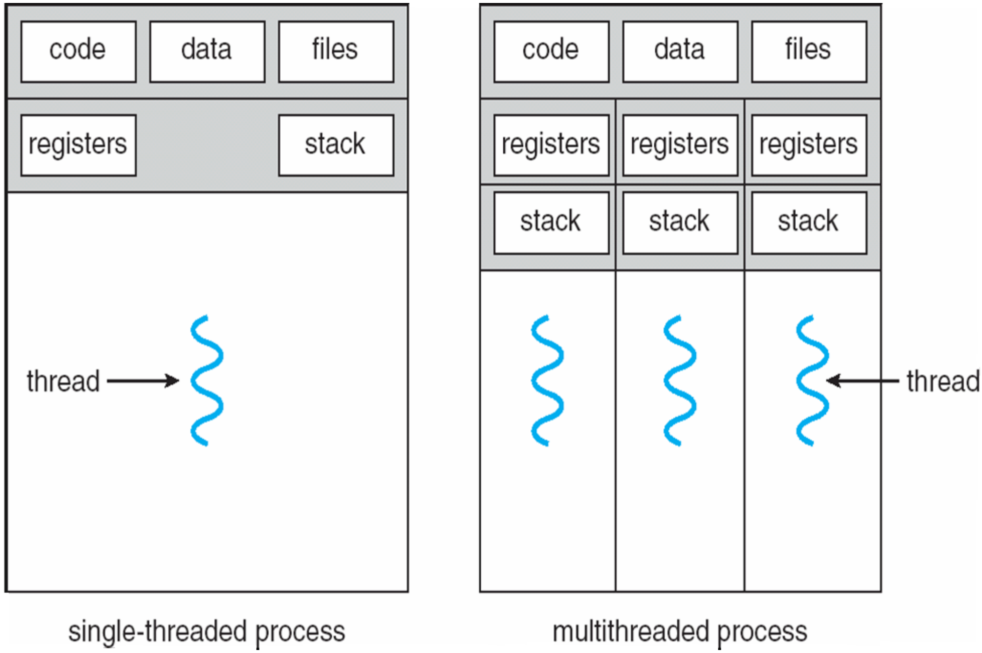
Threads

CS3026 Operating Systems

Lecture 06

Multithreading

- Multithreading is the ability of an operating system to support multiple threads of execution within a single process
- Processes have at least one thread of control
  - Is the CPU context, when process is dispatched for execution

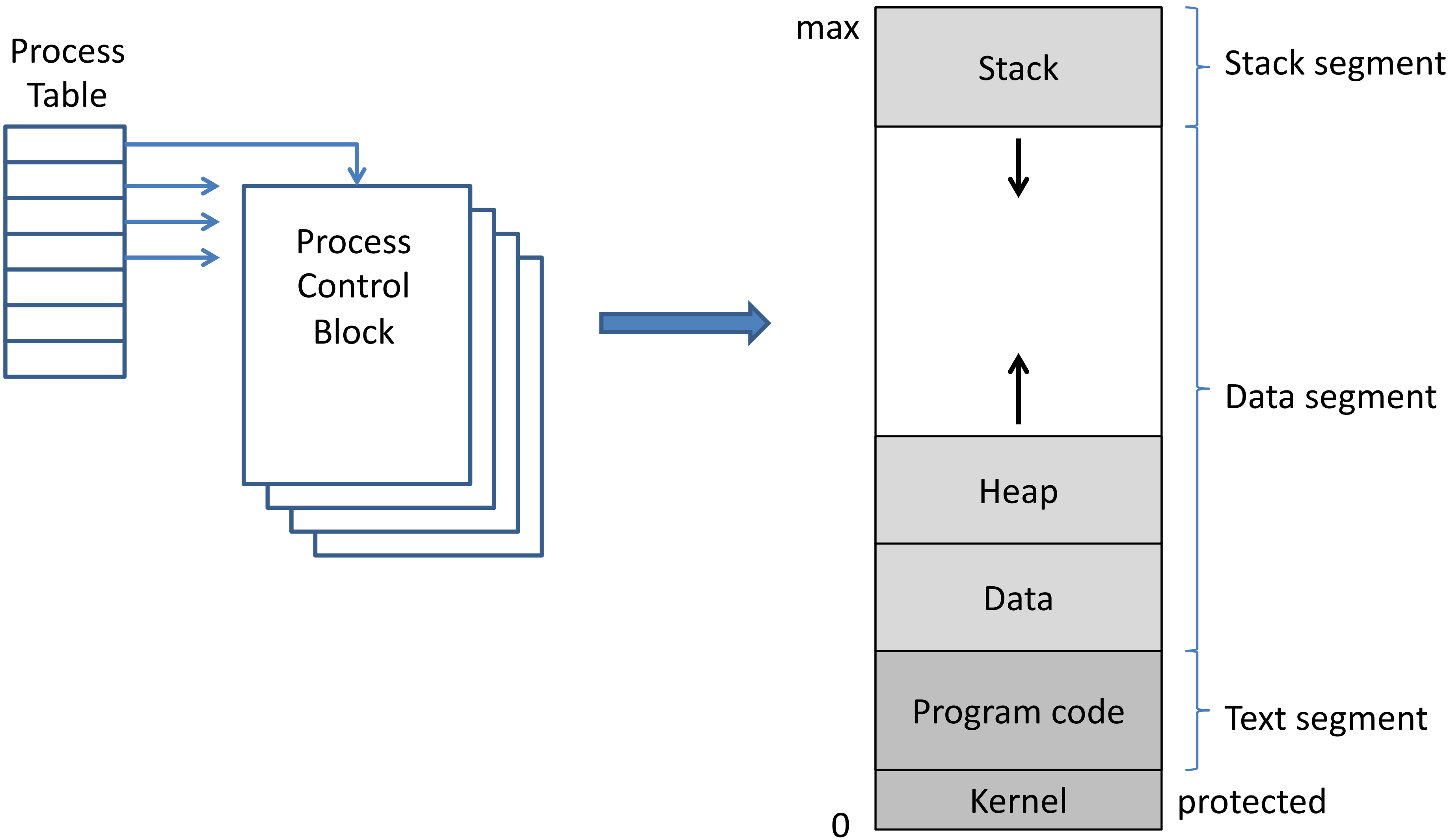


- Multiple threads run in the same address space, share the same memory areas
  - The creation of a thread only creates a new thread control structure, not a separate process image

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
    - CPU may have multiple cores, true parallel execution of threads

Process in Unix



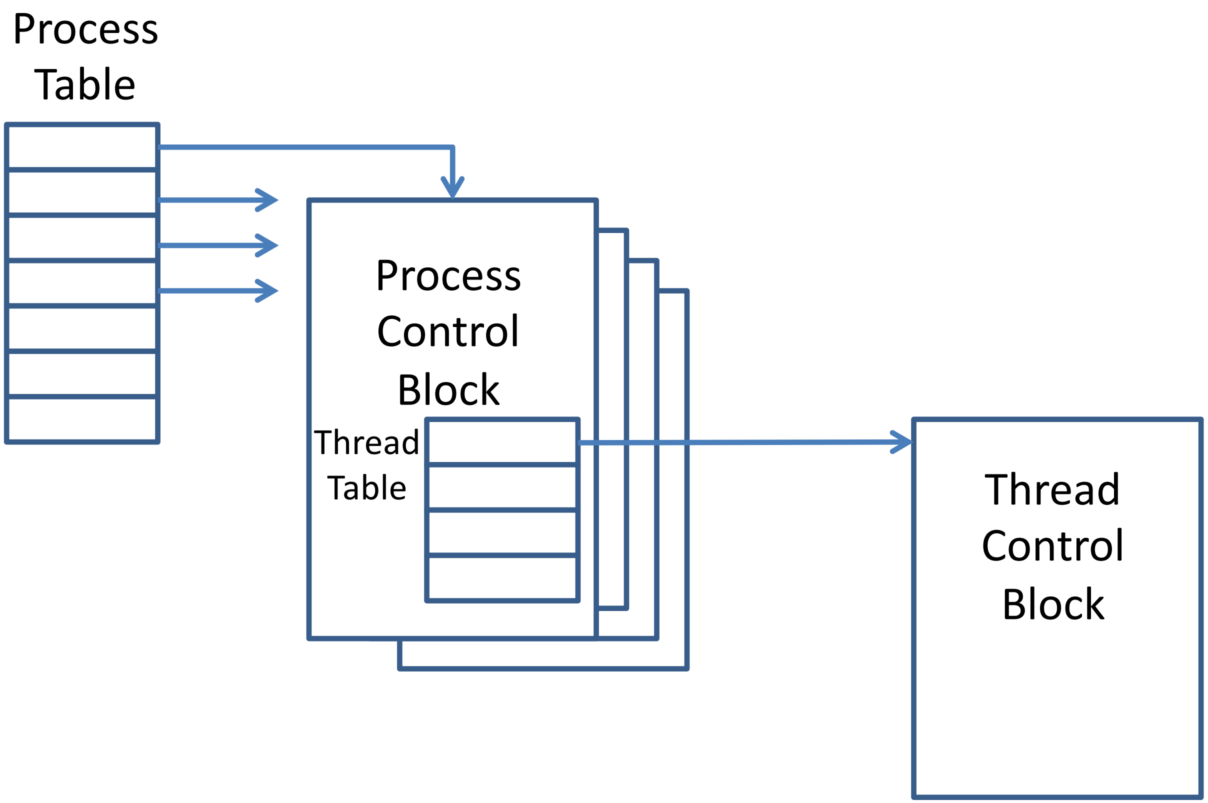
Process – Unit of Resource Ownership

- 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 – Units of Dispatch

- Thread is defined as the **unit of dispatching**:
  - Represents a single thread of execution within a process
- Threads are also called “lightweight processes”
  - Operating system may be able to manage multiple threads of execution directly (e.g. Linux tasks)
- A thread is provided with its own register context and stack space
- 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

Threads in Unix



- It depends on the actual Kernel implementation how threads are managed
- Are threads implemented at User level only or also at kernel level?

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
    - 10-times faster than process creation in Unix
  - Less time to terminate a thread
  - Much faster to switch between threads than to 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

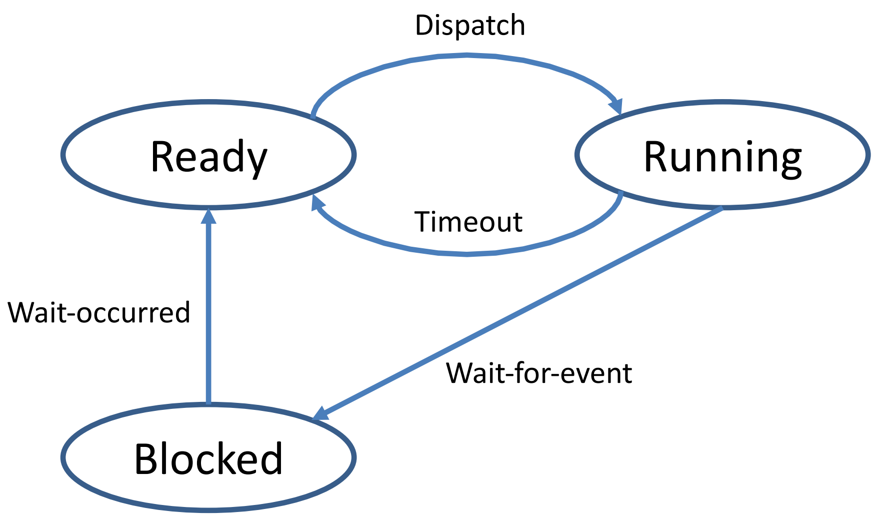
Thread Management

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 also three basic states
  - Running: CPU executes thread
  - Ready: thread control block is placed in Ready queue
  - Blocked: thread awaits event
- 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 operations 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

User and Kernel Threads

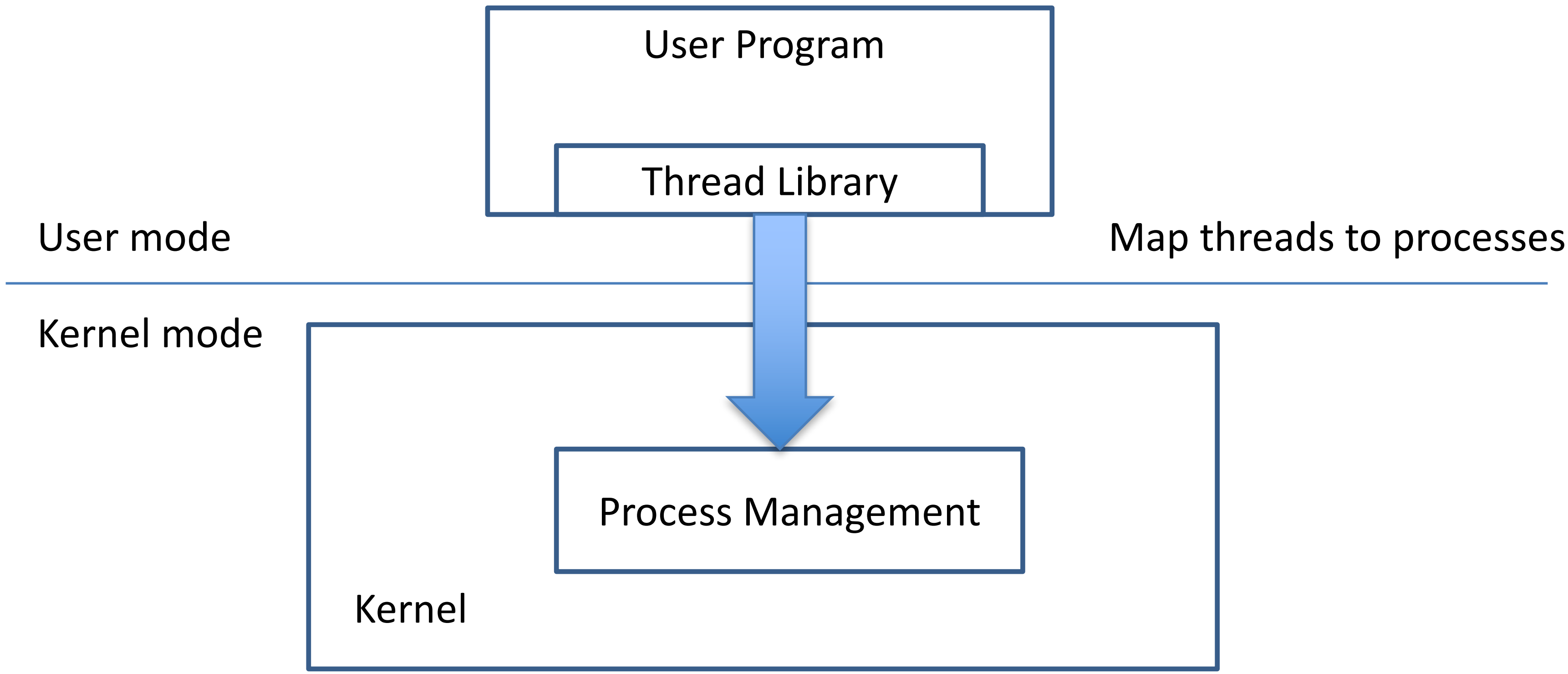
Thread Management

- If one thread blocks
  - Is the whole process with all other threads blocked?
  - Or is only this single thread blocked?
- This depends on the implementation of the kernel
  - The kernel may only know processes, threads have to be managed by the program itself
  - The kernel knows threads, threads in a program can be directly managed by kernel

Thread Implementation

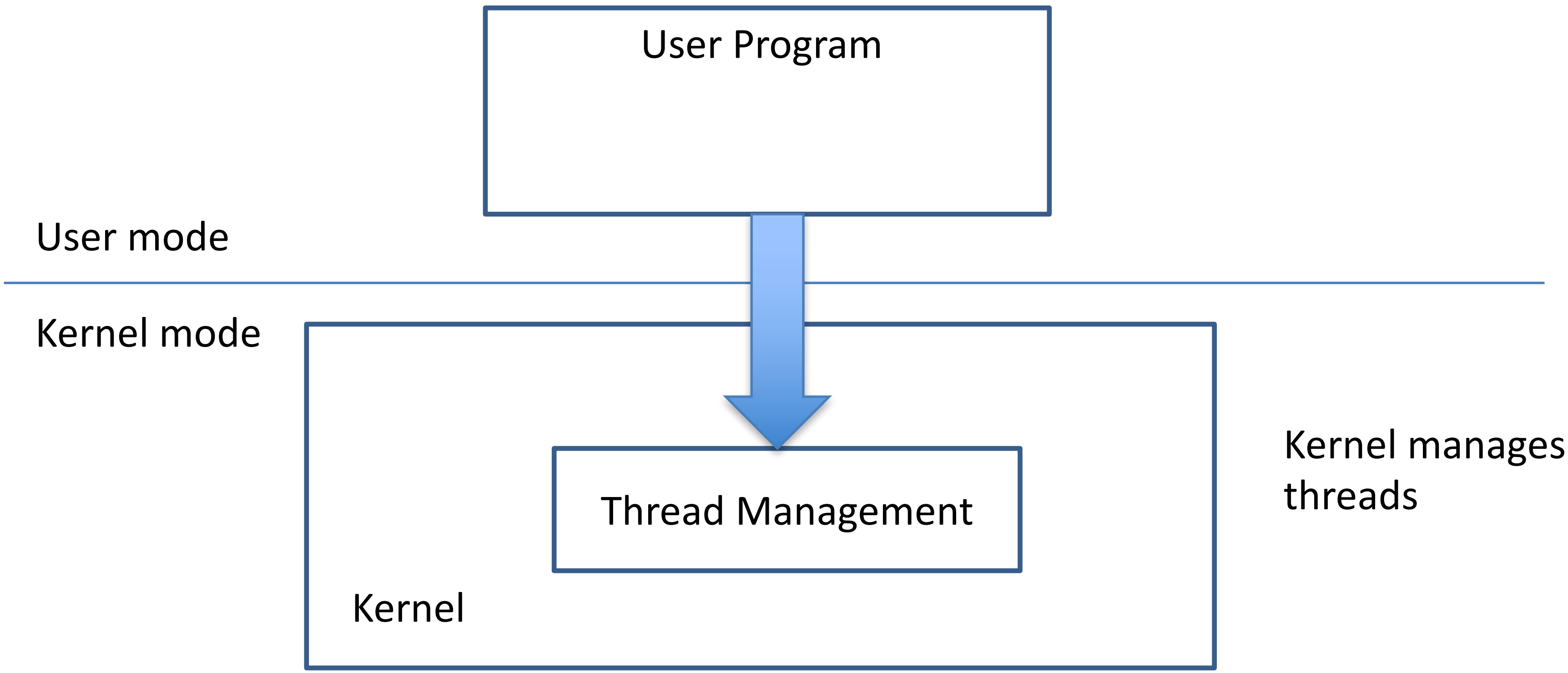
- Three categories of threads
  - User-level threads
  - Kernel-level threads
  - Mixed user-kernel-level threads
- Characterised by the extent of the kernel being involved in their management

User Threads



- Operating System unaware of threads, completely controlled by User program

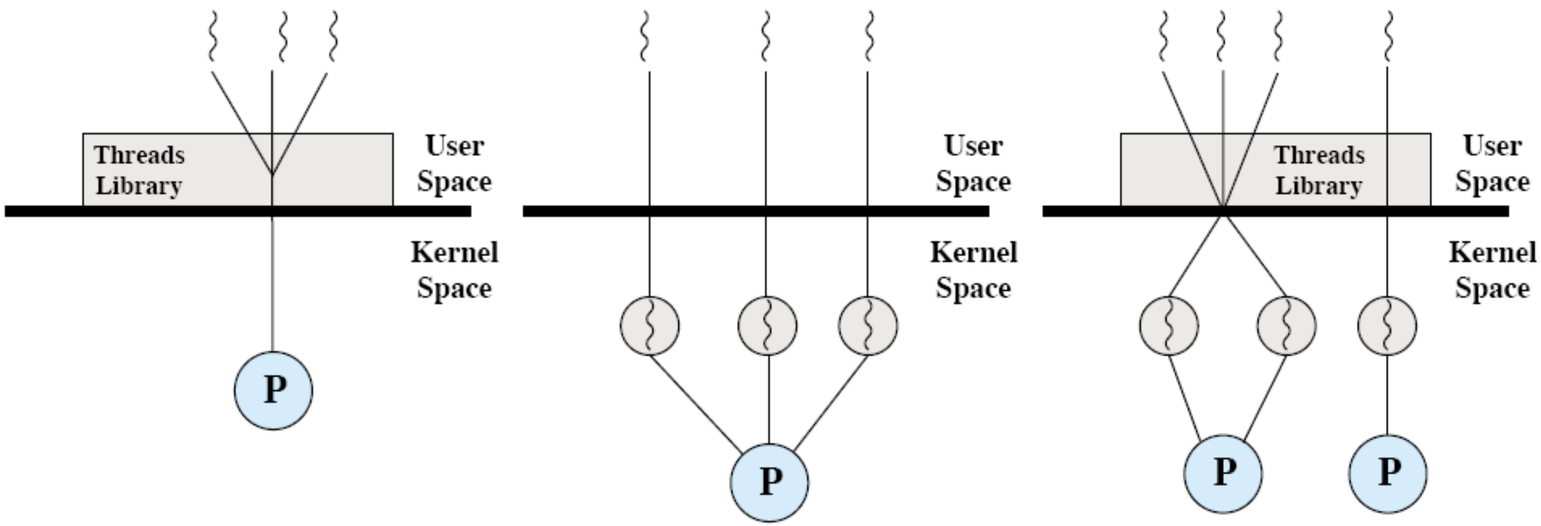
Kernel Threads



- Operating System manages threads, better mapping of threads to multiple CPU cores

Thread Implementation

- Pure User Level Threads
- Pure Kernel Level Threads
- Combined User / Kernel level threads



Pure User-Level
ULT

Pure Kernel-Level
KLT

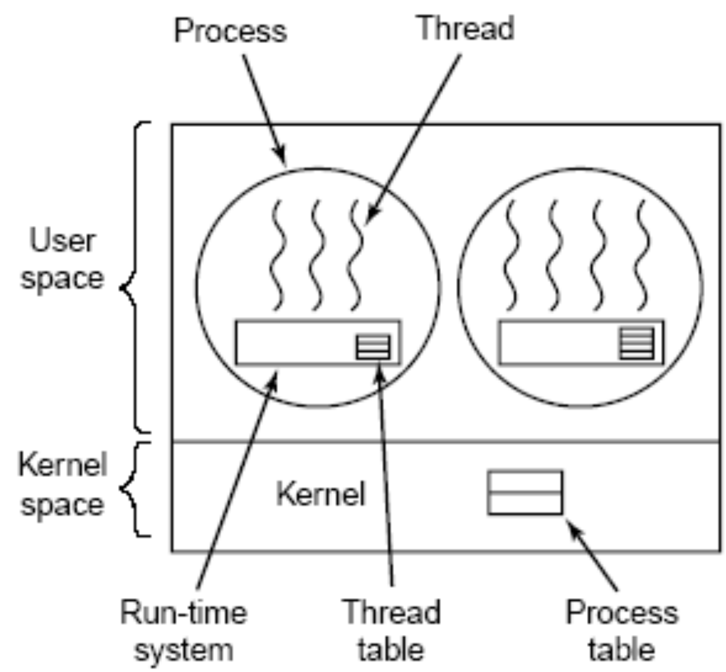
CombinedLevel
ULT/KLT

User-Level Threads

- User-Level 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)

User-Level 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 Multi-processor 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

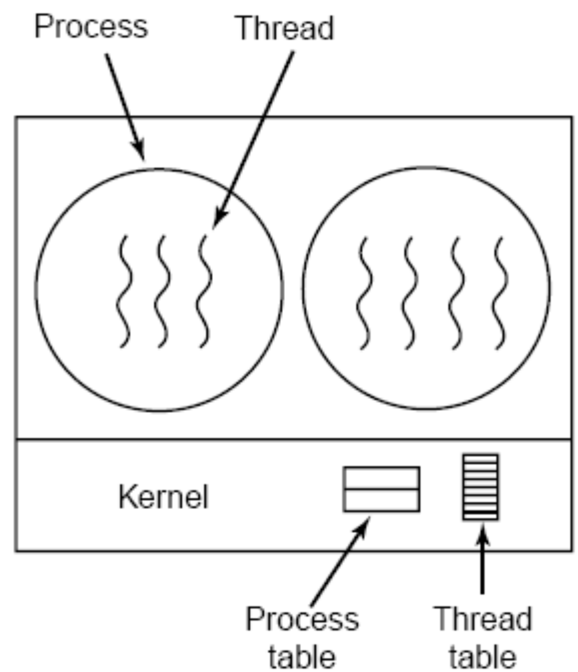


Kernel-level Threads

- Thread is **Unit of dispatch**
  - Kernel is aware of the existence of threads
  - Kernel manages each thread separately
- Benefit
  - Fine-grain 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

Kernel-Level Threads: Disadvantage

- A context switch between threads of the same process involves kernel
  - Mode switch: we have to switch CPU into Kernel mode
- Mode switch is costly
  - 2 mode switches for each thread context switch, is as costly as process switch

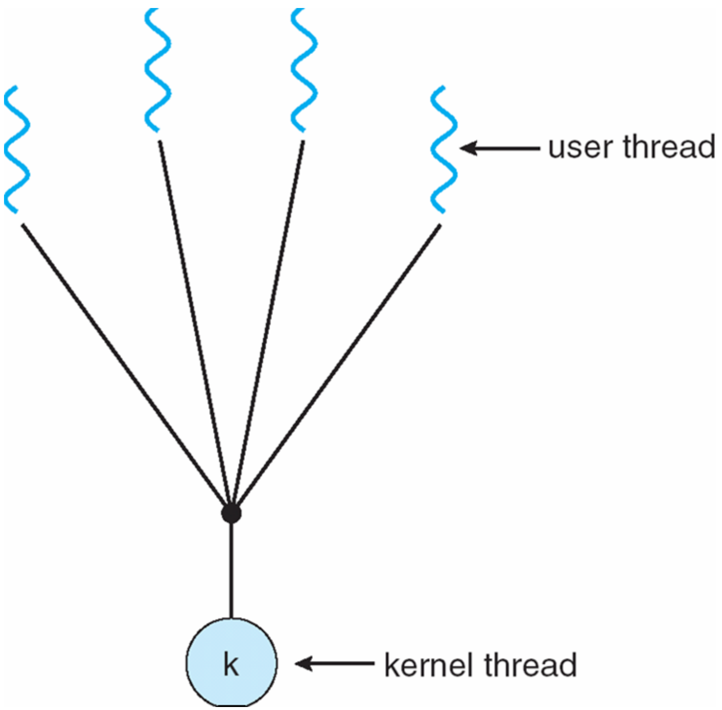


Hybrid Implementations User-Kernel-Level Threads

- Try to combine advantages of both user-level and kernel-level threads
  - User-level: light-weight thread switching
  - Kernel-level: allows dispatch at thread level (same or different process), when one thread blocks
  - true parallelism of threads in multiprocessor systems possible
- Basic technique: Mapping of user-level threads onto a limited set of kernel threads
- Different hybrid Multithreading Models:
  - Many-to-one
  - One-to-one
  - Many-to-many

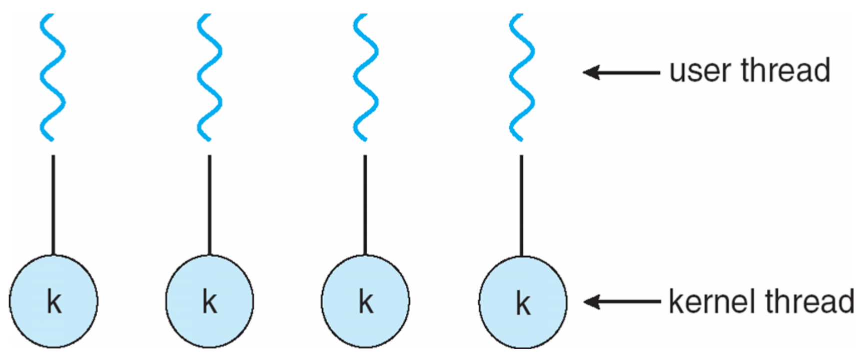
Many-to-One Model

- All user-level threads of one process mapped to a single kernel-level thread / process
- These are the User-level threads as discussed before
- 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



One-to-One Model

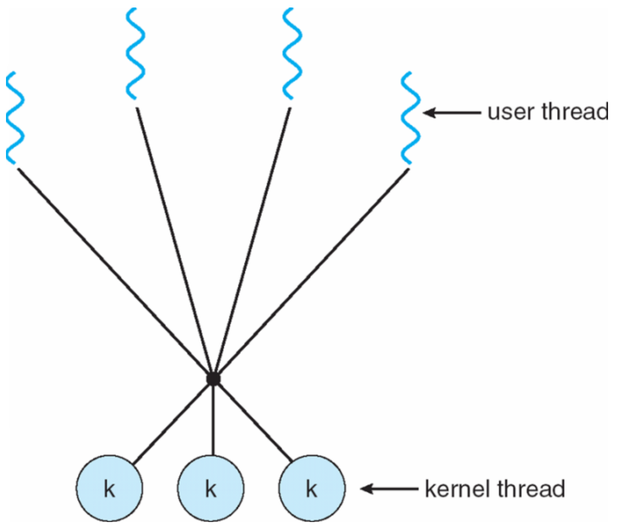
- Each user-level thread mapped to a kernel thread
- These are the Kernel-level threads as discussed before
- One blocking thread does not block other threads
- Multiple threads access kernel concurrently



- Problem
  – Kernel may restrict the number of threads created
- Example systems
  – Windows, Linux, Solaris 9 (and later), Mac OSX

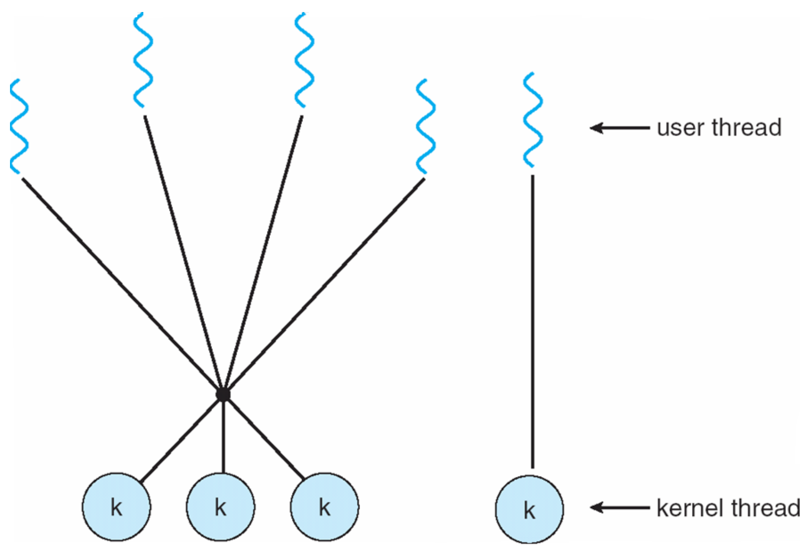
Many-to-Many Model

- Many user-level 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 user-level 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



Two-level Model

- Is a variant of the Many-to-Many model, allows a fixed relationship between a user thread and a kernel thread
- Was used in older Unix-like systems
  - IRIX, HP-UX, 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 pre-allocated 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()

Threading Issues – Signal Handling

- Signals are used in Unix systems to notify a process about the occurrence of an event
  - E.g.: CTRL-C terminates a program
- All signals follow the same pattern
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Once delivered, a signal must be handled
- In multithreaded systems, there are 4 options
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Thread Programming

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)
```

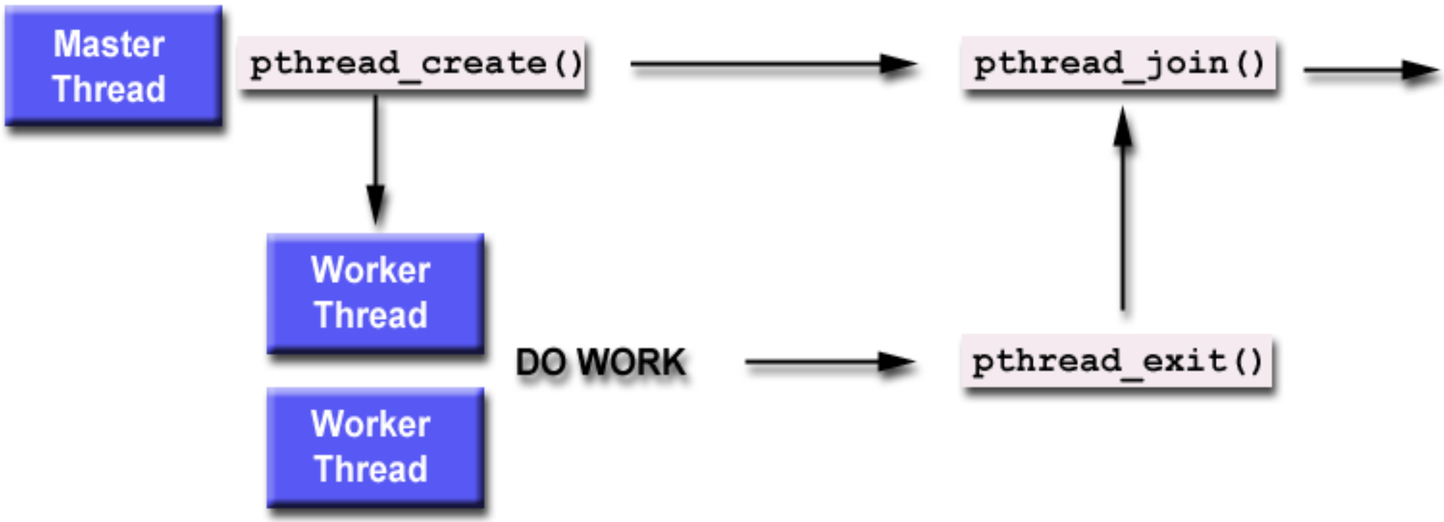
  - Blocks the calling thread until the thread specified by "threadid" terminates
  - The argument "status" passes on the return status of pthread_exit(), called by the thread specified by "threadid"

- Thread yield

```
pthread_yield ( )
```

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

Thread Programming
Wait for completion with join()



SMP Support – Processor Affinity

- Processor Affinity: Relationship between a thread and a processor
  - No affinity
    - Threads of any process may run on any processor
    - No guarantee that they are rescheduled on the same processor after interruption
  - Soft affinity:
    - Dispatcher tries to re-assign threads to the same processor after interruption
    - Helps to reuse data still held on processor cache, less cleanup and reload necessary
  - Hard affinity
    - Application restricts execution of a thread to a particular processor / processor core

Linux Threads

- No distinction between processes and threads
- User-level threads are mapped into kernel-level processes (called “tasks”)
- A new process is created by copying the attributes of the current process
- A clone() system call exists
  – Cloned processes share resources, but have separate stacks

Mac OSX Grand Central Dispatch

- Thread pool
- Mac OSX can operate even more fine-grained:
  - Blocks of program code are **Units of dispatch**
    - A software designer can designate portions of program code, called blocks, that may be dispatched independently and run concurrently
    - These are extensions to a programming language
- Concurrency depends on the number of CPU cores available and the thread capacity of a system

Summary

- User-level threads
  - Created and managed by a thread library that runs in the user space of a process
  - No mode switch required to switch from one thread to another
  - Only a single user-level thread within a process can execute at a time, no execution in parallel on a multiprocessor system
  - If one thread blocks, the entire process is blocked
- Kernel-level threads
  - Threads within a process are maintained by the kernel
  - A mode switch is required to switch from one thread to another
  - Multiple threads within the same process can execute in parallel on a multiprocessor system
  - Blocking of one thread does not block the entire process
- Process: resource ownership
- Thread: program execution / dispatching