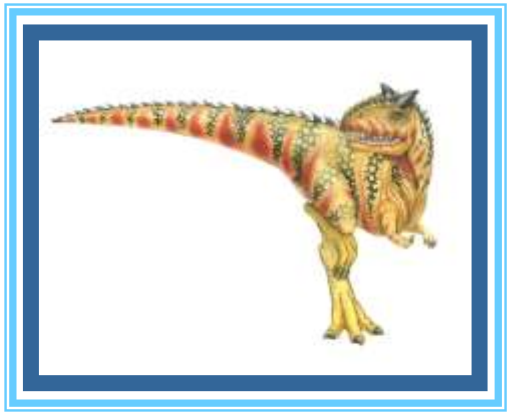
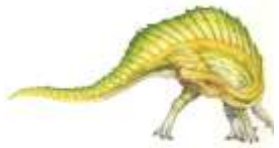
Chapter 4: Threads

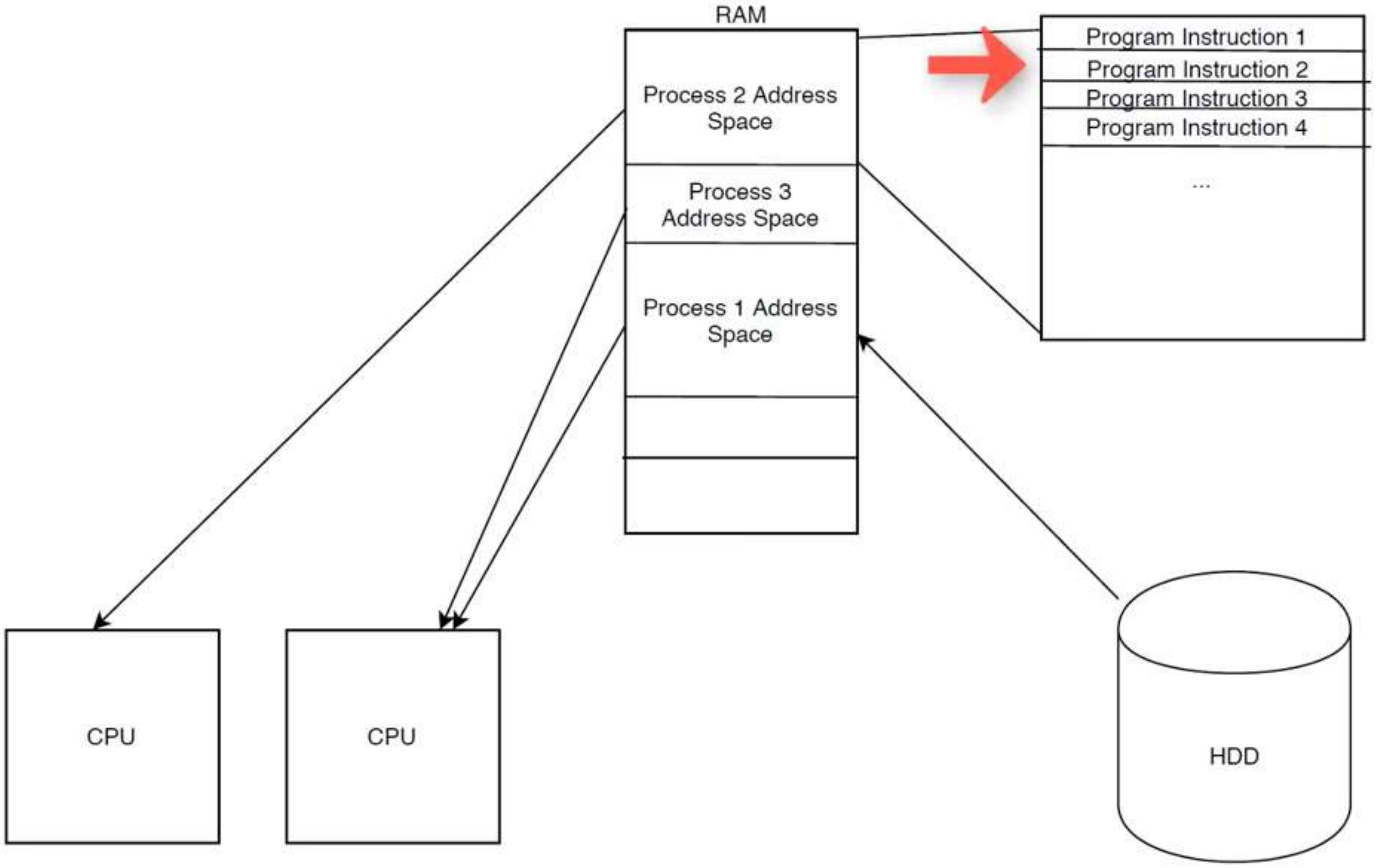


The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control.

Virtually all modern operating system, however, provide features enabling a process to contain multiple threads of control



Process in CPU-Memory Cycle

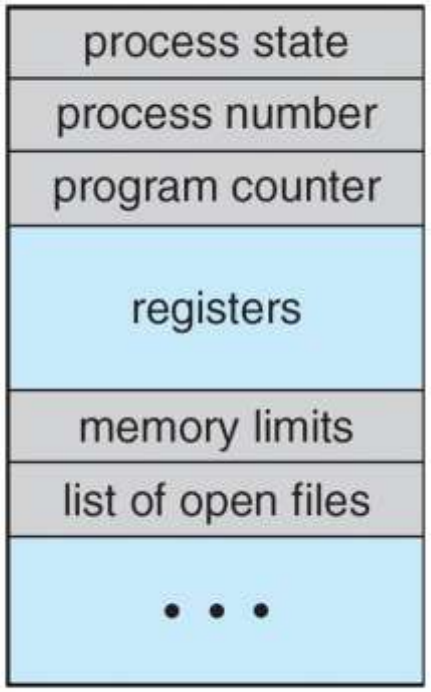


Execution context

Program counter – address of next instruction to be executed

Execution context – the state of a running process

- ➢ the state of the **processor's registers**
- ➢ instruction pointer (**program counter**)



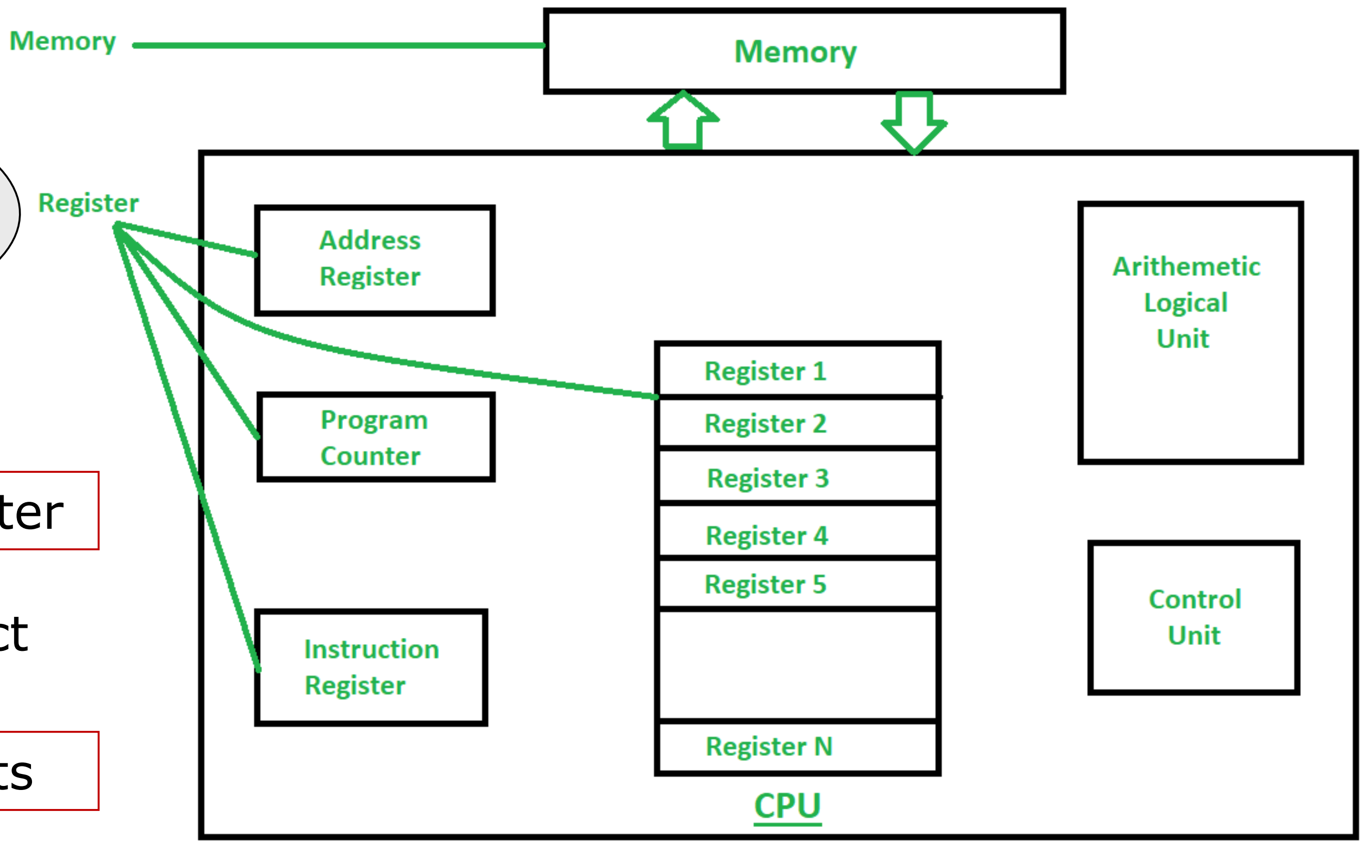
A process was a program in memory along with dynamically-allocated storage (the heap), the stack, and the **execution context**, which comprises the state of the processor's registers and instruction pointer (program counter).

Execution context

Break a process into two components

1. The program and dynamically allocated memory.
2. The **stack**, **instruction pointer**, and **registers**.

Crucial for the execution flow of the program



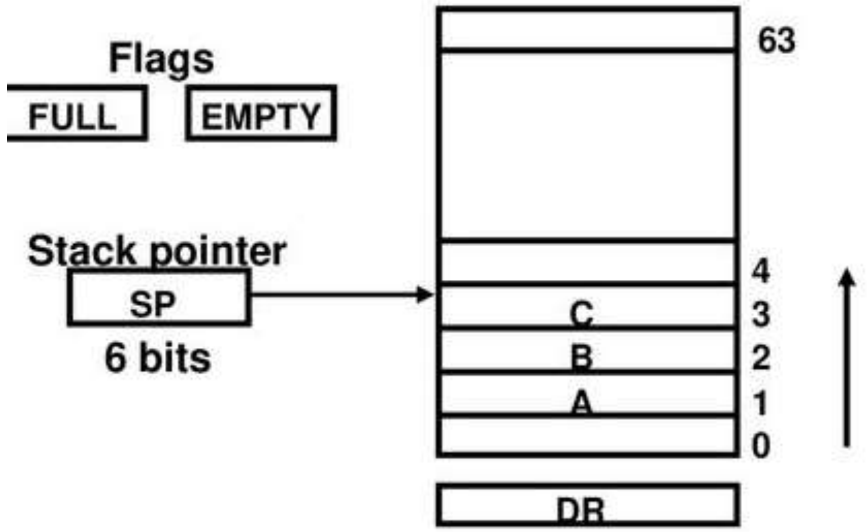
$$\begin{tikzcd}\text{Instruction pointer} \arrow[d, "\text{impact}"] \\ \text{Register contents}\end{tikzcd}$$

Thread of execution

Break a process into two components

1. The program and dynamically allocated memory.
2. The **stack**, **instruction pointer**, and **registers**.

Crucial for the execution flow of the program



$$\begin{tikzcd}
\boxed{\text{Instruction pointer}} \arrow[d, "\text{impact}"] & \\
\boxed{\text{Register contents}} \arrow[r, "\text{impact}"] & \boxed{\text{Stack contents and pointer}}
\end{tikzcd}$$

This stream of instructions is the process' thread of execution.

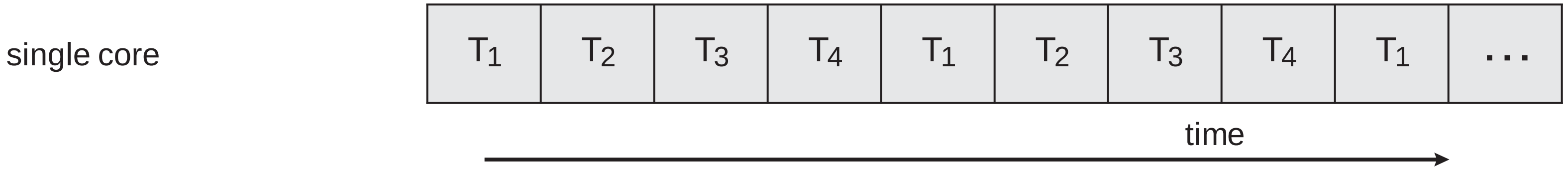
What’s “in” a process?

- A process consists of (at least):
  - An address space, containing
    - the code (instructions) for the running program
    - the data for the running program
  - Thread state, consisting of
    - The program counter (PC), indicating the next instruction
    - The stack pointer register (implying the stack it points to)
    - Other general purpose register values
  - A set of OS resources
    - open files, network connections, sound channels, …
- Decompose …
  - address space
  - thread of control (stack, stack pointer, program counter, registers)
  - OS resources

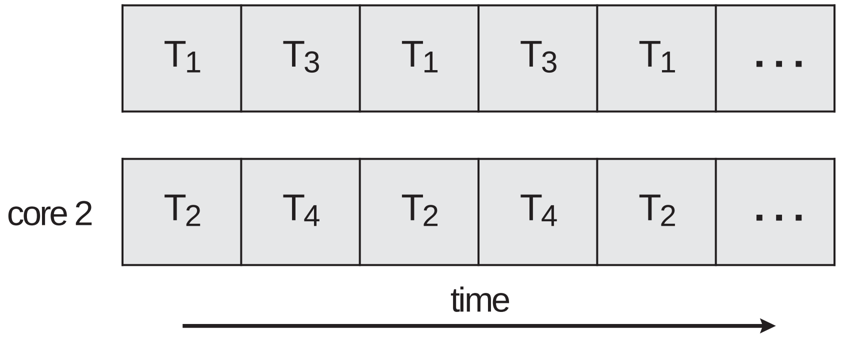
Thread: Concurrency vs. Parallelism

- Threads are about concurrency and parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



Motivation

- Threads are about concurrency and parallelism
- One way to get concurrency and parallelism is to use multiple processes
  - The programs (code) of distinct processes are isolated from each other
- Threads are another way to get concurrency and parallelism
  - Threads "share a process" – same address space, same OS resources
  - Threads have private stack, CPU state – are schedulable

What’s needed?

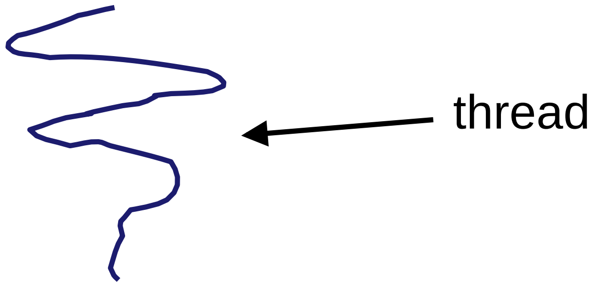
- In many cases
  - Everybody wants to run the same code
  - Everybody wants to access the same data
  - Everybody has the same privileges
  - Everybody uses the same resources (open files, network connections, etc.)
- But you’d like to have multiple hardware execution states:
  - an execution stack 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
- This is really inefficient
  - space: PCB, page tables, etc.
  - time: creating OS structures, fork/copy address space, etc.

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

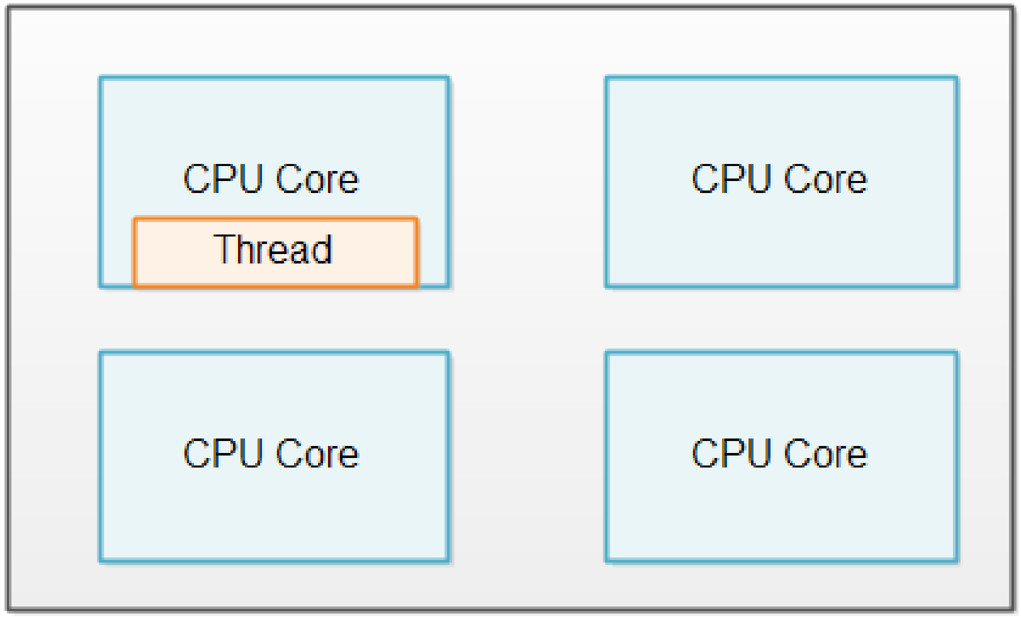


Threads and processes

- 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

Single-threaded Systems

- Conventional programming model & OS structure:
  - – Single threaded
  - – **One process = one thread**



A CPU with 4 cores

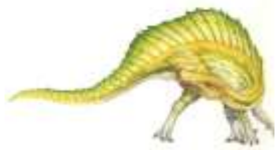
single-threaded systems do not fully utilize modern CPUs

Each core functions as an individual CPU. A single-threaded system can only utilize one of the cores

Thread

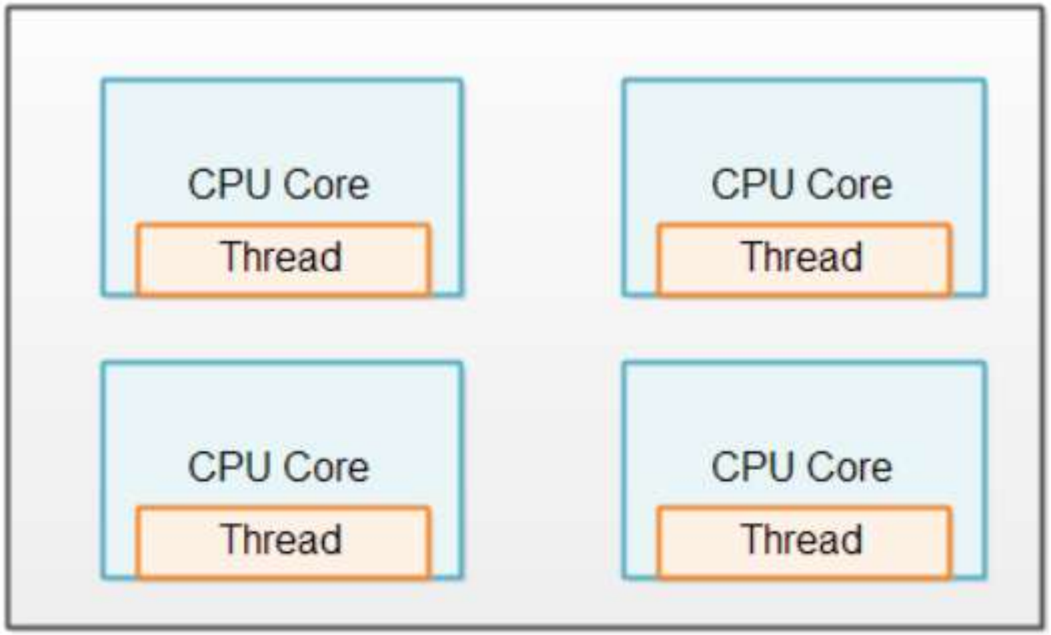
- If we want a process to be able to execute on multiple CPUs at a time, to take advantage of the **multi-core systems**, the process must have **several execution-context** called **threads**.

- A thread is an **active entity** which executes a part of a process



Same-threading: Single-threading Scaled Out

- To **utilize all the cores** in the CPU, a single-threaded system can be **scaled out** to utilize the whole computer



has 1 thread running per CPU

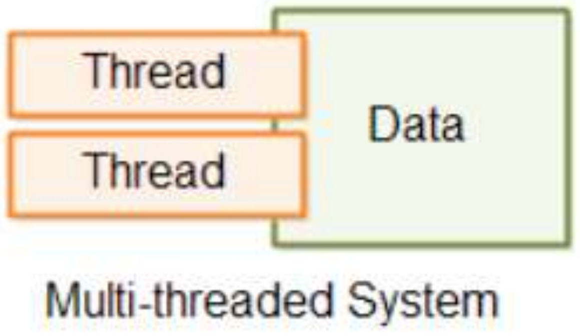
run 4 instances of the **same-threaded system** (4 single-threaded systems)

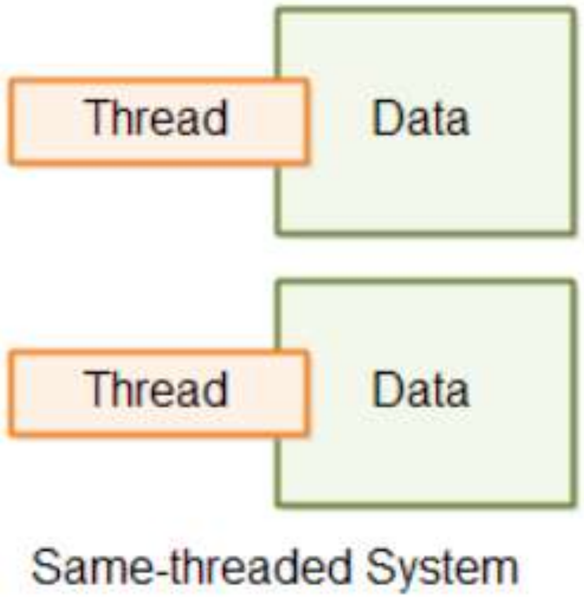
Drawback: the threads in a same-threaded system do **not share data**

Same-threading: Single-threading Scaled Out

- Multiple threads execute simultaneously with each other which results in **the execution of a single whole process**.
- As several threads of a process execute at the same time they are required to maintain **coordination** with each other.
- Coordination between the threads is required in order to **share system resources** like I/O devices, memory, and of course CPU.

Multi-threaded system



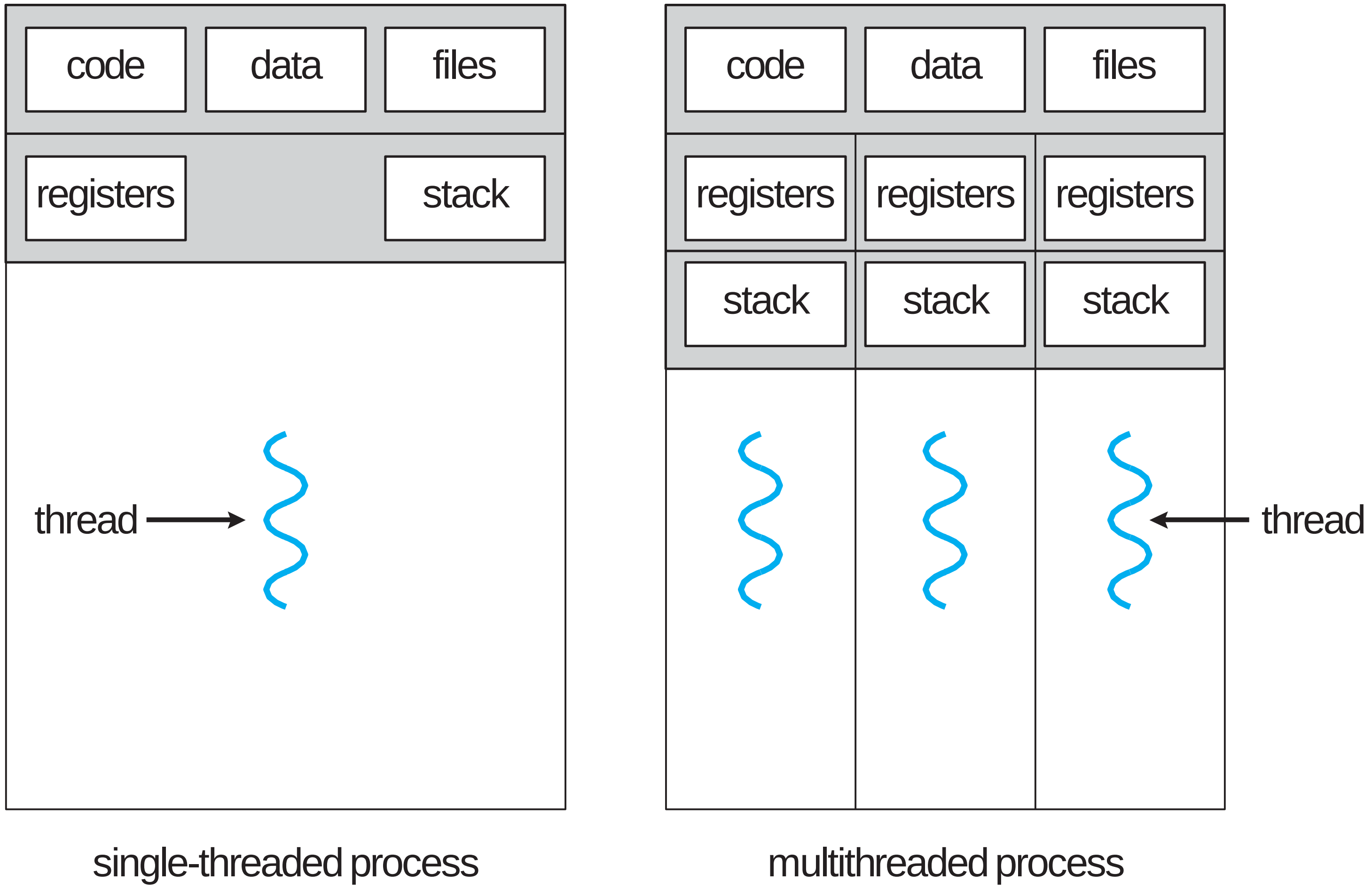


No concurrent data structures

The lack of shared data is what makes each thread behave as it if was a single-threaded system.

However, since a same-threaded system can contain more than a single thread - it is not really a "single-threaded system‟

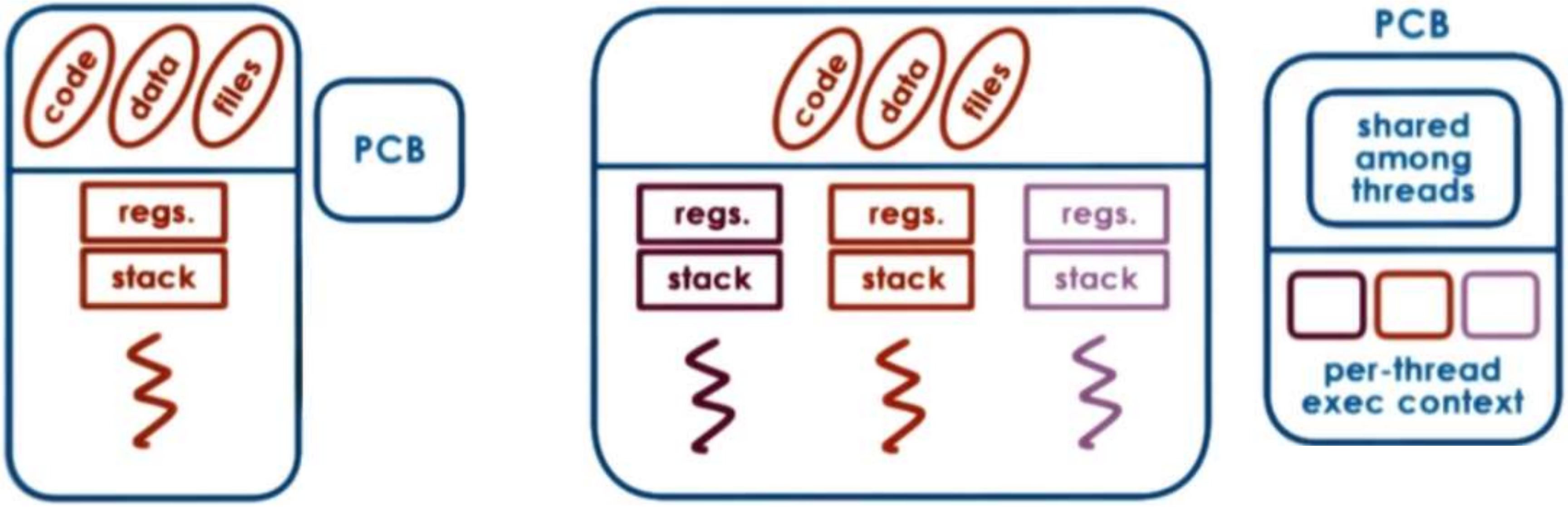
Single and Multithreaded Processes



Communication

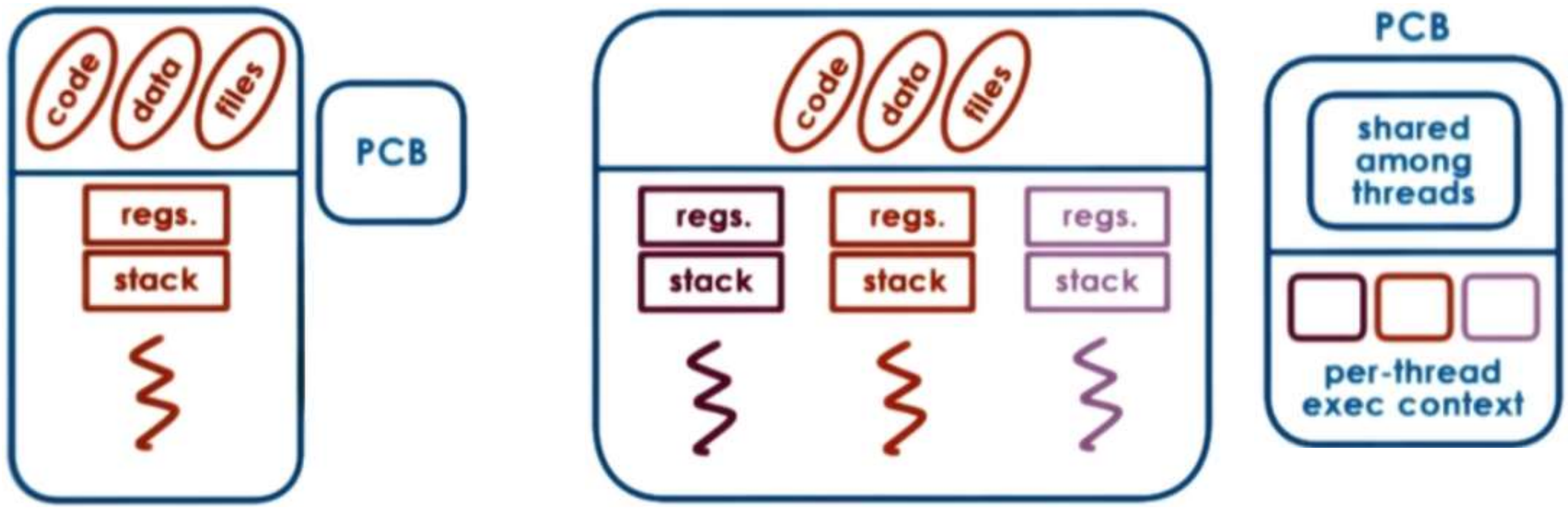
- Threads are concurrent executions sharing an address space (and some OS resources)
- Address spaces provide isolation
  - If you can’t name it, you can’t read or write it
- Hence, communicating between processes is expensive
  - Must go through the OS to move data from one address space to another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

Process v.s. Threads



- The threads of a process are **the part of** the **same virtual address space** (i.e. they share all virtual to physical mapping), they **share** all the **data, code and file.**
- However, they will be **executing** the **different instruction**, they will be accessing the different portion of the address space or different in other ways.

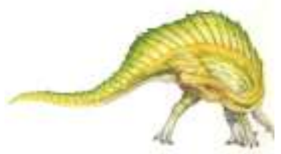
Process v.s. Threads



- A **different thread** has a different **stack**, a different **stack pointer register**, a different **Program Counter**, and other **registers**.
- The Process Control Block (PCB) of a multi-threaded process is more **complex** than a single threaded process. It contains **all the information** which is shared among the threads and **separate execution context** of all threads.

Thread

- A **Thread** also called **lightweight process**
- It compromises a thread ID, a program counter, a register set, and a stack
- A thread is an entity within a process that can be scheduled for execution.



Multi-threaded model

A thread is a **subset** of a process:

– A process contains **one or more** threads

Share memory and open files

–BUT: separate program counter, registers, and stack

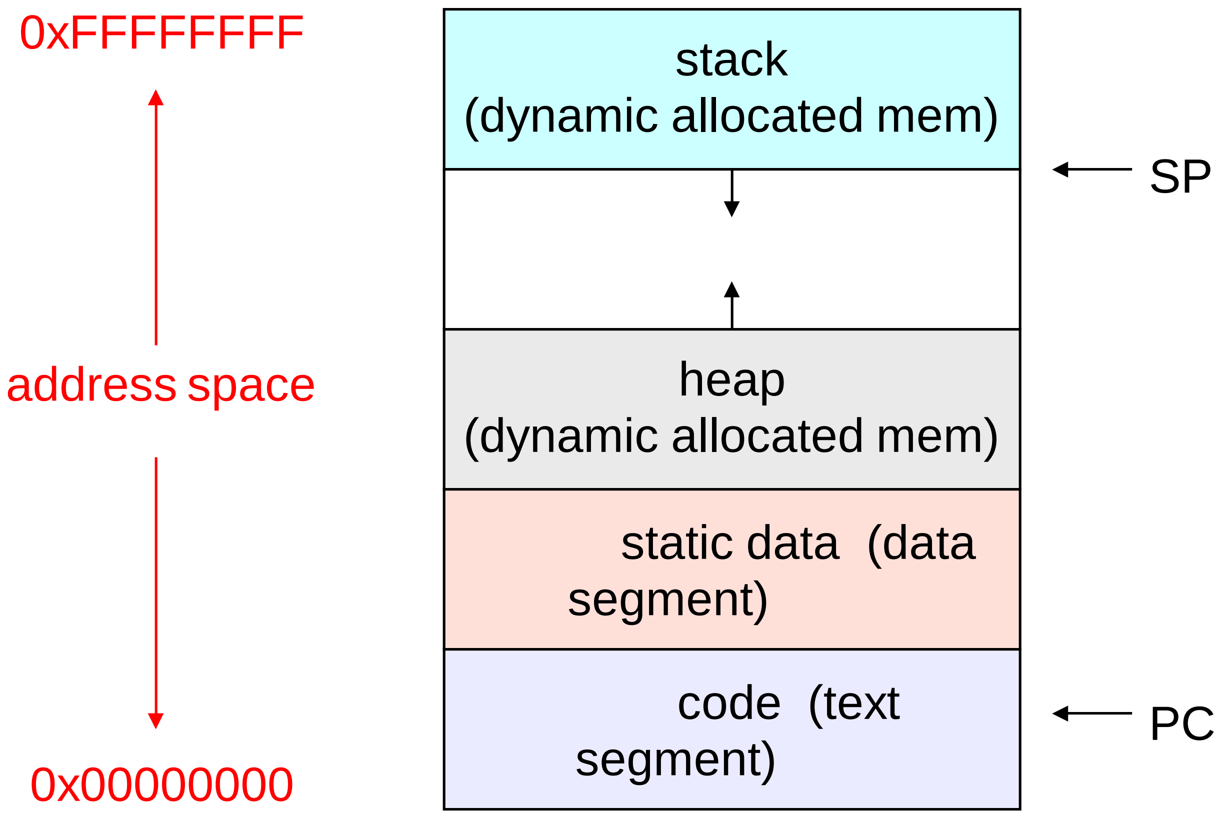
–Shared memory includes the heap and global/static data

–No memory protection among the threads

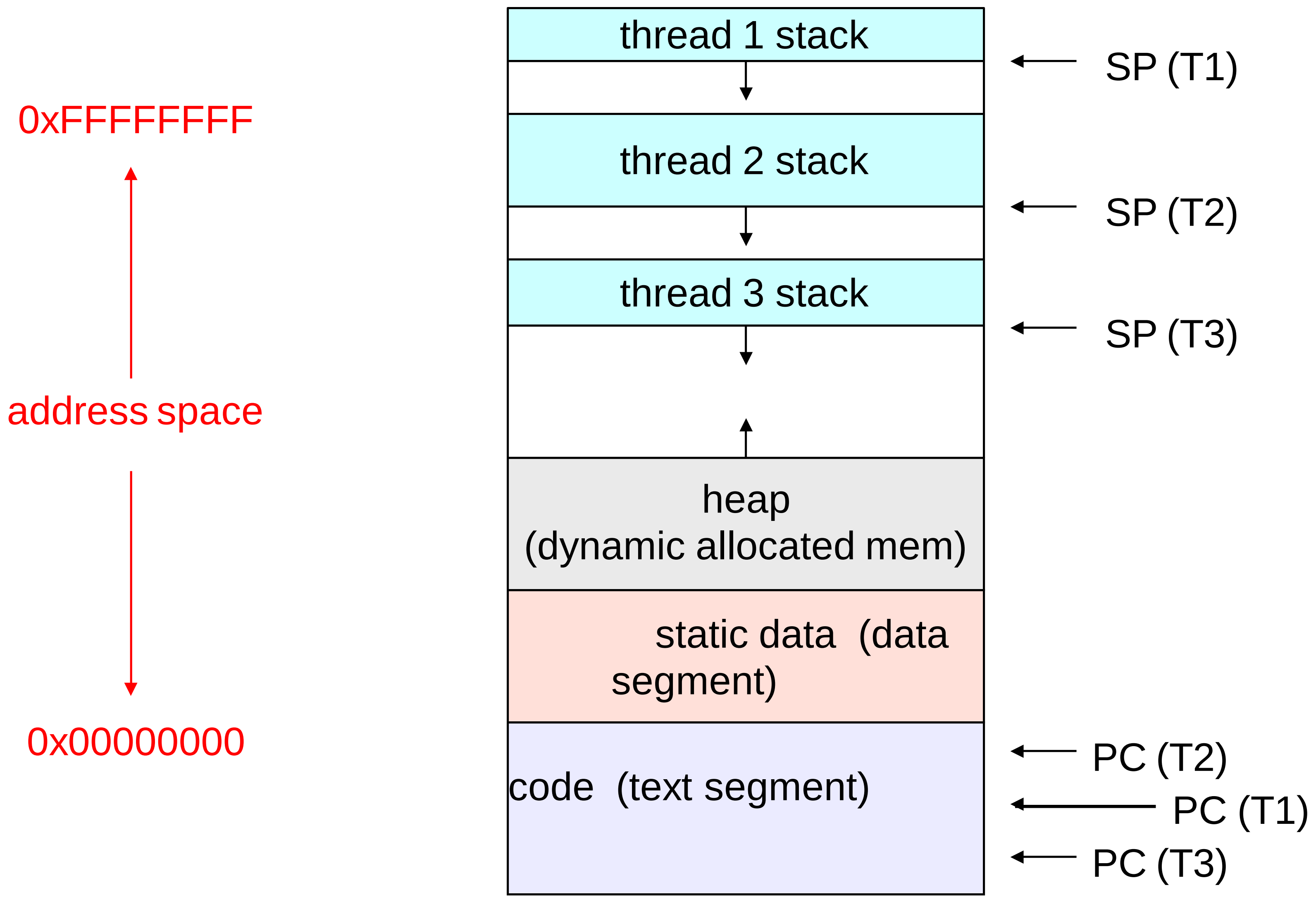
Preemptive multitasking:

– Operating system preempts & schedules threads

(old) Process address space



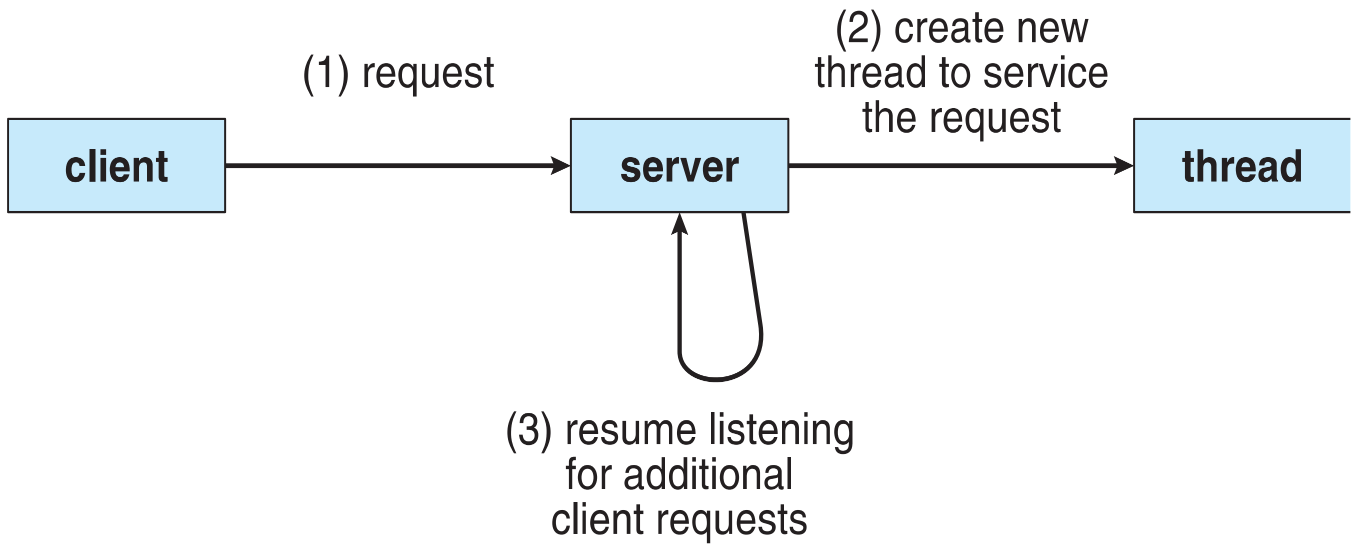
(new) Address space with threads



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"

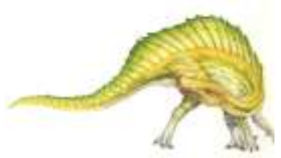
Multithreaded Server Architecture


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

If the web-server process is **multithreaded**, when a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests

Benefits of Threads

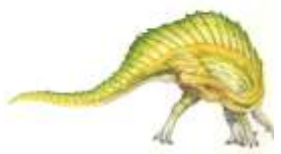
- **Responsiveness** – may allow continued execution if part of process is blocked, or performing lengthy operation, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than *shared memory* or *message passing*
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures



Forking vs threading

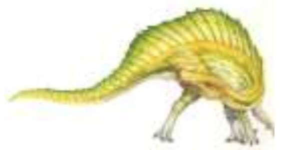
- Few things to note about forking are:
  - The child process will be having it’s own unique process ID.
  - The child process shall have it’s own copy of parent’s file descriptor.
  - Child will have it’s own address space and memory.

  - Forking is much safer and more secure because each forked process runs in its own virtual address space. If one process crashes or has a buffer overrun, it does not affect any other process at all.



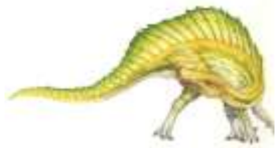
Forking vs threading

- Pitfalls in Fork::
  - In fork, every new process should have it’s own memory/address space, hence a longer startup and stopping time.
  - If you fork, you have two independent processes which need to talk to each other in some way. This inter-process communication is really costly.
  - When the parent exits before the forked child, you will get a ghost process. That is all much easier with a thread. You can end, suspend and resume threads from the parent easily. And if your parent exits suddenly the thread will be ended automatically.
  - In-sufficient storage space could lead the fork system to fail.



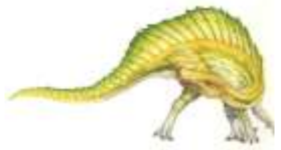
Forking vs threading

- Few things to note about threading are:
  - Thread are most effective on multi-processor or multi-core systems.
  - Threads are more effective in memory management because they uses the same memory block of the parent instead of creating new.



Forking vs threading

- Pitfalls in threads:
  - Race conditions: there is no natural protection from having multiple threads working on the same data at the same time without knowing that others are messing with it
  - They may also execute at different speeds.
  - Mutexes must be utilized to achieve a predictable execution order and outcome.



Forking vs threading

- Why not creating new processes instead of threads?
  - Process creation is **heavy-weight** while thread creation is **light-weight**
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

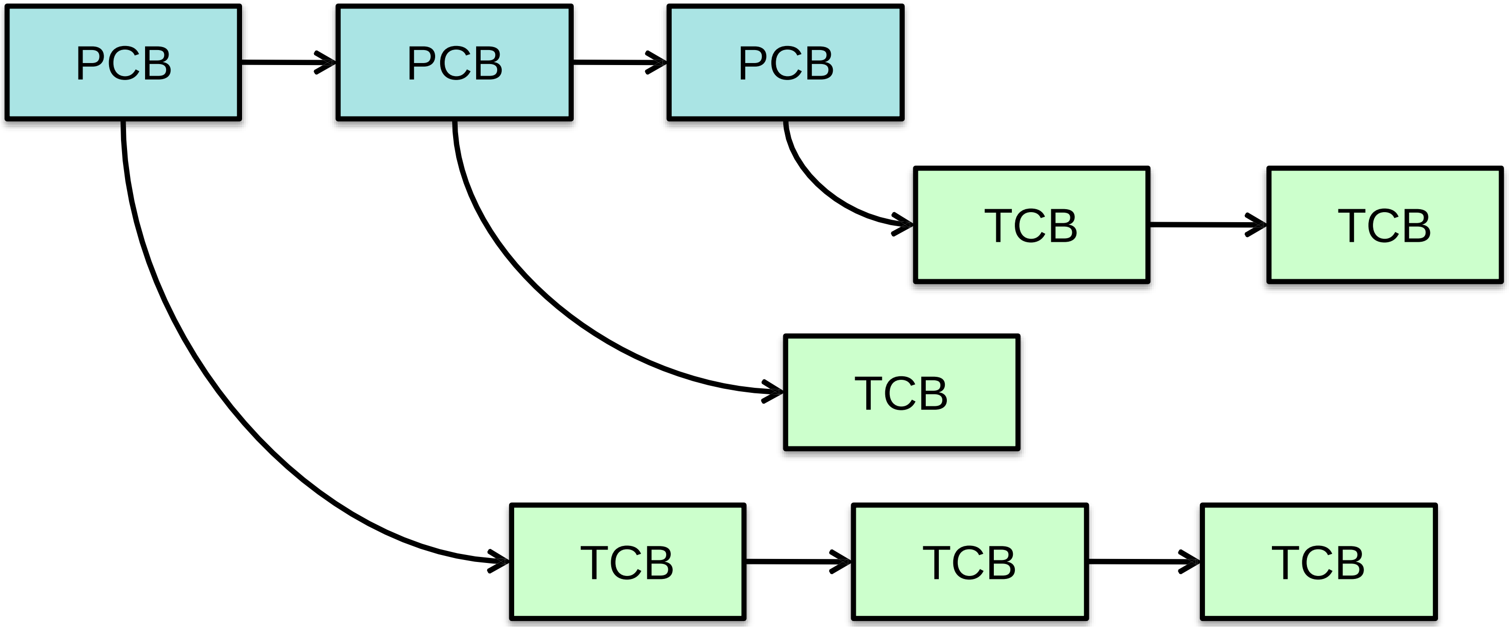
Implementation

Process info (Process Control Block) contains one or more Thread Control Blocks (TCB):

–Thread ID

–Saved registers

–Other per-thread info (signal mask, scheduling parameters)



Scheduling

A thread-aware operating system scheduler schedules *threads*, not *processes*

– A process is just a container for one or more threads

Scheduling Challenges

Scheduler has to realize:

- Context switch among threads of different processes is more expensive
  - Flush cache memory
  - Flush virtual memory translation lookaside buffer
  - Replace page table pointer in memory management unit

- CPU Affinity
  - Rescheduling threads onto a different CPU is more expensive
  - The CPU’s cache may have memory used by the thread cached
  - *Try to reschedule the thread onto the same processor on which it last ran*

Process vs. Thread context switch

A thread switch within the same process is **not** a full context switch – the address space (memory map) does not get switched

```
linux/arch/i386/kernel/process.c:

/* Re-load page tables for a new address space */
{
unsigned long new_cr3 = next->tss.cr3;  if (new_cr3 != prev-
>tss.cr3)
asm volatile("movl %0,%%cr3": :"r" (new_cr3));
}
```

This statement tests if the new task has the same memory map as the current one. If so, we’re switching threads and will not run the instruction to switch the memory mapping tables

Where do threads come from?

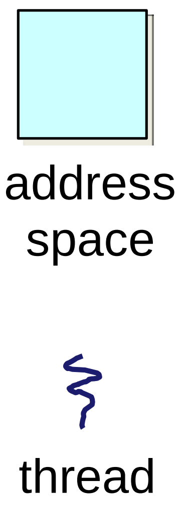
- Natural answer: the OS is responsible for creating/managing threads
  - For example, the kernel call to create a new thread would
    - allocate an execution stack within the process address space
    - create and initialize a Thread Control Block

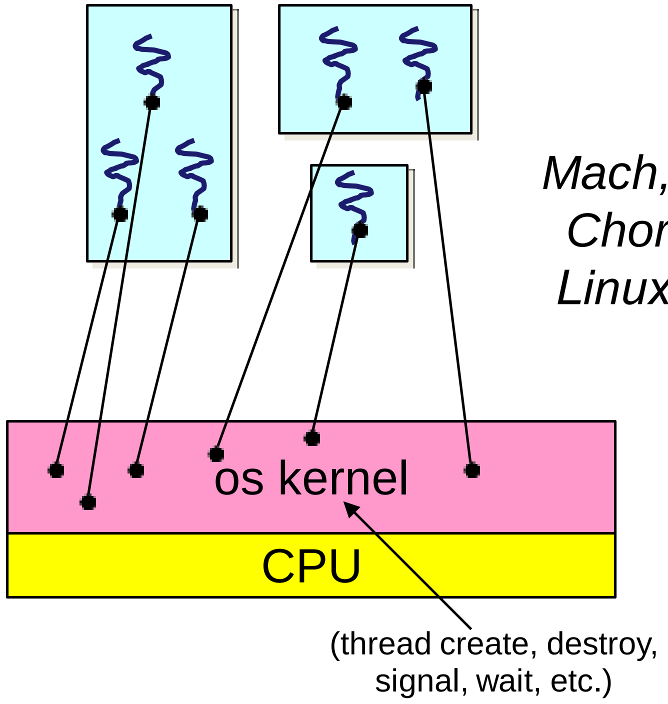
– stack pointer, program counter, register values

- 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

Kernel threads





Mach, NT, Chorus, Linux, …

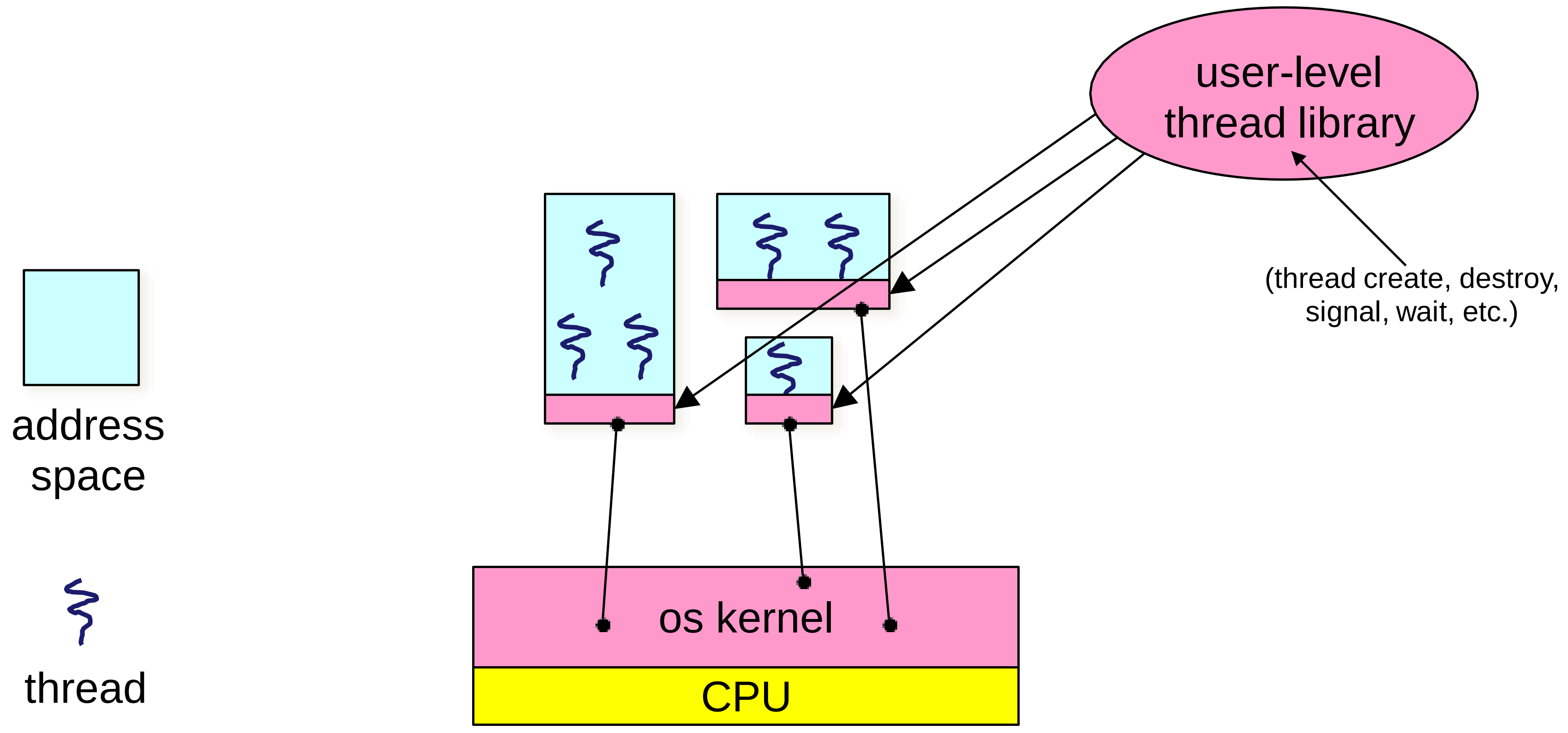
Kernel threads

- 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
    - 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
    - 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**
    - context switch
    - argument checks
  - must maintain kernel state for each thread

Cheaper alternative

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

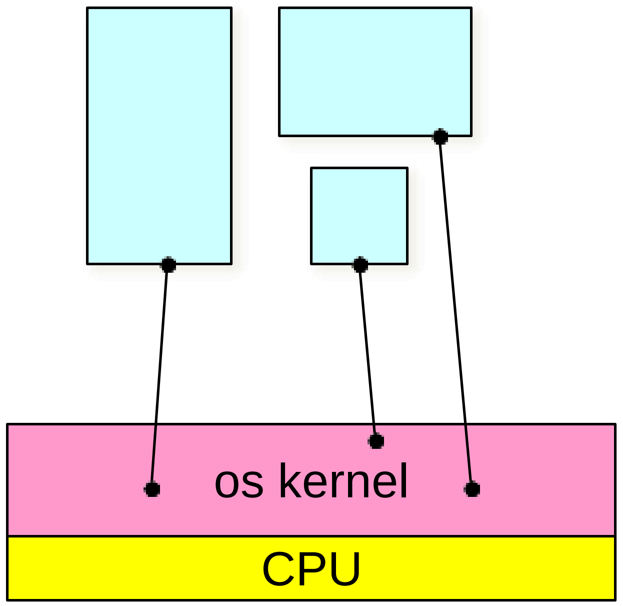
User-level threads


user-level thread library
(thread create, destroy, signal, wait, etc.)
address space
thread
os kernel
CPU

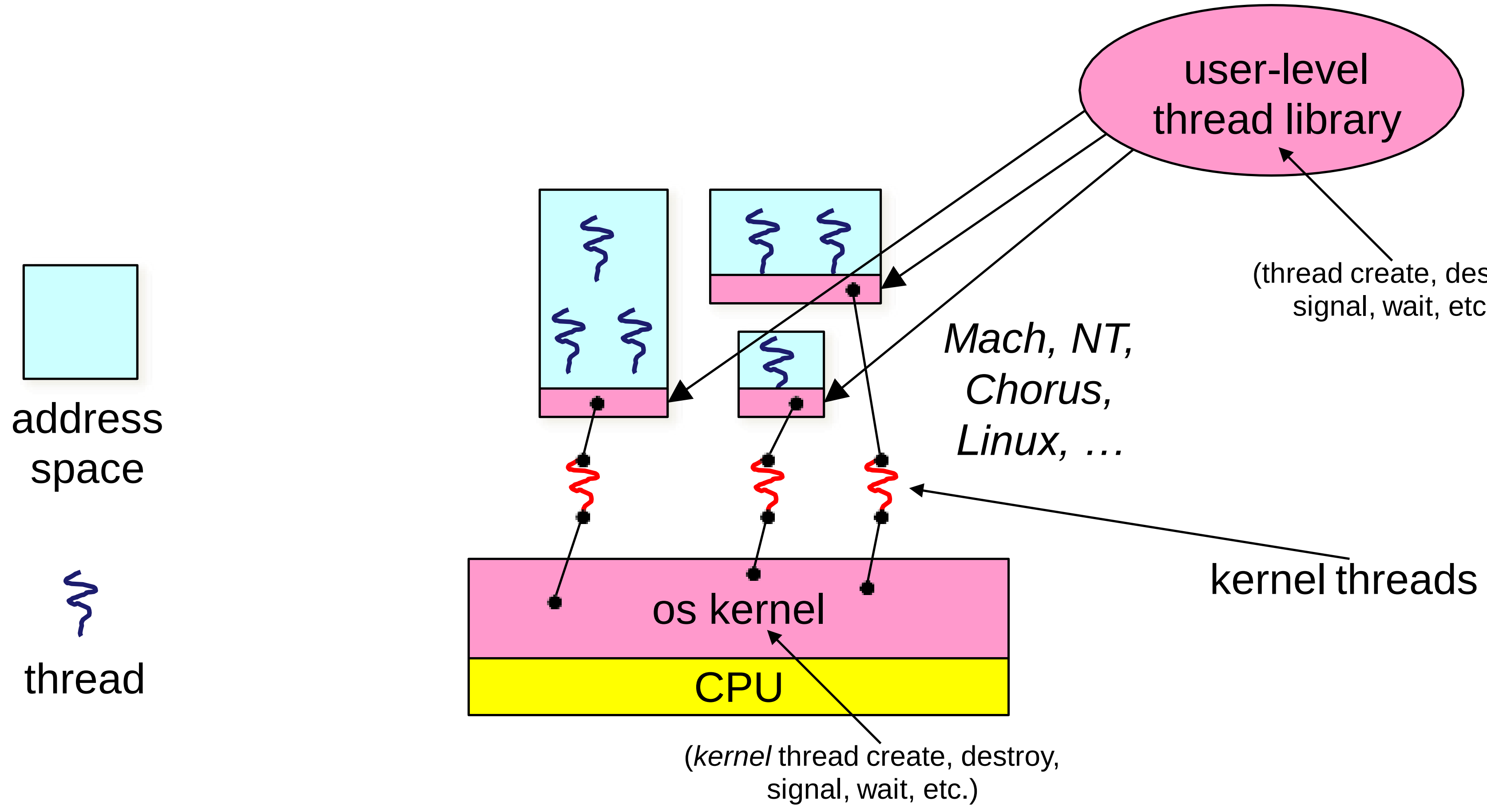
Now thread id is unique within the context of a process, not unique system-wide

User-level threads: what the kernel sees

address space
thread



User-level threads



user-level thread library
(thread create, destroy, signal, wait, etc.)
Mach, NT, Chorus, Linux, …
address space
thread
kernel threads
os kernel
CPU
(*kernel* thread create, destroy, signal, wait, etc.)

One problem: If a user-level thread blocked due to I/O, all other blocked

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

OLD Performance example

- On a 700MHz Pentium running Linux 2.2.16 (only the relative numbers matter; ignore the ancient CPU!):
  - Processes
    - **`fork/exit`**: 251 μs
  - Kernel threads
    - **`pthread_create()/pthread_join()`**: 94 μs **(2.5x faster)** ← Why?
  - User-level threads
    - **`pthread_create()/pthread_join`**: 4.5 μs **(another 20x faster)** ← Why?

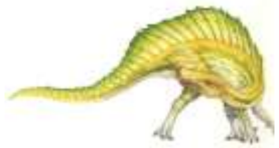
User Threads and Kernel Threads

- **Kernel threads**
  - Supported within the kernel of the OS itself
  - Allowing the kernel to perform **multiple simultaneous tasks** and/or to **service multiple kernel system calls** simultaneously.

- **User threads**
  - These are the threads that application programmers use in their programs.
  - The **thread library** contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution and for saving and restoring thread contexts.

User Threads and Kernel Threads

- **User threads** - management done by **user-level threads library**
- Three primary thread libraries:
  - Provides programmer with API for creating and managing threads
    - POSIX (Portable Operating System Interface): **Pthreads**
    - Windows threads
    - Java threads
- **Kernel threads** - Supported by the Kernel managed by OS
- POSIX **Pthreads**:
  - Provide a library entirely in user space with no kernel support



Kernel-level threads vs. User-level threads

Kernel-level

- Threads supported by operating system
- OS handles scheduling, creation, synchronization

User-level

- Library with code for creation, termination, scheduling
- Kernel sees one execution context: one process

User-level threads

Advantages

- **Low-cost**: user level operations that do not require switching to the kernel
- Scheduling algorithms can be replaced easily & custom to app
- Greater portability

User-level threads

– User-level threads only block under two circumstances:

1. When they hit a page fault or some other condition that the threading library can't handle
2. When the entire process has no way to make forward progress

– That still means that if any user-level thread hits a page fault, the entire process cannot make forward progress until the page fault is serviced

Disadvantages

- If a thread is blocked, all threads for the process are blocked
- Cannot take advantage of multiprocessing

Kernel-level threads

Advantages

–If a thread is blocked, all threads for the process are not influenced

–Can take advantage of multiprocessing

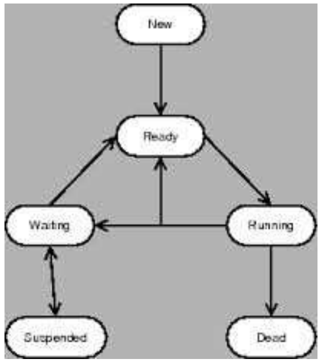
Disadvantages

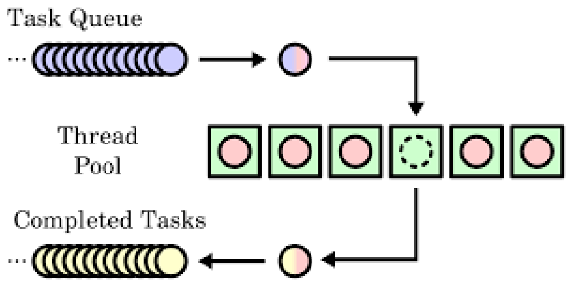
–High cost of context-switch

–Thread tables are stored in a fixed table space or stack space of the operating system, so the number of kernel-level threads is limited and cannot scale as well as user-level threads

User-level thread implementation

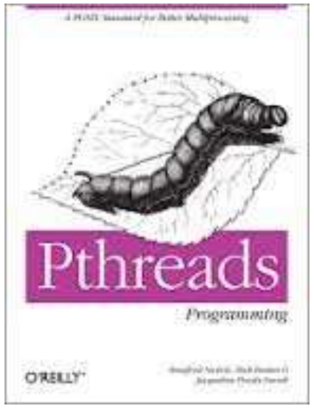
- The OS schedules the kernel thread
- The kernel thread executes user code, including the thread support library and its associated thread scheduler
- The thread scheduler determines when a user-level thread runs
  - it uses queues to keep track of what threads are doing: run, ready, wait
    - just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



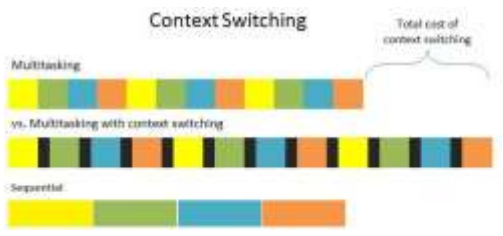


Thread interface

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



Thread context switch


Context Switching
Total cost of context switching
Multitasking
vs. Multitasking with context switching
Sequential

- Very simple for user-level threads:
  - save context of currently running thread
    - push CPU state onto thread stack
  - restore context of the next thread
    - pop CPU state from next thread's stack
  - return as the new thread
    - execution resumes at PC of next thread
  - Note: no changes to memory mapping required
- This is all done in assembly language
  - it works at the level of the procedure calling convention

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

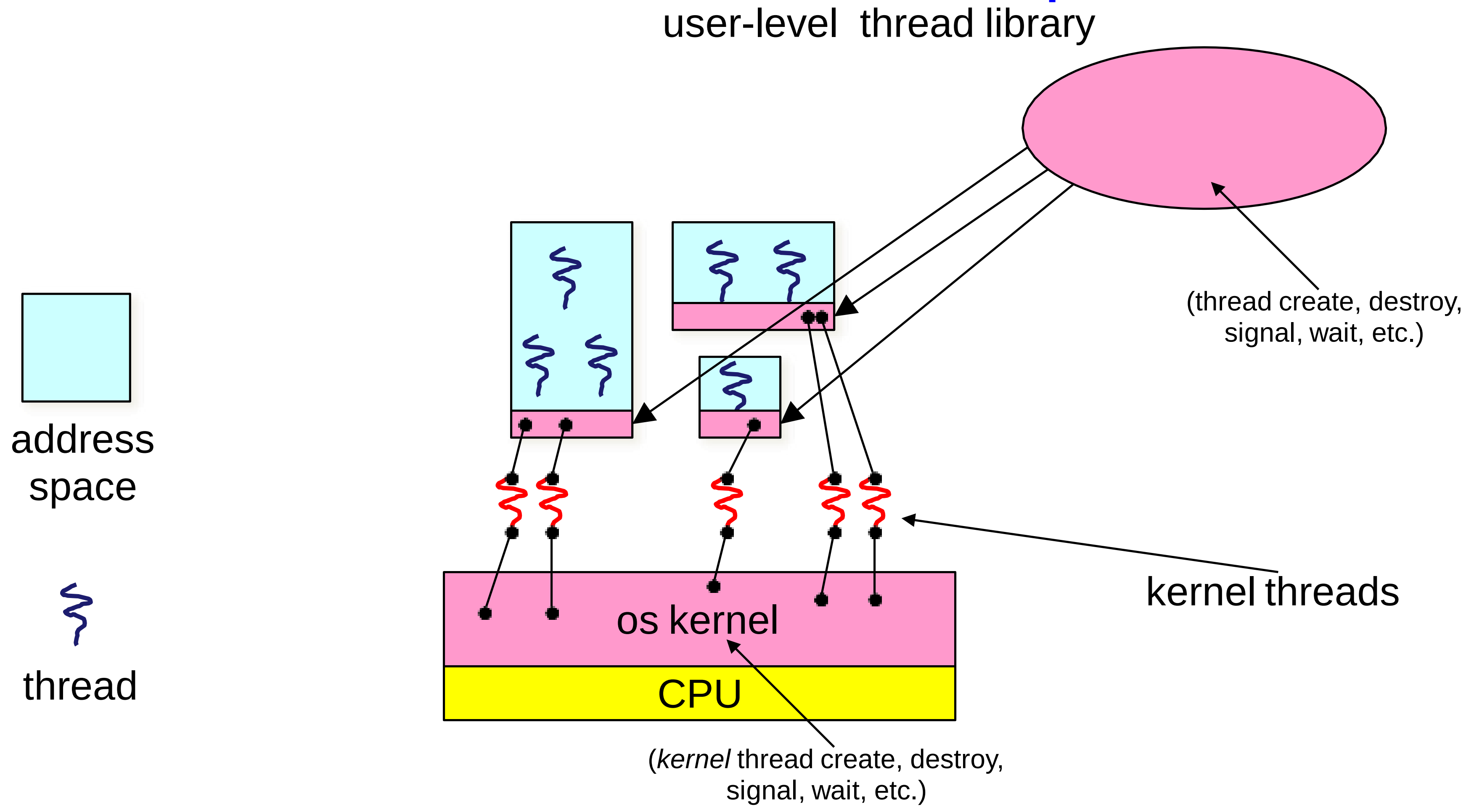
- Strategy 1: force everyone to cooperate
  - a thread willingly gives up the CPU by calling **`yield()`**
  - **`yield()`** calls into the scheduler, which context switches to another ready thread
  - what happens if a thread never calls **`yield()`**?

- 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

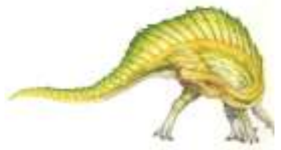

user-level thread library
(thread create, destroy, signal, wait, etc.)
address space
thread
kernel threads
os kernel
CPU
(*kernel* thread create, destroy, signal, wait, etc.)

Multithreading Models

Relationship between user threads and kernel threads

Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling

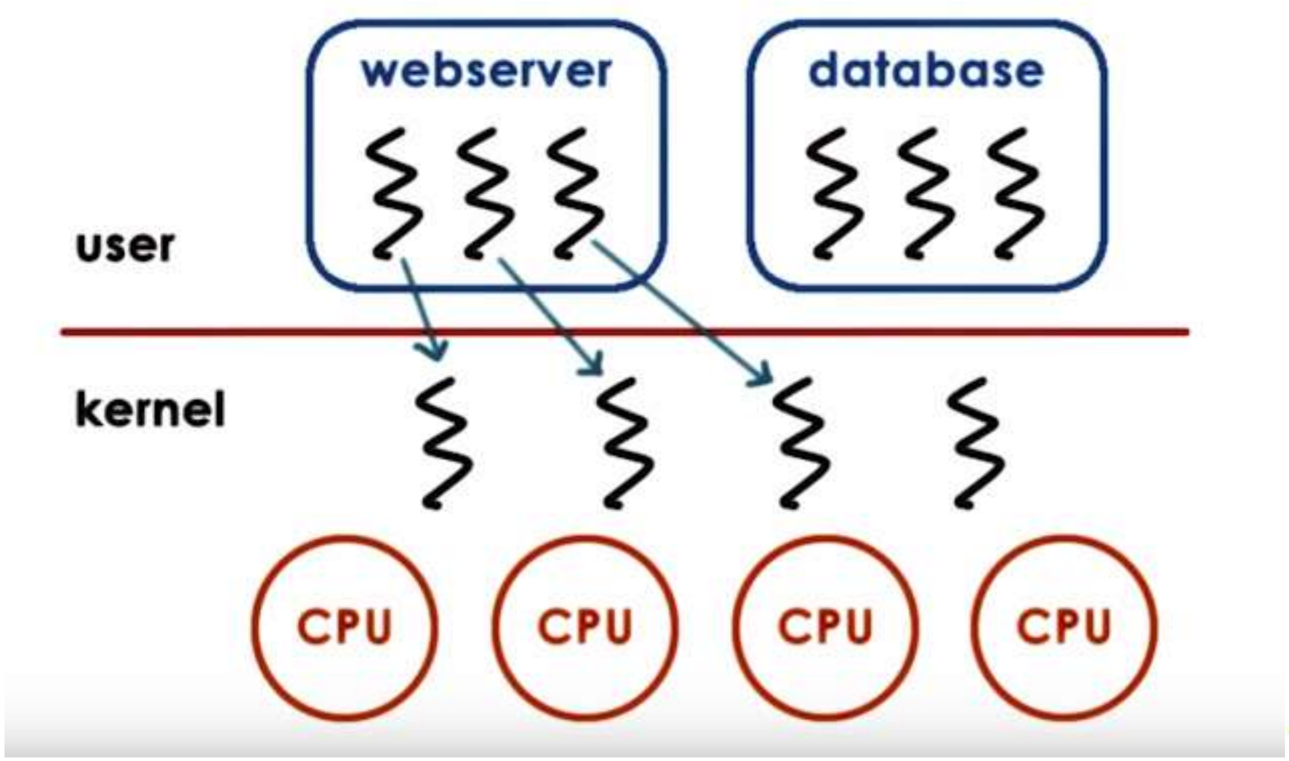
- Many-to-One
- One-to-One
- Many-to-Many



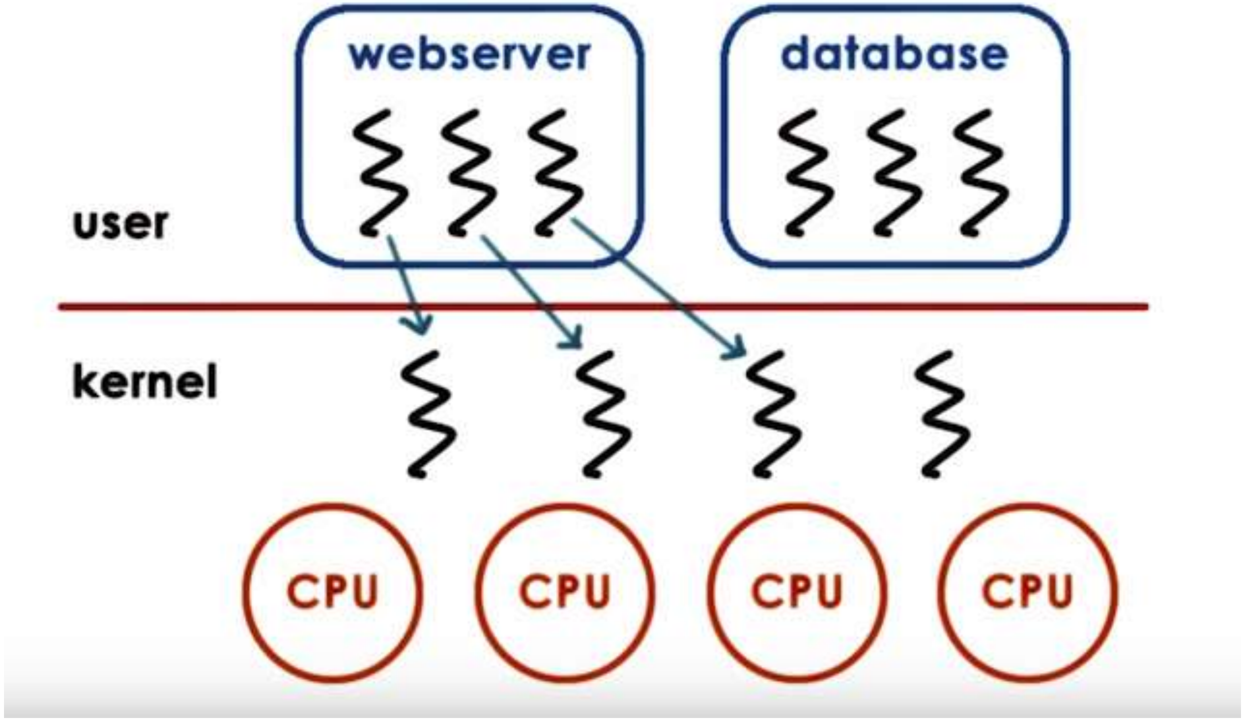
One-to-one

In this model, each **user-level thread** is associated with **a kernel-level thread**.

When a user process creates a user-level thread, there is a kernel-level thread either created or already present.

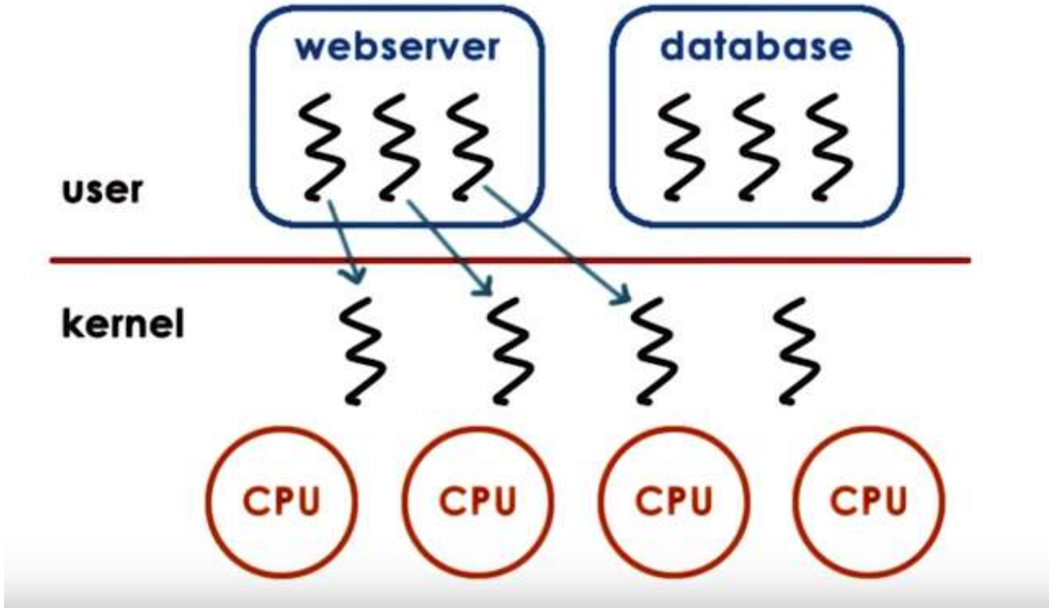


One-to-one



Advantages: Operating system **can directly** see, manage, synchronize, schedule, or block threads. Since OS already supports the threading application can directly take advantage of this support.

One-to-one



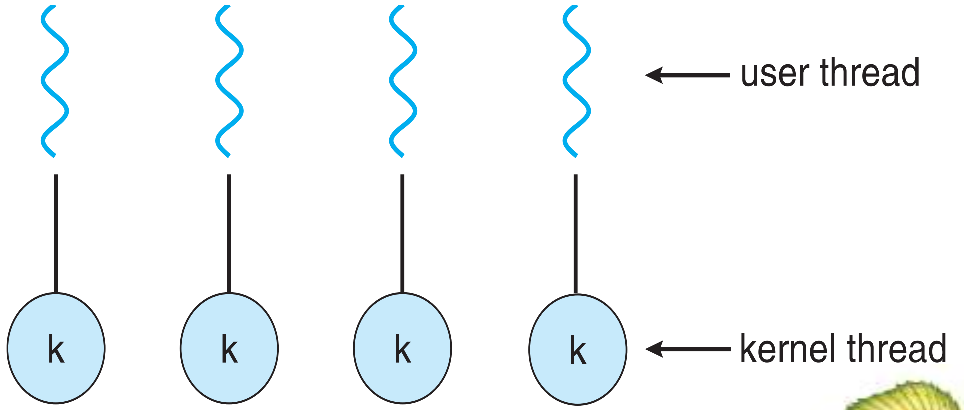
Disadvantage: For every operation, the application has to do a **system call** which involves **changing the user to kernel mode** which is an **expensive** operation.

This model is **not much flexible** as applications have to rely on thread support provided by OS.

As OS may have **limits** on the number of threads it can create or may have any other policies.

One-to-One

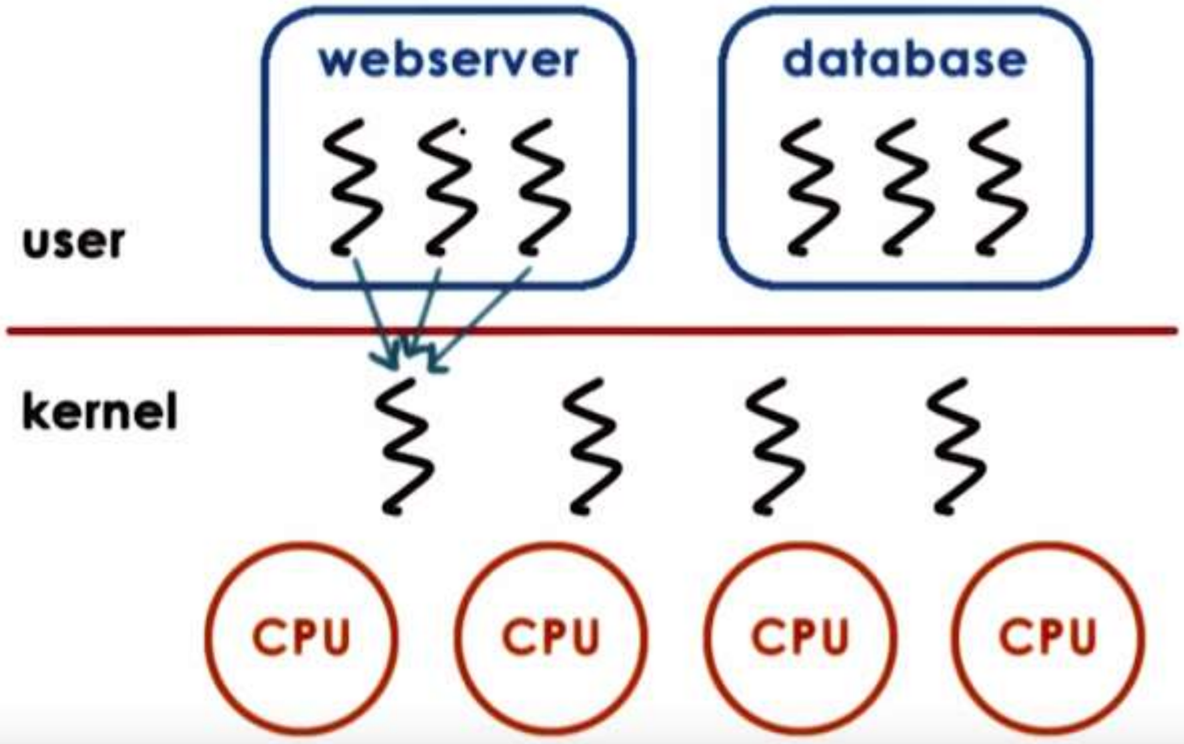
- Each user-level thread is maps to a kernel thread
- More concurrency than many-to-one
- Allows multiple threads to run in parallel on multiprocessors
- Drawback: creating a user thread requires creating the corresponding kernel thread
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



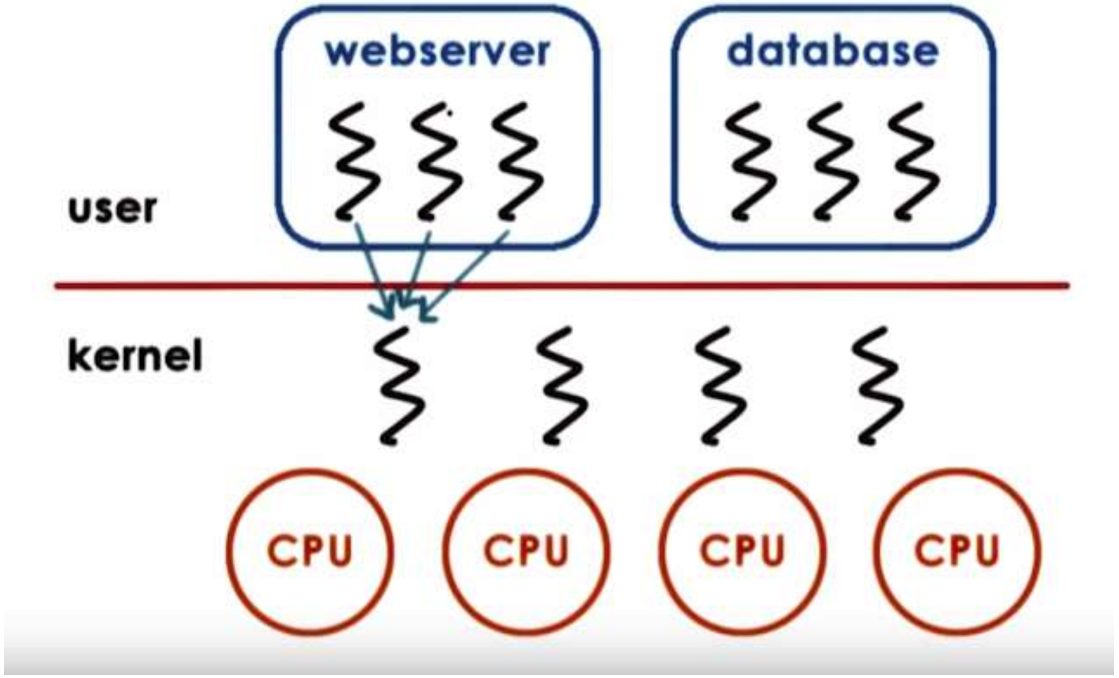
Many-to-one

In this model, **many user level threads** are mapped to **a single kernel level** thread.

At user-level, there is a thread management library which is present at the user-level which decides which of the user-level thread will actually be mapped to the kernel thread.



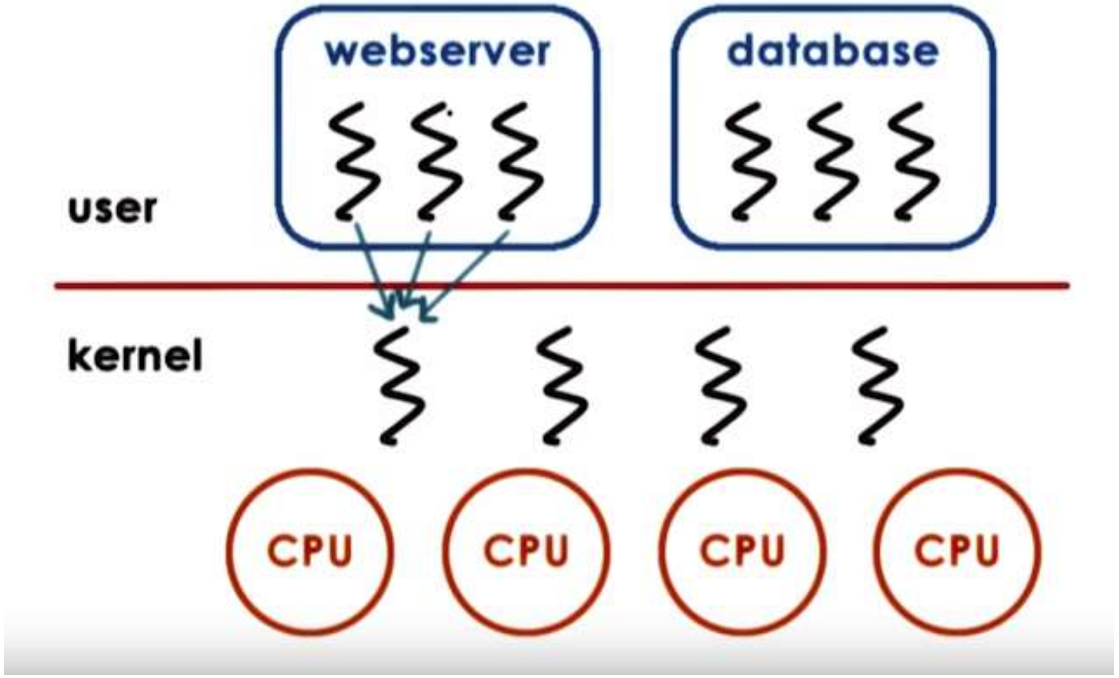
Many-to-one



Advantages: As all of the management is done by thread management library present at the user-level, everything is done by user-level thread library like scheduling, synchronization, context change, etc. So no kernel support is required hence no constraints on limits, policies, etc.

Also, we don’t need to do frequent system calls which is an expensive task.

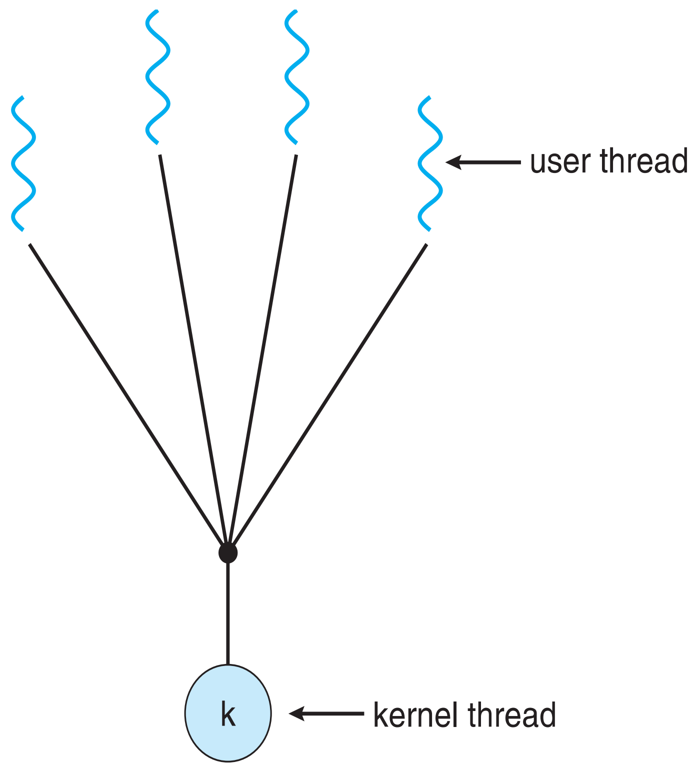
Many-to-one



Disadvantages: OS does not have any information regarding the scheduling done at the user-level, so OS may block the entire process if one user-level thread blocks on I/O.

Many-to-One

- Many user-level threads are mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads are unable to run in parallel on muticore systems because only one thread can access the kernel at a time
- Few systems currently use this model

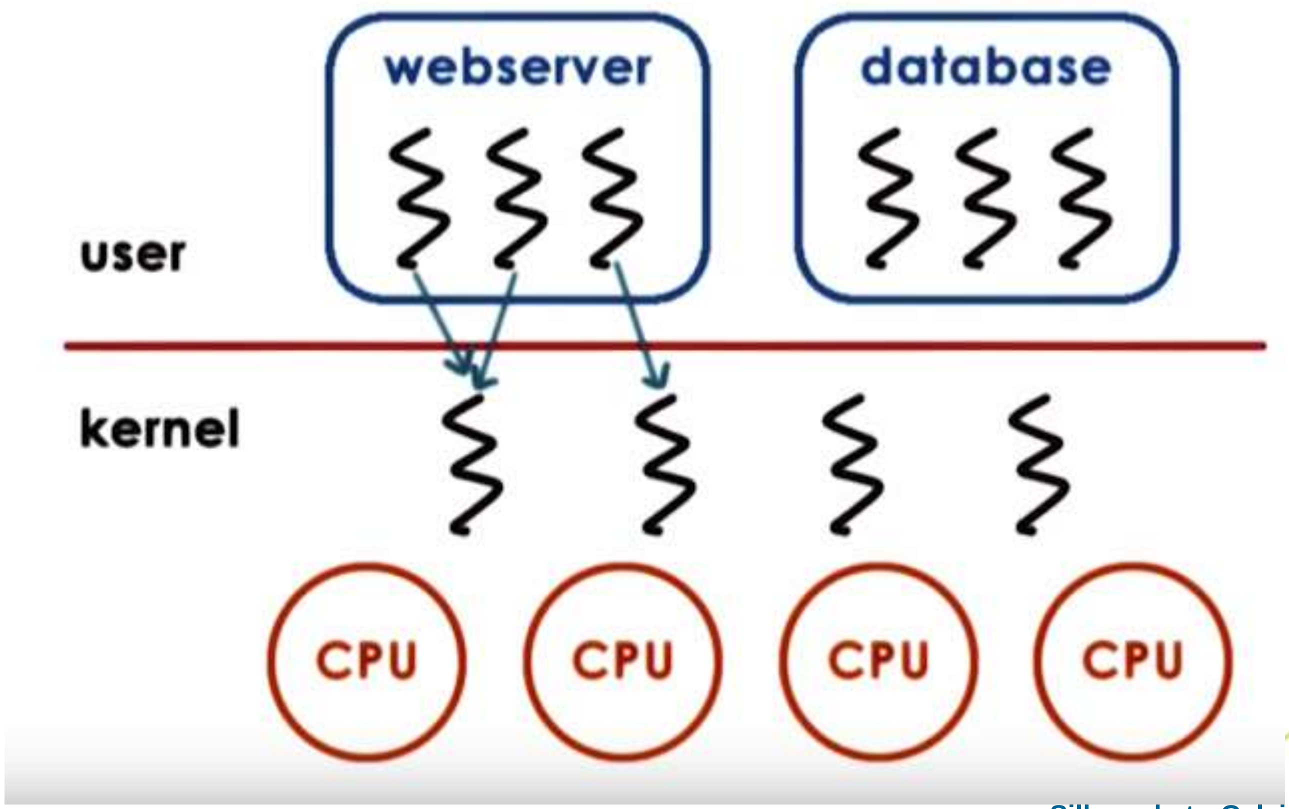


It does not result in true concurrency

Many-to-Many

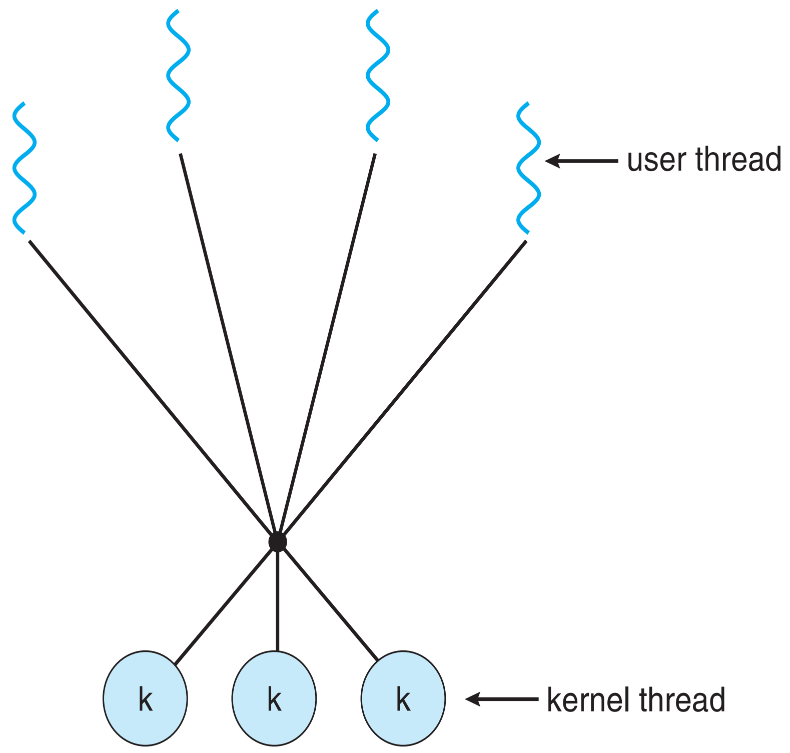
In this model, some threads are mapped directly to kernel threads while in some cases many threads are mapped to a single kernel thread like many-to-one and are managed by the thread management library at user-level.

However, this model requires coordination between user-level thread manager and kernel-level thread manager.



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Developers can create as many user threads as necessary, and the corresponding kernels can run in parallel on the multiprocessor
- When a thread performs a blocking system call, the kernel can schedule another thread for execution



Summary

- Multiple threads per address space
- Kernel threads are much more efficient than processes, but still expensive
  - all operations require a kernel call and parameter validation
- User-level threads are:
  - much cheaper and faster
  - great for common-case operations
    - creation, synchronization, destruction
  - can suffer in uncommon cases due to kernel obliviousness
    - I/O
    - preemption of a lock-holder

End of Chapter 4