Threads

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

Threads

- Effectiveness of parallel computing depends on the *performance* of the primitives used to express and control parallelism
- Separate *notion of execution* from Process abstraction
- Useful for *expressing intrinsic concurrency of a program* regardless of resulting performance
- Discuss three examples of threading:
  - User threads,
  - Kernel threads and
  - Lightweight processes

Concurrency/Parallelism

- Imagine a web server, which might like to handle multiple requests concurrently
  - While waiting for the credit card server to approve a purchase for one client, it could be retrieving the data requested by another client from disk, and assembling the response for a third client from cached information
- Imagine a web client (browser), which might like to initiate multiple requests concurrently
  - The CSE home page has dozens of “src= …” html commands, each of which is going to involve a lot of sitting around! Wouldn’t it be nice to be able to launch these requests concurrently?
- Imagine a parallel program running on a multiprocessor, which might like to employ “physical concurrency”

What's needed?

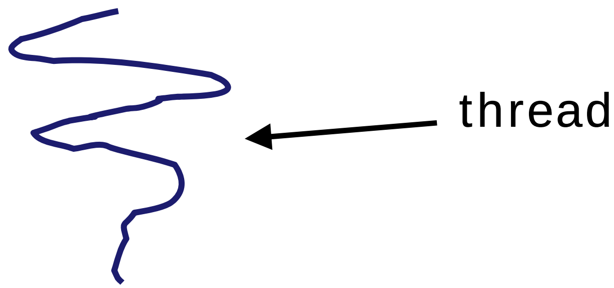
- In each of these examples of concurrency (web server, web client, parallel program):
  - Everybody wants to run the same code
  - Everybody wants to access the same data
  - Everybody has the same privileges
  - Everybody uses the same resources (open files, network connections, etc.)
- But you'd like to have multiple hardware execution states:
  - an execution stack and stack pointer (SP)
    - traces state of procedure calls made
  - the program counter (PC), indicating the next instruction
  - a set of general-purpose processor registers and their values

How could we achieve this?

- Given the process abstraction as we know it:
  - fork several processes
  - cause each to *map* to the same physical memory to share data
    - see the **`shmget()`** system call for one way to do this (kind of)
- This is like making a pig fly – it's really inefficient
  - space: PCB, page tables, etc.
  - time: creating OS structures, fork/copy address space, etc.
- Some equally bad alternatives for some of the examples:
  - Entirely separate web servers
  - Manually programmed asynchronous programming (non-blocking I/O) in the web client (browser)

Can we do better?

- Key idea:
  - separate the concept of a process (address space, OS resources)
  - … from that of a minimal "thread of control" (execution state: stack, stack pointer, program counter, registers)
- This execution state is usually called a thread, or sometimes, a lightweight process

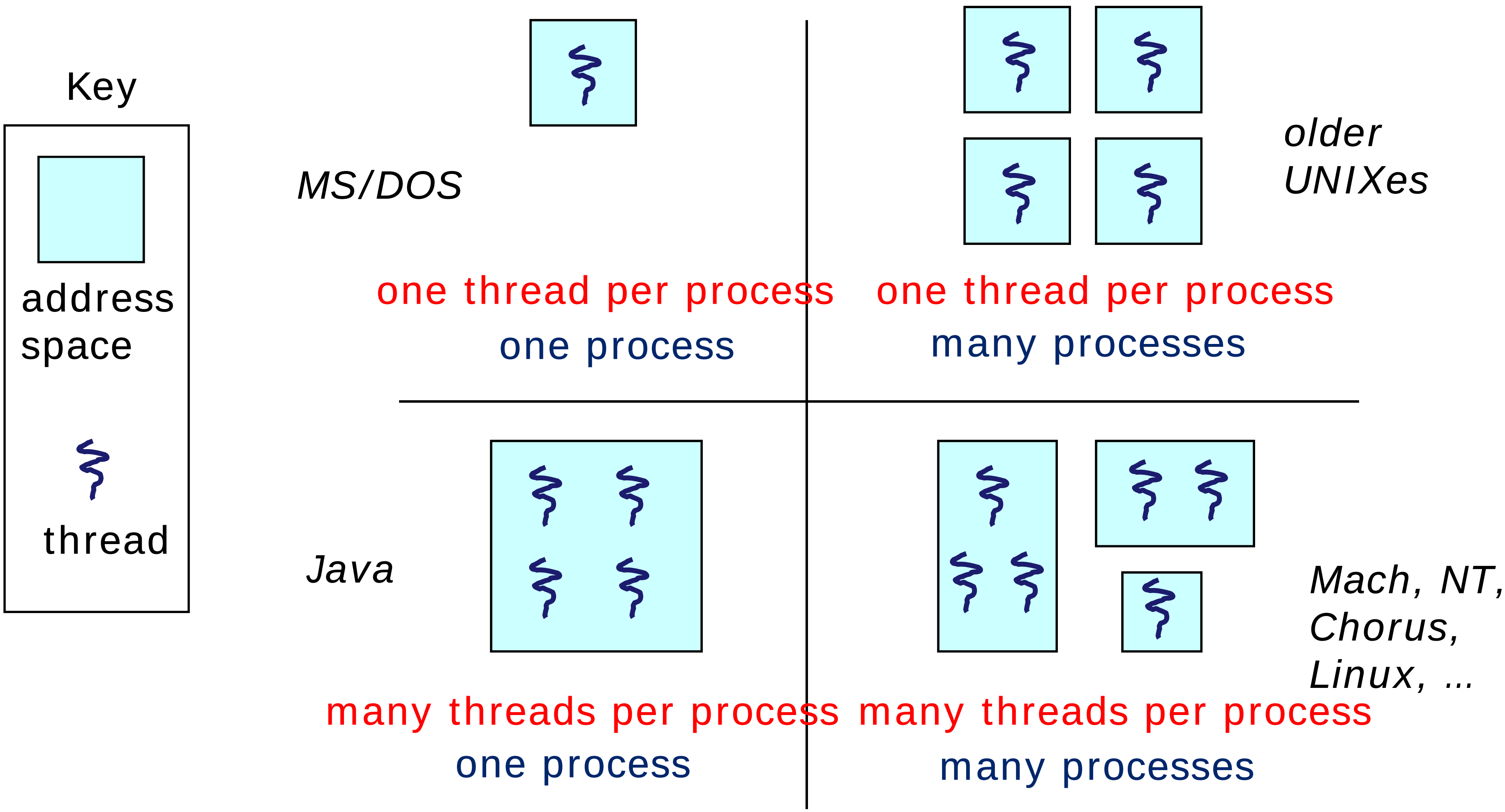


Threads and processes

- Most modern OS's (Mach (Mac OS), Chorus, Windows, UNIX) therefore support two entities:
  - the process, which defines the address space and general process attributes (such as open files, etc.)
  - the thread, which defines a sequential execution stream within a process
- A thread is bound to a single process / address space
  - address spaces, however, can have multiple threads executing within them
  - sharing data between threads is cheap: all see the same address space
  - creating threads is cheap too!
- Threads become the unit of scheduling
  - processes / address spaces are just containers in which threads execute

- Threads are <u>concurrent executions sharing an address space</u> (and some OS resources)
- Address spaces provide isolation
  - If you can't name it, you can't read or write it
- Hence, communicating between processes is expensive
  - Must go through the OS to move data from one address space to another
- Because threads are in the same address space, communication is simple/cheap
  - Just update a shared variable!

The design space



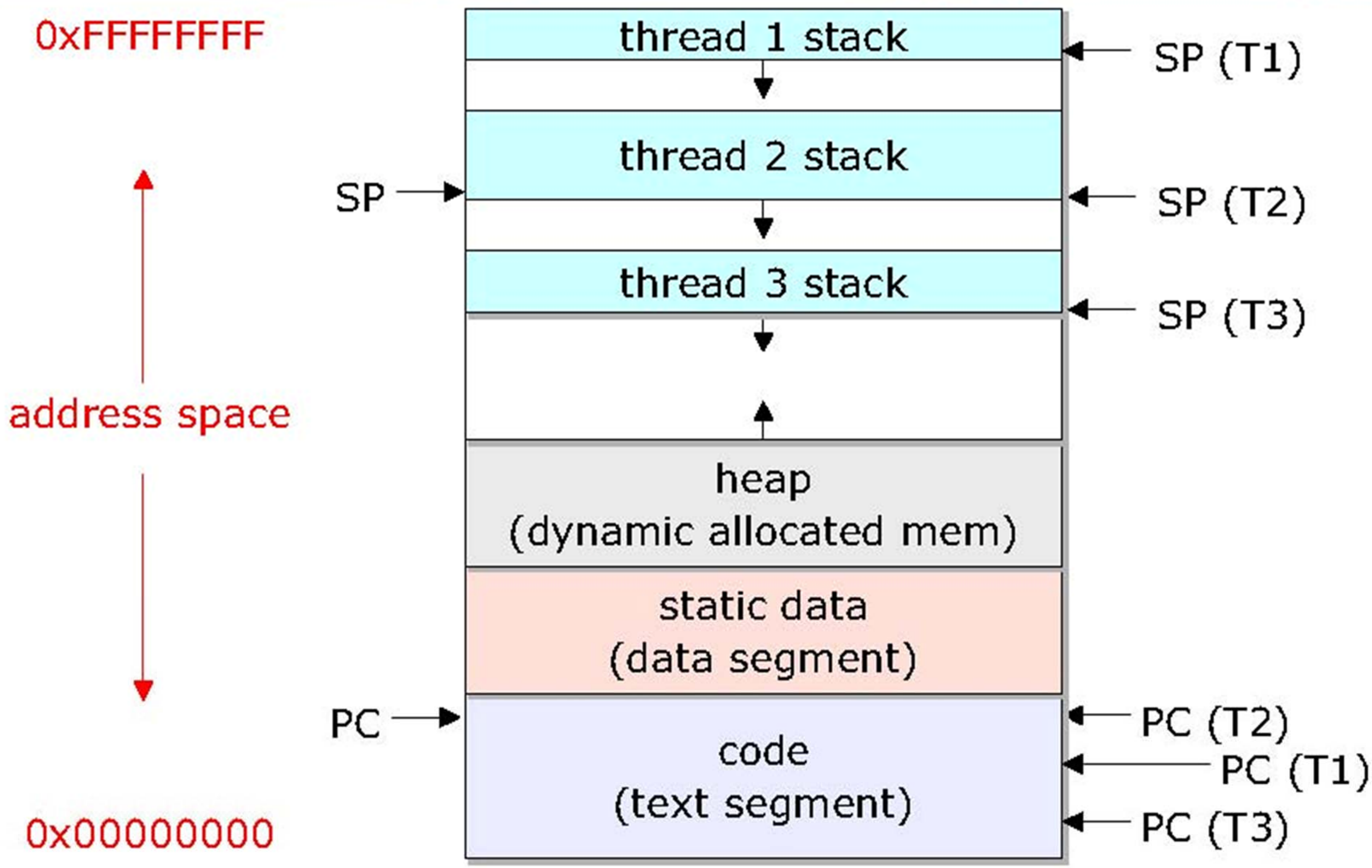
Processes vs. Threads

Processes	Threads
Poor communication	Tight communication
Heavy-weight	Light-weight
Poor performance	Fast performance
Protection	No protection
No Blocking	Blocking

Threads- cont'd.

- Thread : Dynamic object representing an execution path and computational state.
  - One or more threads per process, each having:
    - o Execution state (running, ready, etc.)
    - o Saved thread context when not running
    - o Execution stack
    - o Per-thread static storage for local variables
    - o Shared access to process resources
      - all threads of a process share a common address space.

Address space of a multi-threaded program



Process/thread separation

- Concurrency (multithreading) is useful for:
  - handling concurrent events (e.g., web servers and clients)
  - building parallel programs (e.g., matrix multiply, ray tracing)
  - improving program structure (the Java argument)
- Multithreading is useful even on a uniprocessor
  - even though only one thread can run at a time
- Supporting multithreading – that is, separating the concept of a process (address space, files, etc.) from that of a minimal thread of control (execution state), is a big win
  - creating concurrency does not require creating new processes
  - “faster / better / cheaper”

Terminology

- Just a note that there's the potential for some confusion ...
  - Old world: "process" == "address space + OS resources + single thread"
  - New world: "process" typically refers to an address space + system resources + all of its threads ...
    - When we mean the "address space" we need to be explicit

    "thread" refers to a single thread of control within a process / address space

- A bit like "kernel" and "operating system" ...
  - Old world: "kernel" == "operating system" and runs in "kernel mode"
  - New world: "kernel" typically refers to the microkernel; lots of the operating system runs in user mode

“Where do threads come from?”

- Natural answer: the OS is responsible for creating/managing threads
  - For example, the kernel call to create a new thread would
    - allocate an execution stack within the process address space
    - create and initialize a Thread Control Block
      - stack pointer, program counter, register values
    - stick it on the ready queue
  - We call these kernel threads
  - There is a “thread name space”
    - Thread id’s (TID’s)
    - TID’s are integers (surprise!)

Kernel threads

- OS now manages threads *and* processes / address spaces
  - all thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - o if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
    - o possible to overlap I/O and computation inside a process
- Kernel threads are cheaper than processes
  - less state to allocate and initialize
- But, they're still pretty expensive for fine-grained use
  - orders of magnitude more expensive than a procedure call
  - thread operations are all system calls
    - o context switch
    - o argument checks
  - must maintain kernel state for each thread

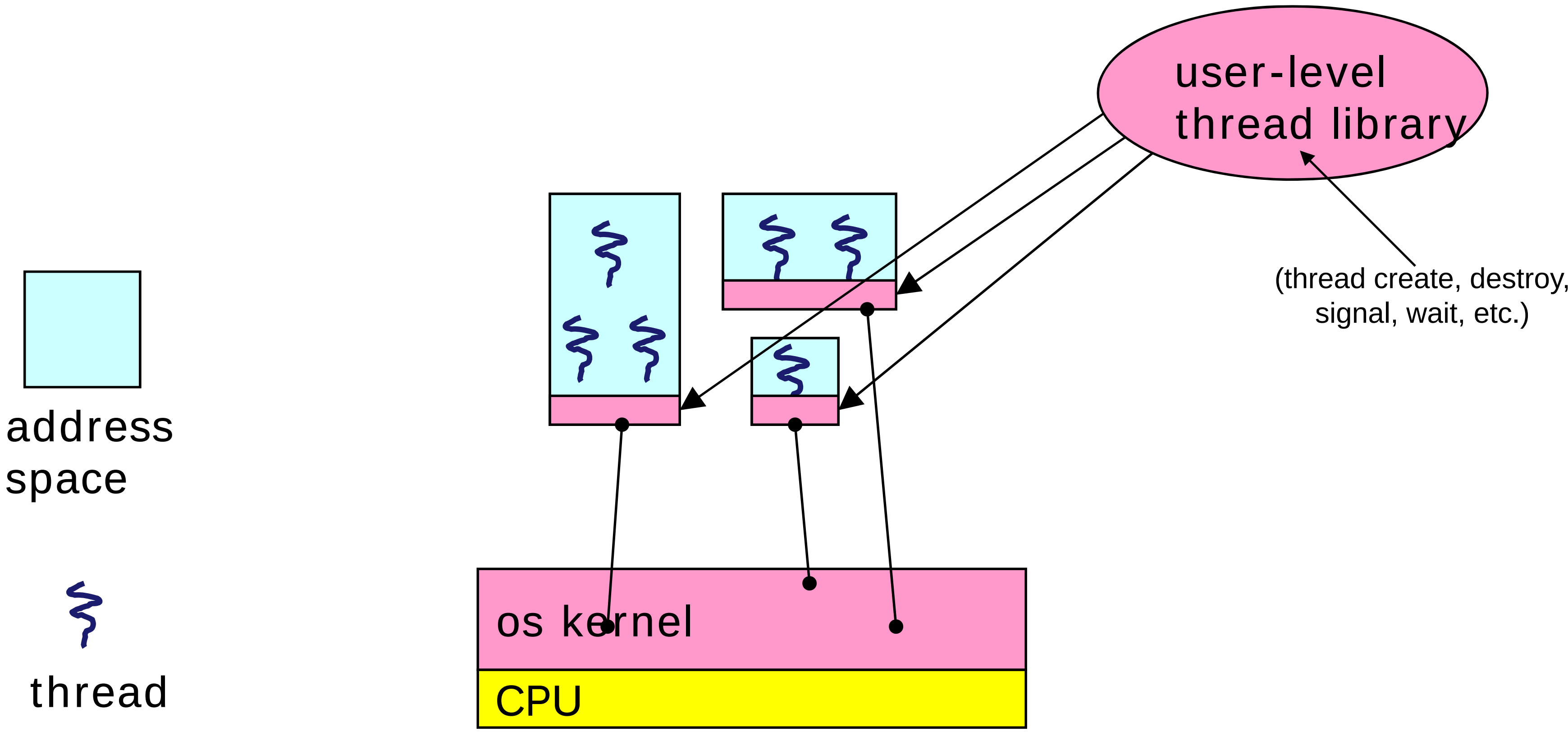
Kernel level threads - drawbacks

- More expensive than user-level threads
  - Overhead of switching in and out of supervisory mode
  - Overhead of features not used by many applications
    - e.g. application may not need to save all floating point registers
- Large kernel size
- Semantic inflexibility:
  - Different scheduling policies
  - Different relationship among threads (cooperative vs. competitive)
- Hard to maintain

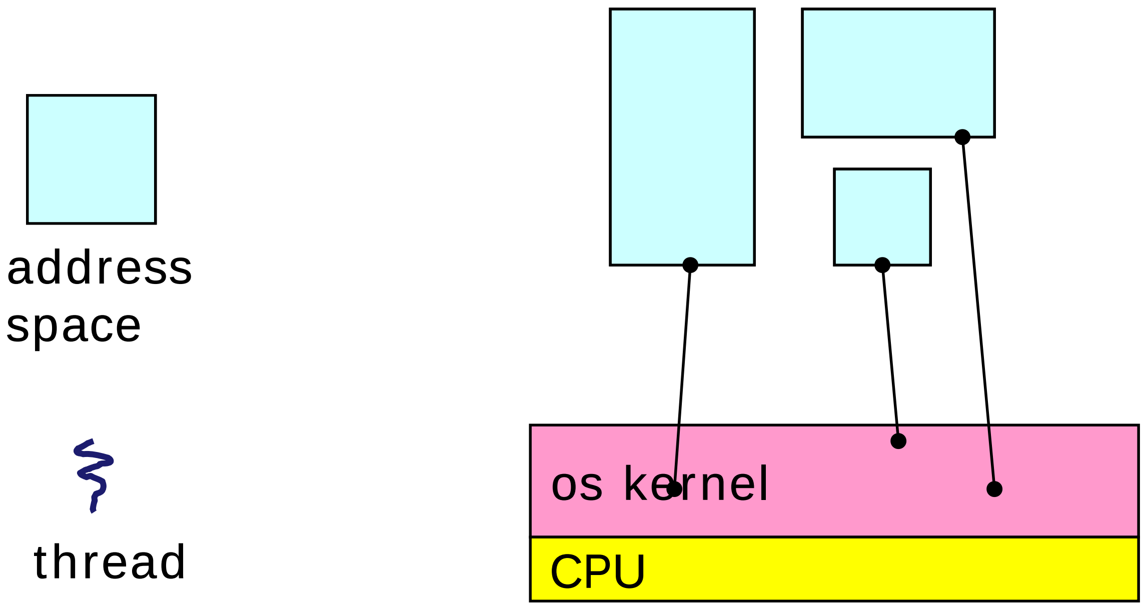
"Where do threads come from? – cont'd"

- There is an alternative to kernel threads
- Threads can also be managed at the user level (that is, entirely from within the process)
  - a library linked into the program manages the threads
    - o because threads share the same address space, the thread manager doesn't need to manipulate address spaces (which only the kernel can do)
    - o threads differ (roughly) only in hardware contexts (PC, SP, registers), which can be manipulated by user-level code
    - o the thread package multiplexes user-level threads on top of kernel thread(s)
    - o each kernel thread is treated as a "virtual processor"
  - we call these user-level threads

User-level threads

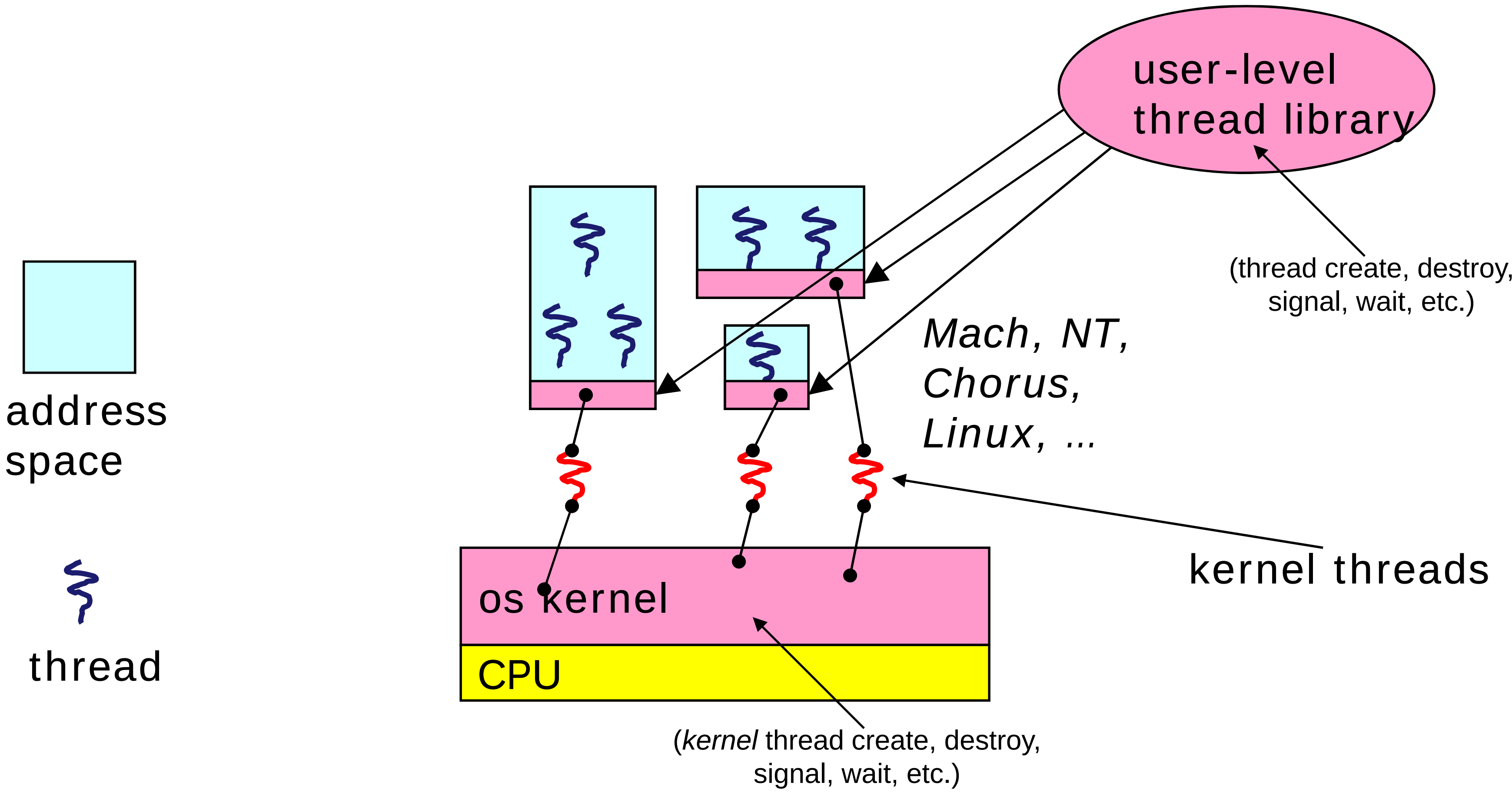


User-level threads: what the kernel sees



4/4/2011
20

User-level threads: the full story



User-level threads

- User-level threads are small and fast
  - managed entirely by user-level library
    - E.g., pthreads (**`libpthreads.a`**)
  - each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
  - creating a thread, switching between threads, and synchronizing threads are done via procedure calls
    - no kernel involvement is necessary!
  - user-level thread operations can be 10-100x faster than kernel threads as a result

Performance example

- On a 700MHz Pentium running Linux 2.2.16 (only the relative numbers matter; ignore the ancient CPU!):

  - Processes
    - o **fork/exit**: 251 μs
  - Kernel threads
    - o **pthread_create()/pthread_join()**: 94 μs **(2.5x faster)**
  - User-level threads
    - o **pthread_create()/pthread_join**: 4.5 μs **(another 20x faster)**

Thread States

- Primary states:
  - Running, Ready and Blocked.
- Operations to change state:
  - Spawn: new thread provided register context and stack pointer.
  - Block: event wait, save user registers, PC and stack pointer
  - Unblock: moved to ready state
  - Finish: deallocate register context and stacks.

User-level thread implementation

- The OS schedules the kernel thread
- The kernel thread executes user code, including the thread support library and its associated thread scheduler
- The thread scheduler determines when a user-level thread runs
  - it uses queues to keep track of what threads are doing: run, ready, wait
    - just like the OS and processes
    - but, implemented at user-level as a library

Thread context switch

- Save context of currently running thread
  - Push all machine state on its stack
- Restore context of next thread
  - Pop machine state from next thread's stack
- Architectures may support techniques for saving states efficiently
- Make next thread current thread
- Return called as new thread
  - Assembly as works at the level of procedure calling
- This is all done by assembly language
  - it works at the level of the procedure calling convention
    - thus, it cannot be implemented using procedure calls

Thread interface

- This is taken from the POSIX pthreads API:
  - `rcode = pthread_create(&t, attributes, start_procedure)`
    - o creates a new thread of control
    - o new thread begins executing at start_procedure
  - `pthread_cond_wait(condition_variable, mutex)`
    - o the calling thread blocks, sometimes called thread_block()
  - `pthread_signal(condition_variable)`
    - o starts a thread waiting on the condition variable
  - `pthread_exit()`
    - o terminates the calling thread
  - `pthread_wait(t)`
    - o waits for the named thread to terminate

User Level Threads: Benefits

- No modifications required to kernel
  - Development and maintenance easier
- Flexible
  - User defined schecduling, communication and process management
- Low cost
  - No kernel cost of thread managment

User Level Threads: Drawbacks

- May block all thread during blocking system calls
  - Kernel may need to provide non-blocking system calls
  - Or implement through auxiliary processes
- Cannot exploit physical parallelism
- Lack of coordination between kernel-level scheduling and thread-level synchronization
  - Kernel pre-empts a thread that other threads depend on

Multithreading Models

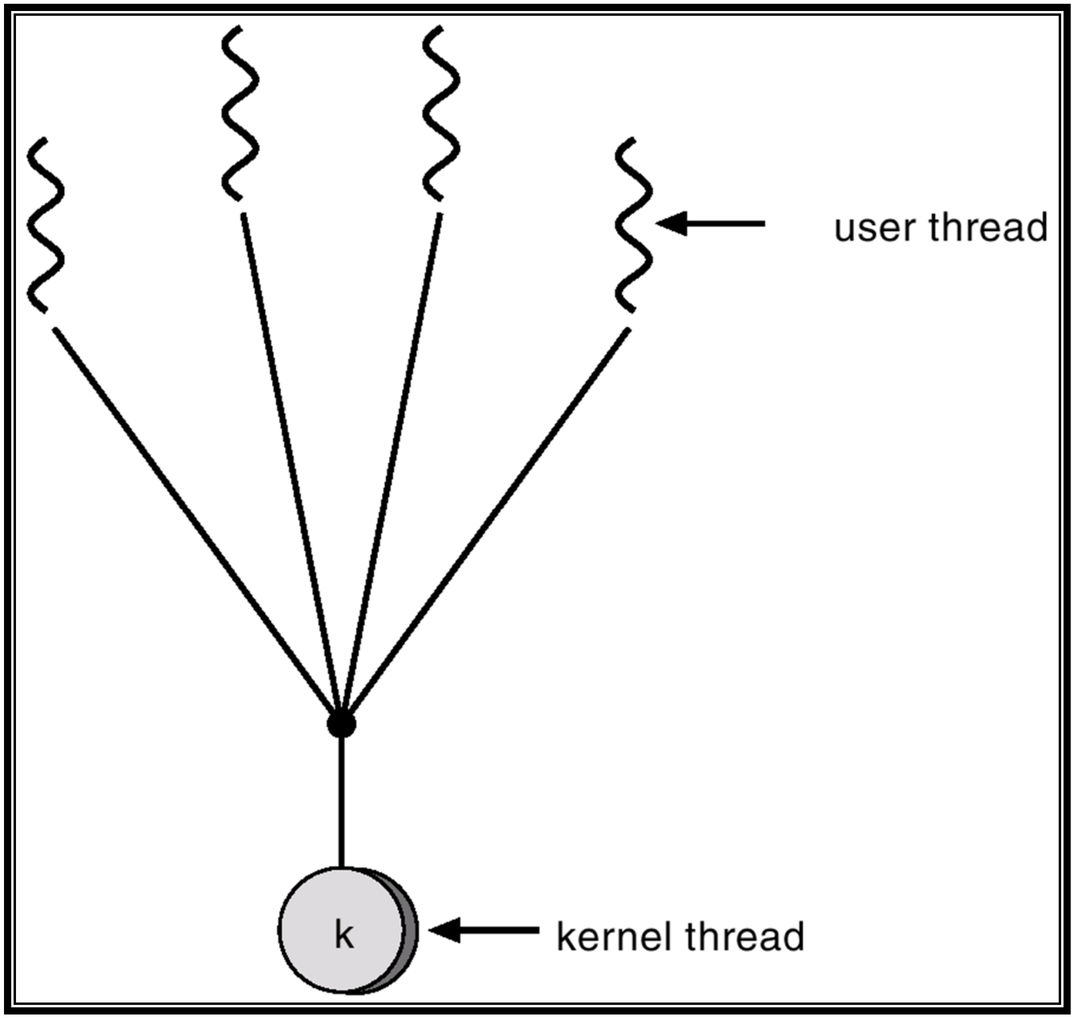
- Many-to-One

- One-to-One

- Many-to-Many

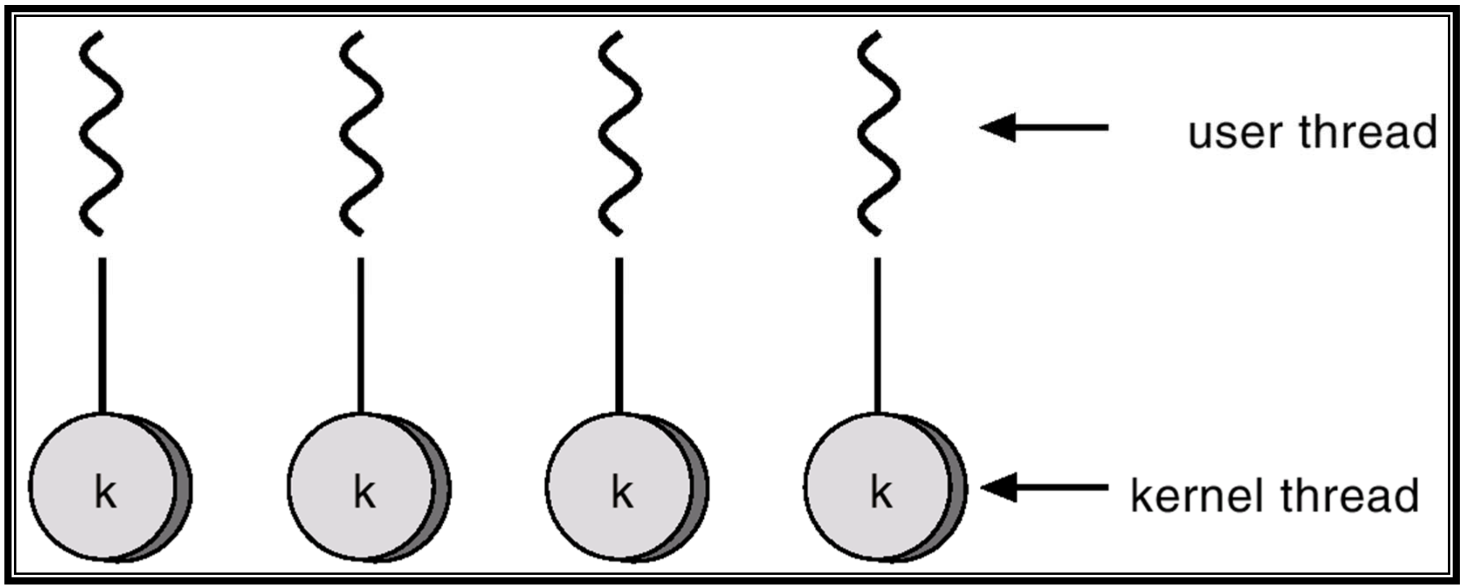
Many-to-One

- Many user-level threads mapped to single kernel thread.
- Used on systems that do not support kernel threads.



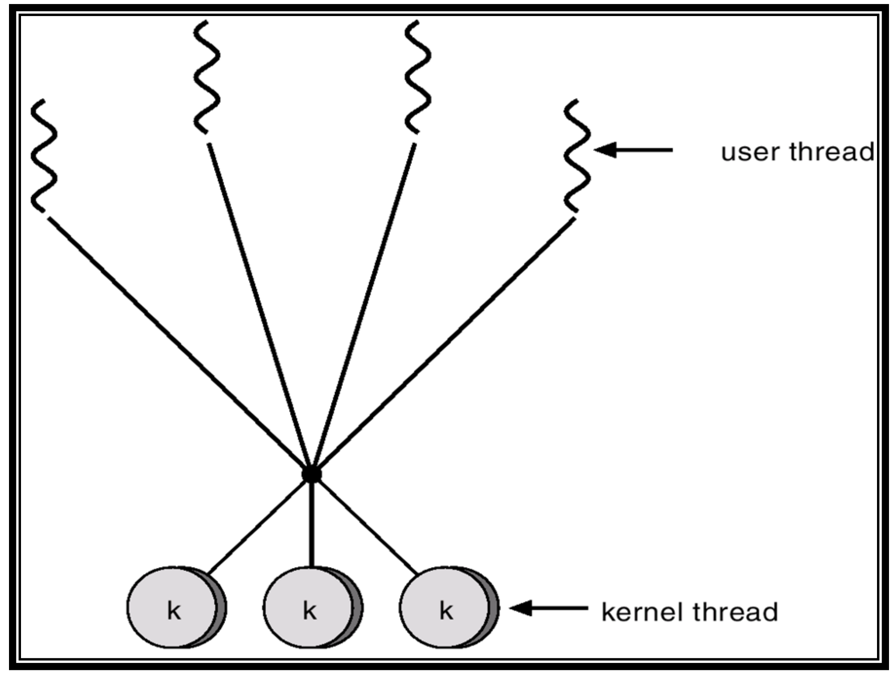
One-to-One

- Each user-level thread maps to kernel thread.
- Examples
  - Windows 95/98/NT/2000
  - OS/2



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
  - Solaris 2
  - Windows NT/2000 with the *ThreadFiber* package



Thread scheduling – cont'd.

- Non-preemptive scheduling: force everyone to cooperate
  - Threads give up CPU by calling **yield**
  - Yield calls into scheduler, which context switches to another ready thread

```
Thread ping() {
  while (1) {
    printf("ping \n");
    yield();
  }
}
```

```
Thread pong() {
  while (1) {
    printf("pong \n");
    yield();
  }
}
```

- Pre-emptive Scheduling:
  - Regain control of processor asynchronously
  - Scheduler requests OS to deliver a timer signal
    - Usually delivered as a UNIX signal (software interrupt)
  - At each interrupt, scheduler gains control and context switches as appropriate

Thread scheduling

- Determines when a thread runs
  - Similar to OS and processes
  - Implemented at library level
- Queues:
  - Run queue
  - Ready queue
  - Wait queue
    - Blocked for some reason
- Thread scheduling issues:
  - How to ensure threads share CPU fairly?
  - What if thread tries to do I/O?
  - What if a thread holding lock is pre-empted?

How to keep a user-level thread from hogging the CPU?

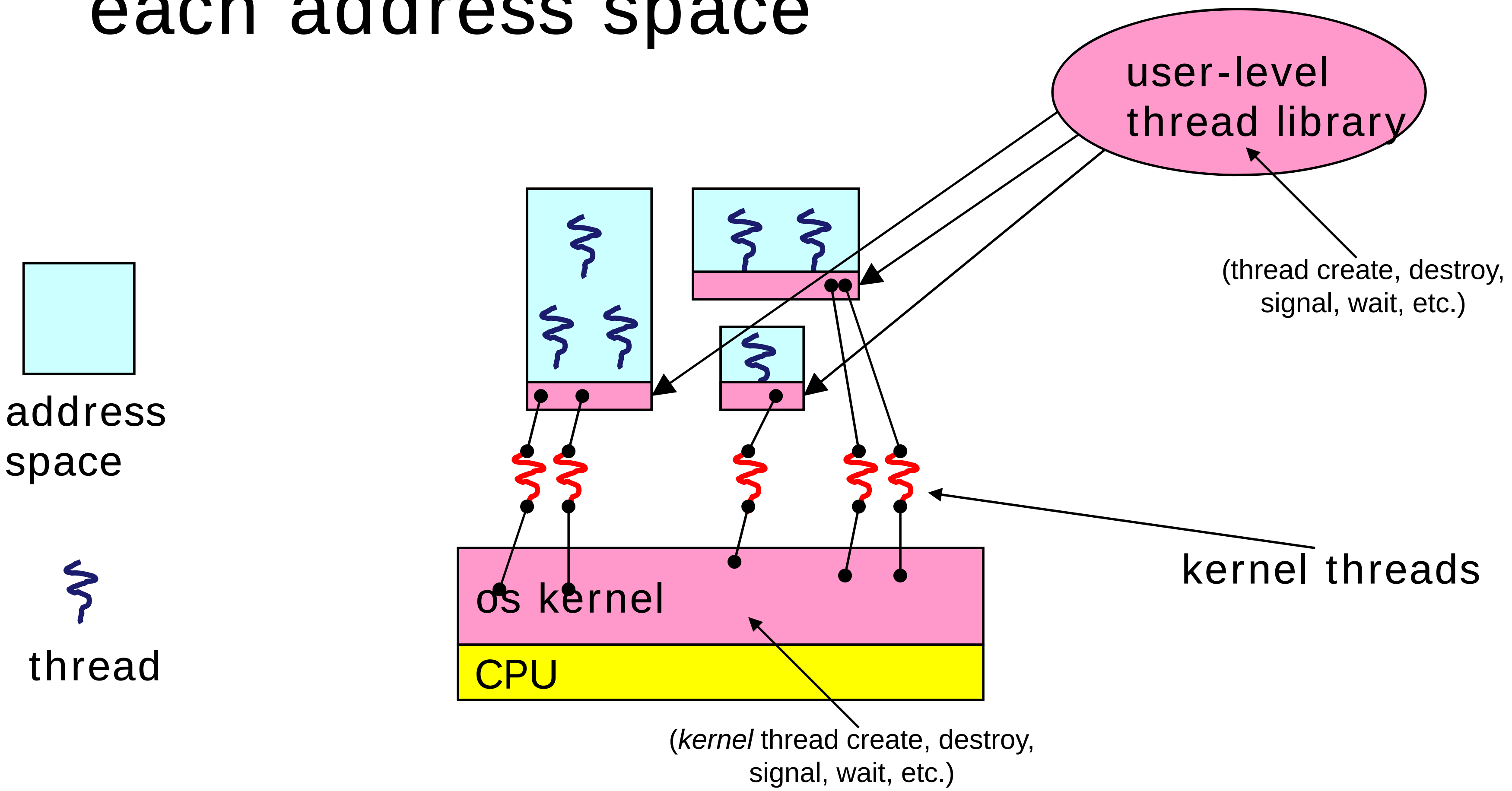
- Strategy 1: force everyone to cooperate
  - a thread willingly gives up the CPU by calling **`yield()`**
  - **`yield()`** calls into the scheduler, which context switches to another ready thread
  - what happens if a thread never calls **`yield()`**?

- Strategy 2: use preemption
  - scheduler requests that a timer interrupt be delivered by the OS periodically
    - usually delivered as a UNIX signal (`man signal`)
    - signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
  - at each timer interrupt, scheduler gains control and context switches as appropriate

What if a thread tries to do I/O?

- The kernel thread "powering" it is lost for the duration of the (synchronous) I/O operation!
  - The kernel thread blocks in the OS, as always
  - It maroons with it the state of the user-level thread
- Could have one kernel thread "powering" each user-level thread
  - "common case" operations (e.g., synchronization) would be quick
- Could have a limited-size "pool" of kernel threads "powering" all the user-level threads in the address space
  - the kernel will be scheduling these threads, obliviously to what's going on at user-level

Multiple kernel threads "powering" each address space



What if the kernel preempts a thread holding a lock?

- Other threads will be unable to enter the critical section and will block (stall)

Addressing these problems

- Effective coordination of kernel decisions and user-level threads requires OS-to-user-level communication
  - OS notifies user-level that it is about to suspend a kernel thread
- This is called “scheduler activations”
  - a research paper from UW with huge effect on practice
  - each process can request one or more kernel threads
    - process is given responsibility for mapping user-level threads onto kernel threads
    - kernel promises to notify user-level before it suspends or destroys a kernel thread
  - *ACM TOCS 10*,1

Summary

- Processes:
  - Representation of a running program
  - States: ready, blocked, swapped, running, terminated...
  - How do these transitions take place? (I/O, timers, interrupts, traps...)
  - How does operating system maintain this state? (PCB)
    - What kind of information stored?
- Threads:
  - Lightweight version of process
  - User level and kernel level threads: how are they different?
  - Mapping of threads on machine resources