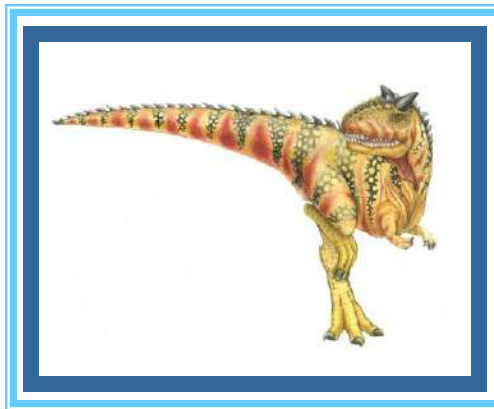


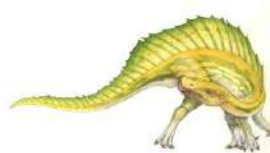
Chapter 3: Processes





Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems





Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
 - The program code, also called **text section**
 - Current activity including **program counter**, processor registers
 - **Stack** containing temporary data
 - ▶ Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time

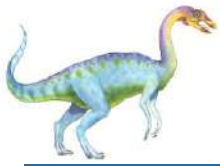




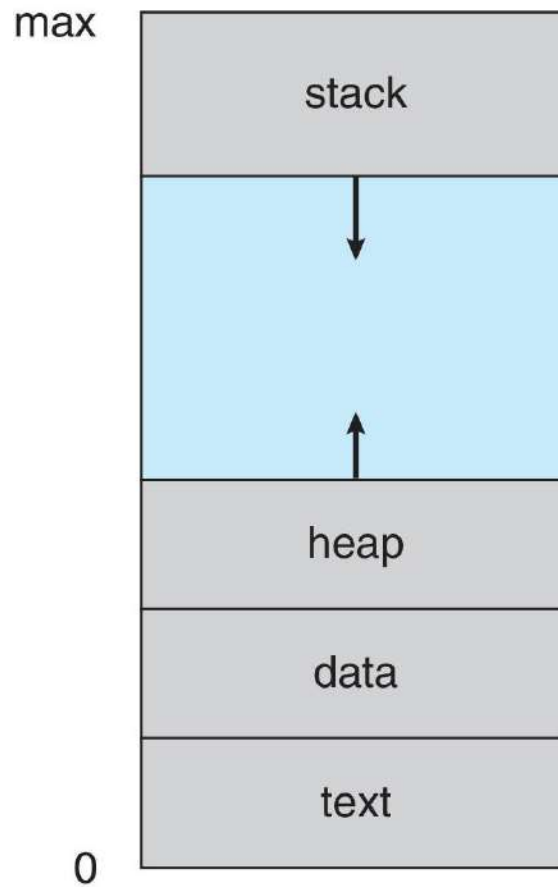
Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**
 - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
 - Consider multiple users executing the same program



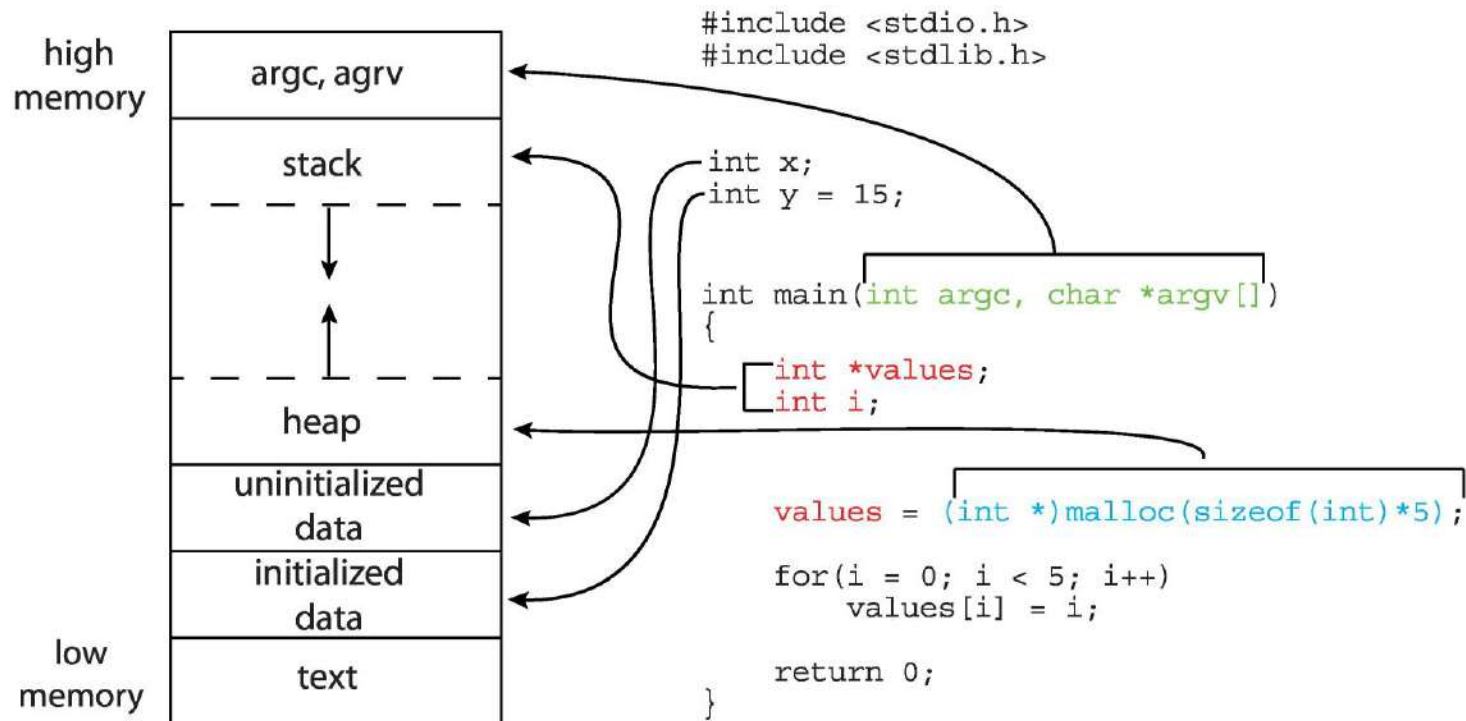


Process in Memory





Memory Layout of a C Program





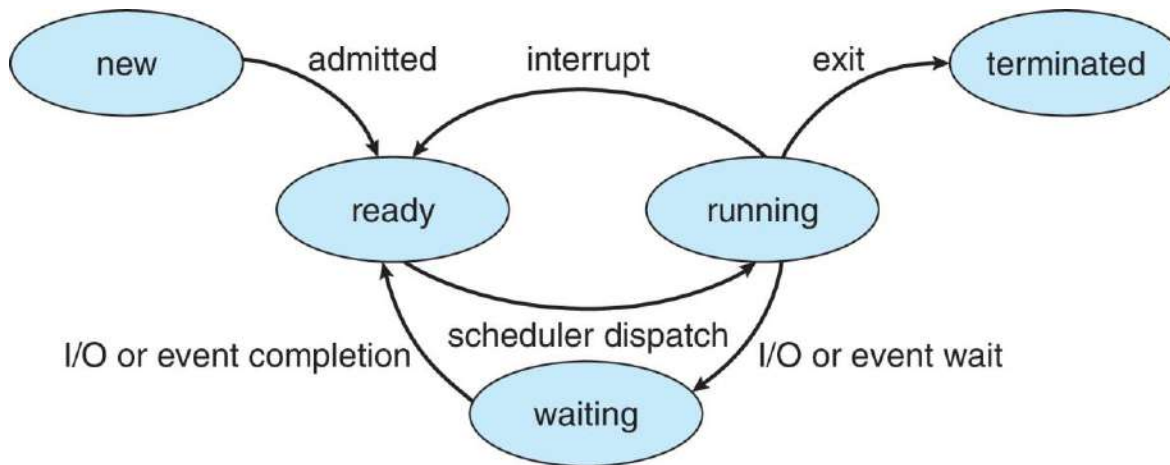
Process State

- As a process executes, it changes **state**
 - **New**: The process is being created
 - **Running**: Instructions are being executed
 - **Waiting**: The process is waiting for some event to occur
 - **Ready**: The process is waiting to be assigned to a processor
 - **Terminated**: The process has finished execution





Diagram of Process State





Process Control Block (PCB)

Information associated with each process(also called **task control block**)

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

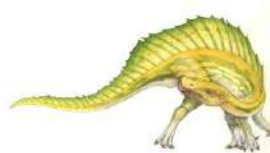
| |
|--------------------|
| process state |
| process number |
| program counter |
| registers |
| memory limits |
| list of open files |
| ... |





Threads

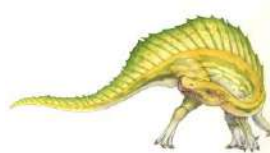
- So far, process has a single thread of execution
- Consider having multiple program counters per process
 - Multiple locations can execute at once
 - ▶ Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4





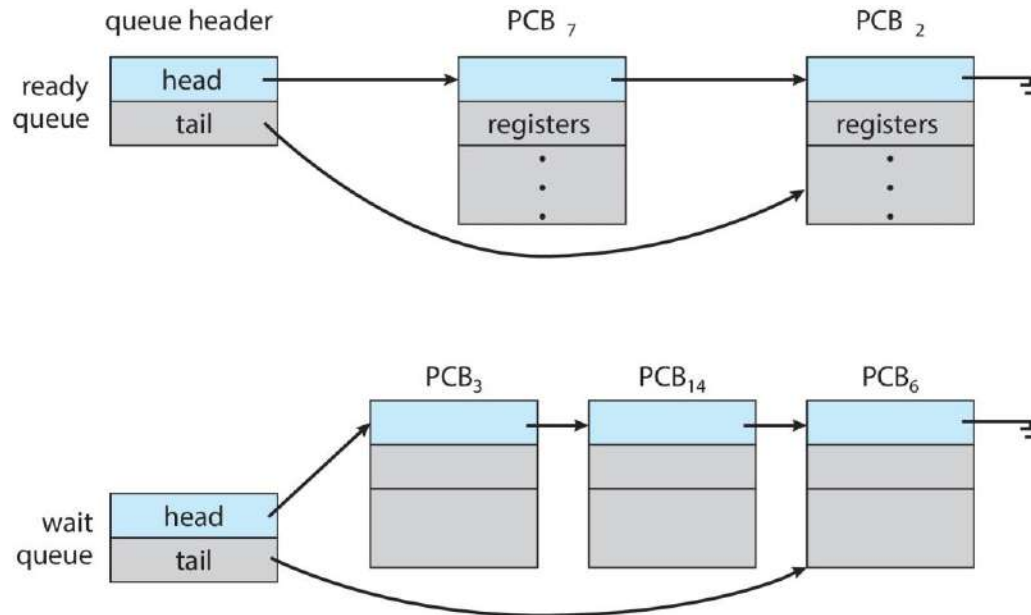
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - **Wait queues** – set of processes waiting for an event (i.e., I/O)
 - Processes migrate among the various queues



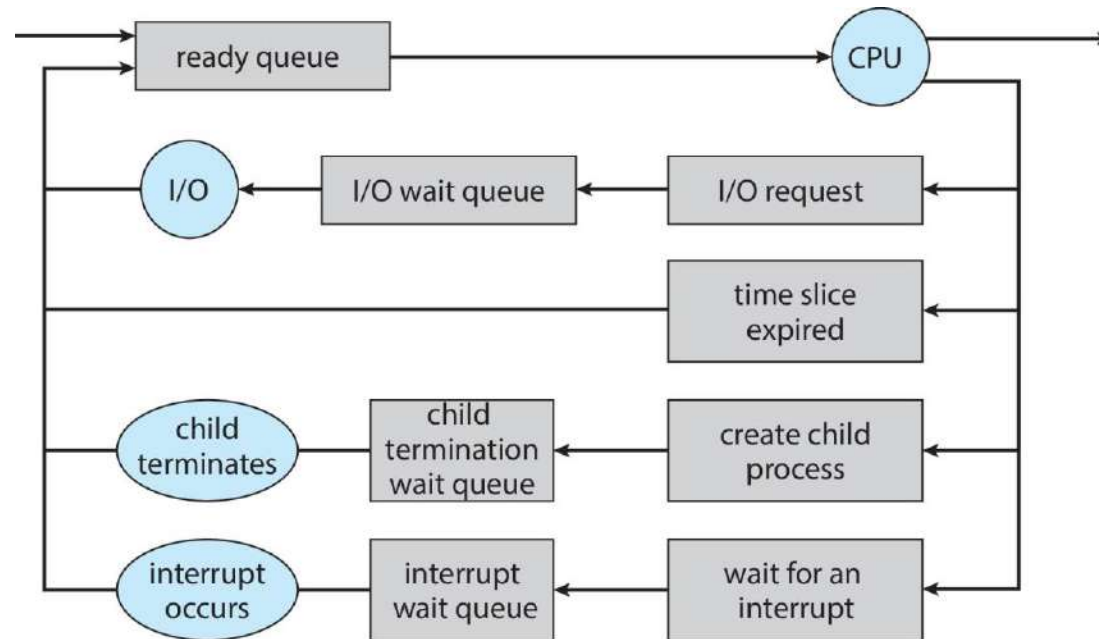


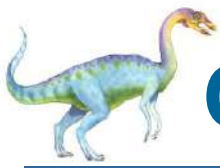
Ready and Wait Queues





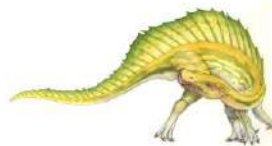
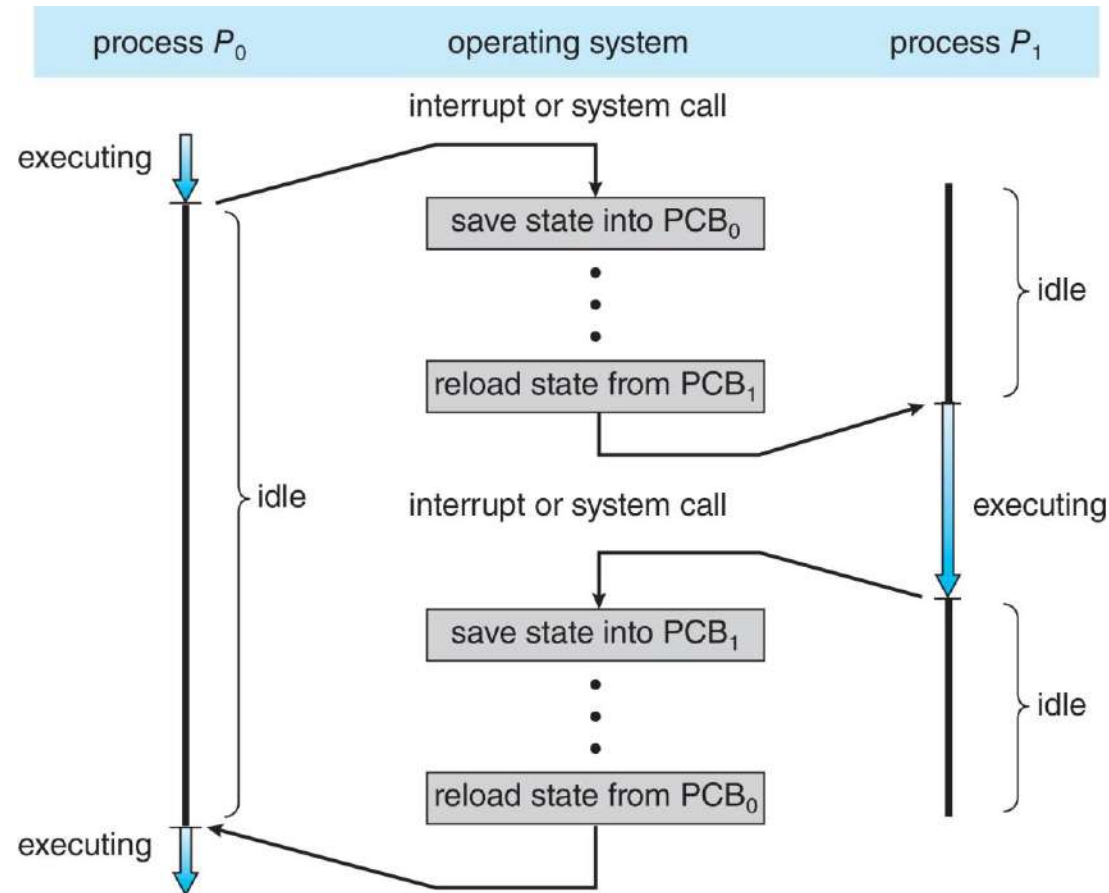
Representation of Process Scheduling





CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.





Context Switch

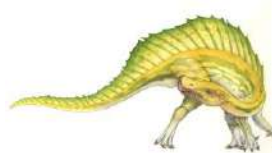
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
 - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once





Multitasking in Mobile Systems

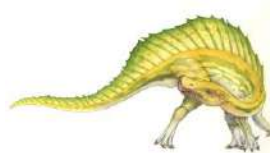
- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
 - Single **foreground** process- controlled via user interface
 - Multiple **background** processes– in memory, running, but not on the display, and with limits
 - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
 - Background process uses a **service** to perform tasks
 - Service can keep running even if background process is suspended
 - Service has no user interface, small memory use





Operations on Processes

- System must provide mechanisms for:
 - Process creation
 - Process termination





Process Creation

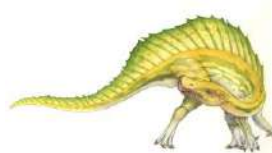
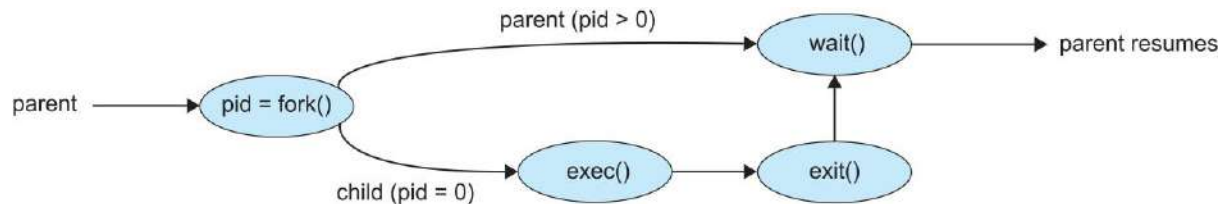
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options
 - Parent and children execute concurrently
 - Parent waits until children terminate





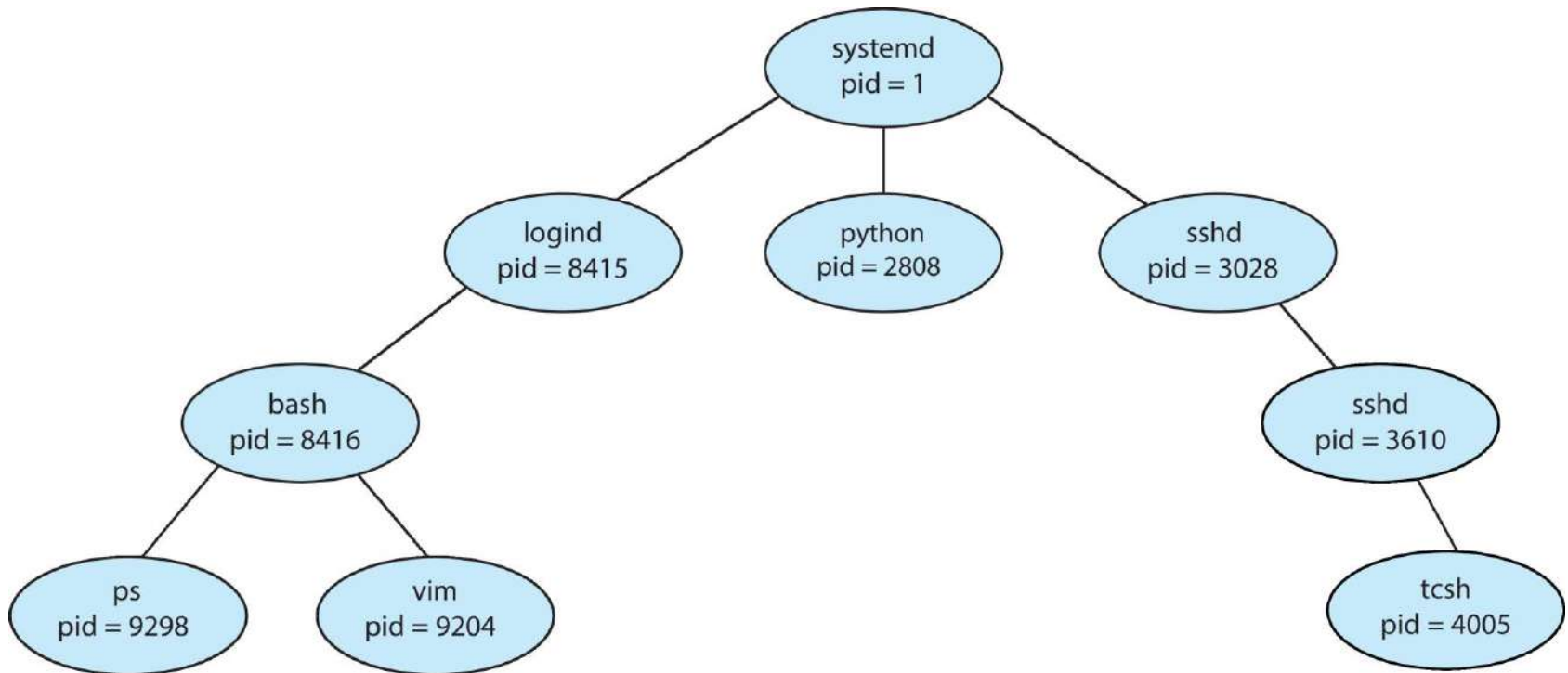
Process Creation (Cont.)

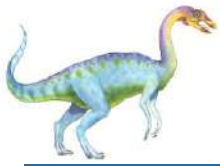
- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - **fork()** system call creates new process
 - **exec()** system call used after a **fork()** to replace the process' memory space with a new program
 - Parent process calls **wait()** waiting for the child to terminate





A Tree of Processes in Linux





Process Termination

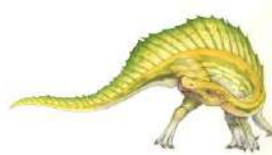
- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
 - Returns status data from child to parent (via **wait()**)
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting, and the operating systems does not allow a child to continue if its parent terminates





Android Process Importance Hierarchy

- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
 - Foreground process
 - Visible process
 - Service process
 - Background process
 - Empty process
- Android will begin terminating processes that are least important.





Interprocess Communication

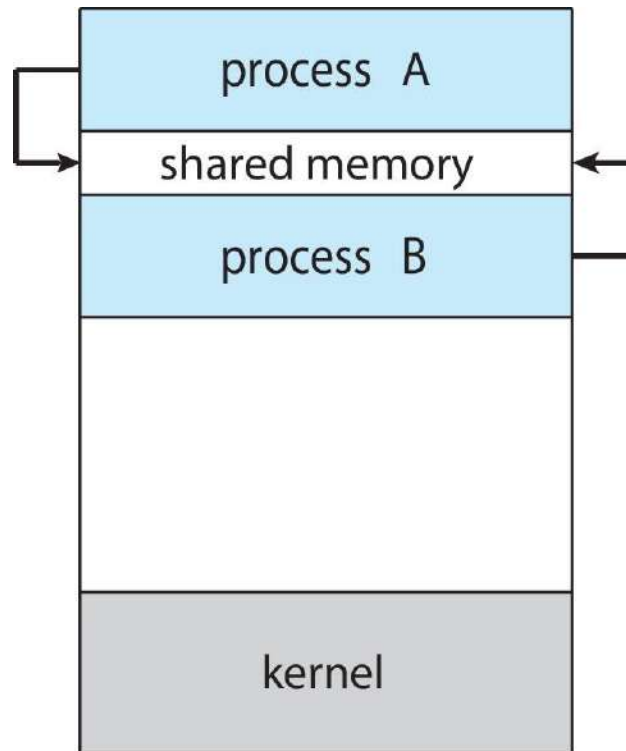
- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 - **Shared memory**
 - **Message passing**





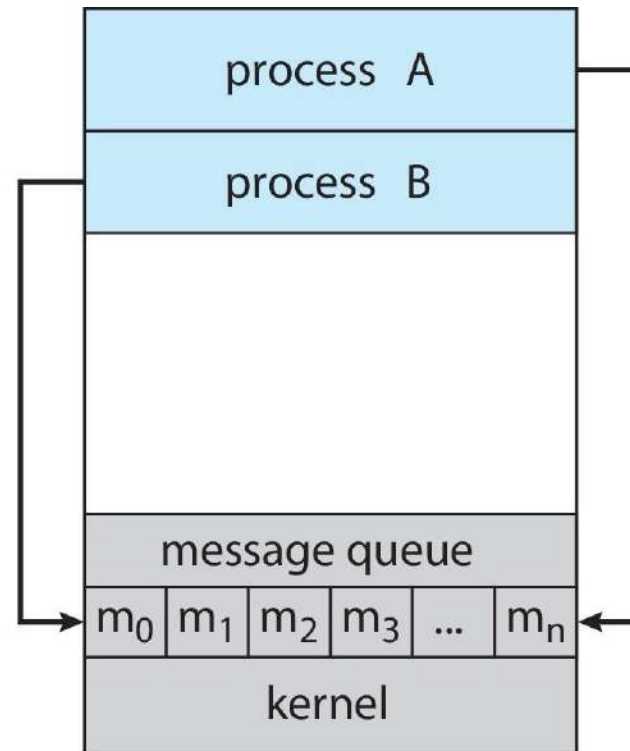
Communications Models

(a) Shared memory.



(a)

(b) Message passing.



(b)





IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.





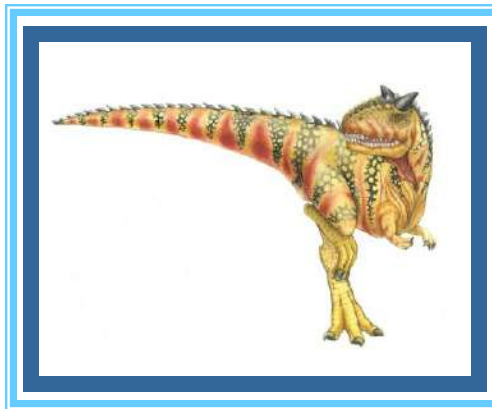
Synchronization

Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
 - **Blocking send** -- the sender is blocked until the message is received
 - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** -- the sender sends the message and continue
 - **Non-blocking receive** -- the receiver receives:
 - ▶ A valid message, or
 - ▶ Null message
- Different combinations possible
 - If both send and receive are blocking, we have a **rendezvous**



Threads

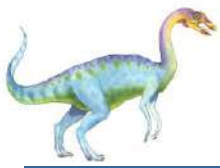




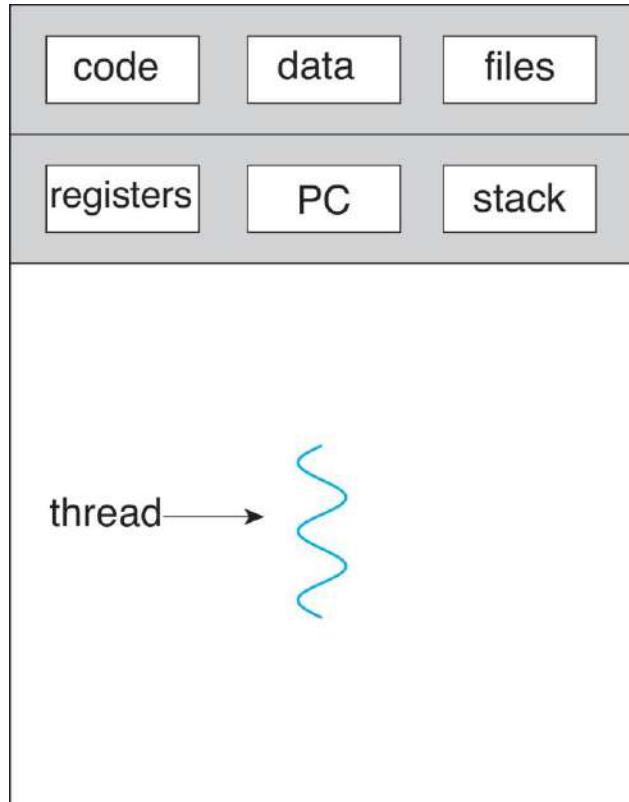
Threads

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

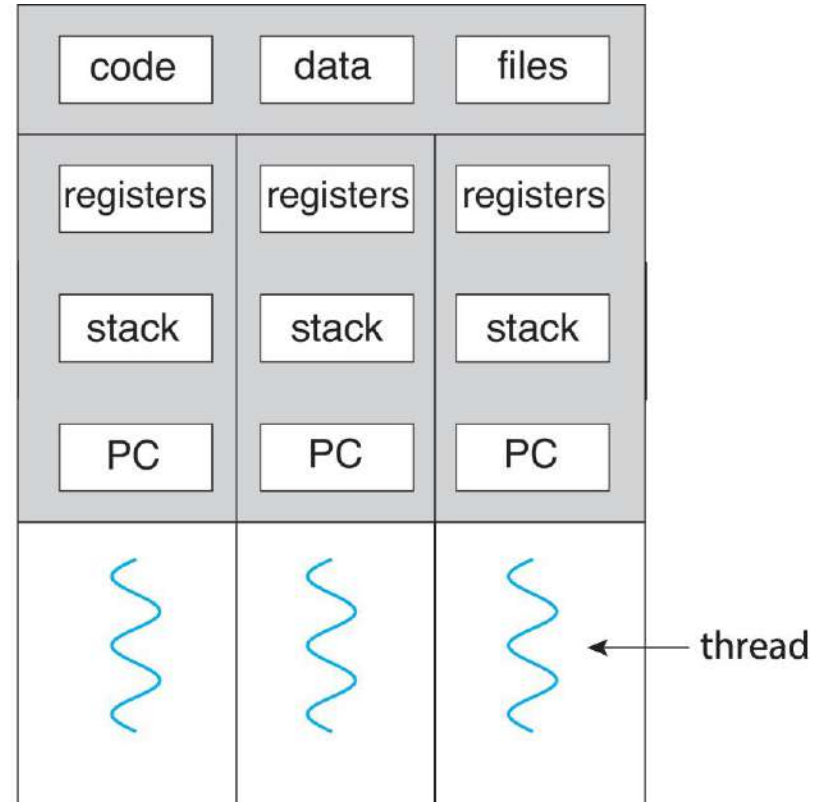




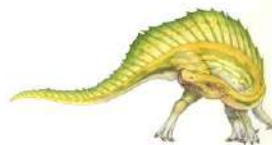
Single and Multithreaded Processes



single-threaded process

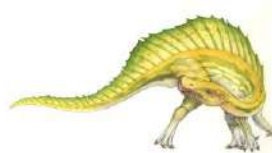
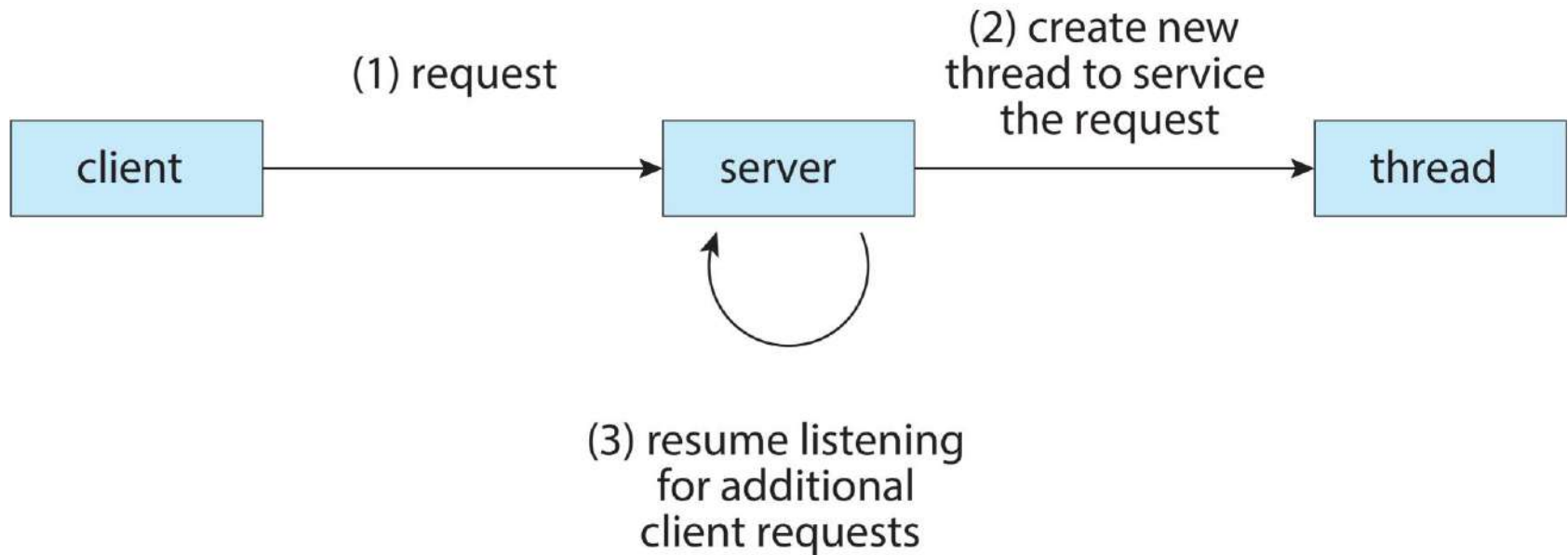


multithreaded process





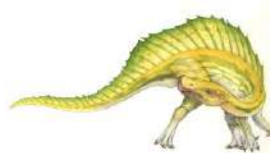
Multithreaded Server Architecture





Benefits

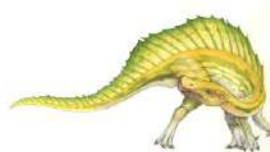
- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower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





Multicore Programming

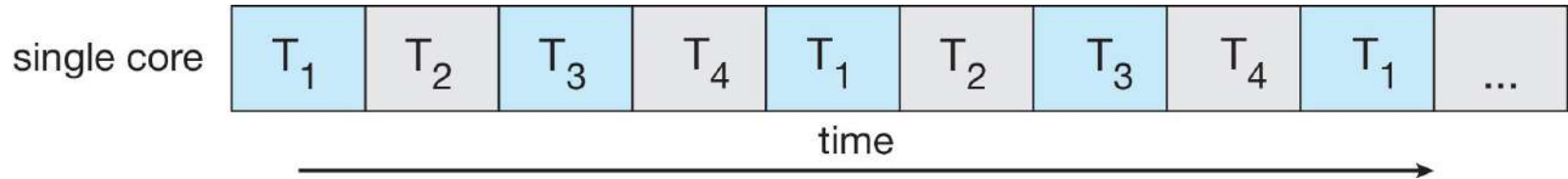
- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency



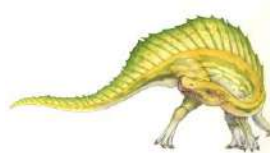
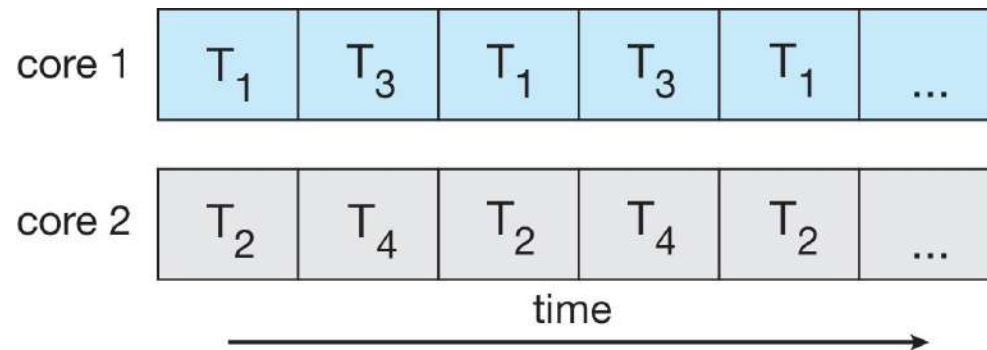


Concurrency vs. Parallelism

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



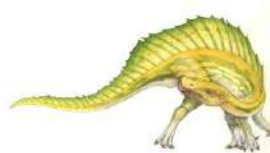
- **Parallelism on a multi-core system:**

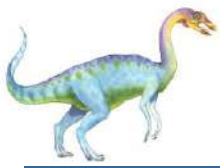




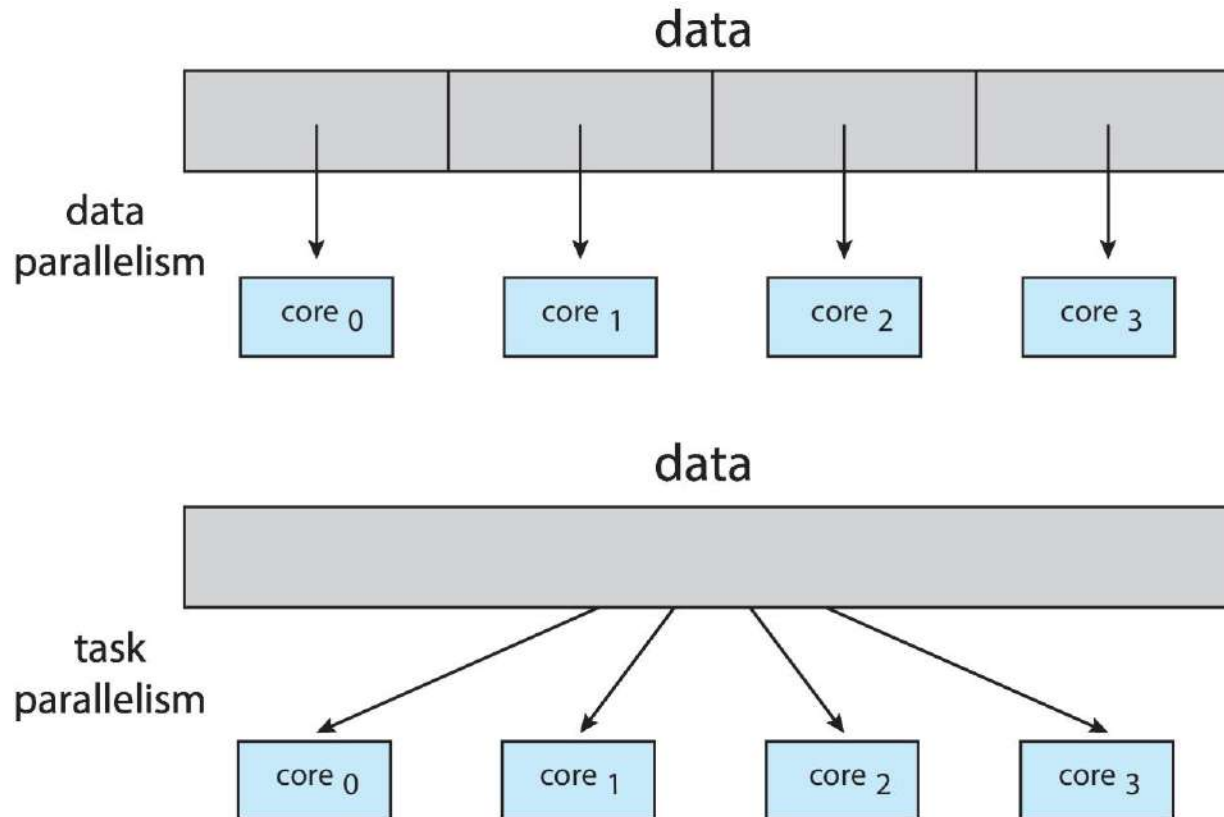
Multicore Programming

- Types of parallelism
 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - **Task parallelism** – distributing threads across cores, each thread performing unique operation





Data and Task Parallelism

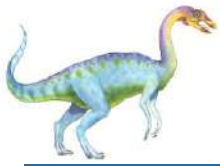




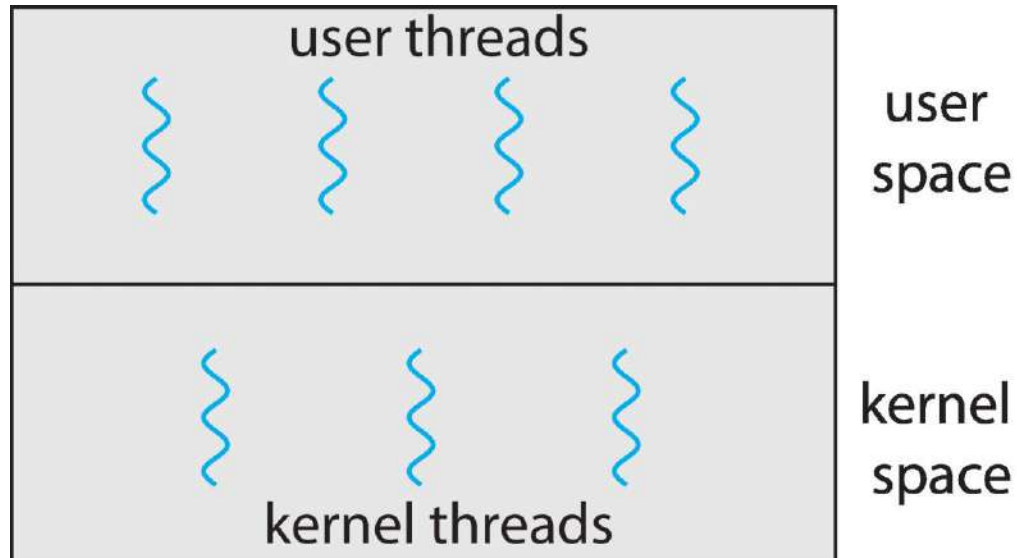
User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
 - Windows
 - Linux
 - Mac OS X
 - iOS
 - Android





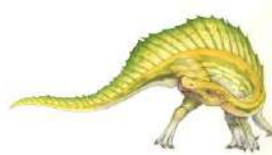
User and Kernel Threads





Multithreading Models

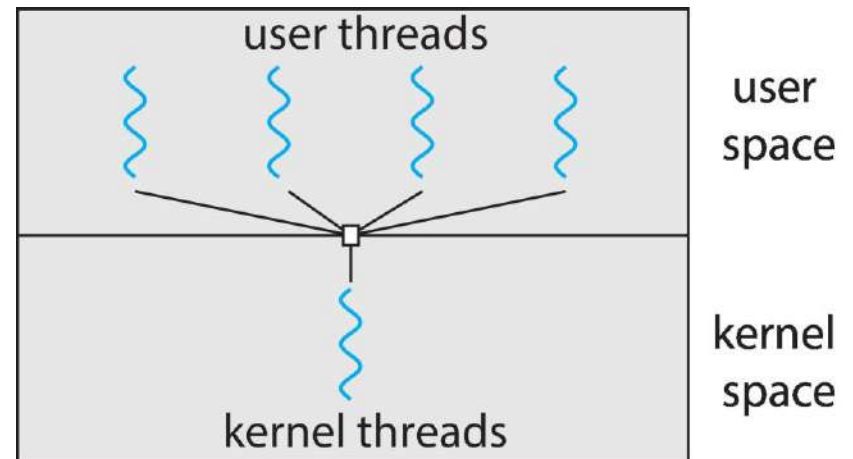
- Many-to-One
- One-to-One
- Many-to-Many





Many-to-One

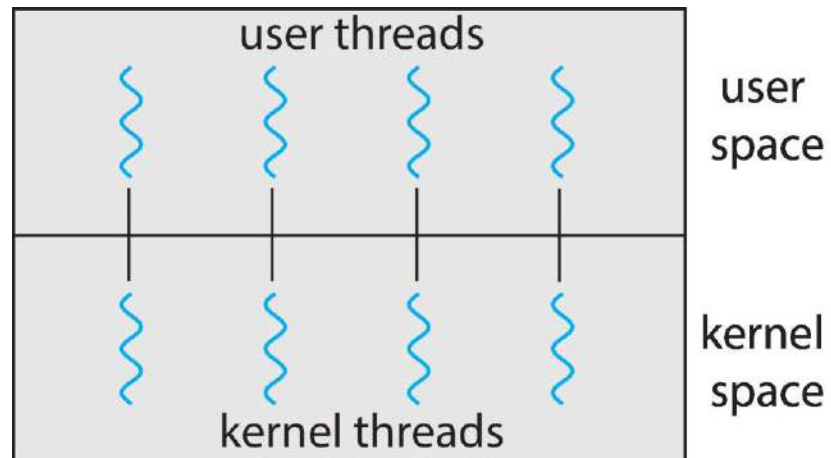
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - **Solaris Green Threads**
 - **GNU Portable Threads**





One-to-One

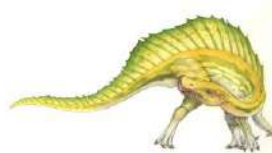
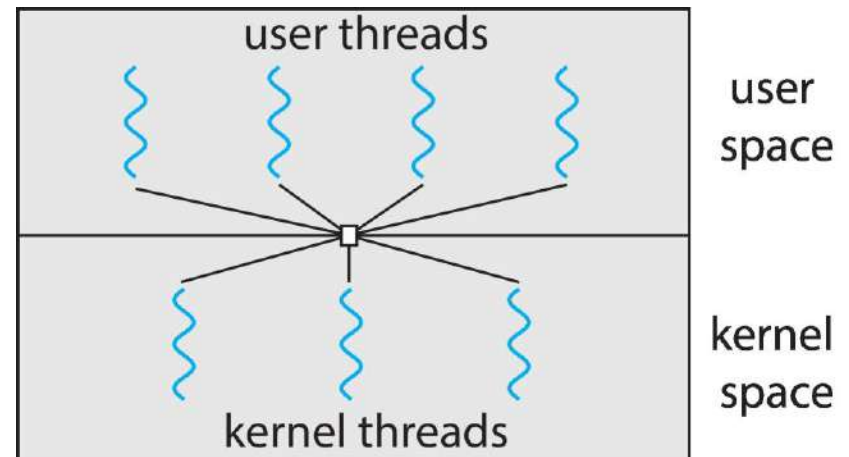
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux





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
- Windows with the *ThreadFiber* package
- Otherwise not very common





Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

