Motivation

- Operating systems (and application programs) often need to be able to handle multiple things happening at the same time
  - Process execution, interrupts, background tasks
- Humans are not very good at keeping track of multiple things happening simultaneously
- Threads are an abstraction to help bridge this gap

Why Concurrency?

- Servers
  - Multiple connections handled simultaneously
- Parallel programs
  - To achieve better performance
- Programs with user interfaces
  - To achieve user responsiveness while doing computation
- Network and disk bound programs
  - To hide disk latency

Definitions

- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time

Multithreading

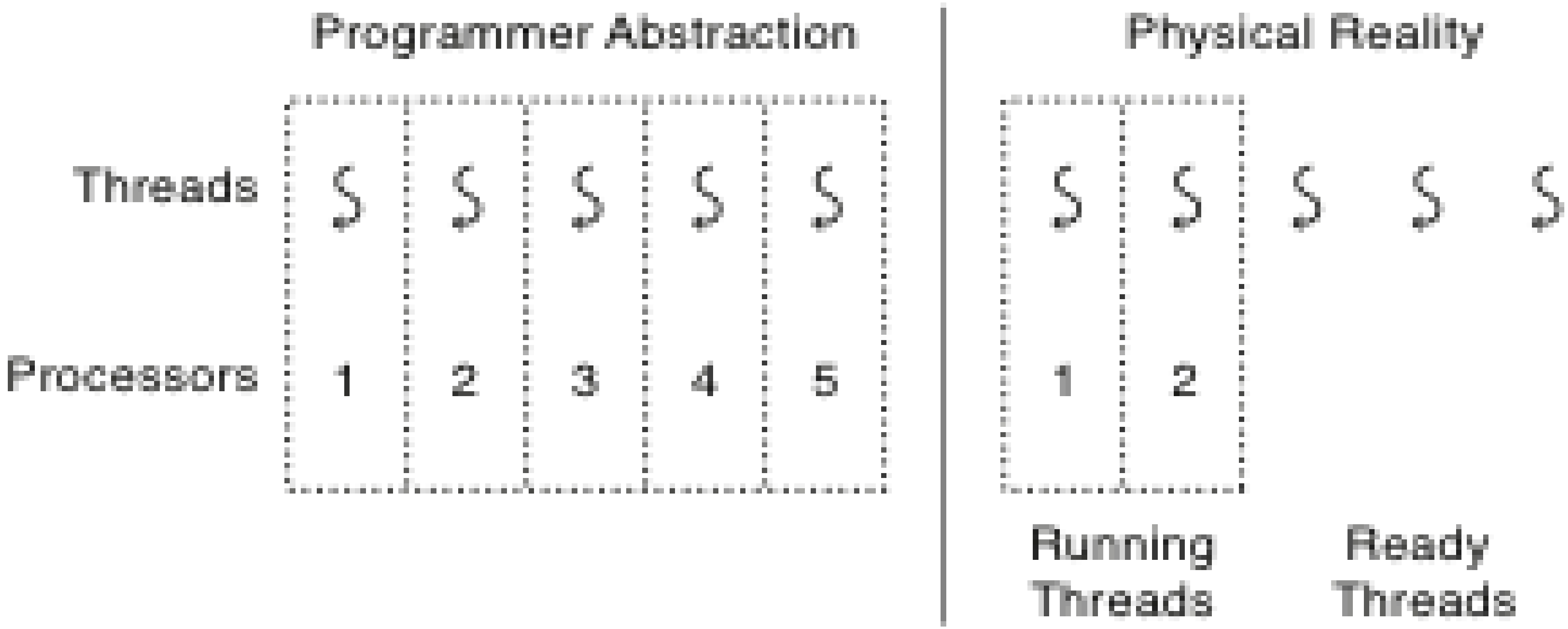
- Our process model so far: we defined a process as the Unit of resource ownership as well as the Unit of dispatching
- We want to separate these two concerns
  - Resource ownership:
    - Process remains unit of resource ownership
  - Program Execution / Dispatching:
    - A process can have multiple Threads of execution, Threads (lightweight processes) become the unit of dispatching

Multithreading

- Processes have at least one thread of control
  - Is the CPU context, when process is dispatched for execution
- Multithreading is the ability of an operating system to support multiple threads of execution within a single process
- Multiple threads run in the same address space, share the same memory areas
  - The creation of a thread only creates a new thread control structure, not a separate process image

Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule



Programmer vs. Processor View

Programmer's View

```
.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.
```

Possible Execution #1

```
.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.
```

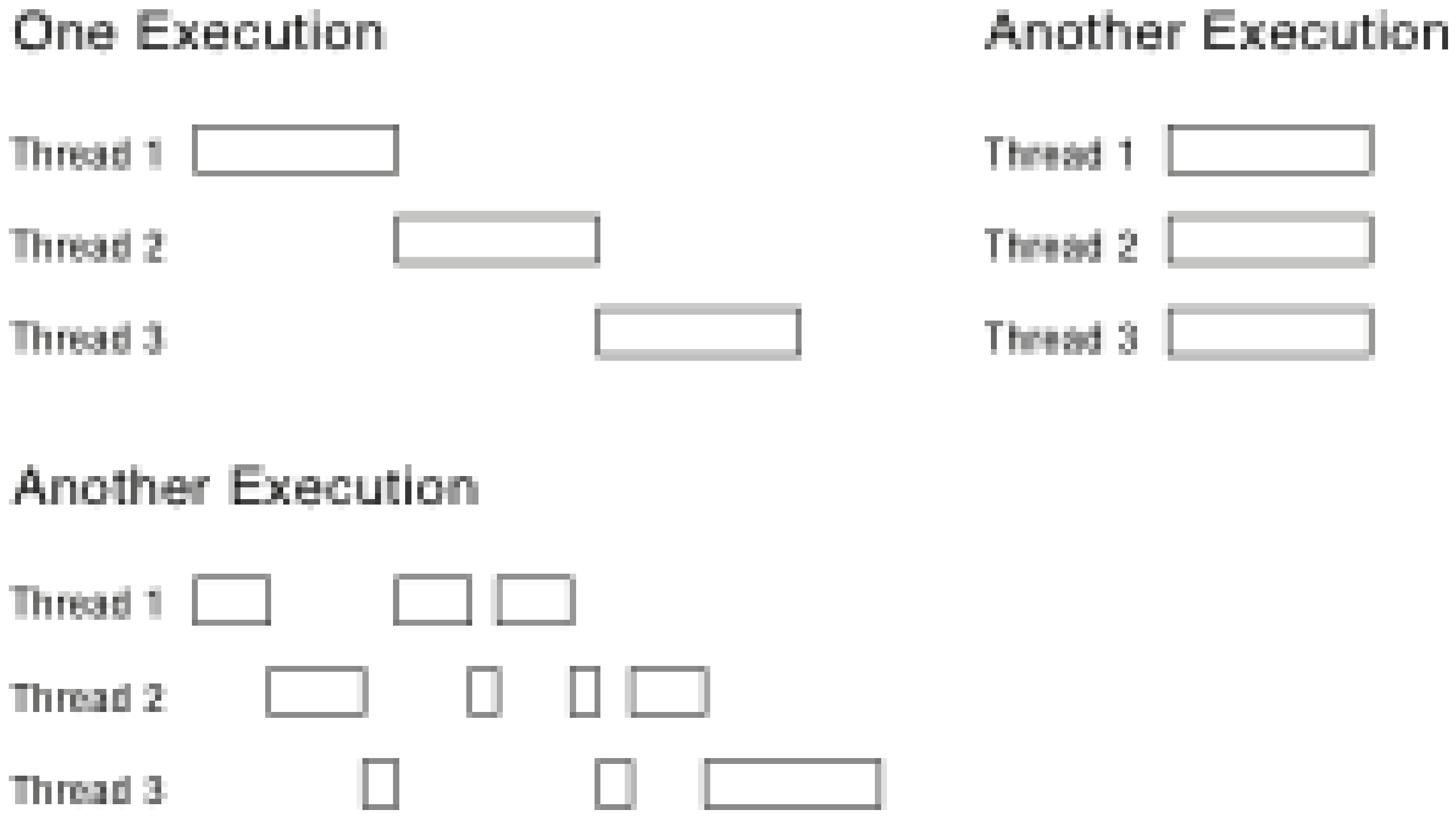
Possible Execution #2

```
.
.
.
x = x + 1;
................
Thread is suspended.
Other thread(s) run.
Thread is resumed.
................
y = y + x;
z = x + 5y;
```

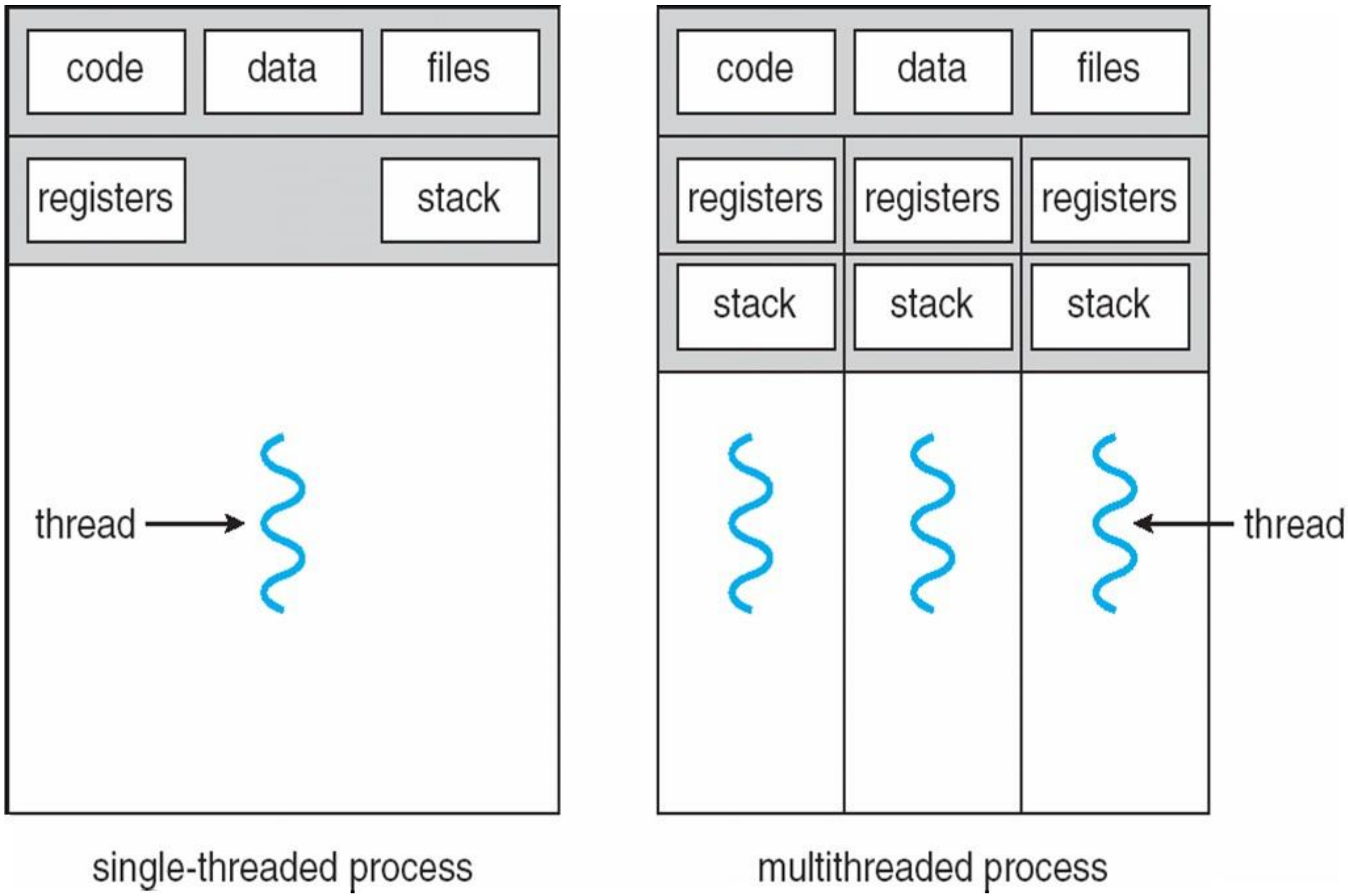
Possible Execution #3

```
.
.
.
x = x + 1;
y = y + x;
................
Thread is suspended.
Other thread(s) run.
Thread is resumed.
................
z = x + 5y;
```

Possible Executions



Multithreaded Process Model



Process

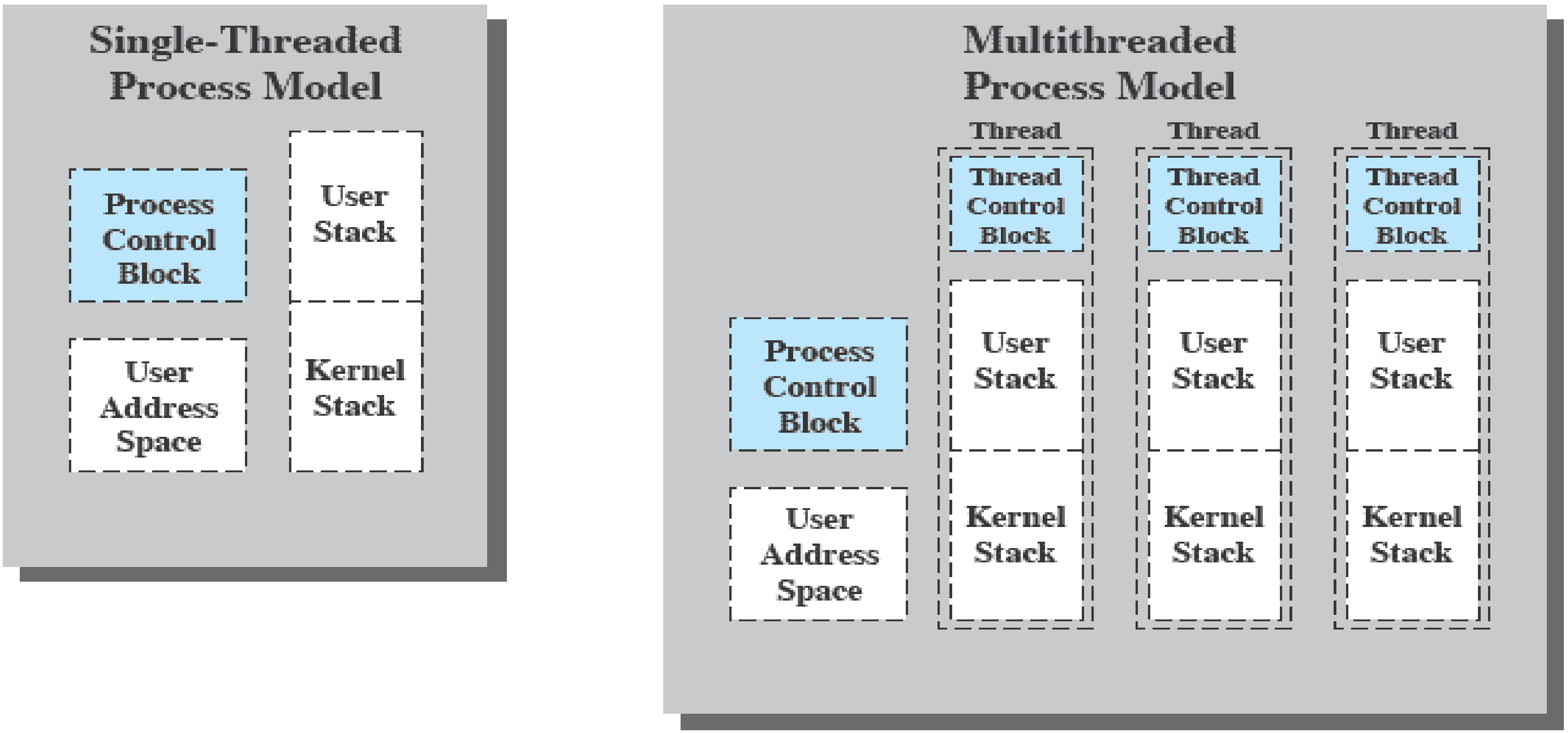
- Unit of resource ownership and protection
  - Resource ownership:
    - Process image, virtual address space
    - Resources (I/O devices, I/O channels, files, main memory)
  - Protection
    - Processors, other processes
      - Operating system protects process to prevent unwanted interference between processes

        memory, files, I/O resources

Threads

- Thread is defined as the unit of dispatching:
  - Represent a single thread of execution within a process
  - Operating system can manage multiple threads execution within a process
  - The thread is provided with its own register context and stack space
  - Threads are also called "lightweight processes"

Singlethreaded vs Multithreaded



Threads

- All threads share the same address space
  - Share global variables
- All threads share the same open files, child processes, signals, etc.
- There is no protection between threads
  - As they share the same address space they may overwrite each others data
- As a process is owned by one user, all threads are owned by one user

Threads vs Processes: Advantages

- Advantages of Threads
  - Much faster to create a thread than a process
    - Spawning a new thread only involves allocating a new stack and a new thread control block
    - 10times faster than process creation in Unix
  - Less time to terminate a thread
  - Much faster to switch between threads thanto switch between processes
  - Threads share data easily
  - Thread communication very efficient, no need to call kernel routines, as all threads live in same process context

Threads vs Processes: Disadvantages

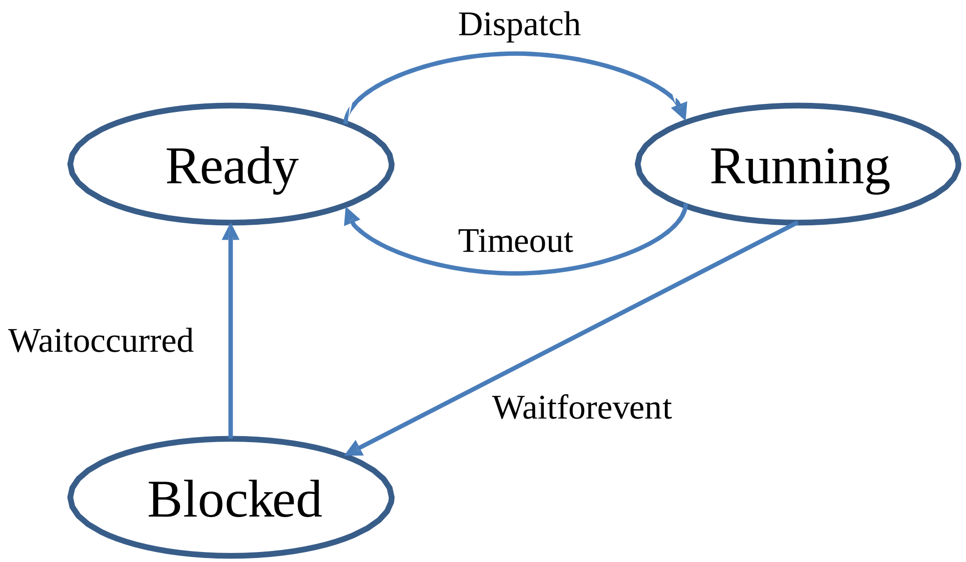
- Disadvantages
  - Processes are more flexible
    - They don’t have to run on the same processor
  - No protection between threads
    - Share same memory, may interfere with each other
  - If threads are implemented as user threads instead of kernel threads
    - If one thread blocks, all threads in process block

Thread Management

- Threads are described by the following:
  - –Thread execution state
    - running, ready, blocked
  - –Thread Control Block
    - A saved thread context when not running (each thread has a separate program counter)
  - –An execution stack
  - –Some per thread static storage for local variables
  - –Access to memory and resources of its process, shared with all other threads of that process

Thread States

- Threads have now three states
  - Running: CPU executes thread
  - Ready: thread control block is placed in Ready queue
  - Blocked: thread awaits event
- There is no suspend, as the process is suspended
- If one thread blocks
  - Is the whole process with all other threads blocked?
  - Or is only this single thread blocked?



Thread Operations

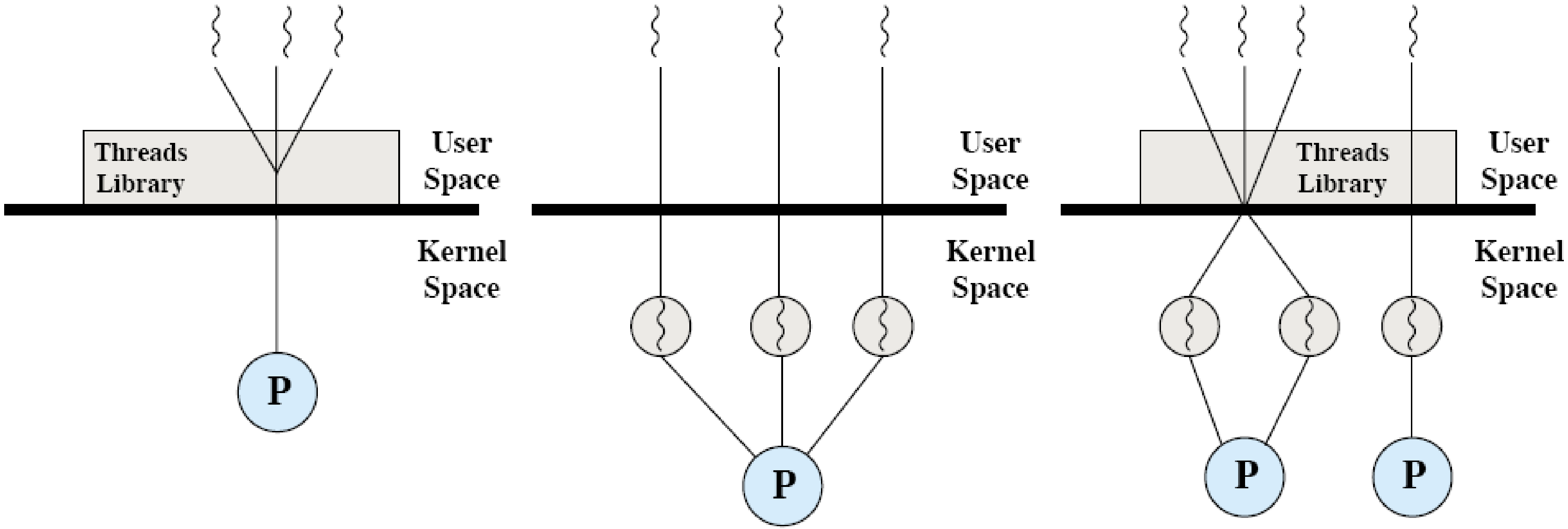
- There are four basic operation for managing threads
  - Spawn / create
    - A thread is created and provided with its own register context and stack space, it can spawn further threads
  - Block:
    - if a thread waits for an event, it will block
    - If the kernel manages threads: the processor may switch to another thread in the same or a different process
  - Unblock:
    - When the event occurs, for which the thread is waiting, it will be queued for execution
  - Finish:
    - When a thread completes, its register context and stacks are de allocated

Thread Implementation

- Two basic categories of threads
  - Userlevel threads
  - Kernellevel threads
- Characterised by the extent of the kernel being involved in their management

Thread Implementation

- Two main categories of thread implementation
  - –Userlevel Threads (ULTs)
  - –Kernellevel Threads (KLTs)
- Characterised by the extent of the kernel being involved in their management



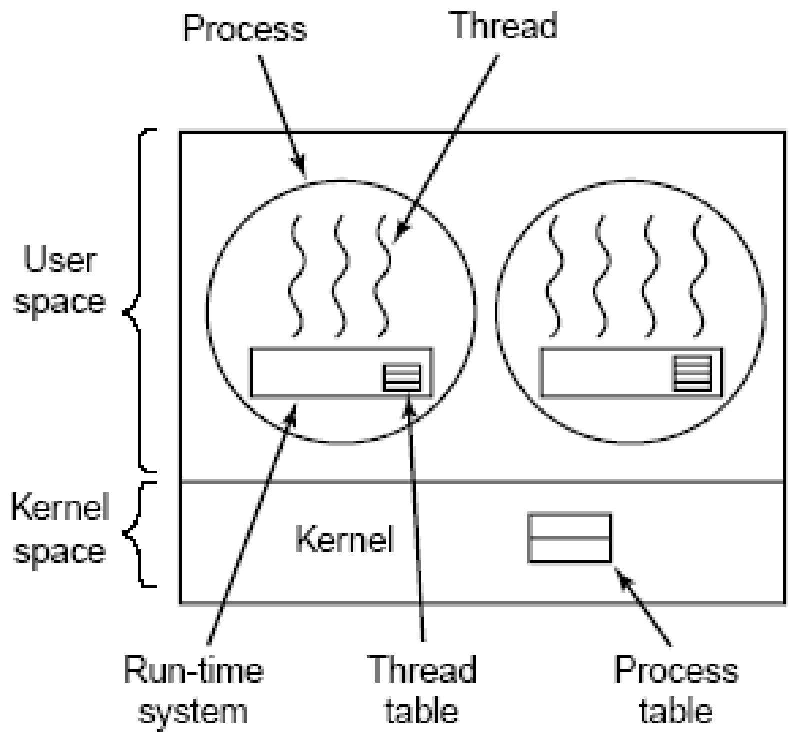
Pure UserLevel ULT | Pure KernelLevel KLT | CombinedLevel ULT/KLT

UserLevel Threads

- UserLevel Threads
  - Kernel not aware of the existence of threads
  - Process uses thread library functions to manage its threads
- Benefit
  - Light thread switching in user mode
  - No mode switch necessary (no call of kernel functions)
  - We can implement our own thread scheduling
- Also called “green threads” on some systems (e.g. Solaris)

UserLevel Threads: Disadvantage

- Process is still the Unit of Dispatch, not a thread:
  - –Kernel doesn’t know threads
- Disadvantage:
  - –Blocking of one thread blocks entire process, including all other threads in it
  - –Only one thread can access the kernel at a time, as the process is the unit of execution known by kernel
  - –No Distribution in Multiprocessor systems:
    - All threads run on the same processor in a multiprocessor system
    - Threads cannot run in parallel utilising different processors, as the process is dispatched on one processor

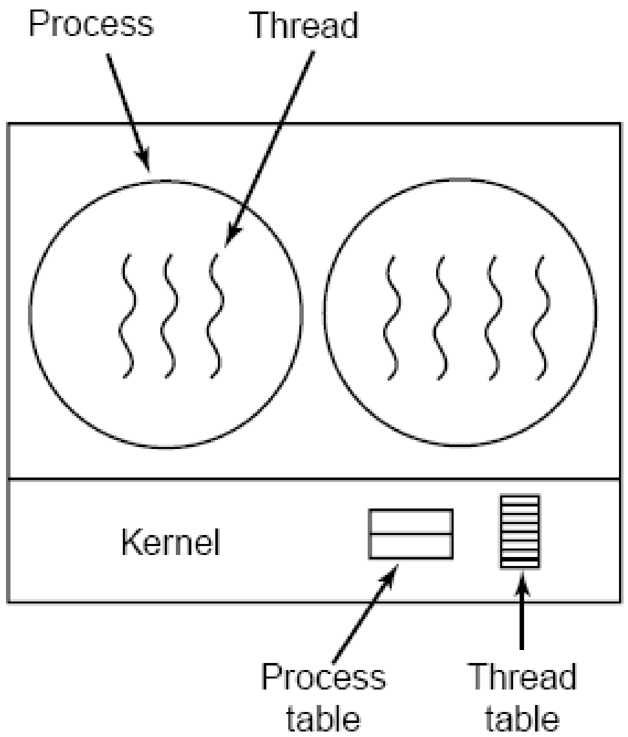


Kernellevel Threads

- Thread is Unit of dispatch
  - Kernel is aware of the existence of threads
  - Kernel manages each thread separately
- Benefit
  - Finegrain scheduling by kernel on thread basis
  - If a thread blocks (e.g. waiting for I/O), another one can be scheduled by kernel without blocking the whole process
  - Threads can be distributed to multiple processors and run in parallel
- Example Systems: Windows XP/7/8, Solaris, Linux, Mac OSX

KernelLevel Threads: Disadvantage

- A switch between threads of the same process involves kernel
  - 2 mode switches for each thread context switch, is as costly as process switch

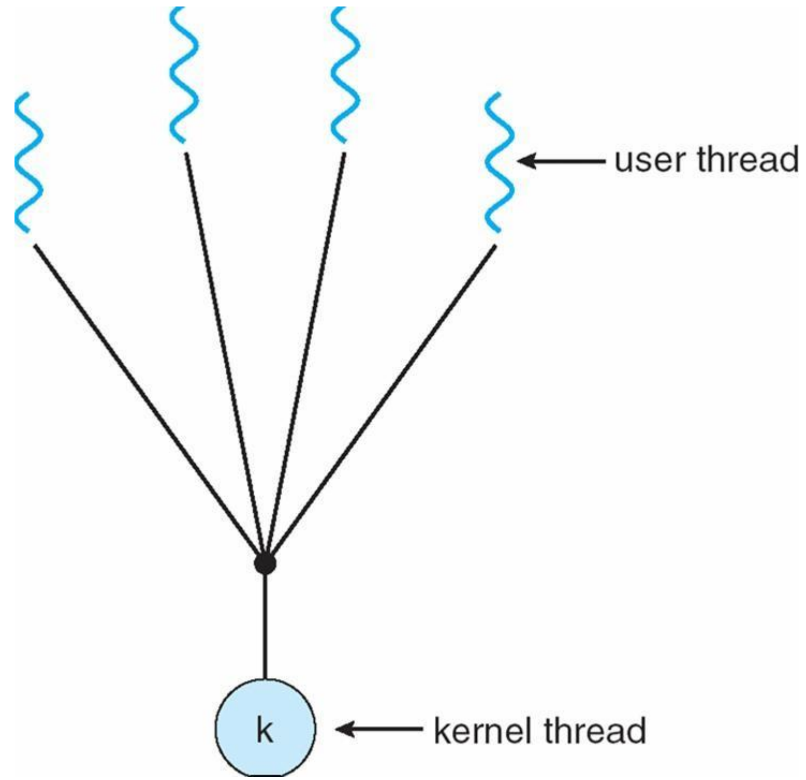


Hybrid Implementations

- Try to combine advantages of both userlevel and kernellevel threads
  - Userlevel: lightweight thread switching
  - Kernellevel: allows dispatch at thread level (same or different process), when one threads blocks
  - true parallelism of threads in multiprocessor systems possible
- Basic technique: Mapping of userlevel threads onto a limited set of kernel threads
- Different hybrid Multithreading Models:
  - Manytoone
  - Onetoone
  - Manytomany

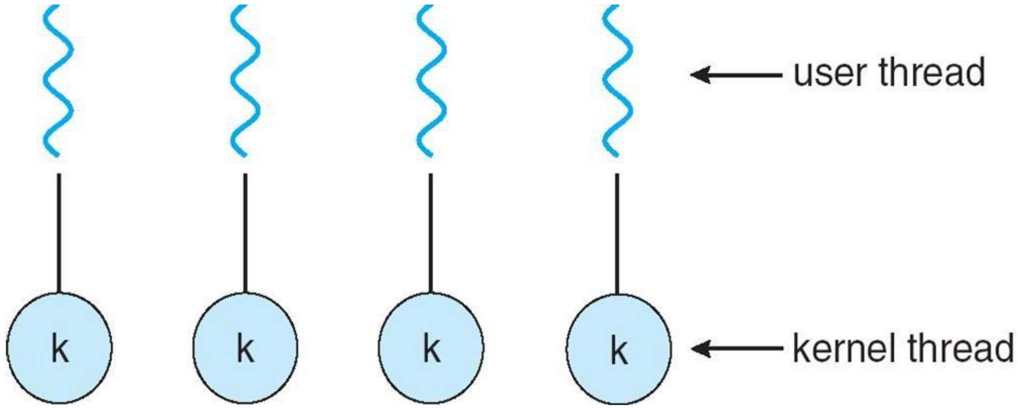
ManytoOne Model

- All userlevel threads of one process mapped to a single kernel level thread
- Thread management in user space
  - –Efficient
  - –Application can run its own scheduler implementation
- One thread can access the kernel at a time
  - –Limited concurrency, limited parallelism
- Examples
  - –“Green threads” (e.g. Solaris)
  - –Gnu Portable Threads



OnetoOne Model

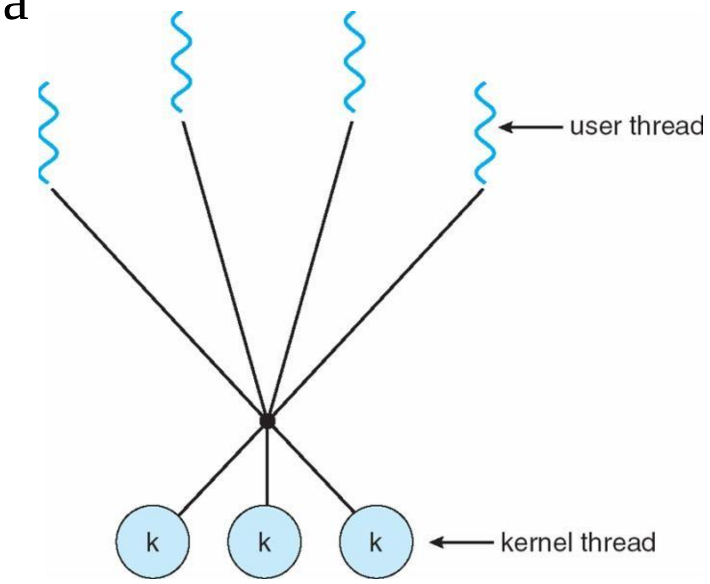
- Each userlevel thread mapped to a kernel thread
- One blocking thread does not block other threads
- Multiple threads access kernel concurrently



- Problem
  - Creating a userlevel thread requires creation of corresponding kernel thread
  - Kernel may restrict the number of threads created
- Example systems
  - Windows, Linux, Solaris 9 (and later), Mac OSX

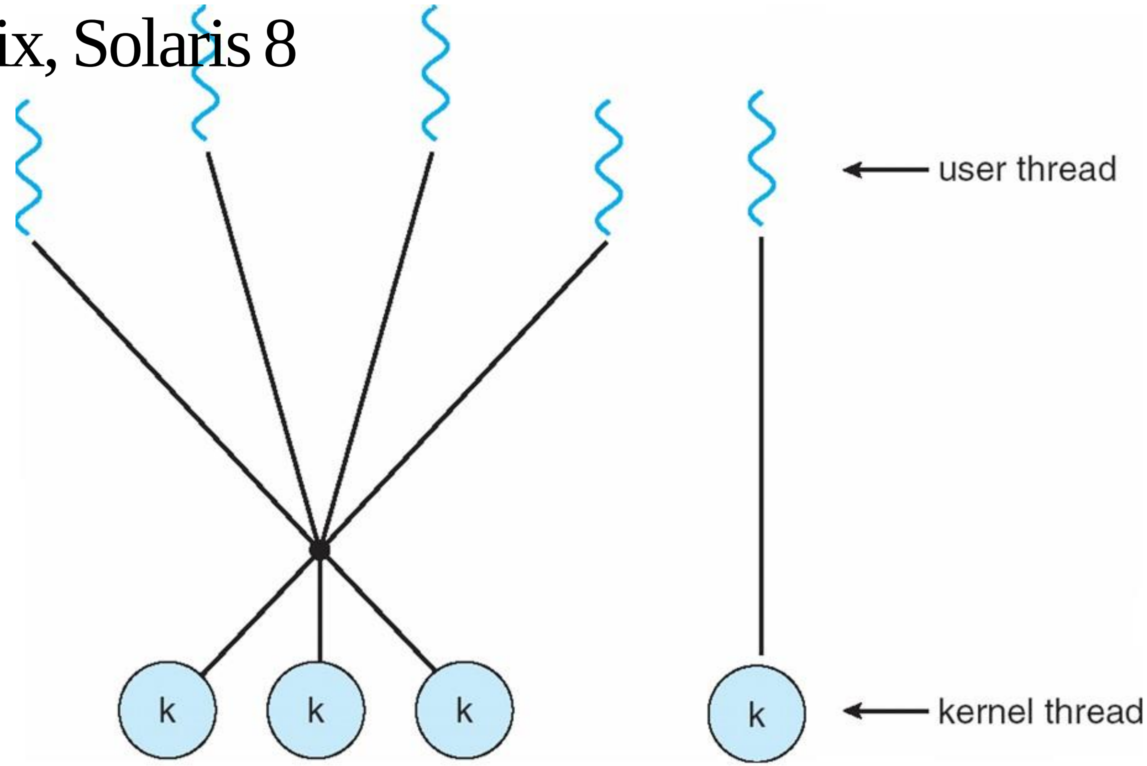
ManytoMany Model

- Many userlevel threads are multiplexed (mapped dynamically) to a smaller or equal number of kernel threads
  - –Thread pool, no fixed binding between a user and a kernel thread
- The number of kernel threads is specific to a particular application or computer system
  - –Application may be allocated more kernel threads on a multiprocessor architecture as on a single processor architecture
- No restriction on userlevel threads
  - –Applications can be designed with as many user level threads as needed
  - –Threads are then mapped dynamically onto a smaller set of currently available kernel threads for execution



Twolevel Model

- Is a variant of the ManytoMany model, allows a fixed relationship between a user thread and a kernel thread
- Was used in older Unixlike systems
  - –IRIX, HPUX, True64 Unix, Solaris 8



Threading Issues – Thread Pools

- Threads come with some overhead
- Unlimited thread creation may exhaust memory and CPU
- Solution
  - Thread pool: create a number of threads at system startup and put them in a pool, from where they will be allocated
  - When an application needs to spawn a thread, an allocated thread is taken from the pool and adapted to the application's needs
- Advantage
  - Usually faster to service a request with already instantiated thread then creating a new one
  - Allows number of threads in applications to be bound by thread pool size
- Number of preallocated threads in pool may depend on
  - Number of CPUs, memory size
  - Expected number of concurrent requests

Threading Issues – fork() and exec()

- Semantics of fork() and exec() changes in a multithreaded program
  - Remember:
    - fork() creates an identical copy of the calling process
  - In case of a multithreaded program
    - Should the new process duplicate all threads?
    - Or should the new process be created with only one thread?
      - If after fork(), the new process calls exec() to start a new program within the created process image, only one thread may be sufficient
  - Solution: some Unix systems implement two versions of fork()

Thread Programming

- POSIX standard threads: pthreads
- Describes an API for creating and managing threads
- There is at least one thread that is created by executing main()
- Other threads are spawned / created from this initial thread

POSIX Thread Programming

- Thread creation

```
pthread_create ( thread, attr, start_routine, arg )
```

  - Returns a new thread ID with parameter "thread"
  - Executes the routine specified by "start_routine" with argument specified by "arg"

- Thread termination

```
pthread_exit ( status )
```

  - Terminates the thread, sends "status" to any thread waiting by calling pthread_join()

POSIX Thread Programming

- Thread synchronisation

```
pthread_join ( threadid, status)
```

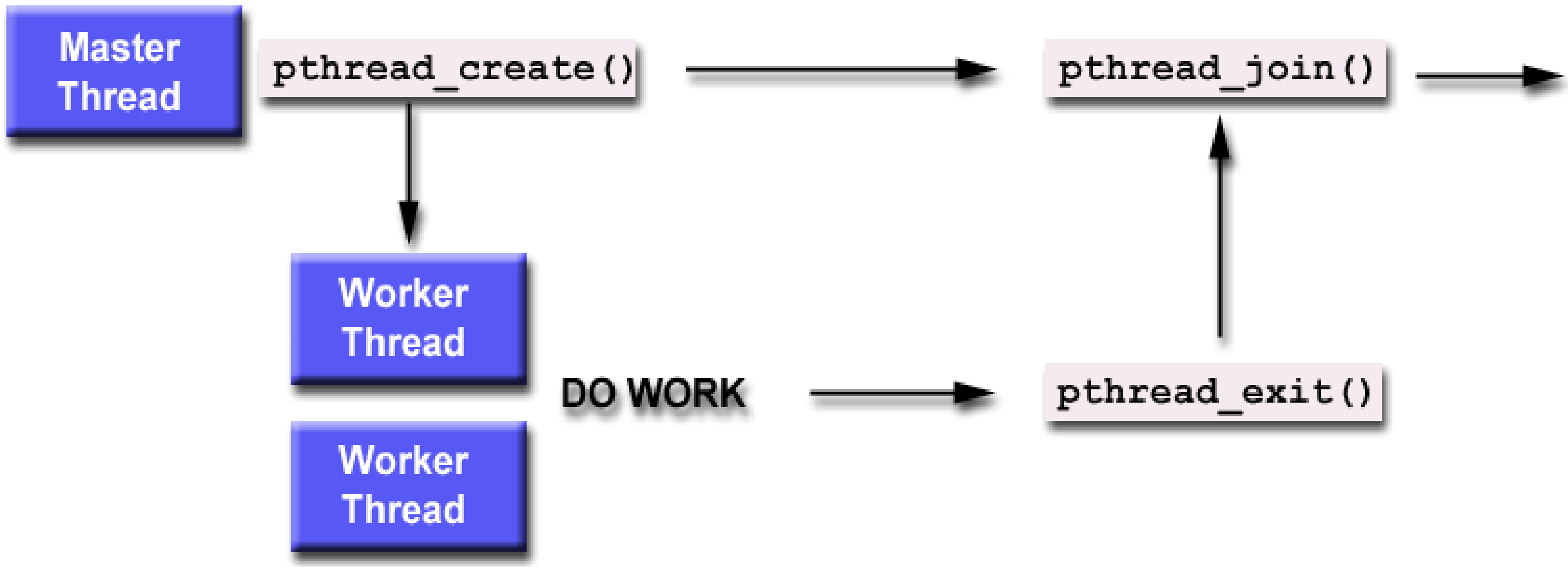
  - Blocks the calling thread until the thread specified by "threadid" terminates
  - The argument "status" passes on the return status of pthread_exit(), called by the thread specified by "threadid"

- Thread yield

```
pthread_yield ( )
```

  - Calling thread gives up the CPU and enters the Ready queue

Thread Programming



Implementing threads

- Thread_fork(func, args)
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)

Thread Context Switch

- Voluntary
  - Thread_yield
  - Thread_join (if child is not done yet)
- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority

Voluntary thread context switch

- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return
- Exactly the same with kernel threads or user threads

MULTICORE AND MULTI-THREADING

- **Performance of software on Multicore**
  - Effective exploitation of parallel resources
  - Amdahl's law states that:

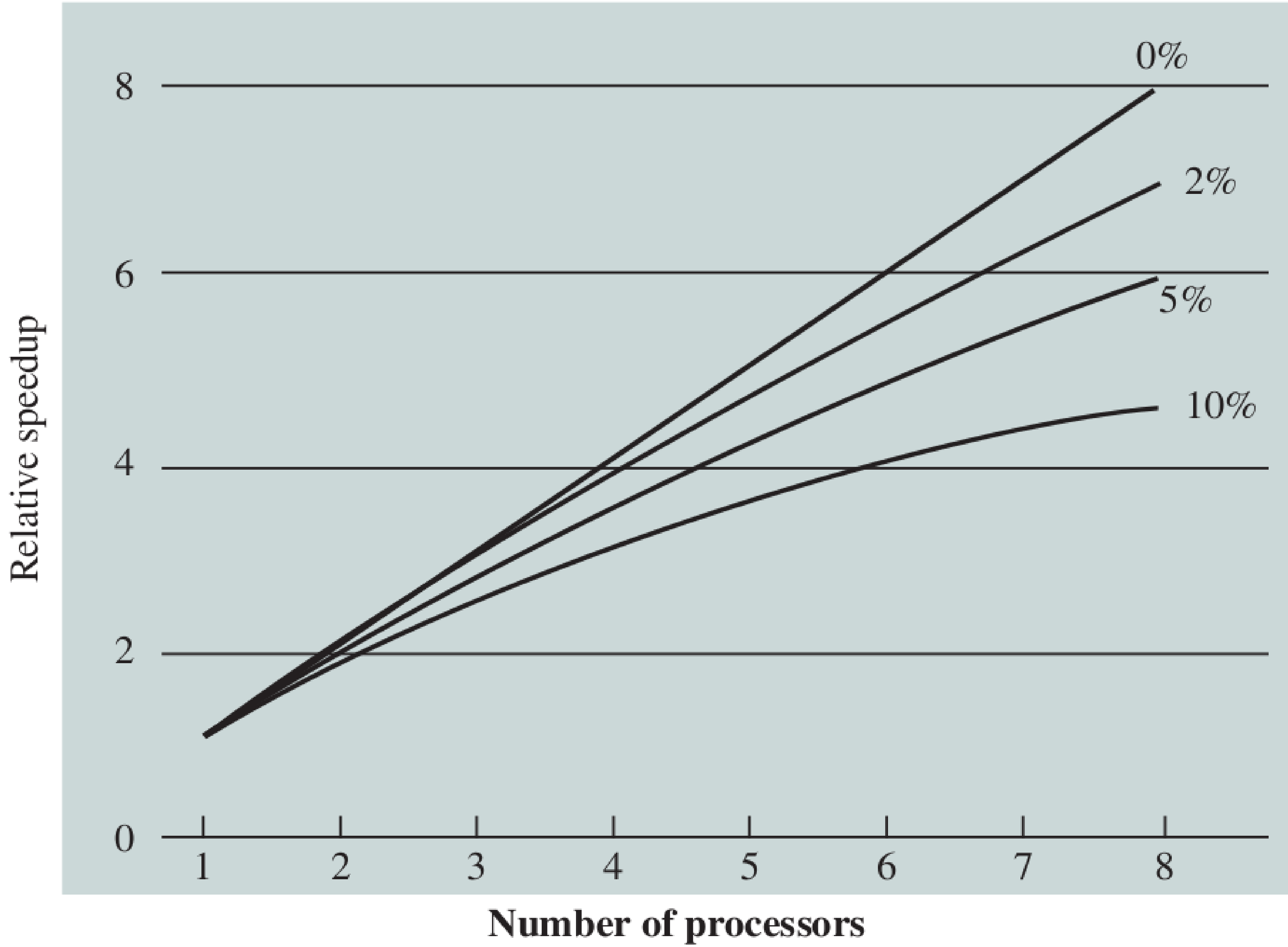
$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} = \frac{1}{(1-f) + \dfrac{f}{N}}$$

  - **(1-f)** : Inherently serial code
  - **f** : infinitely parallelizable code with no scheduling overhead

MULTICORE AND MULTI-THREADING

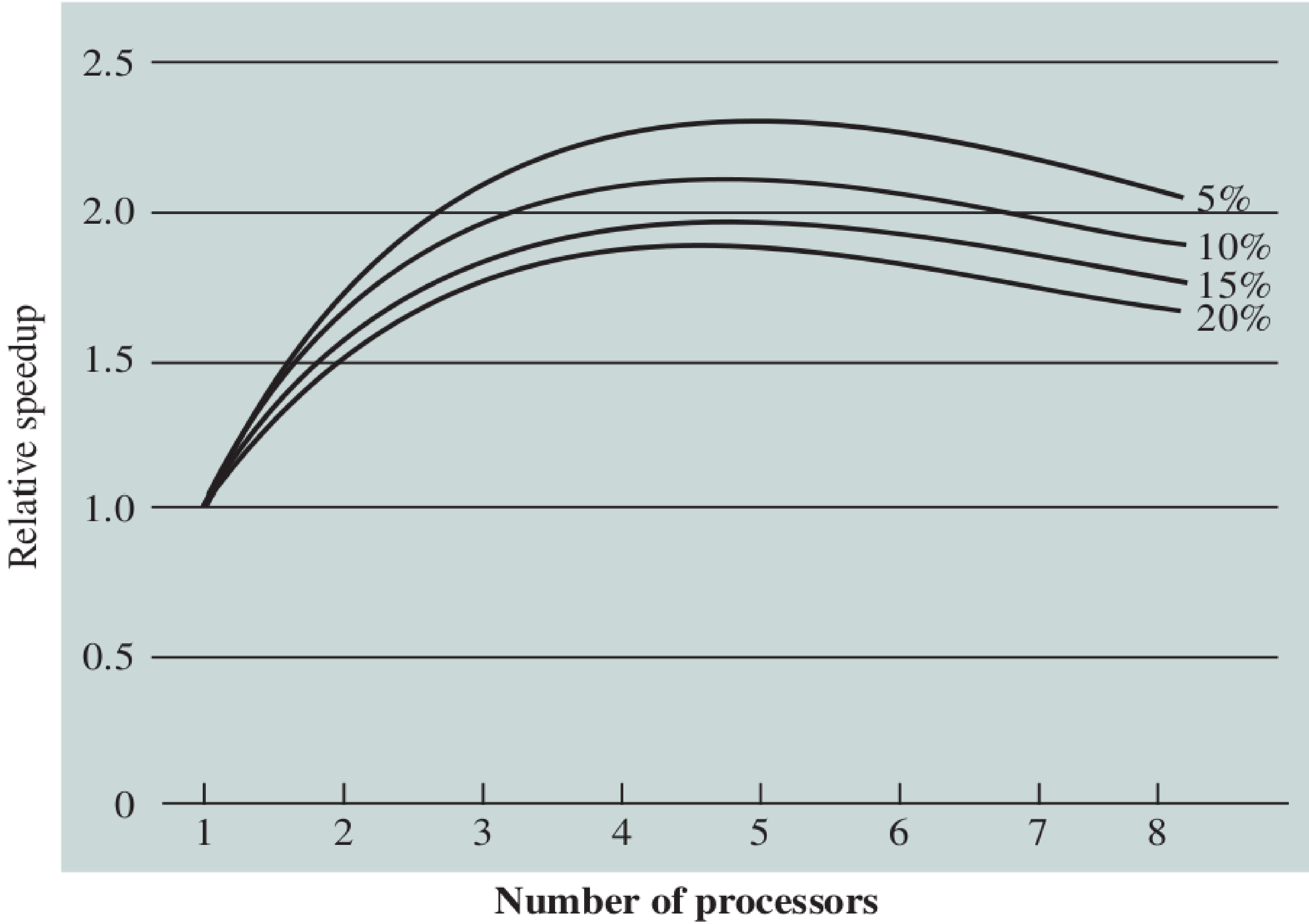
- **Amdahl’s law makes the multicore organizations look attractive!**
  - But even a small amount of serial code has noticeable impact on the overall performance
  - **Example**: 10% serial, 90% parallel, 8 CPUs →~4.7x speedup
- **Other overheads include communication and cache coherence**

MULTICORE AND MULTI-THREADING



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

MULTICORE AND MULTI-THREADING



(b) Speedup with overheads