Processes

Raju Pandey
Department of Computer Sciences
University of California, Davis
Spring 2011

Objectives

- What is a Process?
- What are states of a process?
- How are they created?
- How are they represented inside OS?
- What's OS's process namespace?
- How can this be made faster?

Operating System as Virtual Machine

- Virtualize processor
  - Interleave the execution of several processes to maximize processor utilization while providing reasonable response time
- Virtualize resources
  - Virtualize memory, devices
  - Allocate resources to processes
- Manage resource
  - Safety
  - Fairness
- What is core abstraction for virtualization?

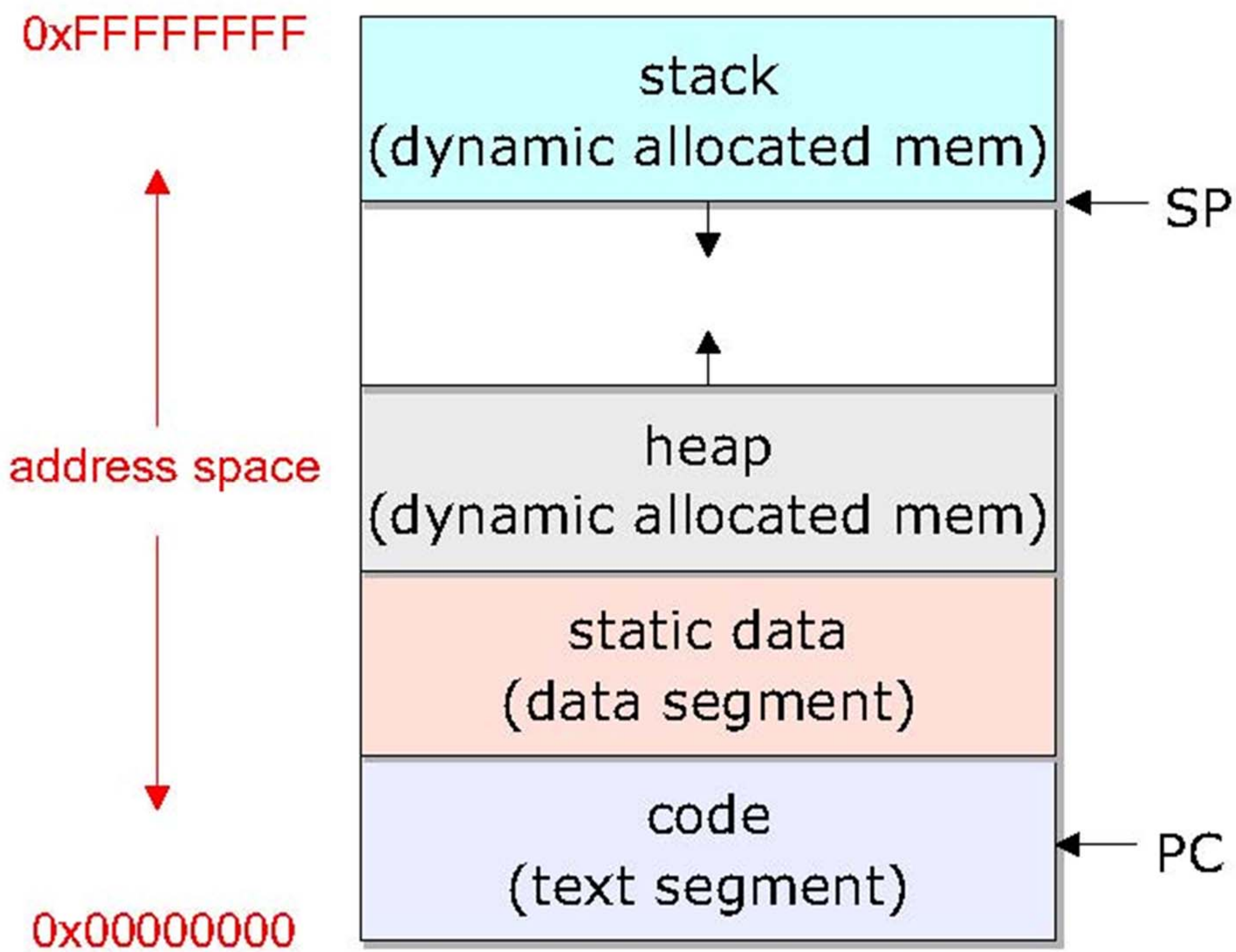
What is a Process?

- Core OS abstraction for virtualization
  - Also called task
- Process =
  - **Unit of execution**: follows an execution path that may be interleaved with other processes
  - **Unit of scheduling**
    - o CPU
    - o : I/O, File, Networking, Display and others
  - **Unit of Execution Context**
    - o Address space: Memory abstraction for holding program executable, state and execution context

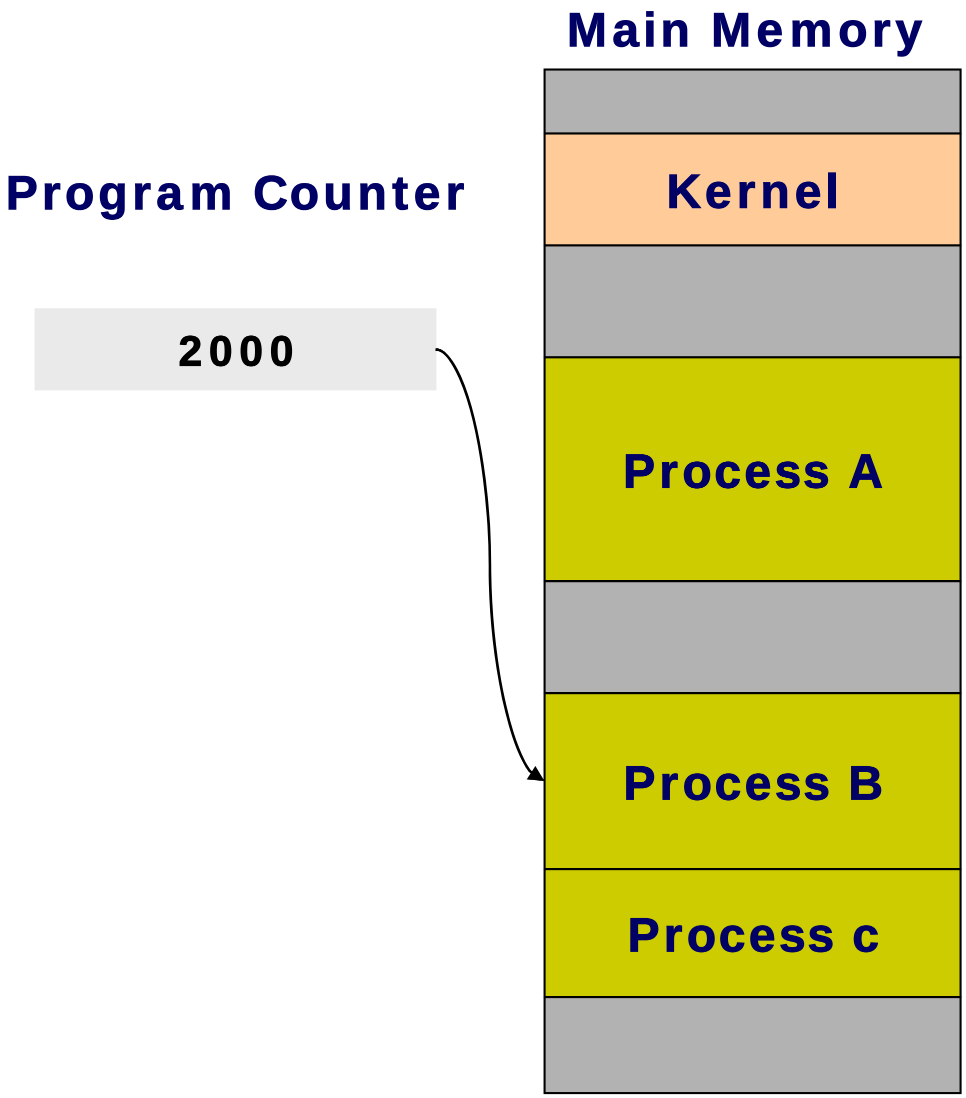
What's "in" a process?

- A process consists of (at least):
  - An address space, containing
    - o the code (instructions) for the running program
    - o the data for the running program
  - Thread state, consisting of
    - o The program counter (PC), indicating the next instruction
    - o The stack pointer register (implying the stack it points to)
    - o Other general purpose register values
  - A set of OS resources
    - o open files, network connections, sound channels, …
- In other words, it's all the stuff you need to run the program
  - or to re-start it, if it's interrupted at some point

Address Space of Processes



Memory Organization



OS Control Structures: Tables

- Memory table
  - Allocation of main memory to processes
  - Allocation of secondary memory to processes
  - Protection attributes for access to shared memory regions
  - Information needed to manage virtual memory
- I/O table:
  - Status of /O device
  - Status of I/O operation
  - Location in main memory being used as the source or destination of the I/O transfer
- File table:
  - Location on secondary memory
  - Current Status
  - Attributes
- Process table
  - Process ID
  - Process state
  - Location in memory

OS Control Structures: Tables

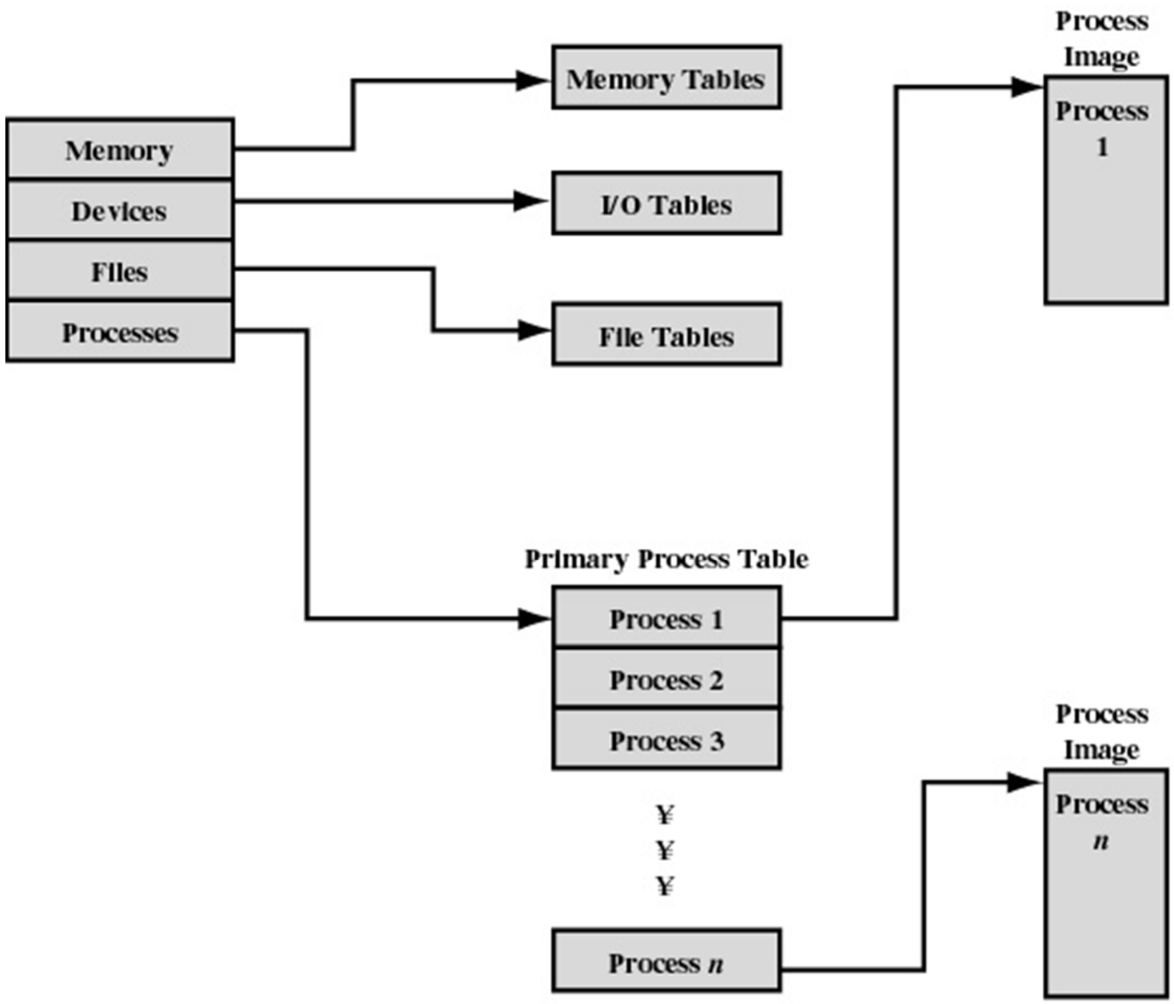


Figure 3.10 General Structure of Operating System Control Tabl

Representation of processes by the OS

- The OS maintains a data structure to keep track of a process's state
  - Called the process control block (PCB)
  - Identified by the PID
- OS keeps all of a process's hardware execution state in the PCB when the process isn't running
  - PC, SP, registers, etc.
  - when a process is unscheduled, the state is transferred out of the hardware into the PCB
  - (when a process is running, its state is spread between the PCB and the CPU)
- Note: It's natural to think that there must be some esoteric techniques being used
  - fancy data structures that'd you'd never think of yourself

Wrong! It's pretty much just what you'd think of!

The OS's process namespace

- (Like most things, the particulars depend on the specific OS, but the principles are general)
- The name for a process is called a process ID (PID)
  - An integer
- The PID namespace is global to the system
  - Only one process at a time has a particular PID
- Operations that create processes return a PID
  - E.g., fork(), clone()
- Operations on processes take PIDs as an argument
  - E.g., kill(), wait(), nice()

The PCB

- The PCB is a data structure with many, many fields:
  - process ID (PID)
  - parent process ID
  - execution state
  - program counter, stack pointer, registers
  - address space info
  - UNIX user id, group id
  - scheduling priority
  - accounting info
  - pointers for state queues
- In Linux:
  - defined in **`task_struct`** (**`include/linux/sched.h`**)
  - over 95 fields!!!

PCBs and hardware state

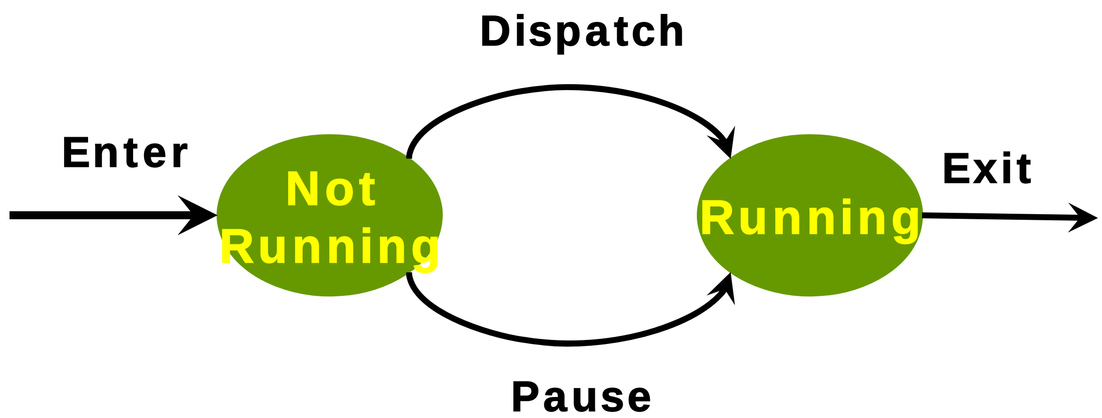
- When a process is running, its hardware state is inside the CPU
  - PC, SP, registers
  - CPU contains current values
- When a process is transitioned to the waiting state, the OS saves its CPU state in the PCB
  - when the OS returns the process to the running state, it loads the hardware registers with values from that process's PCB
- The act of switching the CPU from one process to another is called a context switch
  - systems may do 100s or 1000s of switches/sec.
  - takes a few microseconds on today's hardware
- Choosing which process to run next is called scheduling

Process execution states

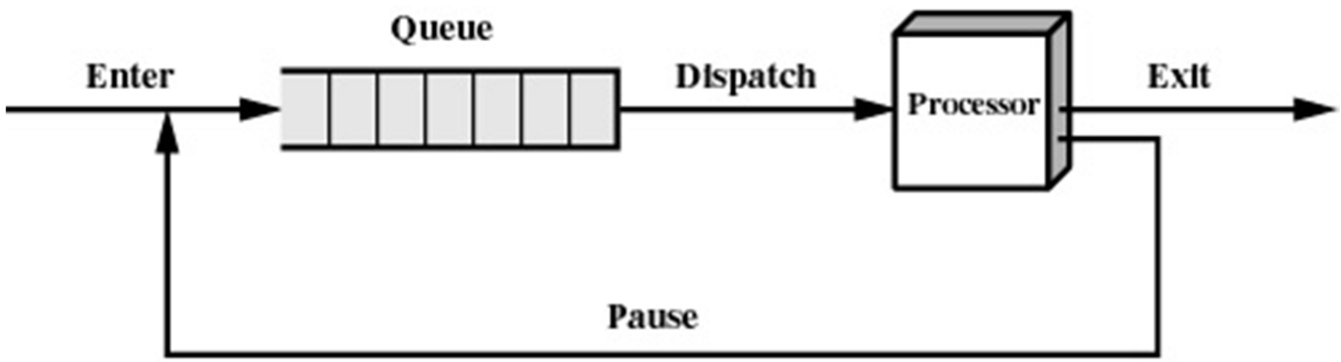
- Each process has an execution state, which indicates what it is currently doing
  - ready: waiting to be assigned to a CPU
    - could run, but another process has the CPU
  - running: executing on a CPU
    - is the process that currently controls the CPU
    - pop quiz: how many processes can be running simultaneously?
  - waiting (aka "blocked"): waiting for an event, e.g., I/O completion
    - cannot make progress until event happens
- As a process executes, it moves from state to state
  - UNIX: run **ps**, STAT column shows current state
  - which state is a process in most of the time?

Two-State Process Model

- States of processes:
  - Running
  - Not-running



- Representation of processes within OS



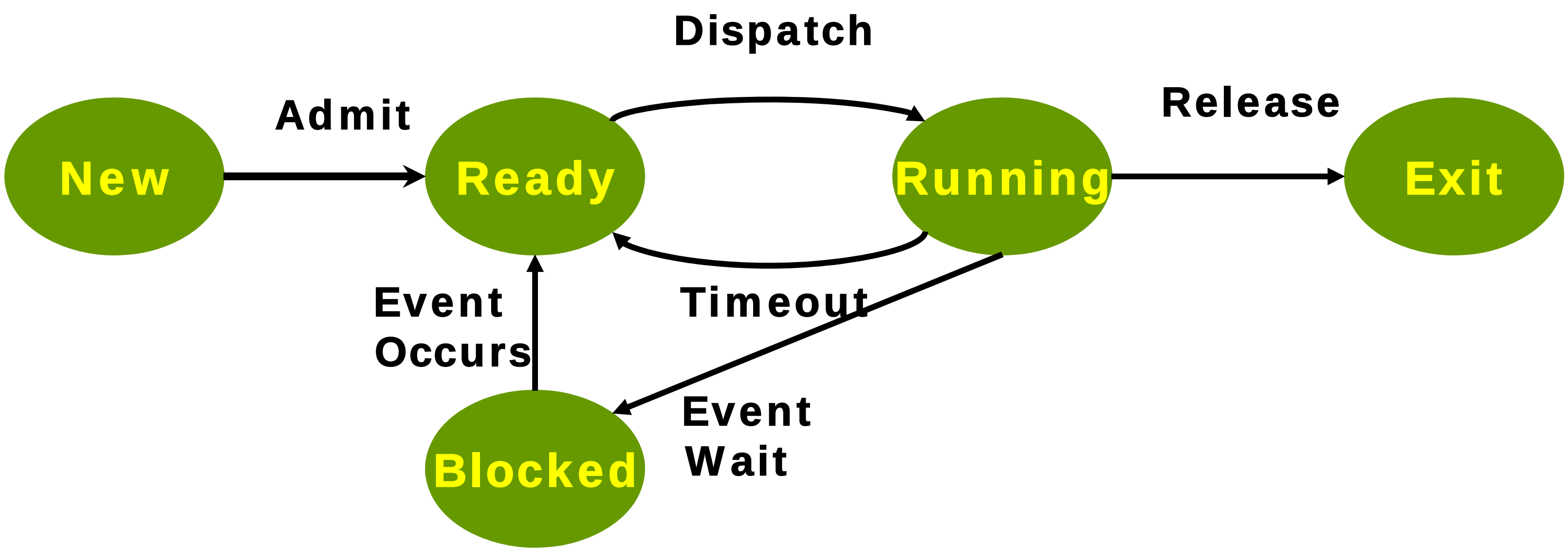
(b) Queuing diagram

How are process created and terminated?

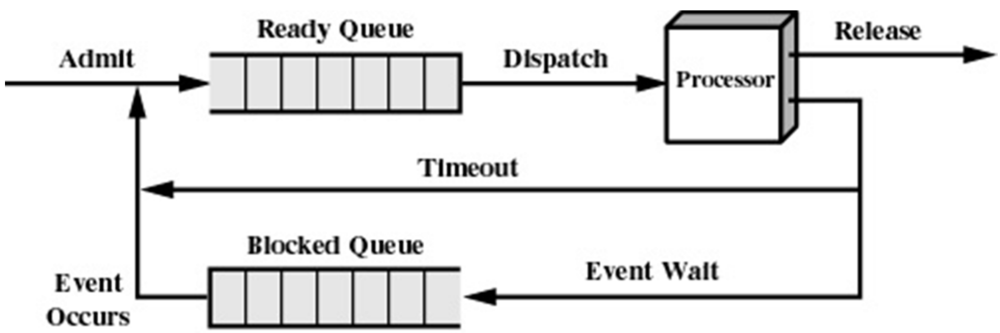
- Creation:
  - Submission of a batch job
  - User logs on
  - Created to provide a service such as printing
  - Process creates another process
- Termination:
  - Normal completion
  - Time limit exceeded
  - Memory unavailable
  - Bounds violation
  - Protection error
    - example write to read-only file
  - Arithmetic error
  - Time overrun
    - process waited longer than a specified maximum for an event

A Five-State Model

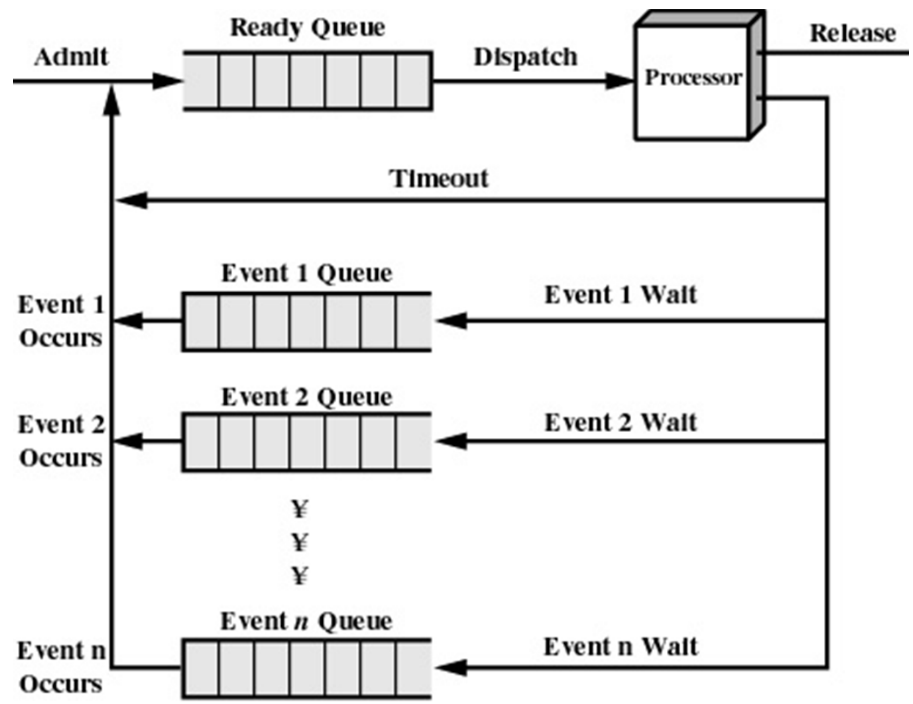
- States: Running, Ready, Blocked, New, Exit



Internal structure



Single Blocked Queue

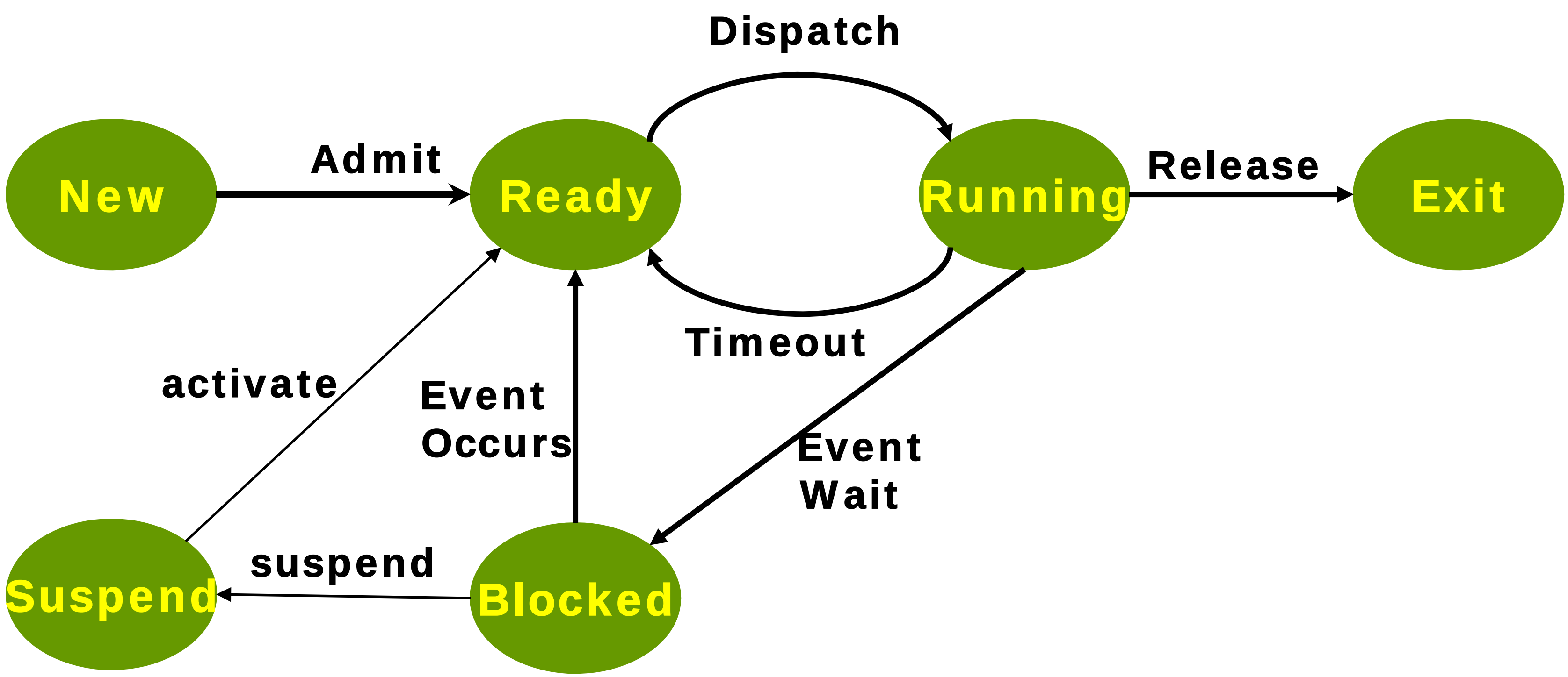


Multiple Blocked Queue

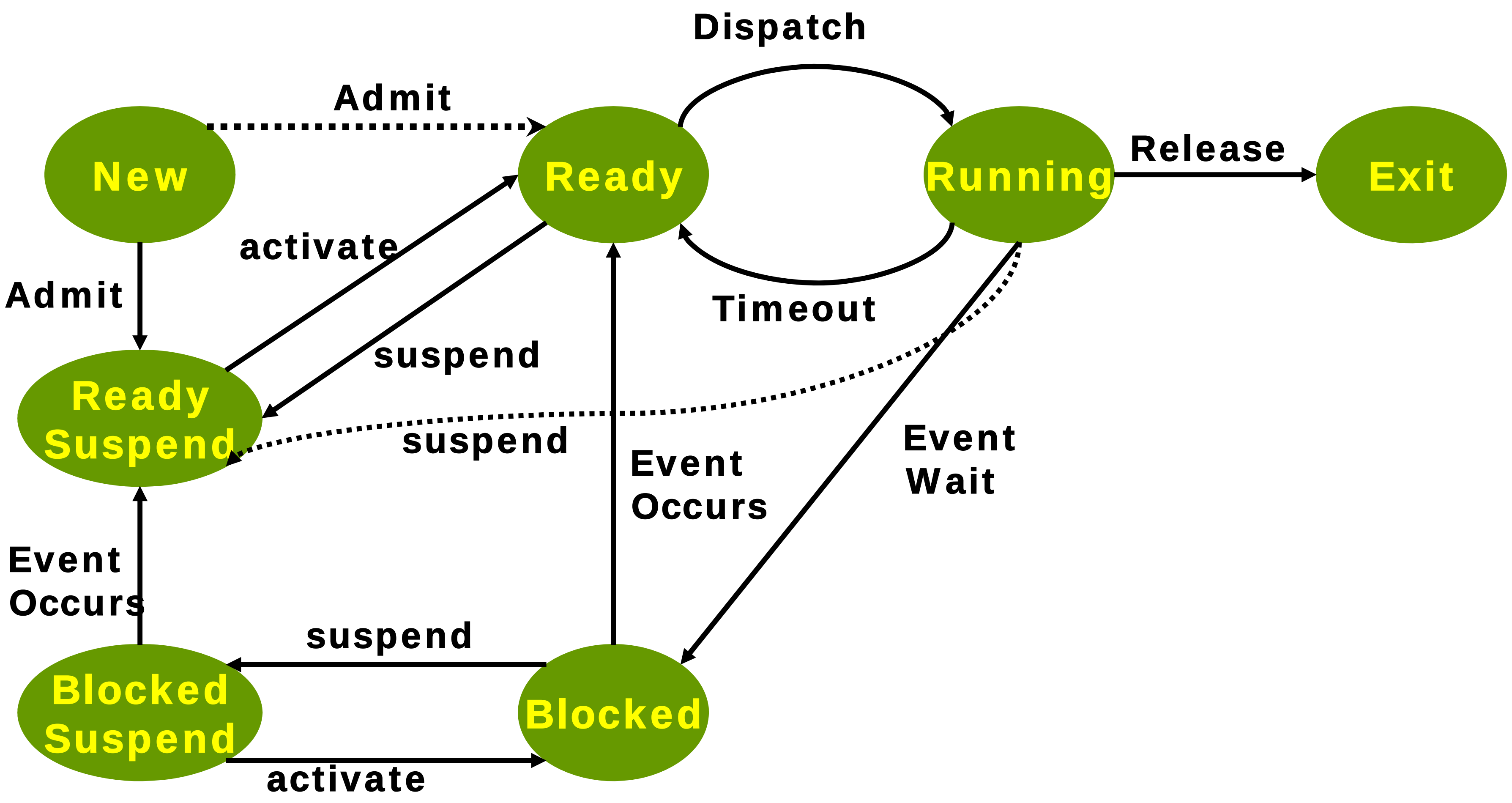
Suspended Processes

- Processor is faster than I/O so all processes could be waiting for I/O
- Reasons for suspension:
  - Swapping: Release main memory
  - Interactive user request: Suspend a program
  - Timing: Periodic execution
  - Parent process request
  - OS initiated: Block a process due to errors
- Swap these processes to disk to free up more memory
- Blocked state becomes suspend state when swapped to disk
  - Blocked, suspend
  - Ready, suspend

One Suspend State



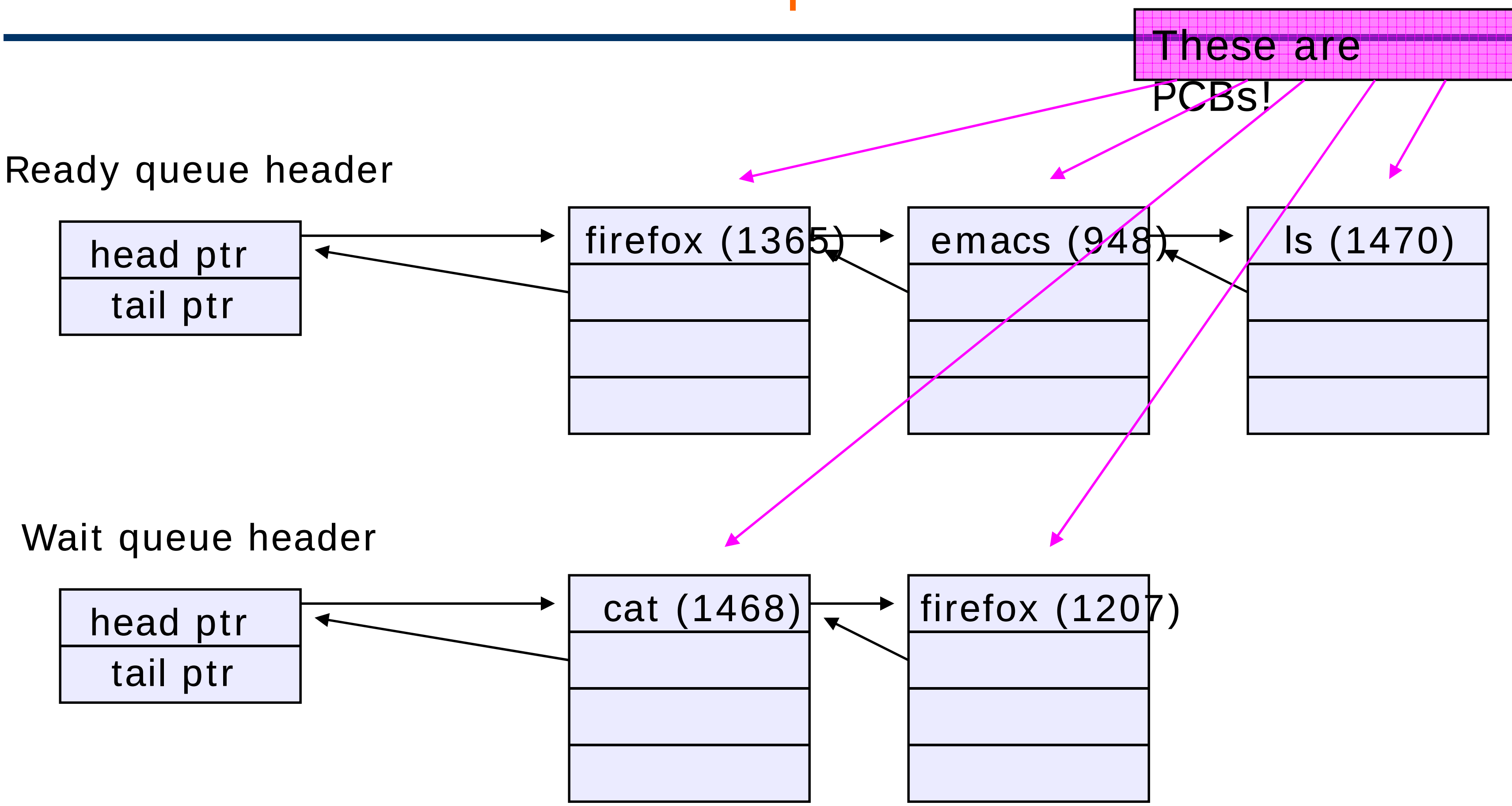
Two Suspend States



State queues

- The OS maintains a collection of queues that represent the state of all processes in the system
  - typically one queue for each state
    - e.g., ready, waiting, …
  - each PCB is queued onto a state queue according to the current state of the process it represents
  - as a process changes state, its PCB is unlinked from one queue, and linked onto another
- Once again, *this is just as straightforward as it sounds!* The PCBs are moved between queues, which are represented as linked lists. *There is no magic!*

State queues



- There may be many wait queues, one for each type of wait (particular device, timer, message, …)

PCBs and state queues

- PCBs are data structures
  - dynamically allocated inside OS memory
- When a process is created:
  - OS allocates a PCB for it
  - OS initializes PCB
  - OS puts PCB on the correct queue
- As a process computes:
  - OS moves its PCB from queue to queue
- When a process is terminated:
  - PCB may hang around for a while (exit code, etc.)
  - eventually, OS deallocates the PCB

Process Creation

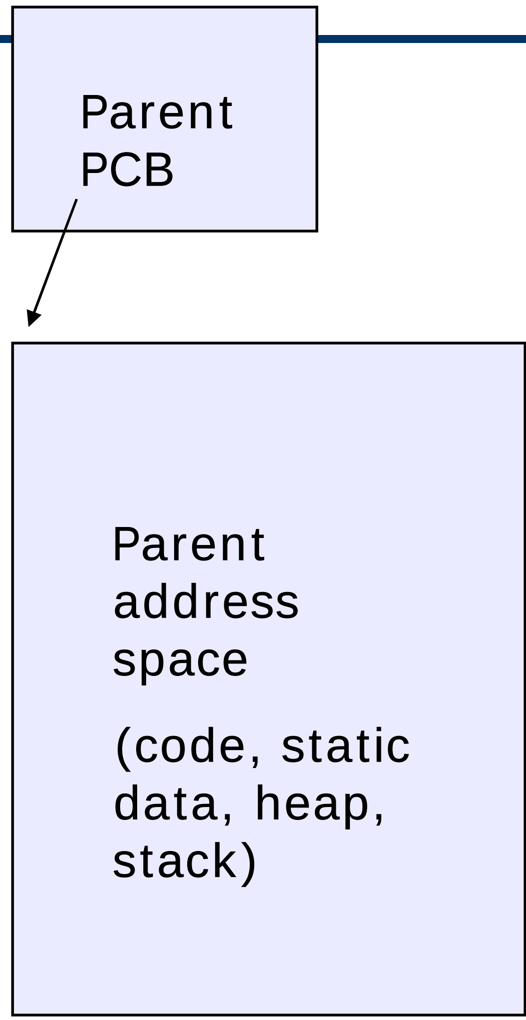
- Assign a unique process identifier
- Allocate space for the process
- Initialize process control block
- Set up appropriate linkages
  - Ex: add new process to linked list used for scheduling queue
- Create of expand other data structures
  - Ex: maintain an accounting file

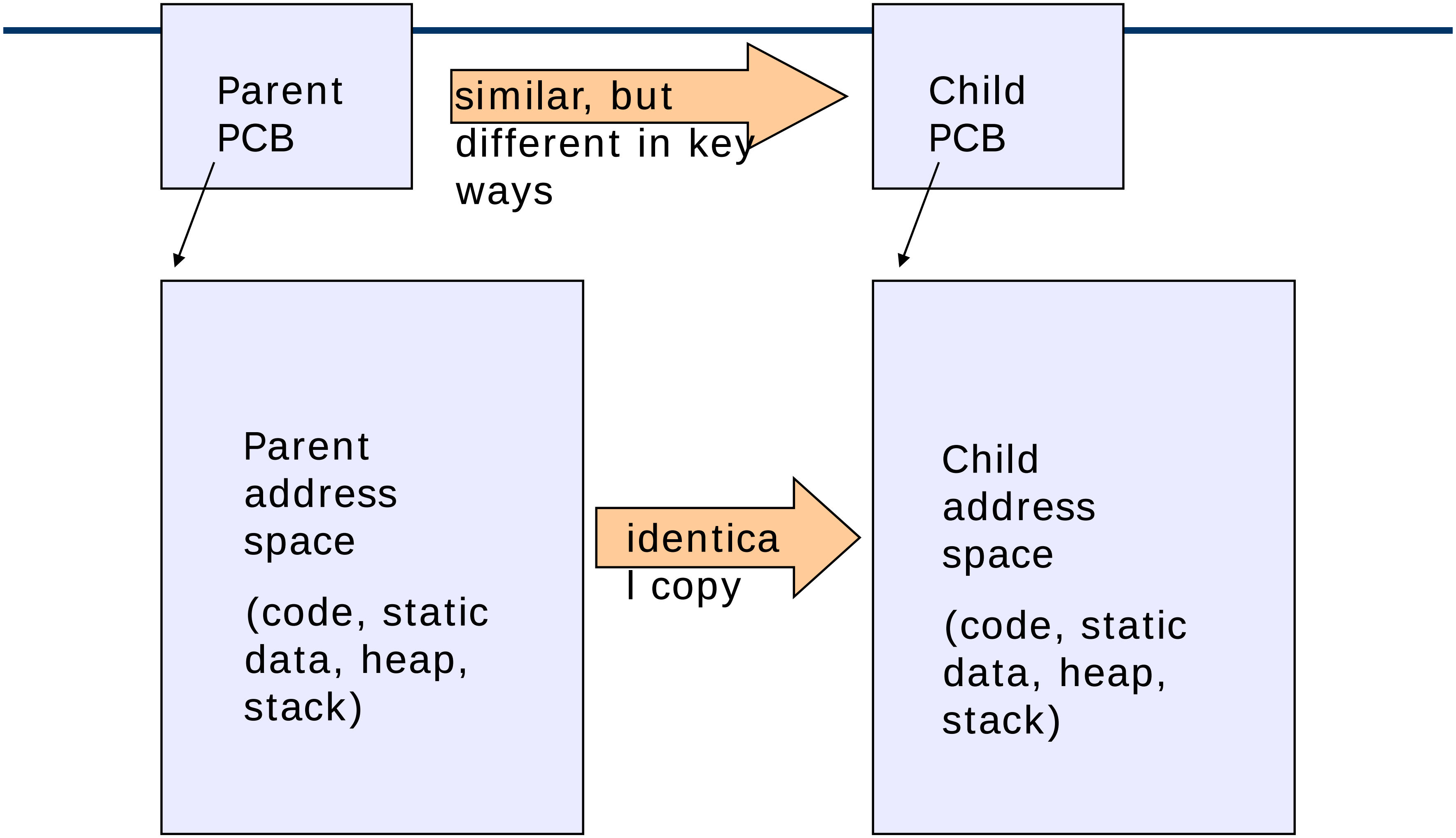
Process creation semantics

- (Depending on the OS) child processes inherit certain attributes of the parent
  - Examples:
    - o Open file table: implies stdin/stdout/stderr
    - o On some systems, resource allocation to parent may be divided among children
- (In Unix) when a child is created, the parent may either wait for the child to finish, or continue in parallel

UNIX process creation details

- UNIX process creation through **`fork()`** system call
  - creates and initializes a new PCB
  - creates a new address space
  - initializes new address space with a copy of the entire contents of the address space of the parent
  - initializes kernel resources of new process with resources of parent (e.g., open files)
  - places new PCB on the ready queue
- the **`fork()`** system call "returns twice"
  - once into the parent, and once into the child
  - returns the child's PID to the parent
  - returns 0 to the child
- **`fork()`** = "clone me"


Parent PCB
Parent address space
(code, static data, heap, stack)


Parent
PCB
similar, but
different in key
ways
Child
PCB
Parent
address
space
(code, static
data, heap,
stack)
identica
l copy
Child
address
space
(code, static
data, heap,
stack)

testparent – use of fork( )

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  char *name = argv[0];
  int pid = fork();
  if (pid == 0) {
    printf(“Child of %s is %d\n”, name, pid);
    return 0;
  } else {
    printf(“My child is %d\n”, pid);
    return 0;
  }
}
```

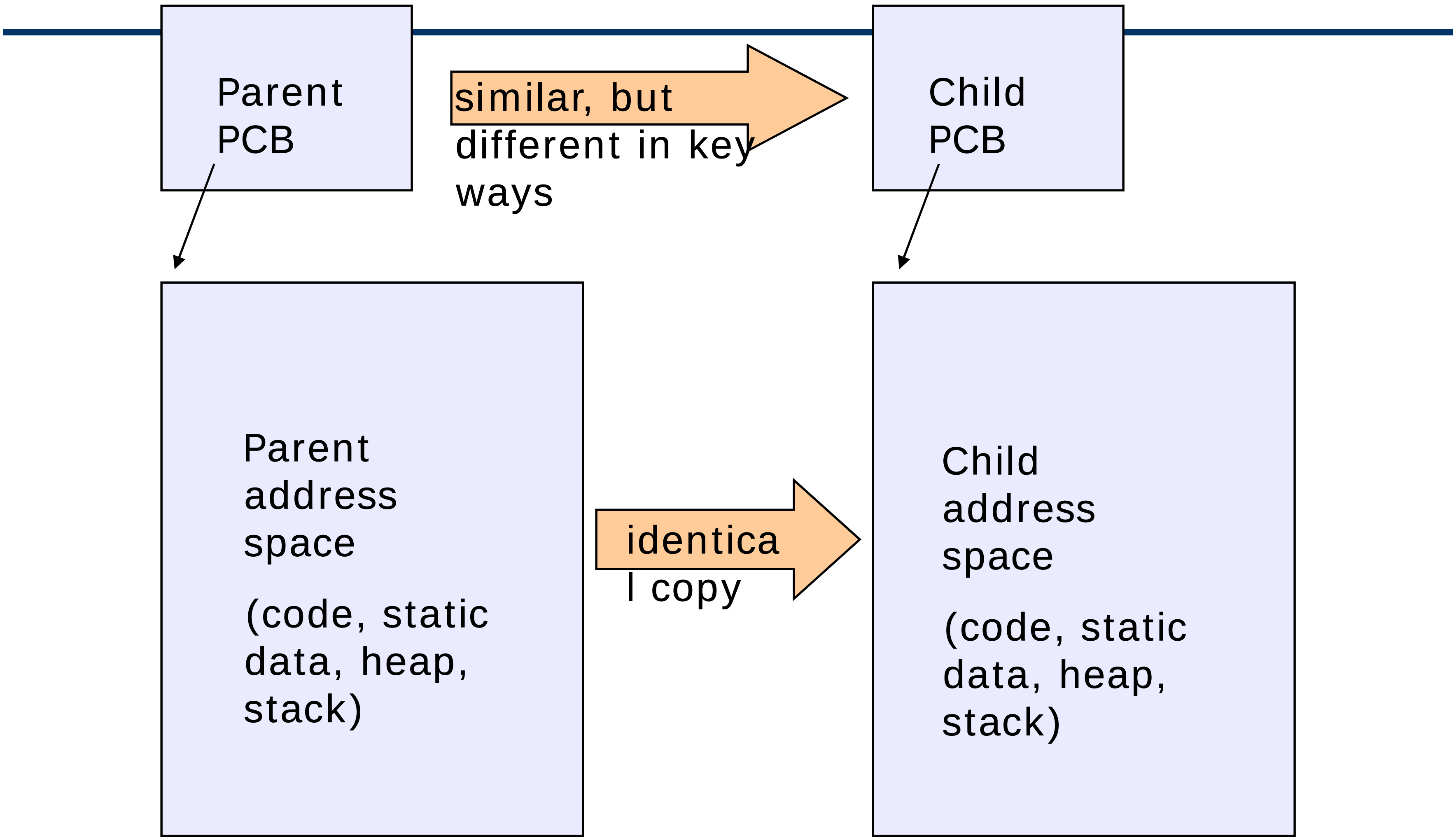
testparent output

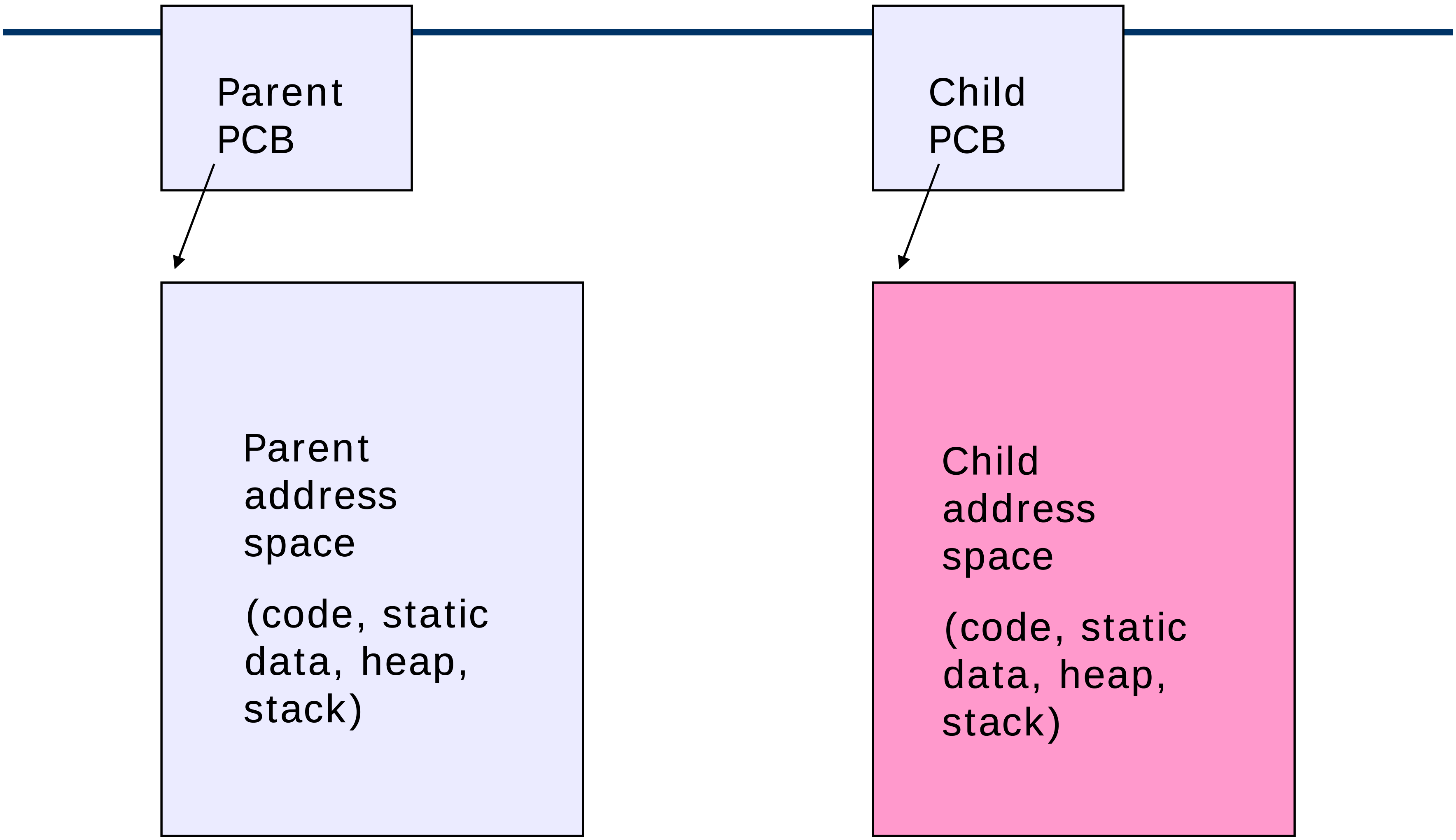
```
spinlock% gcc -o testparent testparent.c
spinlock% ./testparent
My child is 486
Child of testparent is 0
spinlock% ./testparent
Child of testparent is 0
My child is 571
```

exec() vs. fork()

- Q: So how do we start a new program, instead of just forking the old program?
- A: First fork, then exec
  - **`int exec(char * prog, char * argv[])`**
- **`exec()`**
  - stops the current process
  - loads program ‘prog’ into the address space
    - i.e., over-writes the existing process image
  - initializes hardware context, args for new program
  - places PCB onto ready queue
  - note: *<u>does not create a new process!</u>*

- So, to run a new program:
  - fork()
  - Child process does an exec()
  - Parent either waits for the child to complete, or not


Parent
PCB
similar, but
different in key
ways
Child
PCB
Parent
address
space
(code, static
data, heap,
stack)
identica
l copy
Child
address
space
(code, static
data, heap,
stack)

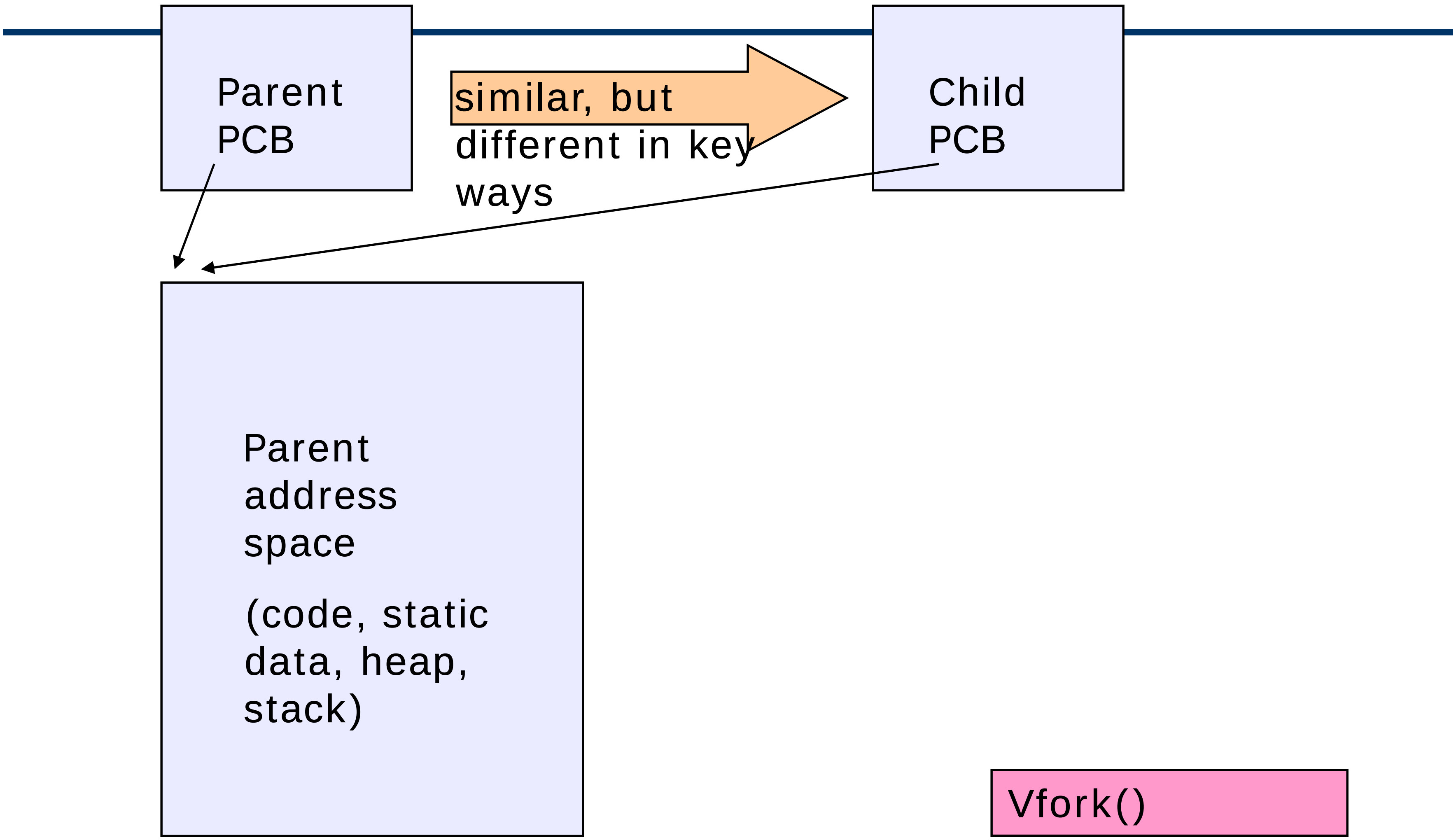

Parent
PCB
Child
PCB
Parent
address
space
(code, static
data, heap,
stack)
Child
address
space
(code, static
data, heap,
stack)

Making process creation faster

- The semantics of fork() say the child's address space is a copy of the parent's
- Implementing fork() that way is slow
  - Have to allocate physical memory for the new address space
  - Have to set up child's page tables to map new address space
  - Have to copy parent's address space contents into child's address space (which you will immediately blow away with an exec())

Method 1: vfork()

- vfork() is the older of the two approaches we'll talk about
- "Change the problem definition into something we can implement efficiently"
- Instead of "child's address space is a copy of the parent's," the semantics are "child's address space *is* the parent's"
  - With a "promise" that the child won't modify the address space before doing an exec()
    - Unenforced! You use vfork() at your own peril
  - When exec() is called, a new address space is created, new page tables set up for it, and it's loaded with the new executable
  - Saves wasted effort of duplicating parent's address space, just to blow it away


Parent
PCB
similar, but
different in key
ways
Child
PCB
Parent
address
space
(code, static
data, heap,
stack)
Vfork()

Method 2: copy-on-write

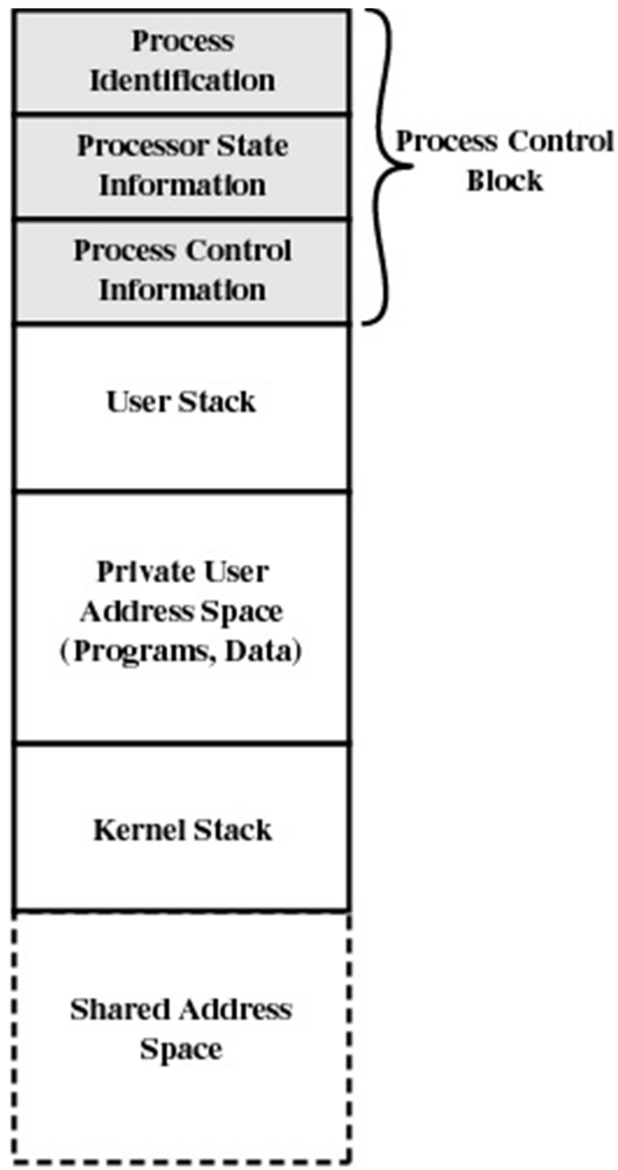
- Retains the original semantics, but copies “only what is necessary” rather than the entire address space
- On fork():
  - Create a new address space
  - Initialize page tables with same mappings as the parent’s (i.e., they both point to the same physical memory)
    - No copying of address space contents have occurred at this point
  - Set both parent and child page tables to make all pages read-only
  - If either parent or child writes to memory, an exception occurs
  - When exception occurs, OS copies the page, adjusts page tables, etc.

When to Switch a Process

- Clock interrupt
  - process has executed for the maximum allowable time slice
- I/O interrupt
- Memory fault
  - memory address is in virtual memory so it must be brought into main memory
- Trap
  - error occurred
  - may cause process to be moved to Exit state
- Supervisor call
  - such as file open

Change of Process State

- Save context of processor including program counter and other registers
- Update the process control block of the process that is currently running
- Move process control block to appropriate queue - ready, blocked
- Select another process for execution
- Update the process control block of the process selected
- Update memory-management data structures
- Restore context of the selected process


Process Identification
Processor State Information
Process Control Information
Process Control Block
User Stack
Private User Address Space (Programs, Data)
Kernel Stack
Shared Address Space

Inter-process communication via signals

- Notification of events to process
- Synchronous: results of program actions
  - **SIGFPE** (floating point exception
  - **SIGSEGV** (segmentation violation)
- Asynchronous

- Processes can register event handlers
  - Feels a lot like event handlers in Java, which ..
  - Feel sort of like catch blocks in Java programs
- When the event occurs, process jumps to event handler routine
- Used to catch exceptions
- Also used for inter-process (process-to-process) communication
  - A process can trigger an event in another process using signal

Signals

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no read
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Example use

- You're implementing Apache, a web server
- Apache reads a configuration file when it is launched
  - Controls things like what the root directory of the web files is, what permissions there are on pieces of it, etc.
- Suppose you want to change the configuration while Apache is running
  - If you restart the currently running Apache, you drop some unknown number of user connections
- Solution: send the running Apache process a signal
  - It has registered an signal handler that gracefully re-reads the configuration file

Signal Handling in multi-threaded applications

- Key issue:
  - Which thread receives a signal? Do all threads receive it?
  - How to control?
- Synchronous: deliver to thread that generates signal
  - Set up a handler for signal in each thread
- Asynchronous:
  - Currently executing thread or
  - Thread that did not mask signal
  - Another approach:
    - o Mask all signals in all threads
    - o Create a separate thread for handling signals