mutex( signaled_proc)
       mutex_server.get_exclusion()
       count–
       get any updates before processing
       do ?Pass_globals ≠ 0 →
         receive Pass_globals( pin, pout, vin, vout, vdest)
       od
     fi
   od
   vin++
   update monitor globals
   fa i := 1 to total_processes
             st i ≠ process_number →
     send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
     num_update_msgs++
   af
   give up exclusive rights to monitor when exiting
   mutex_server.release()
end

proc car_leaving( )
   get monitor exclusively by calling mutex server
   mutex_server.get_exclusion()
   get any updates before processing
   do ?Pass_globals ≠ 0 →
```

```
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
    if ( vout + 1) * N > pin →
      implementing wait( passengers_on_car)
      op return_from_wait1()
      call passengers_on_car.wait_request( return_from_wait1)
      mutex_server.release()
      receive return_from_wait1()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
    fi
    write("Car has filled, doing ride, now unloading")
    var count := N
    var b : bool := true
    var signaled_proc : int
    do count > 0 and b →
      b :=  wait_to_arrive.signal_request( signaled_proc)
      if b →
        mutex_server.pass_mutex( signaled_proc)
        mutex_server.get_exclusion()
        count–
        get any updates before processing
        do ?Pass_globals ≠ 0 →
          receive Pass_globals( pin, pout, vin, vout, vdest)
        od
      fi
    od
    vout++
    update monitor globals
    fa i := 1 to total_processes
              st i ≠ process_number →
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    give up exclusive rights to monitor when exiting
    mutex_server.release()
  end
```

```
proc car_empty( )
   get monitor exclusively by calling mutex server
   mutex_server.get_exclusion()
   get any updates before processing
   do ?Pass_globals ≠ 0 →
     receive Pass_globals( pin, pout, vin, vout, vdest)
   od
   if (vdest + 1) * N > pout →
     implementing wait( passengers_off)
     op return_from_wait1()
     call passengers_off.wait_request( return_from_wait1)
     mutex_server.release()
     receive return_from_wait1()
     get any updates before processing
     do ?Pass_globals ≠ 0 →
       receive Pass_globals( pin, pout, vin, vout, vdest)
     od
   fi
   vdest++
   write("car is now empty")
   update monitor globals
   fa i := 1 to total_processes
               st i ≠ process_number →
     send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
     num_update_msgs++
   af
   give up exclusive rights to monitor when exiting
   mutex_server.release()
end

final
   writes("Monitor server for process ",process_number)
   write(" sent ",num_update_msgs," update messages")
end

end
```

```
resource to handle in a distributed manner the signal and wait
operations on a monitors condition variable for the SW and SU
protocols.
resource Cond_var
  op wait_request( return_from_wait : cap())      local wait request
                                  from "our" process
  op signal_request( res waiting_proc_num : int)
                returns proc_waiting : bool local signal request
                                  from "our" process
  op wait_update( return_from_wait : cap(); from : int) global wait queue
  op signal_update()                      global signal message
  op empty() returns result : bool           check if global wait Q
                                  is empty
  op links( other_servers[ 1: *] : cap Cond_var)

body Cond_var( me, N : int; name:string(*))

  var local_wait_requests := 0, global_wait_msgs := 0
  var local_signal_requests :=0, global_signal_msgs := 0

  process cond_var_handler
    get the capabilities of the handlers of this condition variable
    var cond_var_handlers[ 1 : N] : cap Cond_var
    receive links( cond_var_handlers)

    do true →
      in wait_request( return_from_wait) →
        this handlers process is requesting a wait on the condition
        variable, notify all cond_var_handlers to put request on
        wait queue
        local_wait_requests++
        fa i := 1 to N →
          send cond_var_handlers[ i].wait_update( return_from_wait,
                                     me)
          global_wait_msgs++
        af
      [] signal_request( waiting_proc_num) returns proc_waiting →
        this handlers process is signaling the condition variable
        check if there are any waiting processes
```

```
      local_signal_requests++
      if ?wait_update ≠ 0 →
        there is a wait request pending
        notify other handlers of signal
        fa i := 1 to N st i ≠ me →
          send cond_var_handlers[ i].signal_update()
          global_signal_msgs++
        af
        get operation from local wait_queue and send to it
        in wait_update( return_oper, from) →
          send return_oper()
          waiting_proc_num := from
          global_signal_msgs++
        ni
        proc_waiting := true
      [] else →
        no waits pending, drop signal on floor
        proc_waiting := false
      fi
    [] empty() returns result →
      result := ( ?wait_update = 0)
    [] signal_update() →
      a process has signaled the condition variable
      remove the first element off of wait queue
      if ?wait_update > 0 →
        in wait_update( temp, from) → ni
      [] else →
        write("ERROR: erroneous signal_update received")
      fi
    ni
  od
end

final
  write("For ",name," handler for process #",me,"there were")
  writes(local_wait_requests," local wait requests and ")
  write(global_wait_msgs," global wait messages")
  writes(local_signal_requests," local signal requests and ")
  write(global_signal_msgs," global signal messages")
```

end
end

This resource is a server which is associated with a node and *implements* the Suzuki and Kasami mutual exclusion algorithm *from* the 85 ACM paper. The algorithm has been modified for *use* with the SW monitor protocol. This modification allows *a* process to receive the Privilege without having explicitly *requested* it (i.e., waiting on a condition variable).

```
resource Mutex_server
   op invoke()
   op release()
   op privilege( Q[1:*]:int; front, rear, count: int; LN[ 1 : *] : int)
   op request( from, its_seq_num:int)
   op pass_mutex( signaled_proc : int)
   op links( others[1:*]: cap Mutex_server)
   op get_exclusion() routine called by node to get mut. excl.

body Mutex_server( me, total_processes : int)
   var RN[ 1 : total_processes] :int := ([total_processes]0)
   var LN[ 1 : total_processes] :int := ([total_processes]0)
   var HavePrivilege : bool := false
   var Q [ 0 : total_processes-1] : int
   var front := 0, rear := 0, count := 0

   op have_crit()
   var other_servers[ 1 : total_processes]: cap Mutex_server
   var release_for_signal : bool
   var pass_priv_to : int
   op pass_or_release()

   var obtain_msgs := 0, num_releases := 0

   proc get_exclusion()
         send invoke()
         receive have_crit()
         obtain_msgs+
   end

   process server
         receive links( other_servers)
         if me = 1 →
```

```
      if we are node 1's server, then we initially have the privilege
     HavePrivilege := true
   fi
      do true →
         in invoke() →
               if not HavePrivilege →
                 RN[ me]+
                 fa j := 1 to total_processes st  j ≠ me →
                    send other_servers[ j].request( me, RN[ me])
                    obtain_msgs+
                 af
                 wait for privilege
                 receive privilege( Q, front, rear, count, LN)
                 HavePrivilege := true
                 obtain_msgs+
               fi
               send have_crit()
               call pass_or_release()
         [] request( from, its_seq_num) →
               #
               #
               RN[ from] := max( RN[ from], its_seq_num)
               if HavePrivilege and ( RN[ from] = LN[ from] + 1) →
                 HavePrivilege := false
                 send other_servers[ from].privilege( Q, front,
                                                      rear, count, LN)
               fi
         [] privilege( temp_Q, temp_front, temp_rear, temp_count, temp_LN) →
               obtain_msgs+
               Q := temp_Q
               front := temp_front
               rear := temp_rear
               count := temp_count
               LN := temp_LN
               HavePrivilege := true
               call pass_or_release()
         ni
      od
end
```

```
proc pass_or_release()
    do true →
      in request( from, its_seq_num) →
        update request number for requesting node
        RN[ from] := max( RN[ from], its_seq_num)
      [] release() →
        node is releasing mutual exclusion
        release_for_signal := false
        num_releases+
        exit
      [] pass_mutex( signaled_proc) →
        release_for_signal := true
        pass_priv_to := signaled_proc
        num_releases+
        exit
      ni
    od
    LN[ me] := RN[ me]
    fa j := 1 to total_processes st j ≠ me →
      if not in(Q,j) and (RN[j]=LN[j]+1)then Q:=append(Q,j)
      if RN[ j] = (LN[j] +1) →
        var found : bool := false
        var search_count : int := 0
        var search_ptr := front
        do search_count < count →
            if Q[ search_ptr] = j →
          found := true
              exit
            fi
            search_count+
            search_ptr := (search_ptr + 1) % total_processes
        od
        if not found →
            Q[ rear] := j
            rear := (rear + 1) % total_processes
            count+
        fi
      fi
    af
```

```
        if not release_for_signal →
          if count ≠ 0 →
          HavePrivilege := false
          send other_servers[ Q[ front.privilege( Q,
                                        (front+1)total_processes,
                                        rear, count-1, LN)
          fi
        [] else →
        HavePrivilege := false
        send other_servers[ pass_priv_to].privilege( Q, front,
                                                rear, count, LN)
        fi
    end

    final
      write("For process",me," mutex handler there were ")
      writes(obtain_msgs," messages to get mutex and ")
      write(num_releases," to release mutex.")
    end
end
```

```
Script started on Sat Jul 22 09:52:03 1989
ivy% sw.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Car has arrived at station
passenger # 1  is in the car
passenger # 2  is in the car
passenger # 3  is in the car
passenger # 4  is in the car
Car has filled, doing ride, now unloading
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
Car has arrived at station
passenger # 5  is in the car
passenger # 6  is in the car
passenger # 7  is in the car
passenger # 8  is in the car
Car has filled, doing ride, now unloading
passenger # 5  has exited car
passenger # 6  has exited car
passenger # 7  has exited car
passenger # 8  has exited car
car is now empty
Car has arrived at station
passenger # 1  is in the car
passenger # 2  is in the car
passenger # 3  is in the car
passenger # 4  is in the car
Car has filled, doing ride, now unloading
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
Car has arrived at station
```

passenger # 5 is in the car
passenger # 6 is in the car
passenger # 7 is in the car
passenger # 8 is in the car
Car has filled, doing ride, now unloading
passenger # 5 has exited car
passenger # 6 has exited car
passenger # 7 has exited car
passenger # 8 has exited car
car is now empty
Car has arrived at station
passenger # 1 is in the car
passenger # 2 is in the car
passenger # 3 is in the car
passenger # 4 is in the car
Car has filled, doing ride, now unloading
passenger # 1 has exited car
passenger # 2 has exited car
passenger # 3 has exited car
passenger # 4 has exited car
car is now empty
Car has arrived at station
passenger # 5 is in the car
passenger # 6 is in the car
passenger # 7 is in the car
passenger # 8 is in the car
Car has filled, doing ride, now unloading
passenger # 5 has exited car
passenger # 6 has exited car
passenger # 7 has exited car
passenger # 8 has exited car
car is now empty
Car has arrived at station
passenger # 1 is in the car
passenger # 2 is in the car
passenger # 3 is in the car
passenger # 4 is in the car
Car has filled, doing ride, now unloading
passenger # 1 has exited car

passenger # 2 has exited car
passenger # 3 has exited car
passenger # 4 has exited car
car is now empty
Car has arrived at station
passenger # 5 is in the car
passenger # 6 is in the car
passenger # 7 is in the car
passenger # 8 is in the car
Car has filled, doing ride, now unloading
passenger # 5 has exited car
passenger # 6 has exited car
passenger # 7 has exited car
passenger # 8 has exited car
car is now empty
Car has arrived at station
passenger # 1 is in the car
passenger # 2 is in the car
passenger # 3 is in the car
passenger # 4 is in the car
Car has filled, doing ride, now unloading
passenger # 1 has exited car
passenger 1 is done
passenger # 2 has exited car
passenger 2 is done
passenger # 3 has exited car
passenger 3 is done
passenger # 4 has exited car
passenger 4 is done
car is now empty
Car has arrived at station
passenger # 5 is in the car
passenger # 6 is in the car
passenger # 7 is in the car
passenger # 8 is in the car
Car has filled, doing ride, now unloading
passenger # 5 has exited car
passenger 5 is done
passenger # 6 has exited car

passenger 6 is done
passenger # 7 has exited car
passenger 7 is done
passenger # 8 has exited car
passenger 8 is done
car is now empty
car is done
For passengers_in_car handler for process # 1 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 1 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 1 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 1 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 1 mutex handler there were
101 messages to get mutex and 20 to release mutex.
Monitor server for process 1 sent 80 update messages
For passengers_in_car handler for process # 2 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 2 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 2 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 2 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 2 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 2 sent 80 update messages
For passengers_in_car handler for process # 3 there were
0 local wait requests and 0 global wait messages

0 local signal requests and 0 global signal messages
For car_arrived handler for process # 3 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 3 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 3 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 3 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 3 sent 80 update messages
For passengers_in_car handler for process # 4 there were
0 local wait requests and 0 global wait messages
5 local signal requests and 0 global signal messages
For car_arrived handler for process # 4 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 4 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 4 there were
0 local wait requests and 0 global wait messages
5 local signal requests and 0 global signal messages
For process 4 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 4 sent 80 update messages
For passengers_in_car handler for process # 5 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 5 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 5 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 5 there were
0 local wait requests and 0 global wait messages

0 local signal requests and 0 global signal messages
For process 5 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 5 sent 80 update messages
For passengers_in_car handler for process # 6 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 6 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 6 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 6 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 6 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 6 sent 80 update messages
For passengers_in_car handler for process # 7 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 7 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 7 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 7 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 7 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 7 sent 80 update messages
For passengers_in_car handler for process # 8 there were
0 local wait requests and 0 global wait messages
5 local signal requests and 0 global signal messages
For car_arrived handler for process # 8 there were
5 local wait requests and 45 global wait messages

0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 8 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 8 there were
0 local wait requests and 0 global wait messages
5 local signal requests and 0 global signal messages
For process 8 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 8 sent 80 update messages
For passengers_in_car handler for process # 9 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 9 there were
0 local wait requests and 0 global wait messages
40 local signal requests and 360 global signal messages
For wait_to_arrive handler for process # 9 there were
0 local wait requests and 0 global wait messages
40 local signal requests and 360 global signal messages
For passengers_off handler for process # 9 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 9 mutex handler there were
1001 messages to get mutex and 110 to release mutex.
Monitor server for process 9 sent 240 update messages
ivy% exit
ivy%
script done on Sat Jul 22 09:52:21 1989

```
This resource implements the monitor server for the
roller coaster problem using distributed monitors.
This version uses the signal and urgent wait protocol.
resource Monitor_server
   import Mutex_server  distributed mutual exclusion server
   import Roller_globals
   import Cond_var     handles monitor signal and wait calls
                    in a distrubuted fashion
   op links( others[ 1:* ] : cap Monitor_server)
   op board_car()
   op exit_car()
   op car_arriving()
   op car_leaving()
   op car_empty()
   op Pass_globals Monitor_Pass_globals

body Monitor_server( mutex_server: cap Mutex_server;
                process_number, total_processes : int;
                car_arrived, passengers_on_car,
                wait_to_arrive, passengers_off : cap Cond_var)


   var pin := 0, pout := 0, vin := 0, vout := 0, vdest := 0
   var others [ 1 : total_processes] : cap Monitor_server

   var num_update_msgs := 0

   initial
     reply
     receive links( others)
   end

   proc board_car()
      get monitor exclusively by calling exclusion server
      mutex_server.get_exclusion()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
         receive Pass_globals( pin, pout, vin, vout, vdest)
      od
```

```
    if pin ≥ vin * N →
      # wait( car_arrived)
      op return_from_wait1()
      call car_arrived.wait_request( return_from_wait1)
      mutex_server.release()
      receive return_from_wait1()
      mutex_server.get_exclusion()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
    fi
    pin++
    write("passenger #",process_number," is in the car")
    update monitor globals
    fa i := 1 to total_processes
              st i ≠ process_number →
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    if ( vout + 1) * N ≤ pin →
      there are enough passengers in line for the car, signal a car
      var b : bool
      var signaled_proc : int
      this signal and all subsequent signals have code that
      differs from what was given in Chapter 5.  This is
      due to a bug currently in the SR compiler.
      b := passengers_on_car.signal_request( signaled_proc)
      if b →
        mutex_server.pass_mutex( signaled_proc)
        mutex_server.get_exclusion()
      fi
    fi
    # giving up exclusive rights when exiting
    mutex_server.release()
end

proc exit_car()
    get monitor exclusively by calling exclusion server
```

```
    mutex_server.get_exclusion()
    get any updates before processing
    do ?Pass_globals ≠ 0 →
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
    now try to exit car
    if pout ≥ ( vout * N) →
      no car with room, wait
      implementing wait( )
      op return_from_wait2()
      call wait_to_arrive.wait_request( return_from_wait2)
      mutex_server.release()
      receive return_from_wait2()
      mutex_server.get_exclusion()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
    fi
    pout++
    write( "passenger #",process_number," has exited car")
    update monitor globals
    fa i := 1 to total_processes
              st i ≠ process_number →
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    if ( vdest + 1) * N ≤ pout →
      var b : bool
      var signaled_proc : int
      b := passengers_off.signal_request( signaled_proc)
      if b →
        mutex_server.pass_mutex( signaled_proc)
        mutex_server.get_exclusion()
      fi
    fi
    # giving up exclusive rights when exiting
    mutex_server.release()
  end
```

```
proc car_arriving( )
   get monitor exclusively by calling mutex server
   mutex_server.get_exclusion()
   get any updates before processing
   do ?Pass_globals ≠ 0 →
     receive Pass_globals( pin, pout, vin, vout, vdest)
   od
   car arriving
   vin++
   write("Car has arrived at station")
   update monitor globals
   fa i := 1 to total_processes
                st i ≠ process_number →
     send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
     num_update_msgs++
   af
   var count := N
   var b : bool := true
     var signaled_proc : int
   do count > 0 and b →
     b := car_arrived.signal_request( signaled_proc)
     if b →
       mutex_server.pass_mutex( signaled_proc)
       mutex_server.get_exclusion()
       count–
     fi
   od
   give up exclusive rights to monitor when exiting
   mutex_server.release()
end

proc car_leaving( )
   get monitor exclusively by calling mutex server
   mutex_server.get_exclusion()
   get any updates before processing
   do ?Pass_globals ≠ 0 →
     receive Pass_globals( pin, pout, vin, vout, vdest)
   od
   if ( vout + 1) * N > pin →
```

```
    implementing wait( passengers_on_car)
    op return_from_wait1()
    call passengers_on_car.wait_request( return_from_wait1)
    mutex_server.release()
    receive return_from_wait1()
    mutex_server.get_exclusion()
    get any updates before processing
    do ?Pass_globals ≠ 0 →
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
  fi
  vout++
  write("Car has filled, doing ride, now unloading")
  update monitor globals
  fa i := 1 to total_processes
              st i ≠ process_number →
    send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
    num_update_msgs++
  af
  var count := N
  var b : bool := true
  var signaled_proc : int
  do count > 0 and b →
    b :=  wait_to_arrive.signal_request( signaled_proc)
    if b →
      mutex_server.pass_mutex( signaled_proc)
      mutex_server.get_exclusion()
      count–
    fi
  od
  give up exclusive rights to monitor when exiting
  mutex_server.release()
end

proc car_empty( )
  get monitor exclusively by calling mutex server
  mutex_server.get_exclusion()
  get any updates before processing
  do ?Pass_globals ≠ 0 →
```

```
      receive Pass_globals( pin, pout, vin, vout, vdest)
    od
    if (vdest + 1) * N > pout →
      implementing wait( passengers_off)
      op return_from_wait1()
      call passengers_off.wait_request( return_from_wait1)
      mutex_server.release()
      receive return_from_wait1()
      mutex_server.get_exclusion()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( pin, pout, vin, vout, vdest)
      od
    fi
    vdest++
    write("car is now empty")
    update monitor globals
    fa i := 1 to total_processes
                st i ≠ process_number →
      send others[ i].Pass_globals( pin, pout, vin, vout, vdest)
      num_update_msgs++
    af
    give up exclusive rights to monitor when exiting
    mutex_server.release()
  end

  final
    writes("Monitor server for process ",process_number)
    write(" sent ",num_update_msgs," update messages")
  end
end
```

```
Script started on Sat Jul 22 09:52:28 1989
ivy% su.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Car has arrived at station
passenger # 1  is in the car
passenger # 2  is in the car
passenger # 3  is in the car
passenger # 4  is in the car
Car has filled, doing ride, now unloading
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
Car has arrived at station
passenger # 5  is in the car
passenger # 6  is in the car
passenger # 7  is in the car
passenger # 8  is in the car
Car has filled, doing ride, now unloading
passenger # 5  has exited car
passenger # 6  has exited car
passenger # 7  has exited car
passenger # 8  has exited car
car is now empty
Car has arrived at station
passenger # 1  is in the car
passenger # 2  is in the car
passenger # 3  is in the car
passenger # 4  is in the car
Car has filled, doing ride, now unloading
passenger # 1  has exited car
passenger # 2  has exited car
passenger # 3  has exited car
passenger # 4  has exited car
car is now empty
Car has arrived at station
```

passenger # 5 is in the car
passenger # 6 is in the car
passenger # 7 is in the car
passenger # 8 is in the car
Car has filled, doing ride, now unloading
passenger # 5 has exited car
passenger # 6 has exited car
passenger # 7 has exited car
passenger # 8 has exited car
car is now empty
Car has arrived at station
passenger # 1 is in the car
passenger # 2 is in the car
passenger # 3 is in the car
passenger # 4 is in the car
Car has filled, doing ride, now unloading
passenger # 1 has exited car
passenger # 2 has exited car
passenger # 3 has exited car
passenger # 4 has exited car
car is now empty
Car has arrived at station
passenger # 5 is in the car
passenger # 6 is in the car
passenger # 7 is in the car
passenger # 8 is in the car
Car has filled, doing ride, now unloading
passenger # 5 has exited car
passenger # 6 has exited car
passenger # 7 has exited car
passenger # 8 has exited car
car is now empty
Car has arrived at station
passenger # 1 is in the car
passenger # 2 is in the car
passenger # 3 is in the car
passenger # 4 is in the car
Car has filled, doing ride, now unloading
passenger # 1 has exited car

passenger # 2 has exited car
passenger # 3 has exited car
passenger # 4 has exited car
car is now empty
Car has arrived at station
passenger # 5 is in the car
passenger # 6 is in the car
passenger # 7 is in the car
passenger # 8 is in the car
Car has filled, doing ride, now unloading
passenger # 5 has exited car
passenger # 6 has exited car
passenger # 7 has exited car
passenger # 8 has exited car
car is now empty
Car has arrived at station
passenger # 1 is in the car
passenger # 2 is in the car
passenger # 3 is in the car
passenger # 4 is in the car
Car has filled, doing ride, now unloading
passenger # 1 has exited car
passenger 1 is done
passenger # 2 has exited car
passenger 2 is done
passenger # 3 has exited car
passenger 3 is done
passenger # 4 has exited car
passenger 4 is done
car is now empty
Car has arrived at station
passenger # 5 is in the car
passenger # 6 is in the car
passenger # 7 is in the car
passenger # 8 is in the car
Car has filled, doing ride, now unloading
passenger # 5 has exited car
passenger 5 is done
passenger # 6 has exited car

passenger 6 is done
passenger # 7 has exited car
passenger 7 is done
passenger # 8 has exited car
passenger 8 is done
car is now empty
car is done
For passengers_in_car handler for process # 1 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 1 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 1 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 1 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 1 mutex handler there were
101 messages to get mutex and 20 to release mutex.
Monitor server for process 1 sent 80 update messages
For passengers_in_car handler for process # 2 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 2 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 2 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 2 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 2 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 2 sent 80 update messages
For passengers_in_car handler for process # 3 there were
0 local wait requests and 0 global wait messages

0 local signal requests and 0 global signal messages
For car_arrived handler for process # 3 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 3 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 3 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 3 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 3 sent 80 update messages
For passengers_in_car handler for process # 4 there were
0 local wait requests and 0 global wait messages
5 local signal requests and 0 global signal messages
For car_arrived handler for process # 4 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 4 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 4 there were
0 local wait requests and 0 global wait messages
5 local signal requests and 0 global signal messages
For process 4 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 4 sent 80 update messages
For passengers_in_car handler for process # 5 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 5 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 5 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 5 there were
0 local wait requests and 0 global wait messages

0 local signal requests and 0 global signal messages
For process 5 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 5 sent 80 update messages
For passengers_in_car handler for process # 6 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 6 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 6 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 6 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 6 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 6 sent 80 update messages
For passengers_in_car handler for process # 7 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 7 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 7 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 7 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 7 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 7 sent 80 update messages
For passengers_in_car handler for process # 8 there were
0 local wait requests and 0 global wait messages
5 local signal requests and 0 global signal messages
For car_arrived handler for process # 8 there were
5 local wait requests and 45 global wait messages

0 local signal requests and 0 global signal messages
For wait_to_arrive handler for process # 8 there were
5 local wait requests and 45 global wait messages
0 local signal requests and 0 global signal messages
For passengers_off handler for process # 8 there were
0 local wait requests and 0 global wait messages
5 local signal requests and 0 global signal messages
For process 8 mutex handler there were
110 messages to get mutex and 20 to release mutex.
Monitor server for process 8 sent 80 update messages
For passengers_in_car handler for process # 9 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For car_arrived handler for process # 9 there were
0 local wait requests and 0 global wait messages
40 local signal requests and 360 global signal messages
For wait_to_arrive handler for process # 9 there were
0 local wait requests and 0 global wait messages
40 local signal requests and 360 global signal messages
For passengers_off handler for process # 9 there were
0 local wait requests and 0 global wait messages
0 local signal requests and 0 global signal messages
For process 9 mutex handler there were
1001 messages to get mutex and 110 to release mutex.
Monitor server for process 9 sent 240 update messages
ivy% exit
ivy%
script done on Sat Jul 22 09:52:46 1989

```
resource main
   import disk_scheduler
   import useds

body main()
   var num_users : int := 6
   var ds : cap disk_scheduler
   var users[ 1 : num_users] : cap useds

   process start
      op done()
      create the disk scheduler
      ds := create disk_scheduler()
      create the user processes
      fa i := 1 to num_users →
         users[ i] := create useds( i, ds, done)
      af
      wait for the users to complete
      fa i := 1 to num_users →
         receive done()
      af
      fa i := 1 to num_users →
         destroy users[ i]
      af
      destroy ds
   end
end
```

Process to make use of the centralized disk scheduler

```
resource useds
        import disk_scheduler

body useds( me : int; ds : cap disk_scheduler; done:cap())

    process disk_user
        var cylinder := me % 3
        fa i:= 1 to 3 →
           write ("Process ",me," requesting cylinder",cylinder)
           ds.request( cylinder, me)
           cylinder := (cylinder + 4) % 3
           ds.release( me)
        af
        write("Process ",me," exiting")
        send done()
    end
end
```

```
resource to implement a CSCAN disk scheduler
using a centralized monitor
resource disk_scheduler
   op request( cyl, from: int)
   op release( from : int)

body disk_scheduler( )
   var position := -1
   var c := 0, n := 1
   op mutex()
   type return_rec = rec ( wait: cap( return_from_wait: cap();priority : int);
                  signal: cap();
                  empty: cap() returns results: bool;
                  minrank: cap( success:bool) returns rank: int;
                  wait_ranks: cap( priority : int))
   var scan[ 0 : 1 ] : return_rec
   op cond_var() returns signal_wait : return_rec

   var num_releases := 0, num_wait_msgs := 0, num_signal_msgs := 0
   var num_wait_ranks_msgs := 0

   initial
      create the two condition variables not_full and not_empty
      scan[ 0] := cond_var()
      scan[ 1] := cond_var()
      initialize the mutual exclusion semaphore mutex
      send mutex()
   end

   proc request( cyl, from)
      get monitor exclusively by waiting on mutex
      receive mutex()
      write("Receiving request from ",from," for cylinder ",cyl)
      if position = -1 →
         position := cyl
      [] position ≠ -1 and cyl > position →
         # implementing wait( scan[ c], cyl)
         op return_from_wait1()
         send scan[ c].wait( return_from_wait1, cyl)
```

```
        send scan[ c].wait_ranks( cyl)
        send mutex()
        num_releases++
        receive return_from_wait1()
        receive mutex()
     [] position ≠ -1 and cyl ≤ position →
        # implementing wait( scan[ n], cyl)
        op return_from_wait2()
        send scan[ n].wait( return_from_wait2, cyl)
        send scan[ n].wait_ranks( cyl)
        send mutex()
        num_releases++
        receive return_from_wait2()
        receive mutex()
    fi
    write("access to cylinder",cyl,"being given to process",from)
    give up exclusive rights to monitor when exiting
    send mutex()
    num_releases++
end

proc release( from)
    get monitor exclusively by waiting on mutex
    var success : bool
    receive mutex()
    write("Process ",from,"releasing access to cylinder", position)
    if not scan[ c].empty() →
        position := scan[ c].minrank( success)
     [] scan[ c].empty() and not scan[ n].empty() →
        c :=: n
        position := scan[ c].minrank( success)
     [] scan[ c].empty() and scan[ n].empty() →
        position := -1
    fi
    send scan[ c].signal()
    # giving up exclusive rights when exiting
    send mutex()
    num_releases++
end
```

```
proc cond_var() returns signal_wait
process to implement the wait and signal operations on
# a condition variable
  op wait( return_to_caller:cap(); priority: int)
  op signal()
  op empty() returns results:bool
  op minrank( res success: bool) returns rank:int
  op wait_ranks( priority : int)
  signal_wait.wait := wait
  signal_wait.signal := signal
  signal_wait.empty := empty
  signal_wait.minrank := minrank
  signal_wait.wait_ranks := wait_ranks
  return the wait, signal, empty and minrank functions
  reply
  do true →
    in wait( return_to_caller, priority) and ?signal ≠ 0 by priority →
      execute wait only if there is a signal
      # receive the signal and then get mutual exclusion
      # before returning
      num_wait_msgs++
      in wait_ranks( priority) by priority →
      ni
      num_wait_ranks_msgs++
      receive signal()
      num_signal_msgs++
      send return_to_caller()
      num_signal_msgs++
    [] signal() and ?wait = 0 →
      # drop a sent signal if there are no waits
      num_signal_msgs++
    [] empty() returns results →
      results := ( ?wait = 0)
    [] minrank( success) returns rank →
      if ?wait_ranks ≠ 0 →
        in wait_ranks( priority) by priority →
          num_wait_ranks_msgs++
          send wait_ranks( priority)
          rank := priority
```

```
                success := true
            ni
          [] else →
            write("minrank called with empty cond_var")
            success := false
            stop
        fi
      ni
    od
  end

  final
    write("There were ",num_releases," messages to release mutex and")
    writes( num_wait_msgs," wait messages and ")
    writes( num_signal_msgs," signal messages and ")
    write( num_wait_ranks_msgs," messages to wait_ranks")
  end
end
```

```
Script started on Fri Jul 21 09:55:38 1989
ivy% a.out
Process  1  requesting cylinder 1
Process  2  requesting cylinder 2
Process  3  requesting cylinder 0
Process  4  requesting cylinder 1
Process  5  requesting cylinder 2
Process  6  requesting cylinder 0
Receiving request from  1  for cylinder  1
access to cylinder 1 being given to process 1
Receiving request from  2  for cylinder  2
Receiving request from  3  for cylinder  0
Receiving request from  4  for cylinder  1
Receiving request from  5  for cylinder  2
Receiving request from  6  for cylinder  0
Process  1 releasing access to cylinder 1
Process  1  requesting cylinder 2
access to cylinder 2 being given to process 2
Receiving request from  1  for cylinder  2
Process  2 releasing access to cylinder 2
Process  2  requesting cylinder 0
access to cylinder 2 being given to process 5
Receiving request from  2  for cylinder  0
Process  5 releasing access to cylinder 2
Process  5  requesting cylinder 0
access to cylinder 0 being given to process 3
Receiving request from  5  for cylinder  0
Process  3 releasing access to cylinder 0
Process  3  requesting cylinder 1
access to cylinder 0 being given to process 6
Receiving request from  3  for cylinder  1
Process  6 releasing access to cylinder 0
Process  6  requesting cylinder 1
access to cylinder 0 being given to process 2
Receiving request from  6  for cylinder  1
Process  2 releasing access to cylinder 0
Process  2  requesting cylinder 1
access to cylinder 1 being given to process 4
Receiving request from  2  for cylinder  1
```

```
Process  4 releasing access to cylinder 1
Process  4  requesting cylinder 2
access to cylinder 1 being given to process 3
Receiving request from  4  for cylinder  2
Process  3 releasing access to cylinder 1
Process  3  requesting cylinder 2
access to cylinder 1 being given to process 6
Receiving request from  3  for cylinder  2
Process  6 releasing access to cylinder 1
Process  6  requesting cylinder 2
access to cylinder 2 being given to process 1
Receiving request from  6  for cylinder  2
Process  1 releasing access to cylinder 2
Process  1  requesting cylinder 0
access to cylinder 2 being given to process 4
Receiving request from  1  for cylinder  0
Process  4 releasing access to cylinder 2
Process  4  requesting cylinder 0
access to cylinder 2 being given to process 3
Receiving request from  4  for cylinder  0
Process  3 releasing access to cylinder 2
Process  3  exiting
access to cylinder 0 being given to process 5
Process  5 releasing access to cylinder 0
Process  5  requesting cylinder 1
access to cylinder 0 being given to process 1
Receiving request from  5  for cylinder  1
Process  1 releasing access to cylinder 0
Process  1  exiting
access to cylinder 0 being given to process 4
Process  4 releasing access to cylinder 0
Process  4  exiting
access to cylinder 1 being given to process 2
Process  2 releasing access to cylinder 1
Process  2  exiting
access to cylinder 1 being given to process 5
Process  5 releasing access to cylinder 1
Process  5  exiting
access to cylinder 2 being given to process 6
```

```
Process  6 releasing access to cylinder 2
Process  6  exiting
There were  53  requests for mutex and
17 wait messages and 35 signal messages and 34  messages to wait_ranks
ivy% exit
ivy%
script done on Fri Jul 21 09:55:46 1989
```

```
resource main
   import CSCAN_monitor
   import Pri_Cond_var
   import Mutex_server
   import useds

body main()
   var num_users : int := 6
   var ds[ 1 : num_users] : cap CSCAN_monitor
   var scan0[ 1 : num_users], scan1[ 1 : num_users] : cap Pri_Cond_var
   var mutex_servers[ 1 : num_users] : cap Mutex_server
   var users[ 1 : num_users] : cap useds

   process start
      op done()
      create disk scheduler and mutex server and
      prioritized condition variable handlers
      fa i := 1 to num_users →
        scan0[ i] := create Pri_Cond_var( i, num_users, "scan[ 0]")
        scan1[ i] := create Pri_Cond_var( i, num_users, "scan[ 1]")
        mutex_servers[ i] := create Mutex_server( i, num_users)
        ds[ i] := create CSCAN_monitor( mutex_servers[ i], i, num_users,
                                        scan0[ i], scan1[ i])
      af
      create the user processes
      fa i := 1 to num_users →
        users[ i] := create useds( i, ds[ i], done)
      af
      fa i := 1 to num_users →
        send scan0[ i].links(scan0)
        send scan1[ i].links(scan1)
        send mutex_servers[ i].links( mutex_servers)
        send ds[ i].links(ds)
      af
      wait for users to complete
      fa i := 1 to num_users →
         receive done()
      af
      fa i := 1 to num_users →
```

```
        destroy scan0[ i]
        destroy scan1[ i]
        destroy mutex_servers[ i]
        destroy ds[ i]
        destroy users[ i]
      af
  end
end

resource useds
  import CSCAN_monitor

body useds( me : int; ds : cap CSCAN_monitor; done:cap())

  process disk_user
    var cylinder := me % 3
    fa i:= 1 to 3 →
      write ("Process ",me," requesting cylinder",cylinder)
      ds.request( cylinder)
      cylinder := (cylinder + 4) % 3
      ds.release()
    af
    write("Process ",me," exiting")
    send done()
  end

end
```

```
resource to implement a CSCAN disk scheduler using
a distributed monitor with prioritizd condition variables
resource CSCAN_monitor
    import Mutex_server
    import CSCAN_globals
    import Pri_Cond_var
    op links( others[ 1:*] : cap CSCAN_monitor)
    op request( cyl: int)
    op release()
    op Pass_globals Monitor_Pass_globals

body CSCAN_monitor( mutex_server : cap Mutex_server;
                    process_number, total_processes : int;
                    scan0, scan1 : cap Pri_Cond_var )
    var position := -1
    var c := 0, n := 1
    var others [ 1 : total_processes] : cap CSCAN_monitor
    var scan[ 0 : 1] : cap Pri_Cond_var

    initial
        reply
        receive links( others)
        scan[ 0] := scan0
        scan[ 1] := scan1
    end

    proc request( cyl)
        get monitor exclusively by waiting on mutex
        mutex_server.get_exclusion()
        get any updates before processing
        do ?Pass_globals ≠ 0 →
           receive Pass_globals( position, c, n)
        od
        write("Receiving request from ",process_number," for cylinder ",cyl)
        if position = -1 →
            position := cyl
        [] position ≠ -1 and cyl > position →
            # implementing wait( scan[ c], cyl)
            op return_from_wait1()
```

```
        call scan[ c].wait_request( return_from_wait1, cyl)
        mutex_server.release()
        receive return_from_wait1()
        mutex_server.get_exclusion()
        do ?Pass_globals ≠ 0 →
          receive Pass_globals( position, c, n)
        od
      [] position ≠ -1 and cyl ≤ position →
        # implementing wait( scan[ n], cyl)
        op return_from_wait2()
        call scan[ n].wait_request( return_from_wait2, cyl)
        mutex_server.release()
        receive return_from_wait2()
        mutex_server.get_exclusion()
        do ?Pass_globals ≠ 0 →
          receive Pass_globals( position, c, n)
        od
      fi
      writes("access to cylinder ",cyl," being given to")
      write(" process",process_number)
      update monitor globals
      fa i := 1 to total_processes
                  st i ≠ process_number →
        send others[ i].Pass_globals( position, c, n)
      af
      give up exclusive rights to monitor when exiting
      mutex_server.release()
    end

    proc release()
      get monitor exclusively by waiting on mutex
      mutex_server.get_exclusion()
      get any updates before processing
      do ?Pass_globals ≠ 0 →
        receive Pass_globals( position, c, n)
      od
      writes("Process ",process_number," releasing access")
      write(" to cyclinder", position)
      var success : bool
```

```
        if not scan[ c].empty() →
            position := scan[ c].minrank( success)
        [] scan[ c].empty() and not scan[ n].empty() →
            c :=: n
            position := scan[ c].minrank( success)
        [] scan[ c].empty() and scan[ n].empty() →
            position := -1
        fi
        send scan[ c].signal_request()
        update monitor globals
        fa i := 1 to total_processes
                st i ≠ process_number →
           send others[ i].Pass_globals( position, c, n)
        af
        # giving up exclusive rights when exiting
        mutex_server.release()
    end
end

global CSCAN_globals
        op type for monitor routines to share the monitor variables
        optype Monitor_Pass_globals = ( position, c, n : int)
end CSCAN_globals
```

```
resource to handle in a distributed manner
operations on a monitors condition variable with prioritized wait

resource Pri_Cond_var
  op wait_request( return_from_wait : cap(); priority : int)
                              local wait request
                              from "our" process
  op signal_request()                local signal request
                              from "our" process
  op wait_update( return_from_wait : cap(); priority : int)
                              global wait queue
  op signal_update()                 global signal message
  op empty() returns result : bool        check if global wait Q is empty
  op minrank( success : bool) returns rank : int
                              local minrank call
  op wait_ranks( priority : int)          global wait ranks Q
  op links( other_servers[ 1: *] : cap Pri_Cond_var)

body Pri_Cond_var( me, N : int; name:string(*))

  var local_wait_requests := 0, global_wait_msgs := 0
  var local_signal_requests :=0, global_signal_msgs := 0
  var wait_ranks_msgs := 0

  process cond_var_handler
    get the capabilities of the handlers of this condition variable
    var cond_var_handlers[ 1 : N] : cap Pri_Cond_var
    receive links( cond_var_handlers)

    do true →
      in wait_request( return_from_wait, priority) →
        this handlers process is requesting a wait on the condition
        variable, notify all cond_var_handlers to put request on
        wait queue
        local_wait_requests++
        fa i := 1 to N →
          send cond_var_handlers[ i].wait_update( return_from_wait,
                                    priority)
          send cond_var_handlers[ i].wait_ranks( priority)
```

```
    global_wait_msgs++
    wait_ranks_msgs++
  af
[] signal_request() →
  this handlers process is signaling the condition variable
  check if there are any waiting processes
  local_signal_requests++
  if ?wait_update ≠ 0 →
    there is a wait request pending
    notify other handlers of signal
    fa i := 1 to N st i ≠ me →
      send cond_var_handlers[ i].signal_update()
      global_signal_msgs++
    af
    get operation from global wait_queue and send to it
    in wait_update( return_oper, priority) by priority →
      send return_oper()
      global_signal_msgs++
    ni
    in wait_ranks( priority) by priority →
    ni
  [] else →
    no waits pending, drop signal on floor
  fi
[] empty() returns result →
  result := ( ?wait_update = 0)
[] signal_update() →
  a process has signaled the condition variable
  remove the first element off of wait queue
  if ?wait_update > 0 →
    in wait_update( return_oper, priority) by priority → ni
    in wait_ranks( priority) by priority → ni
  [] else →
    write("ERROR: erroneous signal_update received")
  fi
[] minrank( success) returns rank →
  if ?wait_ranks ≠ 0 →
    in wait_ranks( priority) by priority →
      send wait_ranks( priority)
```

```
            wait_ranks_msgs++
            rank := priority
            success := true
          ni
        [] else →
          success := false
        fi
      ni
    od
  end

  final
    write("For ",name," handler for process #",me,"there were")
    writes(local_wait_requests," local wait requests and ")
    write(global_wait_msgs," global wait messages")
    writes(local_signal_requests," local signal requests and ")
    write(global_signal_msgs," global signal messages")
    write("and ",wait_ranks_msgs," wait_ranks messages")
  end
end
```

*** Mutex server identical to server for distributed sc above

```
Script started on Fri Jul 21 10:15:34 1989
ivy% a.out
Process  1  requesting cylinder 1
Process  2  requesting cylinder 2
Process  3  requesting cylinder 0
Process  4  requesting cylinder 1
Process  5  requesting cylinder 2
Process  6  requesting cylinder 0
Receiving request from  1  for cylinder  1
access to cylinder 1 being given to process 1
Receiving request from  2  for cylinder  2
Receiving request from  3  for cylinder  0
Receiving request from  4  for cylinder  1
Receiving request from  5  for cylinder  2
Receiving request from  6  for cylinder  0
Process 1 releasing access to cyclinder 1
Process  1  requesting cylinder 2
access to cylinder 2 being given to process 2
Receiving request from  1  for cylinder  2
Process 2 releasing access to cyclinder 2
Process  2  requesting cylinder 0
access to cylinder 2 being given to process 5
Receiving request from  2  for cylinder  0
Process 5 releasing access to cyclinder 2
Process  5  requesting cylinder 0
access to cylinder 0 being given to process 3
Receiving request from  5  for cylinder  0
Process 3 releasing access to cyclinder 0
Process  3  requesting cylinder 1
access to cylinder 0 being given to process 6
Receiving request from  3  for cylinder  1
Process 6 releasing access to cyclinder 0
Process  6  requesting cylinder 1
access to cylinder 0 being given to process 2
Receiving request from  6  for cylinder  1
Process 2 releasing access to cyclinder 0
Process  2  requesting cylinder 1
access to cylinder 1 being given to process 4
Receiving request from  2  for cylinder  1
```

```
Process 4 releasing access to cyclinder 1
Process  4  requesting cylinder 2
access to cylinder 1 being given to process 3
Receiving request from  4  for cylinder  2
Process 3 releasing access to cyclinder 1
Process  3  requesting cylinder 2
access to cylinder 1 being given to process 6
Receiving request from  3  for cylinder  2
Process 6 releasing access to cyclinder 1
Process  6  requesting cylinder 2
access to cylinder 2 being given to process 1
Receiving request from  6  for cylinder  2
Process 1 releasing access to cyclinder 2
Process  1  requesting cylinder 0
access to cylinder 2 being given to process 4
Receiving request from  1  for cylinder  0
Process 4 releasing access to cyclinder 2
Process  4  requesting cylinder 0
access to cylinder 2 being given to process 3
Receiving request from  4  for cylinder  0
Process 3 releasing access to cyclinder 2
Process  3  exiting
access to cylinder 0 being given to process 5
Process 5 releasing access to cyclinder 0
Process  5  requesting cylinder 1
access to cylinder 0 being given to process 1
Receiving request from  5  for cylinder  1
Process 1 releasing access to cyclinder 0
Process  1  exiting
access to cylinder 0 being given to process 4
Process 4 releasing access to cyclinder 0
Process  4  exiting
access to cylinder 1 being given to process 2
Process 2 releasing access to cyclinder 1
Process  2  exiting
access to cylinder 1 being given to process 5
Process 5 releasing access to cyclinder 1
Process  5  exiting
access to cylinder 2 being given to process 6
```

Process 6 releasing access to cyclinder 2
Process 6 exiting
For scan[0] handler for process # 1 there were
1 local wait requests and 6 global wait messages
2 local signal requests and 12 global signal messages
and 8 wait_ranks messages
For scan[1] handler for process # 1 there were
1 local wait requests and 6 global wait messages
1 local signal requests and 6 global signal messages
and 7 wait_ranks messages
For process 1 mutex handler there were
50 messages to get mutex and 8 to release mutex
For scan[0] handler for process # 2 there were
2 local wait requests and 12 global wait messages
2 local signal requests and 12 global signal messages
and 14 wait_ranks messages
For scan[1] handler for process # 2 there were
1 local wait requests and 6 global wait messages
1 local signal requests and 6 global signal messages
and 7 wait_ranks messages
For process 2 mutex handler there were
57 messages to get mutex and 9 to release mutex
For scan[0] handler for process # 3 there were
0 local wait requests and 0 global wait messages
1 local signal requests and 6 global signal messages
and 1 wait_ranks messages
For scan[1] handler for process # 3 there were
3 local wait requests and 18 global wait messages
2 local signal requests and 12 global signal messages
and 20 wait_ranks messages
For process 3 mutex handler there were
63 messages to get mutex and 9 to release mutex
For scan[0] handler for process # 4 there were
1 local wait requests and 6 global wait messages
1 local signal requests and 6 global signal messages
and 7 wait_ranks messages
For scan[1] handler for process # 4 there were
2 local wait requests and 12 global wait messages
2 local signal requests and 12 global signal messages

```
and  14  wait_ranks messages
For process  4  mutex handler there were
57 messages to get mutex and 9  to release mutex
For  scan[ 0]  handler for process # 5 there were
3 local wait requests and 18  global wait messages
2 local signal requests and 12  global signal messages
and  20  wait_ranks messages
For  scan[ 1]  handler for process # 5 there were
0 local wait requests and 0  global wait messages
1 local signal requests and 6  global signal messages
and  1  wait_ranks messages
For process  5  mutex handler there were
51 messages to get mutex and 9  to release mutex
For  scan[ 0]  handler for process # 6 there were
1 local wait requests and 6  global wait messages
1 local signal requests and 0  global signal messages
and  6  wait_ranks messages
For  scan[ 1]  handler for process # 6 there were
2 local wait requests and 12  global wait messages
2 local signal requests and 12  global signal messages
and  14  wait_ranks messages
For process  6  mutex handler there were
57 messages to get mutex and 9  to release mutex
ivy% exit
ivy%
script done on Fri Jul 21 10:15:44 1989
```

APPENDIX IV

CSP Code and Runs

The following are the listings and runs for the roller coaster problem using CSP. Also included is the code and runs for finding the determinant of a 3 x 3 matrix using nested processes. First comes all the code for the centralized solution and runs for eight and six passengers. Next comes all the code for the distributed solution and a run with eight passengers. The six passenger case is omitted since it is identical to the centralized run. Next comes the code and run for the transformed determinant problem using the centralized server. The server code is omitted since it is identical to the server used in the roller coaster problem, except that the constant num_requests is set to 7. Next comes the code and run for the transformed determinant problem using the distributed server. The server code is omitted since it is identical to the server used in the roller coaster problem, except that the constant num_requests is set to 7. Next comes the code and run for the determinant problem using the modified centralized algorithm. Then comes the code and run for the determinant problem using the modified distributed algorithm.

```
This main process for the CSP solution to the
roller coaster problem using a centralized server.
resource cent_roller_main
   import central_csp_server
   import Veh_cent
   import Pas_cent

body cent_roller_main()
   op done()
   var num_passengers, N : int

   process start
      writes("Welcome to the roller coaster!")
      write("  Each passenger will get to ride 5 times")
      write("Enter the total number of passengers who will wish to ride.")
      read( num_passengers)
      N := num_passengers + 1
      var the_server : cap central_csp_server
      var passengers[ 1: num_passengers] : cap Pas_cent
      var the_vehicle : cap Veh_cent

      create the centralized I/O server
      the_server := create central_csp_server( N)

      create passenger processes
      fa i := 1 to num_passengers →
         passengers[ i] := create Pas_cent( i, N, N, the_server,
                                    done)
      af
      create the vehicle process
      the_vehicle := create Veh_cent( N, num_passengers, N,
                               the_server, done)
      send communication links to vehicle process
      send the_vehicle.links( passengers)

      wait for the passengers and vehicle processes to complete
      fa i := 1 to N →
         receive done()
      af
```

```
        destroy instances of all the processes
        destroy the_server
        destroy the_vehicle
        fa i := 1 to num_passengers →
          destroy passengers[ i]
        af
    end
end cent_roller_main
```

```
This is the vehicle process code for the CSP solution
to the roller coaster problem using a centralized server.
resource Veh_cent
    import central_csp_server
    import Pas_cent
    op links( passengers[ 1: *] : cap Pas_cent)
    const N := 4
    const num_requests := 2

body Veh_cent( myid, num_passengers, tot_procs :int;
              my_server : cap central_csp_server; done : cap())

        var passengers[ 1: num_passengers] : cap Pas_cent
        var ready[ 1 : tot_procs, num_requests] : bool
                                := ([tot_procs*num_requests]false)
        op found( success : bool; match_proc, request : int)

        process vehicle
           receive links( passengers)
           var who, what : int
           var success : bool := true
           var runs := 0, on_board := 0
           var total_runs := ( num_passengers * 5
                               + ( N - 1)) / N
           do runs < total_runs →
              inner loop to board passengers
              fa i := 1 to num_passengers →
                 ready[ i, 2] := false
              af
              success := true
              do success →
                 if on_board < N →
                    fa i := 1 to num_passengers →
                        ready[ i, 1] := true
                    af
                    send my_server.find_match( ready, myid, found)
                    receive found( success, who, what)
                    if success →
                       if ( what = 1) →
```

```
          call passengers[ who].board()
          on_board++
        [] else →
          write("ERROR, illegal match in", myid)
          write("what =",what," who = ", who)
          stop
        fi
      fi
    [] else →
      # all the guards are false, the loop terminates
      success := false
    fi
od
inner loop to get passengers to debark
fa i := 1 to num_passengers →
    ready[ i, 1] := false
af
success := true
do success →
    if on_board > 0 →
      fa i := 1 to num_passengers →
          ready[ i, 2] := true
      af
      send my_server.find_match( ready, myid, found)
      receive found( success, who, what)
      if success →
        if ( what = 2) →
          call passengers[ who].debark()
          on_board–
        [] else →
          write("ERROR, illegal match in", myid)
          write("what =",what," who = ", who)
          stop
        fi
      fi
    [] else →
      # all the guards are false, the loop terminates
      success := false
    fi
```

```
        od
        runs++
      od
      write("Vehicle is calling proc_done")
      call my_server.proc_done( myid)
      send done()
    end

end Veh_cent
```

Passenger process code for the CSP solution
to the roller coaster problem using a centralized server.

```
resource Pas_cent
   import central_csp_server
   op board()
   op debark()
   const num_requests := 2

body Pas_cent( myid, vehicle_id, N:int;
          my_server : cap central_csp_server; done : cap())

     var ready[ 1 : N, num_requests] : bool := ([N*num_requests]false)
     op found( success : bool; match_proc, request : int)


     process passenger
       var rides := 0
       var success : bool := true
       do rides < 5 →
         attempt to board
         ready[ vehicle_id, 1] := true
         ready[ myid, 1] := true
         call my_server.find_match( ready, myid, found)
         var who, what : int
         receive found( success, who, what)
         if success →
           receive board()
           write("Passenger ",myid," boarding vehicle")
           ready[ vehicle_id, 1] := false
         [] else →
           write("error, no match in", myid)
           stop
         fi
         attempt to get off
         ready[ vehicle_id, 2] := true
         send my_server.find_match( ready, myid, found)
         receive found( success, who, what)
         if success →
           receive debark()
```

```
            write("Passenger ",myid," getting out of vehicle")
            ready[ vehicle_id, 2] := false
            rides++
          [] else →
            write("error, no match in", myid)
            stop
          fi
      od
      write("Passenger ",myid," is calling proc_done")
      call my_server.proc_done( myid)
      send done()
    end

end Pas_cent
```

Resource to handle in a centralized manner the I/O requests
for CSP processes.

```
resource central_csp_server
    op proc_done( proc_id : int)
    const num_requests := 2
    optype found_op = ( success : bool; who, what : int)
    op find_match( ready[ 1:*, 1:num_requests] : bool; process_num : int
                   found : cap found_op)

body central_csp_server( N:int)

        var term_procs[ 1 : N] : bool := ( [N] false)
        type ready_rec = rec ( procnum : int; ready[ N, num_requests] : bool,
                               ready_count : int; found : cap found_op)
        op ready_op ( ready_record : ready_rec)

        var request_count := 0, found_match := 0, no_match := 0
        var proc_done_msgs := 0

        process main
            var count : int
            do true →
                in find_match( in_ready, who, found) →
                   request_count++
                   var request : int
                   var match_found : bool := false
                   count := ?ready_op
                   do count > 0 and not match_found →
                      in ready_op( next_ready) →
                         fa j := 1 to num_requests →
                            if next_ready.ready[ who, j]
                               and in_ready[ next_ready.procnum, j] →
                               match_found := true
                               request := j
                               exit
                            fi
                         af
                         if not match_found →
                            send ready_op( next_ready)
```

```
        [] else →
          send next_ready.found( true, who, request)
          send found( true, next_ready.procnum, request)
          found_match := found_match + 2
        fi
        count–
    ni
  od
  to maintain the order of requests, requests
  that were not checked above will be moved to the
  end of the queue
  if count ≠ ?ready_op →
    do count > 0 →
      in ready_op( ready_rec) →
        send ready_op( ready_rec)
      ni
      count–
    od
  fi
  if not match_found →
    count := 0
    fa i := 1 to N st i ≠ who, j := 1 to num_requests
    st in_ready[ i, j] →
      if term_procs[ i] →
        in_ready[ i, j] := false
      [] else →
        count++
      fi
    af
    if count > 0 →
      send ready_op(
              ready_rec ( who, in_ready, count, found))
     [] else →
      send found( false, 0, 0)
      no_match++
    fi
  fi
[] proc_done ( who) →
  proc_done_msgs++
```

```
          term_procs[ who] := true
          count := ?ready_op
          do count > 0 →
            in ready_op( next_ready) →
              fa j := 1 to num_requests st next_ready.ready[ who, j] →
                next_ready.ready[ who, j] := false
                next_ready.ready_count--
              af
              if next_ready.ready_count = 0 →
                send next_ready.found( false, 0, 0)
                no_match++
              [] else →
                send ready_op( next_ready)
              fi
            ni
            count--
          od
        ni
      od
    end

    final
      write("The total number of I/O requests msgs was ",request_count)
      write("The number of found messages with match ",found_match)
      write("The number of found messages with no match", no_match)
      write("The number of proc_done messages", proc_done_msgs)
    end
end central_csp_server
```

```
Script started on Tue Aug 29 15:20:57 1989
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Passenger  1  boarding vehicle
Passenger  2  boarding vehicle
Passenger  3  boarding vehicle
Passenger  4  boarding vehicle
Passenger  1  getting out of vehicle
Passenger  2  getting out of vehicle
Passenger  3  getting out of vehicle
Passenger  4  getting out of vehicle
Passenger  5  boarding vehicle
Passenger  6  boarding vehicle
Passenger  7  boarding vehicle
Passenger  8  boarding vehicle
Passenger  5  getting out of vehicle
Passenger  6  getting out of vehicle
Passenger  7  getting out of vehicle
Passenger  8  getting out of vehicle
Passenger  1  boarding vehicle
Passenger  2  boarding vehicle
Passenger  3  boarding vehicle
Passenger  4  boarding vehicle
Passenger  1  getting out of vehicle
Passenger  2  getting out of vehicle
Passenger  3  getting out of vehicle
Passenger  4  getting out of vehicle
Passenger  5  boarding vehicle
Passenger  6  boarding vehicle
Passenger  7  boarding vehicle
Passenger  8  boarding vehicle
Passenger  5  getting out of vehicle
Passenger  6  getting out of vehicle
Passenger  7  getting out of vehicle
Passenger  8  getting out of vehicle
Passenger  1  boarding vehicle
Passenger  2  boarding vehicle
```

Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 1 getting out of vehicle
Passenger 2 getting out of vehicle
Passenger 3 getting out of vehicle
Passenger 4 getting out of vehicle
Passenger 5 boarding vehicle
Passenger 6 boarding vehicle
Passenger 7 boarding vehicle
Passenger 8 boarding vehicle
Passenger 5 getting out of vehicle
Passenger 6 getting out of vehicle
Passenger 7 getting out of vehicle
Passenger 8 getting out of vehicle
Passenger 1 boarding vehicle
Passenger 2 boarding vehicle
Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 1 getting out of vehicle
Passenger 2 getting out of vehicle
Passenger 3 getting out of vehicle
Passenger 4 getting out of vehicle
Passenger 5 boarding vehicle
Passenger 6 boarding vehicle
Passenger 7 boarding vehicle
Passenger 8 boarding vehicle
Passenger 5 getting out of vehicle
Passenger 6 getting out of vehicle
Passenger 7 getting out of vehicle
Passenger 8 getting out of vehicle
Passenger 1 boarding vehicle
Passenger 2 boarding vehicle
Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 1 getting out of vehicle
Passenger 2 getting out of vehicle
Passenger 3 getting out of vehicle
Passenger 4 getting out of vehicle
Passenger 5 boarding vehicle

Passenger 6 boarding vehicle
Passenger 7 boarding vehicle
Passenger 8 boarding vehicle
Passenger 1 is calling proc_done
Passenger 2 is calling proc_done
Passenger 3 is calling proc_done
Passenger 4 is calling proc_done
Passenger 5 getting out of vehicle
Passenger 6 getting out of vehicle
Passenger 7 getting out of vehicle
Passenger 8 getting out of vehicle
Vehicle is calling proc_done
Passenger 5 is calling proc_done
Passenger 6 is calling proc_done
Passenger 7 is calling proc_done
Passenger 8 is calling proc_done
The total number of I/O requests msgs was 160
The number of found messages with match 160
The number of found messages with no match 0
The number of proc_done messages 9
ivy% a.out
Welcome to the roller coaster! Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
6
Passenger 1 boarding vehicle
Passenger 2 boarding vehicle
Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 1 getting out of vehicle
Passenger 2 getting out of vehicle
Passenger 3 getting out of vehicle
Passenger 4 getting out of vehicle
Passenger 5 boarding vehicle
Passenger 6 boarding vehicle
Passenger 1 boarding vehicle
Passenger 2 boarding vehicle
Passenger 5 getting out of vehicle
Passenger 6 getting out of vehicle
Passenger 1 getting out of vehicle

Passenger 2 getting out of vehicle
Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 5 boarding vehicle
Passenger 6 boarding vehicle
Passenger 3 getting out of vehicle
Passenger 4 getting out of vehicle
Passenger 5 getting out of vehicle
Passenger 6 getting out of vehicle
Passenger 1 boarding vehicle
Passenger 2 boarding vehicle
Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 1 getting out of vehicle
Passenger 2 getting out of vehicle
Passenger 3 getting out of vehicle
Passenger 4 getting out of vehicle
Passenger 5 boarding vehicle
Passenger 6 boarding vehicle
Passenger 1 boarding vehicle
Passenger 2 boarding vehicle
Passenger 5 getting out of vehicle
Passenger 6 getting out of vehicle
Passenger 1 getting out of vehicle
Passenger 2 getting out of vehicle
Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 5 boarding vehicle
Passenger 6 boarding vehicle
Passenger 3 getting out of vehicle
Passenger 4 getting out of vehicle
Passenger 5 getting out of vehicle
Passenger 6 getting out of vehicle
Passenger 1 boarding vehicle
Passenger 2 boarding vehicle
Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 1 getting out of vehicle
Passenger 2 getting out of vehicle

```
Passenger  3  getting out of vehicle
Passenger  4  getting out of vehicle
Passenger  5  boarding vehicle
Passenger  6  boarding vehicle
Passenger  1  is calling proc_done
Passenger  2  is calling proc_done
Passenger  3  is calling proc_done
Passenger  4  is calling proc_done
RTS warning: 5 blocked processes during termination
ivy% exit
ivy%
script done on Tue Aug 29 15:21:13 1989
```

```
This is the main process for the CSP solution to
the roller coaster problem using distributed servers.
resource roller_main
    import csp_server
    import Vehicle
    import Passenger

body roller_main()
    op done()
    var num_passengers, N : int

    process start
        writes("Welcome to the roller coaster!")
        write("  Each passenger will get to ride 5 times")
        write("Enter the total number of passengers who will wish to ride.")
        read( num_passengers)
        N := num_passengers + 1
        var servers[ 1 : N ] : cap csp_server
        var passengers[ 1: num_passengers] : cap Passenger
        var the_vehicle : cap Vehicle

        create passenger processes and associated servers
        fa i := 1 to num_passengers →
            servers[ i] := create csp_server( i, N)
            passengers[ i] := create Passenger( i, N, N, servers[ i],
                                    done)
        af
        create vehicle process and associated server
        servers[ N] := create csp_server( N, N)
        the_vehicle := create Vehicle( N, num_passengers, N,
                                servers[ N], done)

        send communication links to the vehicle and the servers
        send the_vehicle.links( passengers)
        fa i := 1 to N →
           send servers[ i].links( servers)
        af

        wait for the passenger and car processes to complete
```

```
        fa i := 1 to N →
           receive done()
        af

        destroy the instances of all the processes created
        fa i := 1 to num_passengers →
          destroy passengers[ i]
          destroy servers[ i]
        af
        destroy the_vehicle
        destroy servers[ N]
    end
end roller_main
```

```
This is the vehicle process code for the CSP solution
to the roller coaster problem using distributed servers.
resource Vehicle
   import csp_server
   import Passenger
   op links( passengers[ 1: *] : cap Passenger)
   const capacity := 4
   const num_requests := 2

body Vehicle( myid, num_passengers, N :int;
          my_server : cap csp_server; done : cap())

     var passengers[ 1: num_passengers] : cap Passenger
     var ready[ 1 : N, num_requests] : bool := ([N*num_requests]false)
     op found( success : bool; match_proc, request : int)

     process vehicle
       receive links( passengers)
       var total_runs := ( num_passengers * 5
                     + ( capacity - 1)) / capacity
       set up tokens to send to passengers both board and debark
       fa request := 1 to num_requests,
         pass_num := 1 to num_passengers →
           send my_server.init_token( pass_num, request)
       af
       send my_server.token_init_done()
       var who, what : int
       var success : bool := true
       var runs := 0, on_board := 0
       do runs < total_runs →
         inner loop to board passengers
         fa i := 1 to num_passengers →
           ready[ i, 2] := false
         af
         success := true
         do success →
           if on_board < capacity →
             fa i := 1 to num_passengers →
               ready[ i, 1] := true
```

```
    af
    send my_server.find_match( ready, found)
    receive found( success, who, what)
    if success →
      if ( what = 1) →
        call passengers[ who].board()
        on_board++
      [] else →
        write("ERROR, illegal match in", myid)
        write("what =",what," who = ", who)
        stop
      fi
    fi
  [] else →
    # all the guards are false, the loop terminates
    success := false
  fi
od
```

inner loop to get passengers to debark

```
fa i := 1 to num_passengers →
  ready[ i, 1] := false
af
success := true
do success →
  if on_board > 0 →
    fa i := 1 to num_passengers →
      ready[ i, 2] := true
    af
    send my_server.find_match( ready, found)
    receive found( success, who, what)
    if success →
      if ( what = 2) →
        call passengers[ who].debark()
        on_board--
      [] else →
        write("ERROR, illegal match in", myid)
        write("what =",what," who = ", who)
        stop
      fi
```

```
            fi
          [] else →
            # all the guards are false, the loop terminates
            success := false
          fi
        od
        runs++
      od
      write("Vehicle is calling proc_done")
      call my_server.proc_done()
      send done()
    end

end Vehicle
```

```
Passenger process code for the CSP solution
to the roller coaster problem using distributed servers.
resource Passenger
   import csp_server
   op board()
   op debark()
   const capacity := 5
   const num_requests := 2

body Passenger( myid, vehicle_id, N:int;
          my_server : cap csp_server; done : cap())

      var ready[ 1 : N, num_requests] : bool := ([N*num_requests]false)
      op found( success : bool; match_proc, request : int)


      process passenger
        passengers initially have no tokens
        send my_server.token_init_done()
        var rides := 0
        var success : bool := true
        do rides < 5 →
          attempt to board
          ready[ vehicle_id, 1] := true
          call my_server.find_match( ready, found)
          var who, what : int
          receive found( success, who, what)
          if success →
            receive board()
            write("Passenger ",myid," boarding vehicle")
            ready[ vehicle_id, 1] := false
          [] else →
            write("error, no match in", myid)
            stop
          fi
          attempt to get off
          ready[ vehicle_id, 2] := true
          send my_server.find_match( ready, found)
          receive found( success, who, what)
```

```
      if success →
        receive debark()
        write("Passenger ",myid," getting out of vehicle")
        ready[ vehicle_id, 2] := false
        rides++
      [] else →
        write("error, no match in", myid)
        stop
      fi
    od
    write("Passenger ",myid," is calling proc_done")
    call my_server.proc_done()
    send done()
  end

end Passenger
```

Resource to handle in a distributed manner the I/O requests of
CSP processes - using Bagrodia's algorithm.

```
resource csp_server
    import Token_decls
    optype found_op = ( success : bool; who, what : int)
    op in_token : in_token_op
    op cancel : cancel_op
    op term : termination_op
    const num_requests := 2
    op find_match( ready_param[ 1 : *, 1 : num_requests] : bool;
                found_param : cap found_op)
    op proc_done()
    op links( io_servers[ 1: *]: cap csp_server)
    op init_token( who, request : int)
    op token_init_done()

body csp_server( myid, N: int)

        type color_stat = enum( black, white, grey)
        var color : color_stat
        var ready_count : int
        op token_q : ( token: token_typ)
        op delay_q : ( token: token_typ)
        op temp_q : ( token: token_typ)
        op not_ready_q : ( token: token_typ)
        var io_servers[ 1 : N] : cap csp_server
        var ready[ 1 : N, 1 : num_requests] : bool
        var found : cap found_op
        var term_procs[ 1 : N] : bool := ([N]false)
        op Refuse_token( token : token_typ)
        op Request_token() returns success : bool
        op Deny_token( token : token_typ)
        op Delay_token( token : token_typ)
        op Commit_token( token : token_typ)
        op Commit_delayed() returns token : token_typ
        op Refuse_delayed()
        op Initialize()
        op main_prot()
```

```
var request_count := 0, token_sends := 0, cancel_sends := 0
var found_sends := 0, termination_msgs := 0

initial
  reply
  var init_done := false
  var token : token_typ
  token.process_id := myid
  token.process_list[ 1] := myid
  do not init_done →
    receive from our process the token
    that this server initially holds
    in init_token( who, request) →
      token.process_list[ 2] := who
      token.request := request
      send not_ready_q( token)
    [] token_init_done() →
      init_done := true
    ni
  od
  receive links( io_servers)
  send main_prot()
end

proc main_prot()
  color := black
  var match_token : token_typ
  var not_done := true
  do not_done →
    do color = black →
      in in_token( token) →
        Refuse_token( token)
      [] find_match( ready_param, found_param) →
        request_count++
        found := found_param
        fa i := 1 to N, j := 1 to num_requests →
          ready[ i, j] := ready_param[ i, j]
        af
        Initialize()
```

```
    color := white
  [] proc_done() →
    termination_msgs++
    fa i := 1 to N st (i ≠ myid and not term_procs[i]) →
      send io_servers[ i].term( myid)
      termination_msgs++
    af
  [] term( who) →
    term_procs[ who] := true
  ni
od
do color ≠ black and ready_count > 0 →
  if color = white and (?token_q ≠ 0) →
    var success : bool
    success := Request_token()
    if success →
      color := grey
    fi
  fi
  in in_token( token) and color = white →
    if ready[ token.process_id, token.request] →
      Commit_token( token)
      match_token := token
      color := black
    [] else →
      Refuse_token( token)
    fi
  [] in_token( token) and color = grey →
    if myid = token.process_id →
      Commit_token( token)
      match_token := token
      color := black
    [] else →
      if not(ready[ token.process_id, token.request]) →
        Refuse_token( token)
      [] else →
        if myid > token.process_id →
          Deny_token( token)
        [] myid < token.process_id →
```

```
              Delay_token( token)
            fi
          fi
        fi
      [] cancel() and color = grey →
        if ?delay_q  = 0 →
          color := white
        [] else →
          match_token := Commit_delayed()
          color := black
          send match
        fi
      [] term( who) →
        fa j := 1 to num_requests st ready[ who, j] →
           ready[ who, j] := false
           ready_count–
        af
        term_procs[ who] := true
    ni
od
if ready_count = 0 →
    did not find a matching I/O command
    send found( false, 0, 0)
    found_sends++
    if color = grey →
      sent request to process that has terminated
      must wait for cancel message
      receive cancel()
    fi
    color := black
[] else →
    notify client of process with matching I/O command
    if myid = match_token.process_list[ 1] →
      send found( true, match_token.process_list[ 2],
                  match_token.request)
    [] else →
      send found( true, match_token.process_list[ 1],
                  match_token.request)
    fi
```

```
      found_sends++
    fi
  od
end

proc Refuse_token( token)
  send io_servers[ token.process_id].cancel()
  cancel_sends++
  send not_ready_q( token)
end

proc Request_token() returns success
  var send_to : int
  var token : token_typ
  success := false
  do ?token_q ≠ 0 and not success →
    receive token_q( token)
    token.process_id := myid
    if myid = token.process_list[ 1] →
        send_to := token.process_list[ 2]
    [] else →
        send_to := token.process_list[ 1]
    fi
    if ready[ send_to, token.request] →
      success := true
    fi
  od
  if success →
    send io_servers[ send_to].in_token( token)
    token_sends++
  fi
end

proc Deny_token( token)
  send io_servers[ token.process_id].cancel()
  cancel_sends++
  send token_q( token)
end
```

```
proc Delay_token( token)
  send delay_q( token)
end

proc Commit_token( token)
  if token.process_id ≠ myid →
    send io_servers[ token.process_id].in_token( token)
    token_sends++
  [] else →
    send token_q( token)
  fi
  Refuse_delayed()
end

proc Commit_delayed() returns token
  receive delay_q( token)
  Commit_token( token)
end

proc Refuse_delayed()
  var token : token_typ
  do ?delay_q ≠ 0 →
    receive delay_q( token)
    Refuse_token( token)
  od
end

proc Initialize()
  var token : token_typ
  var other_proc : int
  ready_count := 0
  fa i := 1 to N st i ≠ myid, j := 1 to num_requests →
      if ready[ i, j] →
        if term_procs[ i] →
            ready[ i, j] := false
        [] else →
            ready_count++
        fi
      fi
```

```
      af
      do ?token_q ≠ 0 →
        receive token_q( token)
        send not_ready_q( token)
      od
      do ?not_ready_q ≠ 0 →
        receive not_ready_q( token)
        if myid = token.process_list[ 1] →
          other_proc := token.process_list[ 2]
         [] else →
          other_proc := token.process_list[ 1]
        fi
        if ready[ other_proc, token.request] →
          send token_q( token)
         [] else →
          send temp_q( token)
        fi
      od
      do ?temp_q ≠ 0 →
       receive temp_q( token)
       send not_ready_q(token)
      od
    end

    final
      write("For server ",myid," there were ")
      writes(request_count," request messages and ")
      write(found_sends," found messages and")
      writes(token_sends," token messages and ")
      write(cancel_sends," cancel messages")
      write(termination_msgs," termination messages")
    end
end csp_server
```

```
global Token_decls
   declare token type
   type token_typ = rec ( process_id, process_list[ 1:2], request:int)
   optype in_token_op = ( req_token :token_typ)
   optype cancel_op = ()
   optype termination_op = ( id : int)
end Token_decls
```

```
Script started on Tue Aug 29 15:19:57 1989
ivy% a.out
Welcome to the roller coaster!  Each passenger will get to ride 5 times
Enter the total number of passengers who will wish to ride.
8
Passenger  1  boarding vehicle
Passenger  2  boarding vehicle
Passenger  3  boarding vehicle
Passenger  4  boarding vehicle
Passenger  1  getting out of vehicle
Passenger  2  getting out of vehicle
Passenger  3  getting out of vehicle
Passenger  4  getting out of vehicle
Passenger  5  boarding vehicle
Passenger  6  boarding vehicle
Passenger  7  boarding vehicle
Passenger  8  boarding vehicle
Passenger  5  getting out of vehicle
Passenger  6  getting out of vehicle
Passenger  7  getting out of vehicle
Passenger  8  getting out of vehicle
Passenger  1  boarding vehicle
Passenger  2  boarding vehicle
Passenger  3  boarding vehicle
Passenger  4  boarding vehicle
Passenger  1  getting out of vehicle
Passenger  2  getting out of vehicle
Passenger  3  getting out of vehicle
Passenger  4  getting out of vehicle
Passenger  5  boarding vehicle
Passenger  6  boarding vehicle
Passenger  7  boarding vehicle
Passenger  8  boarding vehicle
Passenger  5  getting out of vehicle
Passenger  6  getting out of vehicle
Passenger  7  getting out of vehicle
Passenger  8  getting out of vehicle
Passenger  1  boarding vehicle
Passenger  2  boarding vehicle
```

Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 1 getting out of vehicle
Passenger 2 getting out of vehicle
Passenger 3 getting out of vehicle
Passenger 4 getting out of vehicle
Passenger 5 boarding vehicle
Passenger 6 boarding vehicle
Passenger 7 boarding vehicle
Passenger 8 boarding vehicle
Passenger 5 getting out of vehicle
Passenger 6 getting out of vehicle
Passenger 7 getting out of vehicle
Passenger 8 getting out of vehicle
Passenger 1 boarding vehicle
Passenger 2 boarding vehicle
Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 1 getting out of vehicle
Passenger 2 getting out of vehicle
Passenger 3 getting out of vehicle
Passenger 4 getting out of vehicle
Passenger 5 boarding vehicle
Passenger 6 boarding vehicle
Passenger 7 boarding vehicle
Passenger 8 boarding vehicle
Passenger 5 getting out of vehicle
Passenger 6 getting out of vehicle
Passenger 7 getting out of vehicle
Passenger 8 getting out of vehicle
Passenger 1 boarding vehicle
Passenger 2 boarding vehicle
Passenger 3 boarding vehicle
Passenger 4 boarding vehicle
Passenger 1 getting out of vehicle
Passenger 2 getting out of vehicle
Passenger 3 getting out of vehicle
Passenger 4 getting out of vehicle
Passenger 5 boarding vehicle

Passenger 6 boarding vehicle
Passenger 7 boarding vehicle
Passenger 8 boarding vehicle
Passenger 1 is calling proc_done
Passenger 2 is calling proc_done
Passenger 3 is calling proc_done
Passenger 4 is calling proc_done
Passenger 5 getting out of vehicle
Passenger 6 getting out of vehicle
Passenger 7 getting out of vehicle
Passenger 8 getting out of vehicle
Vehicle is calling proc_done
Passenger 5 is calling proc_done
Passenger 6 is calling proc_done
Passenger 7 is calling proc_done
Passenger 8 is calling proc_done
For server 1 there were
10 request messages and 10 found messages and
10 token messages and 1 cancel messages
9 termination messages
For server 2 there were
10 request messages and 10 found messages and
15 token messages and 6 cancel messages
9 termination messages
For server 3 there were
10 request messages and 10 found messages and
15 token messages and 6 cancel messages
9 termination messages
For server 4 there were
10 request messages and 10 found messages and
15 token messages and 6 cancel messages
9 termination messages
For server 5 there were
10 request messages and 10 found messages and
10 token messages and 1 cancel messages
4 termination messages
For server 6 there were
10 request messages and 10 found messages and
15 token messages and 5 cancel messages

```
4  termination messages
For server  7  there were
10 request messages and 10  found messages and
15 token messages and 5  cancel messages
4  termination messages
For server  8  there were
10 request messages and 10  found messages and
15 token messages and 5  cancel messages
4  termination messages
For server  9  there were
80 request messages and 80  found messages and
115 token messages and 30  cancel messages
9  termination messages
ivy% exit
ivy%
script done on Tue Aug 29 15:20:09 1989
```

The six passenger distributed run was omitted, since it was identical to the centralized run shown above.

```
this a CSP solution to calculate the determinant of a 3 x 3
matrix using CSP code transformed to remove nested processes
this version uses the centralized server.
resource det_main
    import central_csp_server
    import determinant

body det_main()
    op done()

    process start
        var the_server : cap central_csp_server
        var processes[ 1: 7] : cap determinant
        the_server := create central_csp_server( 7)
        fa i := 1 to 7 →
            processes[ i] := create determinant( i, 7, the_server,
                                                  done)
        af
        send links to all the processes
        fa i := 1 to 7 →
           send processes[ i].links( processes)
        af
        fa i := 1 to 7 →
            receive done()
        af
        destroy all the processes
        fa i := 1 to 7 →
           destroy processes[ i]
        af
        destroy the server
        destroy the_server
    end
end det_main
```

```
this resource contains the code for the CSP processes used to calculate
the determinant of a 3 x 3 matrix.  The CSP code has been transformed
to remove nesting. This version uses the centralized server.
resource determinant
   import central_csp_server
   op links( others[ 1: *] : cap determinant)
   const num_requests := 7
   op in_matrix( matrix[ 3, 3] : int)
   op in_end()
   op in_int( int_value : int)
   op start()
   op startA( matrix[ 3, 3] : int)

body determinant( myid, tot_procs :int;
          my_server : cap central_csp_server; finished : cap())

      var others[ 1: tot_procs] : cap determinant
      var ready[ 1 : tot_procs, num_requests] : bool
                            := ([tot_procs*num_requests]false)
      op found( success : bool; match_proc, request : int)
      op X1()
      op CALC( num: int)
      op TOTAL()
      op Y()
      op INPUT()

      initial
        reply
        receive links( others)
        if myid = 1 →
          send X1()
        [] myid ≥ 2 and myid ≤ 4 →
          send CALC( myid - 1)
        [] myid = 5 →
          send TOTAL()
        [] myid = 6 →
          send Y()
        [] myid = 7 →
          send INPUT()
```

```
  fi
end

proc X1()
  var who, what : int
  var success : bool := true
  var A [ 3, 3] : int
  var done[ 4] : bool
  do success →
    loop to get next matrix from INPUT
    ready[ 7, 2] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
      if ( what = 2) →
        receive in_matrix( A)
      [] else →
        write("ERROR, illegal match in", myid)
        write("what =",what," who = ", who)
        stop
      fi
      ready[ 7, 2] := false
      do loop to notify X2(1)..X2(3) to start
      var i : int
      i := 0
      do i < 3 →
        i++
        ready[ i+1, 3] := true
        send my_server.find_match( ready, myid, found)
        receive found( success, who, what)
        if success →
          call others[i+1].startA( A)
          done[ i] := false
        [] else →
          write("error, no match in", myid)
          stop
        fi
        ready[ i+1, 3] := false
      od
```

```
  notify X4 to start
  ready[ 5, 4] := true
  send my_server.find_match( ready, myid, found)
  receive found( success, who, what)
  if success →
    call others[ 5].start()
    done[ 4] := false
   [] else →
    write("error, no match in", myid)
    stop
  fi
  ready[ 5, 4] := false
  loop to wait for end msgs from the X2s
  do success →
    var guard_possible := false
    fa i := 1 to 4 →
      ready[ i + 1, 5] := not done[ i]
      guard_possible := guard_possible or not done[ i]
    af
    if guard_possible →
      send my_server.find_match( ready, myid, found)
      receive found( success, who, what)
      if success →
        receive in_end()
        done[ who - 1] := true
       [] else →
        write("error, no match in", myid)
        stop
      fi
     [] else →
      received end from all four X2s, loop terminating
      success := false
    fi
  od
  success := true
  fa i := 1 to 4 →
    ready[ i + 1, 5] := false
  af
[] else →
```

```
      # INPUT terminated, the loop terminates
      success := false
    fi
  od
  send my_server.proc_done( myid)
  send finished()
end

proc CALC( num)
  var who, what : int
  var success : bool := true
  var A[ 3, 3] : int
  do success →
    loop to get START msg and matrix from X1
    ready[ 1, 3] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
      receive startA( A)
    [] else →
      # X1 has terminated, loop terminates
      exit
    fi
    ready[ 1, 3] := false
    var First_col, Second_col, Cofactor : int
    var value : int
    First_col := ( num % 3) + 1
    Second_col := ( First_col % 3) + 1
    Cofactor := A[ 2, First_col] * A[ 3, Second_col] -
            A[ 3, First_col] * A[ 2, Second_col]
    value := Cofactor * A[ 1, num]
    send value to X2( 4)
    ready[ 5, 6] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
      call others[ 5].in_int( value)
    [] else →
      write("error, no match in", myid)
```

```
      stop
    fi
    ready[ 5, 6] := false
    send end msg to X2(4)
    ready[ 5, 7] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
      call others[ 5].in_end()
     [] else →
      write("error, no match in", myid)
      stop
    fi
    ready[ 5, 7] := false
    send end msg to X1
    ready[ 1, 5] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
      call others[ 1].in_end()
     [] else →
      write("error, no match in", myid)
      stop
    fi
    ready[ 1, 5] := false
  od
  send my_server.proc_done( num)
  send finished()
end

proc TOTAL()
  var who, what : int
  var success : bool := true
  var done[ 3] : bool
  do success →
    loop to get START msg from X1
    ready[ 1, 4] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
```

```
if success →
  receive start()
 [] else →
  # X1 has terminated, loop terminates
  exit
fi
ready[ 1, 4] := false
var partial_value, total_value : int
var i : int
total_value := 0
i := 0
do i < 3 →
  i++
  done[ i] := false
od
loop to receive value and end msgs from other X2s
do success →
  fa i := 1 to 3 →
    ready[ i + 1, 6] := not done[ i]
    ready[ i + 1, 7] := not done[ i]
  af
  send my_server.find_match( ready, myid, found)
  receive found( success, who, what)
  if success →
    if ( what = 7) →
      receiving end msg
      receive in_end()
      done[ who - 1] := true
     [] ( what = 6) →
      receiving value
      receive in_int( partial_value)
      total_value := total_value + partial_value
     [] else →
      write("ERROR, illegal match in", myid)
      write("what =",what," who = ", who)
      stop
    fi
   [] else →
    all guards all false, loop terminates
```

```
      fi
    od
    success := true
    send value of determinant to Y
    ready[ 6, 1] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
      call others[ 6].in_int( total_value)
    [] else →
      write("error, no match in", myid)
      stop
    fi
    ready[ 6, 1] := false
    send end message to X1
    ready[ 1, 5] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
      call others[ 1].in_end()
    [] else →
      write("error, no match in", myid)
      stop
    fi
    ready[ 1, 5] := false
  od
  send my_server.proc_done( myid)
  send finished()
end

proc INPUT()
  var Matrix1[ 3, 3] : int := ( 5, 7, 9, -2, 4, 12, 1, 2, 32)
  var Matrix2[ 3, 3] : int := ( 4, 8, 12, 10, 7, 32, 1, 2, 3)
  var who, what : int
  var success : bool := true
  send 1st matrix to X1
  write("The first matrix is:")
  fa i := 1 to 3 →
    fa j := 1 to 3 →
```

```
      writes(Matrix1[ i, j]," ")
    af
    write()
  af
  ready[ 1, 2] := true
  send my_server.find_match( ready, myid, found)
  receive found( success, who, what)
  if success →
    call others[ 1].in_matrix( Matrix1)
  [] else →
    write("error, no match in", myid)
    stop
  fi
  send second matrix to X1
  write("The second matrix is:")
  fa i := 1 to 3 →
    fa j := 1 to 3 →
      writes(Matrix2[ i, j]," ")
    af
    write()
  af
  ready[ 1, 2] := true
  send my_server.find_match( ready, myid, found)
  receive found( success, who, what)
  if success →
    call others[ 1].in_matrix( Matrix2)
  [] else →
    write("error, no match in", myid)
    stop
  fi
  send my_server.proc_done( myid)
  send finished()
end

proc Y()
  var who, what : int
  var success : bool := true
  var det_value, count : int
  count := 1
```

```
    do loop to receive value of determinant
    do success →
      ready[ 1, 1] := true
      ready[ 5, 1] := true
      send my_server.find_match( ready, myid, found)
      receive found( success, who, what)
      if success →
        receive in_int( det_value)
        write("The determinant of Matrix ",count,"is ", det_value)
        count++
      [] else →
        processes have terminated, loop terminates
        exit
      fi
    od
    send my_server.proc_done( myid)
    send finished()
  end

end determinant
```

```
Script started on Fri Aug 25 09:01:21 1989
ivy% a.out
The first matrix is:
5 7 9
-2 4 12
1 2 32
The second matrix is:
The determinant of Matrix  1 is  980
4 8 12
10 7 32
1 2 3
The determinant of Matrix  2 is  0
The total number of I/O requests msgs was  72
The number of found messages with match  64
The number of found messages with no match 8
The number of proc_done messages 7
ivy% exit
ivy%
script done on Fri Aug 25 09:01:27 1989
```

```
this a CSP solution to calculate the determinant of a 3 x 3
matrix using CSP code transformed to remove nested processes
this version uses the distributed servers.
resource det_main
    import csp_server
    import determinant

body det_main()
    op done()

    process start
        var servers[ 1 : 7] : cap csp_server
        var processes[ 1: 7] : cap determinant
        fa i := 1 to 7 →
            servers[ i] := create csp_server( i, 7)
            processes[ i] := create determinant( i, 7, servers[ i],
                                    done)
        af
        send links to all the processes
        fa i := 1 to 7 →
           send servers[ i].links( servers)
           send processes[ i].links( processes)
        af
        fa i := 1 to 7 →
            receive done()
        af
        destroy all the processes
        fa i := 1 to 7 →
           destroy servers[ i]
           destroy processes[ i]
        af
        destroy the server
    end
end det_main
```

```
this resource contains the code for the CSP processes used to calculate
the determinant of a 3 x 3 matrix.  The CSP code has been transformed
to remove nesting.  This version uses the distributed servers.
resource determinant
   import csp_server
   op links( others[ 1: *] : cap determinant)
   const num_requests := 7
   op in_matrix( matrix[ 3, 3] : int)
   op in_end()
   op in_int( int_value : int)
   op start()
   op startA( matrix[ 3, 3] : int)

body determinant( myid, tot_procs :int;
          my_server : cap csp_server; finished : cap())

      var others[ 1: tot_procs] : cap determinant
      var ready[ 1 : tot_procs, num_requests] : bool
                              := ([tot_procs*num_requests]false)
      op found( success : bool; match_proc, request : int)
      op X1()
      op CALC( num: int)
      op TOTAL()
      op Y()
      op INPUT()

      initial
        reply
        receive links( others)
        if myid = 1 →
          send X1()
        [] myid ≥ 2 and myid ≤ 4 →
          send CALC( myid - 1)
        [] myid = 5 →
          send TOTAL()
        [] myid = 6 →
          send Y()
        [] myid = 7 →
          send INPUT()
```

```
  fi
end

proc X1()
  var who, what : int
  var success : bool := true
  var A [ 3, 3] : int
  var done[ 4] : bool
  set up tokens to send startA and start messages
  fa i := 2 to 4 →
    send my_server.init_token( i, 3)
  af
  send my_server.init_token( 5, 4)
  send my_server.token_init_done()
  do success →
    loop to get next matrix from INPUT
    ready[ 7, 2] := true
    send my_server.find_match( ready, found)
    receive found( success, who, what)
    if success →
      if ( what = 2) →
        receive in_matrix( A)
      [] else →
        write("ERROR, illegal match in", myid)
        write("what =",what," who = ", who)
        stop
      fi
      ready[ 7, 2] := false
      do loop to notify X2(1)..X2(3) to start
      var i : int
      i := 0
      do i < 3 →
        i++
        ready[ i+1, 3] := true
        send my_server.find_match( ready, found)
        receive found( success, who, what)
        if success →
          call others[i+1].startA( A)
          done[ i] := false
```

```
    [] else →
      write("error, no match in", myid)
      stop
    fi
    ready[ i+1, 3] := false
  od
  notify X4 to start
  ready[ 5, 4] := true
  send my_server.find_match( ready, found)
  receive found( success, who, what)
  if success →
    call others[ 5].start()
    done[ 4] := false
  [] else →
    write("error, no match in", myid)
    stop
  fi
  ready[ 5, 4] := false
  loop to wait for end msgs from the X2s
  do success →
    var guard_possible := false
    fa i := 1 to 4 →
      ready[ i + 1, 5] := not done[ i]
      guard_possible := guard_possible or not done[ i]
    af
    if guard_possible →
      send my_server.find_match( ready, found)
      receive found( success, who, what)
      if success →
        receive in_end()
        done[ who - 1] := true
      [] else →
        write("error, no match in", myid)
        stop
      fi
    [] else →
      received end from all four X2s, loop terminating
      success := false
    fi
```

```
      od
      success := true
      fa i := 1 to 4 →
        ready[ i + 1, 5] := false
      af
    [] else →
      # INPUT terminated, the loop terminates
      success := false
    fi
  od
  send my_server.proc_done( )
  send finished()
end

proc CALC( num)
  var who, what : int
  var success : bool := true
  var A[ 3, 3] : int
  set up token to send integer to X2(4)
  send my_server.init_token( 5, 6)
  set up tokens for end msgs to X1 and X2(4)
  send my_server.init_token( 1, 5)
  send my_server.init_token( 5, 7)
  send my_server.token_init_done()
  do success →
    loop to get START msg and matrix from X1
    ready[ 1, 3] := true
    send my_server.find_match( ready, found)
    receive found( success, who, what)
    if success →
      receive startA( A)
    [] else →
      # X1 has terminated, loop terminates
      exit
    fi
    ready[ 1, 3] := false
    var First_col, Second_col, Cofactor : int
    var value : int
    First_col := ( num % 3) + 1
```

```
  Second_col := ( First_col % 3) + 1
  Cofactor := A[ 2, First_col] * A[ 3, Second_col] -
          A[ 3, First_col] * A[ 2, Second_col]
  value := Cofactor * A[ 1, num]
  send value to X2( 4)
  ready[ 5, 6] := true
  send my_server.find_match( ready, found)
  receive found( success, who, what)
  if success →
    call others[ 5].in_int( value)
  [] else →
    write("error, no match in", myid)
    stop
  fi
  ready[ 5, 6] := false
  send end msg to X2(4)
  ready[ 5, 7] := true
  send my_server.find_match( ready, found)
  receive found( success, who, what)
  if success →
    call others[ 5].in_end()
  [] else →
    write("error, no match in", myid)
    stop
  fi
  ready[ 5, 7] := false
  send end msg to X1
  ready[ 1, 5] := true
  send my_server.find_match( ready, found)
  receive found( success, who, what)
  if success →
    call others[ 1].in_end()
  [] else →
    write("error, no match in", myid)
    stop
  fi
  ready[ 1, 5] := false
od
send my_server.proc_done( )
```

```
  send finished()
end

proc TOTAL()
  var who, what : int
  var success : bool := true
  var done[ 3] : bool
  set up token to send integer to process Y
  send my_server.init_token( 6, 1)
  set up token to send end msg to X1
  send my_server.init_token( 1, 5)
  send my_server.token_init_done()
  do success →
    loop to get START msg from X1
    ready[ 1, 4] := true
    send my_server.find_match( ready, found)
    receive found( success, who, what)
    if success →
      receive start()
    [] else →
      # X1 has terminated, loop terminates
      exit
    fi
    ready[ 1, 4] := false
    var partial_value, total_value : int
    var i : int
    total_value := 0
    i := 0
    do i < 3 →
      i++
      done[ i] := false
    od
    loop to receive value and end msgs from other X2s
    do success →
      fa i := 1 to 3 →
        ready[ i + 1, 6] := not done[ i]
        ready[ i + 1, 7] := not done[ i]
      af
      send my_server.find_match( ready, found)
```

```
    receive found( success, who, what)
    if success →
      if ( what = 7) →
        receiving end msg
        receive in_end()
        done[ who - 1] := true
      [] ( what = 6) →
        receiving value
        receive in_int( partial_value)
        total_value := total_value + partial_value
      [] else →
        write("ERROR, illegal match in", myid)
        write("what =",what," who = ", who)
        stop
      fi
    [] else →
      all guards all false, loop terminates
    fi
od
success := true
send value of determinant to Y
ready[ 6, 1] := true
send my_server.find_match( ready, found)
receive found( success, who, what)
if success →
  call others[ 6].in_int( total_value)
[] else →
  write("error, no match in", myid)
  stop
fi
ready[ 6, 1] := false
send end message to X1
ready[ 1, 5] := true
send my_server.find_match( ready, found)
receive found( success, who, what)
if success →
  call others[ 1].in_end()
[] else →
  write("error, no match in", myid)
```

```
        stop
      fi
      ready[ 1, 5] := false
    od
    send my_server.proc_done( )
    send finished()
end

proc INPUT()
    var Matrix1[ 3, 3] : int := ( 5, 7, 9, -2, 4, 12, 1, 2, 32)
    var Matrix2[ 3, 3] : int := ( 4, 8, 12, 10, 7, 32, 1, 2, 3)
    var who, what : int
    var success : bool := true
    set up tokens to send matrix to X1
    send my_server.init_token( 1, 2)
    send my_server.token_init_done()
    send 1st matrix to X1
    write("The first matrix is:")
    fa i := 1 to 3 →
      fa j := 1 to 3 →
        writes(Matrix1[ i, j]," ")
      af
      write()
    af
    ready[ 1, 2] := true
    send my_server.find_match( ready, found)
    receive found( success, who, what)
    if success →
      call others[ 1].in_matrix( Matrix1)
    [] else →
      write("error, no match in", myid)
      stop
    fi
    send second matrix to X1
    write("The second matrix is:")
    fa i := 1 to 3 →
      fa j := 1 to 3 →
        writes(Matrix2[ i, j]," ")
      af
```

```
    write()
  af
  ready[ 1, 2] := true
  send my_server.find_match( ready, found)
  receive found( success, who, what)
  if success →
    call others[ 1].in_matrix( Matrix2)
  [] else →
    write("error, no match in", myid)
    stop
  fi
  send my_server.proc_done( )
  send finished()
end

proc Y()
  var who, what : int
  var success : bool := true
  var det_value, count : int
  process initially has no tokens
  send my_server.token_init_done()
  count := 1
  do loop to receive value of determinant
  do success →
    ready[ 1, 1] := true
    ready[ 5, 1] := true
    send my_server.find_match( ready, found)
    receive found( success, who, what)
    if success →
      receive in_int( det_value)
      write("The determinant of Matrix ",count,"is ", det_value)
      count++
    [] else →
      processes have terminated, loop terminates
      exit
    fi
  od
  send my_server.proc_done( )
  send finished()
```

end

end determinant

```
Script started on Fri Aug 25 12:17:18 1989
ivy% a.out
The first matrix is:
5 7 9
-2 4 12
1 2 32
The second matrix is:
The determinant of Matrix  1 is  980
4 8 12
10 7 32
1 2 3
The determinant of Matrix  2 is  0
For server  1  there were
19 request messages and 19  found messages and
18 token messages and 0  cancel messages
6  termination messages
For server  2  there were
9 request messages and 9  found messages and
12 token messages and 3  cancel messages
5  termination messages
For server  3  there were
9 request messages and 9  found messages and
12 token messages and 3  cancel messages
5  termination messages
For server  4  there were
9 request messages and 9  found messages and
12 token messages and 3  cancel messages
5  termination messages
For server  5  there were
21 request messages and 21  found messages and
27 token messages and 12  cancel messages
5  termination messages
For server  6  there were
3 request messages and 3  found messages and
2 token messages and 0  cancel messages
1  termination messages
For server  7  there were
2 request messages and 2  found messages and
2 token messages and 0  cancel messages
```

7 termination messages
ivy% exit
ivy%
script done on Fri Aug 25 12:17:26 1989

```
this a CSP solution to calculate the determinant of a 3 x 3
matrix using CSP code containing nesting,
this version uses the modified centralized server.
resource det_main
   import central_csp_server
   import determinant

body det_main()
   op done()

   process start
      var the_server : cap central_csp_server
      var processes[ 1: 7] : cap determinant
      var parent[ 1 : 7] : int := (0, 1, 1, 1, 1, 0, 0)
      the_server := create central_csp_server( 7, parent)
      fa i := 1 to 7 →
         processes[ i] := create determinant( i, 7, the_server,
                                              done)
      af
      send links to all the processes
      fa i := 1 to 7 →
        send processes[ i].links( processes)
      af
      fa i := 1 to 7 →
         receive done()
      af
      destroy all the processes
      fa i := 1 to 7 →
        destroy processes[ i]
      af
      destroy the server
      destroy the_server
   end
end det_main
```

this resource contains the code for the CSP processes used to calculate
the determinant of a 3 x 3 matrix. This version uses the modified
centralized server.

```
resource determinant
   import central_csp_server
   op links( others[ 1: *] : cap determinant)
   const num_requests := 7
   op in_matrix( matrix[ 3, 3] : int)
   op in_end()
   op in_int( int_value : int)
   op start()
   op startA( matrix[ 3, 3] : int)

body determinant( myid, tot_procs :int;
          my_server : cap central_csp_server; finished : cap())

      var others[ 1: tot_procs] : cap determinant
      var ready[ 1 : tot_procs, num_requests] : bool
                        := ([tot_procs*num_requests]false)
      op found( success : bool; match_proc, request : int)
      op X1()
      op CALC( num: int)
      op TOTAL()
      op Y()
      op INPUT()

      initial
        reply
        receive links( others)
        if myid = 1 →
          send X1()
        [] myid ≥ 2 and myid ≤ 4 →
          send CALC( myid - 1)
        [] myid = 5 →
          send TOTAL()
        [] myid = 6 →
          send Y()
        [] myid = 7 →
          send INPUT()
```

```
  fi
end

proc X1()
  var who, what : int
  var success : bool := true
  var A [ 3, 3] : int
  var done[ 4] : bool
  do success →
    loop to get next matrix from INPUT
    ready[ 7, 2] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
      if ( what = 2) →
        receive in_matrix( A)
      [] else →
        write("ERROR, illegal match in", myid)
        write("what =",what," who = ", who)
        stop
      fi
      ready[ 7, 2] := false
      call server to awake children
      call my_server.awaken( myid)
      do loop to notify X2(1)..X2(3) to start
      fa i := 1 to 3 →
        send others[i+1].startA( A)
      af
      notify X4 to start
      send others[ 5].start()
      loop to wait for end msgs from the X2s
      fa i := 1 to 4 →
        receive in_end()
      af
    [] else →
      # INPUT terminated, the loop terminates
      success := false
    fi
  od
```

```
  notify children that they are finished
  fa i := 1 to 4 →
    send others[ i+1].in_end()
  af
  send my_server.proc_done( myid)
  send finished()
end

proc CALC( num)
  var who, what : int
  var success : bool := true
  var A[ 3, 3] : int
  do true →
    loop to get START msg and matrix from X1
    in startA( temp_A) →
      A := temp_A
    [] in_end() →
      exit
    ni
    var First_col, Second_col, Cofactor : int
    var value : int
    First_col := ( num % 3) + 1
    Second_col := ( First_col % 3) + 1
    Cofactor := A[ 2, First_col] * A[ 3, Second_col] -
                A[ 3, First_col] * A[ 2, Second_col]
    value := Cofactor * A[ 1, num]
    send value to X2( 4)
    ready[ 5, 6] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
      call others[ 5].in_int( value)
    [] else →
      write("error, no match in", myid)
      stop
    fi
    ready[ 5, 6] := false
    send my_server.proc_done( myid)
    send others[ 1].in_end()
```

```
  od
  send finished()
end

proc TOTAL()
  var who, what : int
  var success : bool := true
  var done[ 3] : bool
  do true →
    loop to get START msg from X1
    in start() →
    [] in_end() →
      # X1 has terminated, loop terminates
      exit
    ni
    var partial_value, total_value : int
    var i : int
    total_value := 0
    i := 0
    do i < 3 →
      i++
      done[ i] := false
    od
    loop to receive value and end msgs from other X2s
    do success →
      fa i := 1 to 3 →
        ready[ i + 1, 6] := not done[ i]
      af
      send my_server.find_match( ready, myid, found)
      receive found( success, who, what)
      if success →
        if ( what = 6) →
          receiving value
          receive in_int( partial_value)
          total_value := total_value + partial_value
        [] else →
          write("ERROR, illegal match in", myid)
          write("what =",what," who = ", who)
          stop
```

```
        fi
      [] else →
        all guards all false, loop terminates
      fi
    od
    success := true
    send value of determinant to Y
    ready[ 6, 1] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
      call others[ 6].in_int( total_value)
    [] else →
      write("error, no match in", myid)
      stop
    fi
    ready[ 6, 1] := false
    send end message to X1
    send my_server.proc_done( myid)
    send others[ 1].in_end()
  od
  send finished()
end

proc INPUT()
  var Matrix1[ 3, 3] : int := ( 5, 7, 9, -2, 4, 12, 1, 2, 32)
  var Matrix2[ 3, 3] : int := ( 4, 8, 12, 10, 7, 32, 1, 2, 3)
  var who, what : int
  var success : bool := true
  send 1st matrix to X1
  write("The first matrix is:")
  fa i := 1 to 3 →
    fa j := 1 to 3 →
      writes(Matrix1[ i, j]," ")
    af
    write()
  af
  ready[ 1, 2] := true
  send my_server.find_match( ready, myid, found)
```

```
  receive found( success, who, what)
  if success →
    call others[ 1].in_matrix( Matrix1)
   [] else →
    write("error, no match in", myid)
    stop
  fi
  send second matrix to X1
  write("The second matrix is:")
  fa i := 1 to 3 →
    fa j := 1 to 3 →
      writes(Matrix2[ i, j]," ")
    af
    write()
  af
  ready[ 1, 2] := true
  send my_server.find_match( ready, myid, found)
  receive found( success, who, what)
  if success →
    call others[ 1].in_matrix( Matrix2)
   [] else →
    write("error, no match in", myid)
    stop
  fi
  send my_server.proc_done( myid)
  send finished()
end

proc Y()
  var who, what : int
  var success : bool := true
  var det_value, count : int
  count := 1
  do loop to receive value of determinant
  do success →
    ready[ 1, 1] := true
    send my_server.find_match( ready, myid, found)
    receive found( success, who, what)
    if success →
```

```
            receive in_int( det_value)
            write("The determinant of Matrix ",count,"is ",det_value)
            count++
          [] else →
            processes have terminated, loop terminates
            exit
          fi
        od
        send my_server.proc_done( myid)
        send finished()
      end

end determinant
```

```
Resource to handle in a centralized manner the I/O requests
for CSP processes.  This version is modified to handle
nested processes.
resource central_csp_server
   op proc_done( proc_id : int)
   const num_requests := 7
   optype found_op = ( success : bool; who, what : int)
   op find_match( ready[ 1:*, 1:num_requests] : bool; process_num : int
               found : cap found_op)
   op awaken( myid : int)

body central_csp_server( N:int; parent[1:*] : int)

      var term_procs[ 1 : N] : bool := ( [N] false)
      type ready_rec = rec ( procnum : int; ready[ N, num_requests] : bool,
                      ready_count : int; found : cap found_op)
      op ready_op ( ready_record : ready_rec)
      var parent[ 1 : N] : int

      var request_count := 0, found_match := 0, no_match := 0
      var proc_done_msgs := 0

      process main
         var count : int
         do true →
            in find_match( in_ready, who, found) →
              request_count++
              var request : int
              var match_found : bool := false
              count := ?ready_op
              do count > 0 and not match_found →
                in ready_op( next_ready) →
                  fa j := 1 to num_requests →
                    check to see if any processes are
                    children that match
                    var ancestor_who := who
                    var ancestor_next : int
                    do ancestor_who ≠ 0
                      and not match_found →
```

```
            if next_ready.ready[ ancestor_who, j] →
              ancestor_next := next_ready.procnum
              do ancestor_next ≠ 0
                and not match_found →
                if in_ready[ ancestor_next, j] →
                  match_found := true
                  request := j
                 [] else →
                  ancestor_next := parent[ ancestor_next]
                fi
              od
            fi
            ancestor_who := parent[ ancestor_who]
          od
        af
        if not match_found →
          send ready_op( next_ready)
         [] else →
          send next_ready.found( true, who, request)
          send found( true, next_ready.procnum, request)
          found_match := found_match + 2
        fi
        count–
    ni
od
to maintain the order of requests, requests
that were not checked above will be moved to the
end of the queue
if count ≠ ?ready_op →
  do count > 0 →
    in ready_op( ready_rec) →
      send ready_op( ready_rec)
    ni
    count–
  od
fi
if not match_found →
  count := 0
  fa i := 1 to N st i ≠ who, j := 1 to num_requests
```

```
        st in_ready[ i, j] →
          if term_procs[ i] →
            in_ready[ i, j] := false
          [] else →
            count++
          fi
        af
        if count > 0 →
          send ready_op(
                ready_rec ( who, in_ready, count, found))
        [] else →
          send found( false, 0, 0)
          no_match++
        fi
      fi
    [] proc_done ( who) →
      proc_done_msgs++
      term_procs[ who] := true
      count := ?ready_op
      do count > 0 →
        in ready_op( next_ready) →
          fa j := 1 to num_requests st next_ready.ready[ who, j] →
            next_ready.ready[ who, j] := false
            next_ready.ready_count–
          af
          if next_ready.ready_count = 0 →
            send next_ready.found( false, 0, 0)
            no_match++
          [] else →
            send ready_op( next_ready)
          fi
        ni
        count–
      od
    [] awaken( whose_children) →
     fa i := 1 to N st parent[ i] = whose_children →
          term_procs[ i] := false
     af
  ni
```

```
    od
  end

  final
    write("The total number of I/O requests msgs was ",request_count)
    write("The number of found messages with match ",found_match)
    write("The number of found messages with no match", no_match)
    write("The number of proc_done messages", proc_done_msgs)
  end
end central_csp_server
```

```
Script started on Fri Aug 25 09:06:54 1989
ivy% a.out
The first matrix is:
5 7 9
-2 4 12
1 2 32
The second matrix is:
The determinant of Matrix  1 is  980
4 8 12
10 7 32
1 2 3
The determinant of Matrix  2 is  0
The total number of I/O requests msgs was  24
The number of found messages with match  20
The number of found messages with no match 4
The number of proc_done messages 11
ivy% exit
ivy%
script done on Fri Aug 25 09:06:59 1989
```

```
this a CSP solution to calculate the determinant of a 3 x 3
matrix using CSP code containing nesting,
this version uses the modified distributed servers.
resource det_main
   import csp_server
   import determinant

body det_main()
   op done()

   process start
      var servers[ 1 : 7] : cap csp_server
      var processes[ 1: 7] : cap determinant
      var parent[ 1 : 7] : int := (0, 1, 1, 1, 1, 0, 0)
      fa i := 1 to 7 →
         servers[ i] := create csp_server( i, 7, parent)
         processes[ i] := create determinant( i, 7, servers[ i],
                                  done)
      af
      send links to all the processes
      fa i := 1 to 7 →
        send processes[ i].links( processes)
        send servers[ i].links( servers)
      af
      fa i := 1 to 7 →
         receive done()
      af
      destroy all the processes and servers
      fa i := 1 to 7 →
        destroy processes[ i]
        destroy servers[ i]
      af
   end
end det_main
```

```
this resource contains the code for the CSP processes used to calculate
the determinant of a 3 x 3 matrix.  This version uses the modified
distributed servers.
resource determinant
   import csp_server
   op links( others[ 1: *] : cap determinant)
   const num_requests := 7
   op in_matrix( matrix[ 3, 3] : int)
   op in_end()
   op in_int( int_value : int)
   op start()
   op startA( matrix[ 3, 3] : int)

body determinant( myid, tot_procs :int;
          my_server : cap csp_server; finished : cap())

      var others[ 1: tot_procs] : cap determinant
      var ready[ 1 : tot_procs, num_requests] : bool
                           := ([tot_procs*num_requests]false)
      op found( success : bool; match_proc, request : int)
      op X1()
      op CALC( num: int)
      op TOTAL()
      op Y()
      op INPUT()

      initial
        reply
        receive links( others)
        if myid = 1 →
          send X1()
        [] myid ≥ 2 and myid ≤ 4 →
          send CALC( myid - 1)
        [] myid = 5 →
          send TOTAL()
        [] myid = 6 →
          send Y()
        [] myid = 7 →
          send INPUT()
```

```
  fi
end

proc X1()
  var who, what : int
  var success : bool := true
  var A [ 3, 3] : int
  process X1 has no tokens initially
  send my_server.token_init_done()
  do success →
    loop to get next matrix from INPUT
    ready[ 7, 2] := true
    send my_server.find_match( ready, found)
    receive found( success, who, what)
    if success →
      if ( what = 2) →
        receive in_matrix( A)
      [] else →
        write("ERROR, illegal match in", myid)
        write("what =",what," who = ", who)
        stop
      fi
      ready[ 7, 2] := false
      call server to awake children
      call my_server.awaken()
      do loop to notify X2(1)..X2(3) to start
      fa i := 1 to 3 →
        send others[i+1].startA( A)
      af
      notify X4 to start
      send others[ 5].start()
      loop to wait for end msgs from the X2s
      fa i := 1 to 4 →
        receive in_end()
      af
    [] else →
      # INPUT terminated, the loop terminates
      success := false
    fi
```

```
  od
  notify children that they are finished
  fa i := 1 to 4 →
    send others[ i+1].in_end()
  af
  send my_server.proc_done( )
  send finished()
end

proc CALC( num)
  var who, what : int
  var success : bool := true
  var A[ 3, 3] : int
  set up token to send integer to X2(4)
  send my_server.init_token( 5, 6)
  send my_server.token_init_done()
  do true →
    loop to get START msg and matrix from X1
    in startA( temp_A) →
      A := temp_A
    [] in_end() →
      exit
    ni
    var First_col, Second_col, Cofactor : int
    var value : int
    First_col := ( num % 3) + 1
    Second_col := ( First_col % 3) + 1
    Cofactor := A[ 2, First_col] * A[ 3, Second_col] -
            A[ 3, First_col] * A[ 2, Second_col]
    value := Cofactor * A[ 1, num]
    send value to X2( 4)
    ready[ 5, 6] := true
    send my_server.find_match( ready, found)
    receive found( success, who, what)
    if success →
      call others[ 5].in_int( value)
    [] else →
      write("error, no match in", myid)
      stop
```

```
    fi
    ready[ 5, 6] := false
    send my_server.proc_done( )
    send others[ 1].in_end()
  od
  send finished()
end

proc TOTAL()
  var who, what : int
  var success : bool := true
  var done[ 3] : bool
  set up token to send integer to process Y
  send my_server.init_token( 6, 1)
  send my_server.token_init_done()
  do true →
    loop to get START msg from X1
    in start() →
     [] in_end() →
      # X1 has terminated, loop terminates
      exit
    ni
    var partial_value, total_value : int
    var i : int
    total_value := 0
    i := 0
    do i < 3 →
      i++
      done[ i] := false
    od
    loop to receive value and end msgs from other X2s
    do success →
      fa i := 1 to 3 →
        ready[ i + 1, 6] := not done[ i]
      af
      send my_server.find_match( ready, found)
      receive found( success, who, what)
      if success →
        if ( what = 6) →
```

```
            receiving value
            receive in_int( partial_value)
            total_value := total_value + partial_value
           [] else →
            write("ERROR, illegal match in", myid)
            write("what =",what," who = ", who)
            stop
           fi
         [] else →
           all guards all false, loop terminates
         fi
      od
      success := true
      send value of determinant to Y
      ready[ 6, 1] := true
      send my_server.find_match( ready, found)
      receive found( success, who, what)
      if success →
        call others[ 6].in_int( total_value)
       [] else →
        write("error, no match in", myid)
        stop
      fi
      ready[ 6, 1] := false
      send end message to X1
      send my_server.proc_done( )
      send others[ 1].in_end()
   od
   send finished()
end

proc INPUT()
   var Matrix1[ 3, 3] : int := ( 5, 7, 9, -2, 4, 12, 1, 2, 32)
   var Matrix2[ 3, 3] : int := ( 4, 8, 12, 10, 7, 32, 1, 2, 3)
   var who, what : int
   var success : bool := true
   set up tokens to send matrix to X1
   send my_server.init_token( 1, 2)
   send my_server.token_init_done()
```

```
  send 1st matrix to X1
  write("The first matrix is:")
  fa i := 1 to 3 →
    fa j := 1 to 3 →
      writes(Matrix1[ i, j]," ")
    af
    write()
  af
  ready[ 1, 2] := true
  send my_server.find_match( ready, found)
  receive found( success, who, what)
  if success →
    call others[ 1].in_matrix( Matrix1)
  [] else →
    write("error, no match in", myid)
    stop
  fi
  send second matrix to X1
  write("The second matrix is:")
  fa i := 1 to 3 →
    fa j := 1 to 3 →
      writes(Matrix2[ i, j]," ")
    af
    write()
  af
  ready[ 1, 2] := true
  send my_server.find_match( ready, found)
  receive found( success, who, what)
  if success →
    call others[ 1].in_matrix( Matrix2)
  [] else →
    write("error, no match in", myid)
    stop
  fi
  send my_server.proc_done( )
  send finished()
end

proc Y()
```

```
    var who, what : int
    var success : bool := true
    var det_value, count : int
    process initially has no tokens
    send my_server.token_init_done()
    count := 1
    do loop to receive value of determinant
    do success →
      ready[ 1, 1] := true
      send my_server.find_match( ready, found)
      receive found( success, who, what)
      if success →
        receive in_int( det_value)
        write("The determinant of Matrix ",count,"is ",det_value)
        count++
      [] else →
        processes have terminated, loop terminates
        exit
      fi
    od
    send my_server.proc_done( )
    send finished()
  end

end determinant
```

Resource to handle in a distributed manner the I/O requests of *CSP* processes - using Bagrodia's algorithm. This version is *modified* to handle nested processes.

```
resource csp_server
   import Token_decls
   optype found_op = ( success : bool; who, what : int)
   op in_token : in_token_op
   op cancel : cancel_op
   op term : termination_op
   const num_requests := 7
   op find_match( ready_param[ 1 : *, 1 : num_requests] : bool;
               found_param : cap found_op)
   op proc_done()
   op links( io_servers[ 1: *]: cap csp_server)
   op init_token( who, request : int)
   op token_init_done()
   op awaken()
   op awakened( whose_children : int)

body csp_server( myid, N: int; parent[1:*] : int)

      type color_stat = enum( black, white, grey)
      var color : color_stat
      var ready_count : int
      op token_q : ( token: token_typ)
      op delay_q : ( token: token_typ)
      op temp_q : ( token: token_typ)
      op not_ready_q : ( token: token_typ)
      var io_servers[ 1 : N] : cap csp_server
      var ready[ 1 : N, 1 : num_requests] : bool
      var found : cap found_op
      var term_procs[ 1 : N] : bool := ([N]false)
      op Refuse_token( token : token_typ)
      op Request_token() returns success : bool
      op Deny_token( token : token_typ)
      op Delay_token( token : token_typ)
      op Commit_token( token : token_typ)
      op Commit_delayed() returns token : token_typ
      op Refuse_delayed()
```

```
op Initialize()
op main_prot()
op check_ready( who, what : int) returns is_ready : bool

var request_count := 0, token_sends := 0, cancel_sends := 0
var found_match := 0, no_match := 0, termination_msgs := 0

initial
  reply
  var init_done := false
  var token : token_typ
  token.process_id := myid
  token.process_list[ 1] := myid
  do not init_done →
    receive from our process the token
    that this server initially holds
    in init_token( who, request) →
      token.process_list[ 2] := who
      token.request := request
      send not_ready_q( token)
      if the other process (i.e., who) has children
      add a token for each of its descendants
      var parent_id : int
      fa procs := 1 to N st procs ≠ who →
        parent_id := parent[ procs]
        do parent_id ≠ 0 and parent_id ≠ who →
          parent_id := parent[ parent_id]
        od
        if parent_id = who →
          token.process_list[ 2] := procs
          send not_ready_q( token)
        fi
      af
    [] token_init_done() →
      init_done := true
    ni
  od
  receive links( io_servers)
  send main_prot()
```

```
end

proc main_prot()
  color := black
  var match_token : token_typ
  var not_done := true
  do not_done →
    do color = black →
      in in_token( token) →
        Refuse_token( token)
      [] find_match( ready_param, found_param) →
        request_count++
        found := found_param
        fa i := 1 to N, j := 1 to num_requests →
          ready[ i, j] := ready_param[ i, j]
        af
        Initialize()
        color := white
      [] proc_done() →
        termination_msgs++
        fa i := 1 to N st (i ≠ myid and not term_procs[i]) →
          send io_servers[ i].term( myid)
          termination_msgs++
        af
      [] term( who) →
        term_procs[ who] := true
      [] awaken() →
        fa i := 1 to N →
         send io_servers[ i].awakened( myid)
        af
      [] awakened( whose_children) →
        fa i := 1 to N st parent[ i] = whose_children →
          term_procs[ i] := false
        af
      ni
    od
    do color ≠ black and ready_count > 0 →
      if color = white and (?token_q ≠ 0) →
        var success : bool
```

```
  success := Request_token()
  if success →
    color := grey
  fi
fi
in in_token( token) and color = white →
  if check_ready( token.process_id, token.request) →
    Commit_token( token)
    match_token := token
    color := black
  [] else →
    Refuse_token( token)
  fi
[] in_token( token) and color = grey →
  if myid = token.process_id →
    Commit_token( token)
    match_token := token
    color := black
  [] else →
    if not check_ready( token.process_id, token.request) →
      Refuse_token( token)
    [] else →
      if myid > token.process_id →
        Deny_token( token)
      [] myid < token.process_id →
        Delay_token( token)
      fi
    fi
  fi
[] cancel() and color = grey →
  if ?delay_q  = 0 →
    color := white
  [] else →
    match_token := Commit_delayed()
    color := black
    send match
  fi
[] term( who) →
  fa j := 1 to num_requests st ready[ who, j] →
```

```
            ready[ who, j] := false
            ready_count–
          af
          term_procs[ who] := true
        ni
      od
      if ready_count = 0 →
        did not find a matching I/O command
        notify client, no match was found
        send found( false, 0, 0)
        no_match++
        if color = grey →
          sent request to process that has terminated
          must wait for cancel message
          receive cancel()
        fi
        color := black
      [] else →
        notify client of process with matching I/O command
        if myid = match_token.process_list[ 1] →
          send found( true, match_token.process_list[ 2],
                      match_token.request)
        [] else →
          send found( true, match_token.process_list[ 1],
                      match_token.request)
        fi
        found_match++
      fi
    od
  end

  proc Refuse_token( token)
    send io_servers[ token.process_id].cancel()
    cancel_sends++
    send not_ready_q( token)
  end

  proc Request_token() returns success
    var send_to : int
```

```
  var token : token_typ
  success := false
  do ?token_q ≠ 0 and not success →
    receive token_q( token)
    token.process_id := myid
    if myid = token.process_list[ 1] →
        send_to := token.process_list[ 2]
    [] else →
        send_to := token.process_list[ 1]
    fi
    if check_ready( send_to, token.request) →
      success := true
    [] else →
      send not_ready_q( token)
    fi
  od
  if success →
    send io_servers[ send_to].in_token( token)
    token_sends++
  fi
end

proc Deny_token( token)
  send io_servers[ token.process_id].cancel()
  cancel_sends++
  send token_q( token)
end

proc Delay_token( token)
  send delay_q( token)
end

proc Commit_token( token)
  if token.process_id ≠ myid →
    send io_servers[ token.process_id].in_token( token)
    token_sends++
  [] else →
    send token_q( token)
  fi
```

```
  Refuse_delayed()
end

proc Commit_delayed() returns token
  receive delay_q( token)
  Commit_token( token)
end

proc Refuse_delayed()
  var token : token_typ
  do ?delay_q ≠ 0 →
    receive delay_q( token)
    Refuse_token( token)
  od
end

proc Initialize()
  var token : token_typ
  var other_proc : int
  ready_count := 0
  fa i := 1 to N st i ≠ myid, j := 1 to num_requests →
      if ready[ i, j] →
        if term_procs[ i] →
            ready[ i, j] := false
         [] else →
            ready_count++
        fi
      fi
  af
  do ?token_q ≠ 0 →
    receive token_q( token)
    send not_ready_q( token)
  od
  do ?not_ready_q ≠ 0 →
    receive not_ready_q( token)
    if myid = token.process_list[ 1] →
      other_proc := token.process_list[ 2]
    [] else →
      other_proc := token.process_list[ 1]
```

```
      fi
      if check_ready( other_proc, token.request) →
        send token_q( token)
      [] else →
        send temp_q( token)
      fi
    od
    do ?temp_q ≠ 0 →
      receive temp_q( token)
      send not_ready_q(token)
    od
  end

  proc check_ready( who, what) returns is_ready
    if ready[ who, what] →
      is_ready := true
    [] else →
      is_ready := false
      var proc_num := parent[ who]
      do proc_num ≠ 0 and not is_ready →
        if ready[ proc_num, what] →
          is_ready := true
        [] else →
          proc_num := parent[ proc_num]
        fi
      od
    fi
  end

  final
    write("For server ",myid," there were ")
    writes(request_count," request messages and ")
    write(termination_msgs," termination messages and")
    writes(found_match," found messages with match and ")
    write(no_match," found message with no match and")
    writes(token_sends," token messages and ")
    write(cancel_sends," cancel messages")
  end
end csp_server
```

Script started on Fri Aug 25 09:08:25 1989
ivy% a.out
The first matrix is:
5 7 9
-2 4 12
1 2 32
The second matrix is:
The determinant of Matrix 1 is 980
4 8 12
10 7 32
1 2 3
The determinant of Matrix 2 is 0
For server 1 there were
3 request messages and 2 termination messages and
2 found messages with match and 1 found message with no match and
2 token messages and 0 cancel messages
For server 2 there were
2 request messages and 13 termination messages and
2 found messages with match and 0 found message with no match and
3 token messages and 1 cancel messages
For server 3 there were
2 request messages and 13 termination messages and
2 found messages with match and 0 found message with no match and
3 token messages and 1 cancel messages
For server 4 there were
2 request messages and 10 termination messages and
2 found messages with match and 0 found message with no match and
4 token messages and 3 cancel messages
For server 5 there were
10 request messages and 8 termination messages and
8 found messages with match and 2 found message with no match and
10 token messages and 5 cancel messages
For server 6 there were
3 request messages and 4 termination messages and
2 found messages with match and 1 found message with no match and
2 token messages and 0 cancel messages
For server 7 there were
2 request messages and 3 termination messages and
2 found messages with match and 0 found message with no match and

```
6 token messages and 0  cancel messages
ivy% exit
ivy%
script done on Fri Aug 25 09:08:33 1989
```

REFERENCES

[Andr88] Andrews, G.R., et al. "An Overview of the SR Language and Implementation", *ACM Transactions on Programming Languages and Systems*, 10, 1 (Jan. 1988), 51-86.

[Andr89] Andrews, G.R., *Concurrent Programming: Principles and Practice*, In preparation.

[Apt84] Apt, K.R., and Francez, N., "Modeling the Distributed Termination Convention of CSP", *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 3 (July 1984), 370-379.

[Bagr86] Bagrodia, R., "A Distributed Algorithm to Implement the Generalized Alternative Command of CSP", Proc. The 6th International Conference on Distributed Computing Systems, Cambridge, Mass., (May 1986), 422-427.

[Brin72] Brinch Hansen, P. "Structured multiprogramming", *Communications of the ACM*, 15, 7 (July 1972),574-578.

[Brin73] Brinch Hansen, P. *Operating System Principles*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[Brin75] Brinch Hansen, P. "The Programming Language Concurrent Pascal", IEEE Transaction on Software Engineering, SE-1, 2 (June 1975), 199-206.

[Buck83] Buckley, G.N., and Silberschatz, A. "An Effective Implementation for the Generalized Input-output Construct of CSP", *ACM Transactions on Programming Languages and Systems*, 5, 2 (April 1983), 223-235.

[Burn88] Burns, A., *Programming In Occam 2*, Addison-Wesley, Wonkingham, England, 1988.

[Dijk68] Dijkstra, E.W. "The Structure of 'THE' Multiprogramming System", *Communications of the ACM*, 11, 5 (May 1968), 341-346.

[Dijk76] Dijkstra, E.W. *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. 1976.

[Hoar78] Hoare, C.A.R. "Communicating Sequential Processes", *Communications of the ACM*, 21, 8 (Aug. 1978), 666-677.

[Jeff85] Jefferson, D., "Virtual Time", *ACM Transacations on Programming Languates and Systems*, 7, 3, (July 1985), 404-425.

[Lamp80] Lampson, B.W., and Redell, D. D., "Experience with Processes and Monitors in Mesa", *Communications of the ACM*, 23, 2 (Feb. 80) 105-117.

[Lamp78] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, 21, 7 (July 1978) 558-565.

[Laue78] Lauer, H.C., and Needham, R.M., "On the Duality of Operating Systems Structures", Proc. Second International Symposium on Operating Systems, IRIA, (Oct. 1978), reprinted in *ACM Operating Systems Review*, 13,2 (April 1979), 3-19.

[Rame87] Ramesh, S., "A New Efficient Implementation of CSP With Output Guards", Proc. The 7th International Conference on Distributed Computing Systems, Berlin, W. Germany, (Sept. 1987), 266-273.

[Rayn86] Raynal, M. *Algorithms for Mutual Exclusion*, North Oxford Academic, London, 1986.

[Reed83] Reed, D.P., "Implementing Atomic Actions on Decentralized Data", *ACM Transactions on Computer Systems*, 1, 1 (Feb. 1983), 3-23.

[Rica81] Ricart, G., and Agrawala, A.K., "An Optimal Algorithm for Mutual Exclusion in Computer Networks", *Communications of the ACM*, 24, 1, (Jan. 1981) 9-17.

[Rica83] Ricart, G., and Agrawala, A.K., Author's response to 'On Mutual Exclusion in Computer Networks' by Carvalho and Roucairol, *Communications of the ACM*, 26, 2 (Feb. 1983)147-148.

[Schn80] Schneider, F.B., "Ensuring Consistency in a Distributed Database System by Use of Distributed Semaphores", In Proceedings of International Symposium on Distributed Data Bases, Paris, France, (March 1980) 183-189.

[Schn82] Schneider, F.B., "Synchronization in Distributed Programs", *ACM Transacations on Programming Languages and Systems*, 4, 2, (April 1982), 179-195.

[Schn89] Schneider, F.B., Private Communication, May 1989.

[Scot83] Scott, M.L., "Messages vs. Remote Procedures is a False Dichotomy", *SIGPLAN Notices*, 18, 5 (May 1983), 57-62.

[Silb88] Silberschatz, A. and Peterson, J.L., *Operating System Concepts*, Addison-Wesley, Reading, Massachusetts, 1988.

[Suzu85] Suzuki, I., and Kasami, T., "A Distributed Mutual Exclusion Algorithm", *ACM Transacations on Computer Systems*, 3, 4, (Nov. 1985), 344-349.

[Wels79] Welsh, J., and Bustard, D.W., "Pascal-Plus Another Language for Modula Multiprogramming", Software-Procedure and Experience. 9,11, (Nov. 1979), 947-957.

– –

TABLE OF CONTENTS

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I am greatly indebted to my advisor Dr. Ronald A. Olsson. Thanks to his help, patience, and support I was able to complete this thesis.

I would also like to thank Dr.'s Gene Fisher and Karl Levitt for taking the time to be on my committee.

I am also indebted to my family and friends who listened to my doubts about ever finishing my masters. Their encouragement gave me the strength to finish.