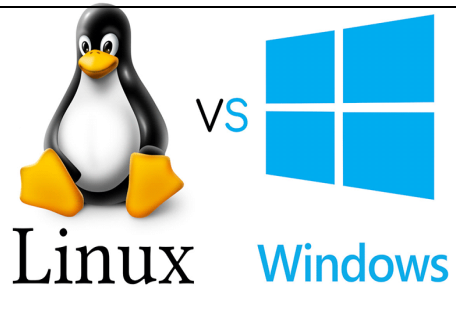
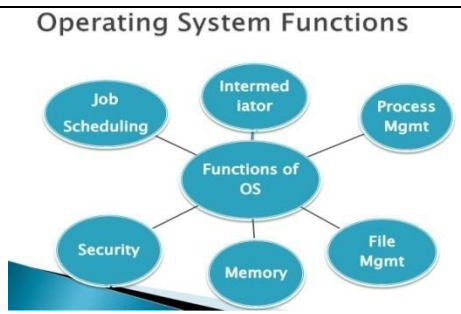
CIT210 OP. SYS. MANAG.

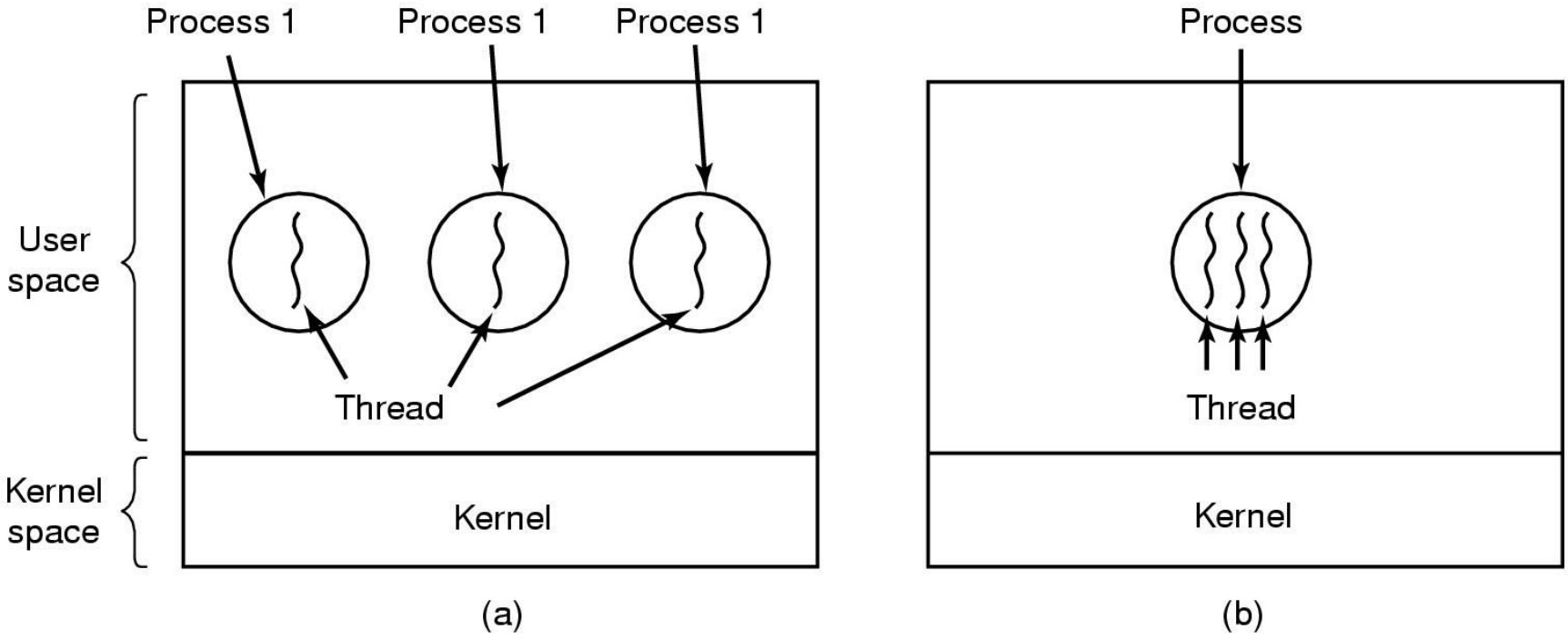
CSUN
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE

Week7Day2 : Process Management, Process vs Thread, Threads States, Multithreading, User-level and Kernel-Level Thread, Hybrid ULT/KLT Approaches, Windows/Linux/Unix Process Management





The Process vs. Thread Model



(a) Three processes each with one thread.

(b) One process with three threads.

Threads Motivation

- 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.

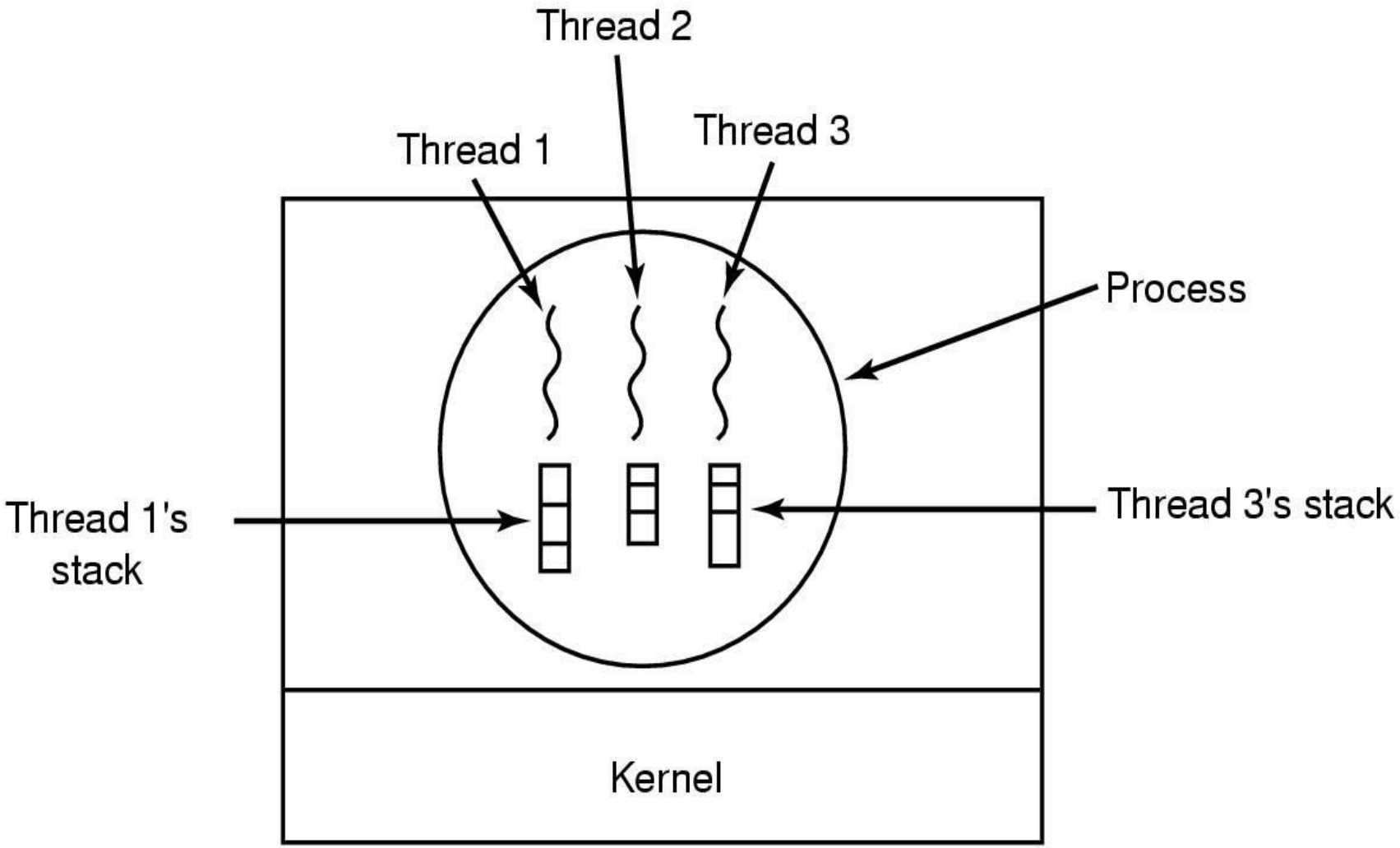
Tasks/Processes and Threads (1)

- **Task Items (shared by all threads of task):**
  - address space which holds the process image
  - global variables
  - protected access to files, I/O and other resources
- **Thread Items:**
  - an execution state (Running, Ready, etc.)
  - program counter, register set
  - execution stack
  - some per-thread static storage for local variables
  - saved thread context when not running

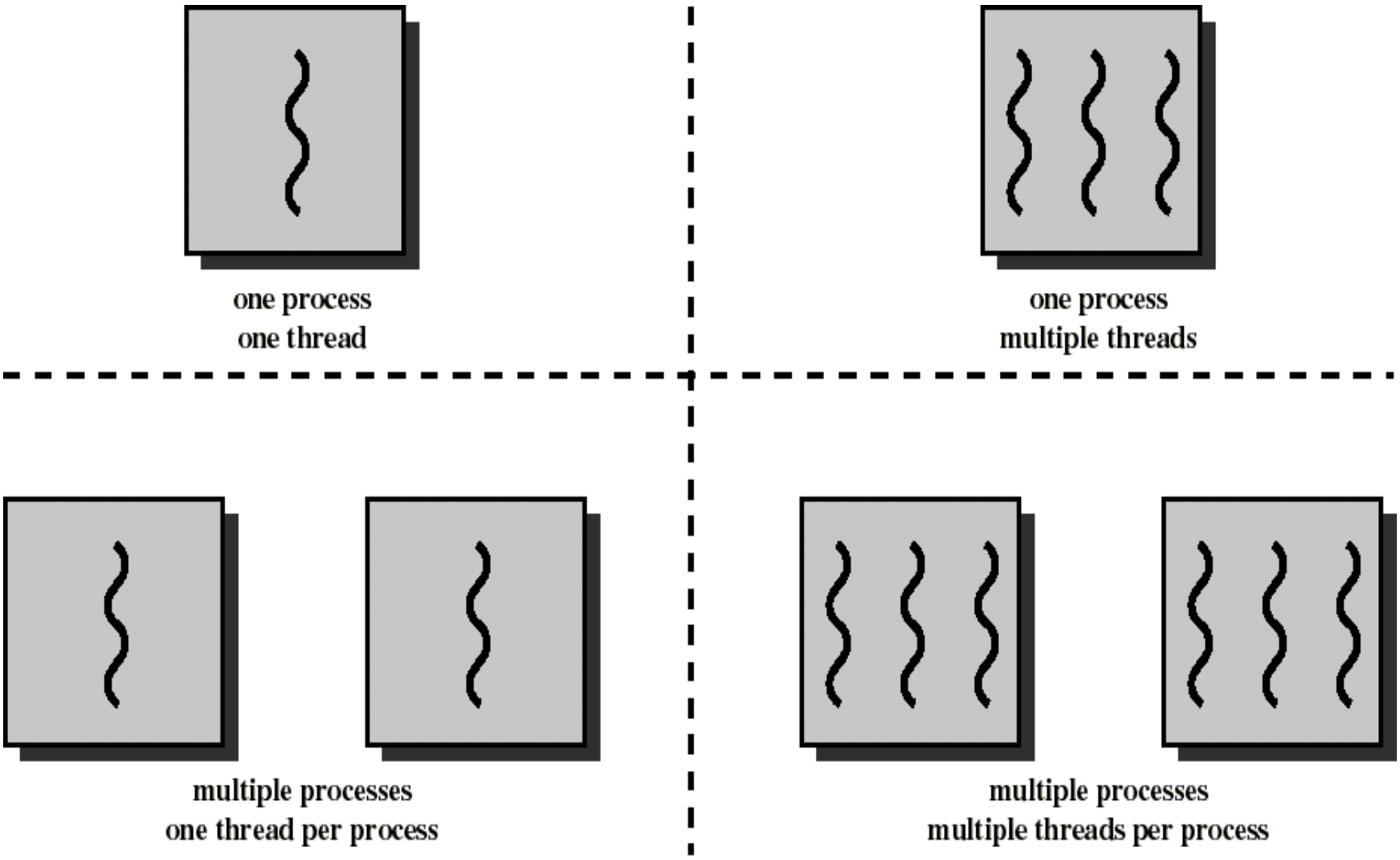
Tasks/Processes and Threads (2)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

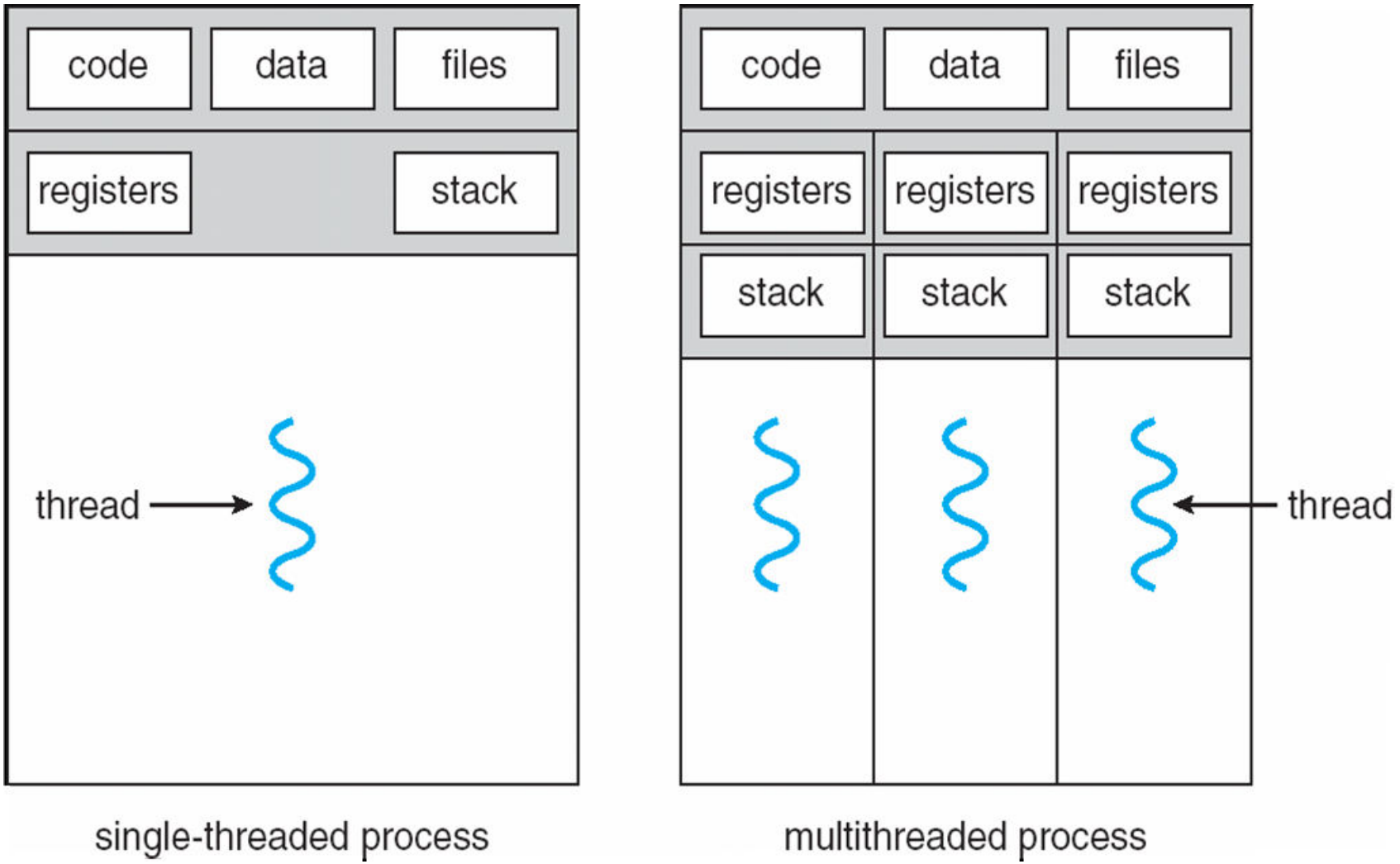
Each thread has its own stack



Combinations of Threads and Processes



Single and Multithreaded Processes



Processes vs. Threads

- Creating and managing processes is generally regarded as an expensive task (fork system call).
- Making sure all the processes peacefully co-exist on the system is not easy (as concurrency transparency comes at a price).
- Threads can be thought of as an “execution of a part of a program (in user-space)”.
- Rather than make the OS responsible for concurrency transparency, it is left to the individual application to manage the creation and scheduling of each thread.

Benefits of Threads (1)

- Responsiveness – may allow continued execution if part of process is blocked, especially important for user interfaces.
- Resource Sharing – threads share process resources, 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 and networked/distributed systems.

Benefits of Threads (2)

Threads allows parallel activity inside a single address space:

- **While one server thread is blocked and waiting, a second thread in the same task can run.**
- **Less time to create and terminate a thread than a process (because we do not need another address space).**
- **Less time to switch between two threads than between processes.**
- **Inter-thread communication and synchronization is very fast.**

Examples of benefits of threads

- **Example 1: Word Processor:**
  - one thread displays menu and reads user input while another thread executes user commands and a third one writes to disk – the application is more responsive.
- **Example 2: a File/Web server on a LAN:**
  - It needs to handle several files/pages requests over a short period.
  - Hence more efficient to create (and destroy) a single thread for each request.
  - On a SMP machine: multiple threads can possibly be executing simultaneously on different processors.

Important Implications

- Two Important Implications:
  1. Threaded applications often run faster than non-threaded applications (as context-switches between kernel and user-space are avoided).
  2. Threaded applications are harder to develop (although simple, clean designs can help here).
- Additionally, the assumption is that the development environment provides a Threads Library for developers to use (most modern environments do).

Application benefits of threads (1)

- Consider an application that consists of several independent parts that do not need to run in sequence.
- Each part can be implemented as a thread.
- Whenever one thread is blocked waiting for an I/O, execution could possibly switch to another thread of the same application (instead of blocking it and switching to another application).

Application benefits of threads (2)

- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel.
- Therefore necessary to synchronize the activities of various threads so that they do not obtain inconsistent views of the current data.

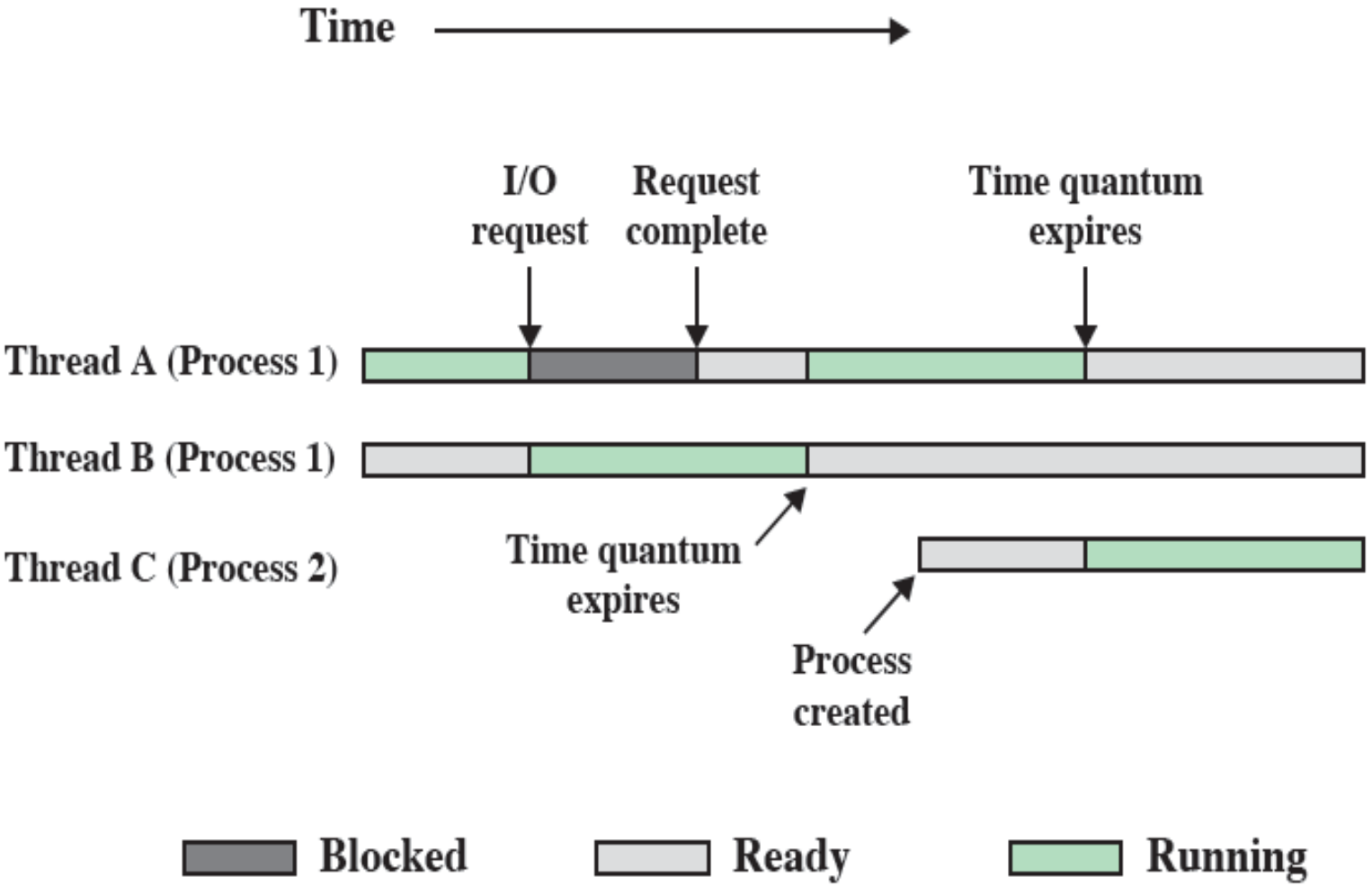
Thread Characteristics

- Has an execution context/state.
- Thread context is saved when not running.
- Has an execution stack and some per-thread static storage for local variables.
- Has access to the memory address space and resources of its task:
  - all threads of a task share this.
  - when one thread alters a (non-private) memory item, all other threads (of the task) see that.
  - a file open with one thread, is available to others.

Threads States

- Three key states: running, ready, blocked
- They have no suspend state because all threads within same task share the same address space
  - Indeed: suspending (i.e., swapping) a single thread involves suspending all threads of the same task.
- Termination of a task, terminates all threads within the task.

Multithreading Example



Multithreading vs. Single threading

- Single threading: when the OS does not recognize the concept of thread.
- Multithreading: when the OS supports multiple threads of execution within a single process.
- MS-DOS supports a single user process and a single thread.
- Older UNIXs supports multiple user processes but only support one thread per process.
- Solaris and Windows NT support multiple threads.

Multithreading Levels

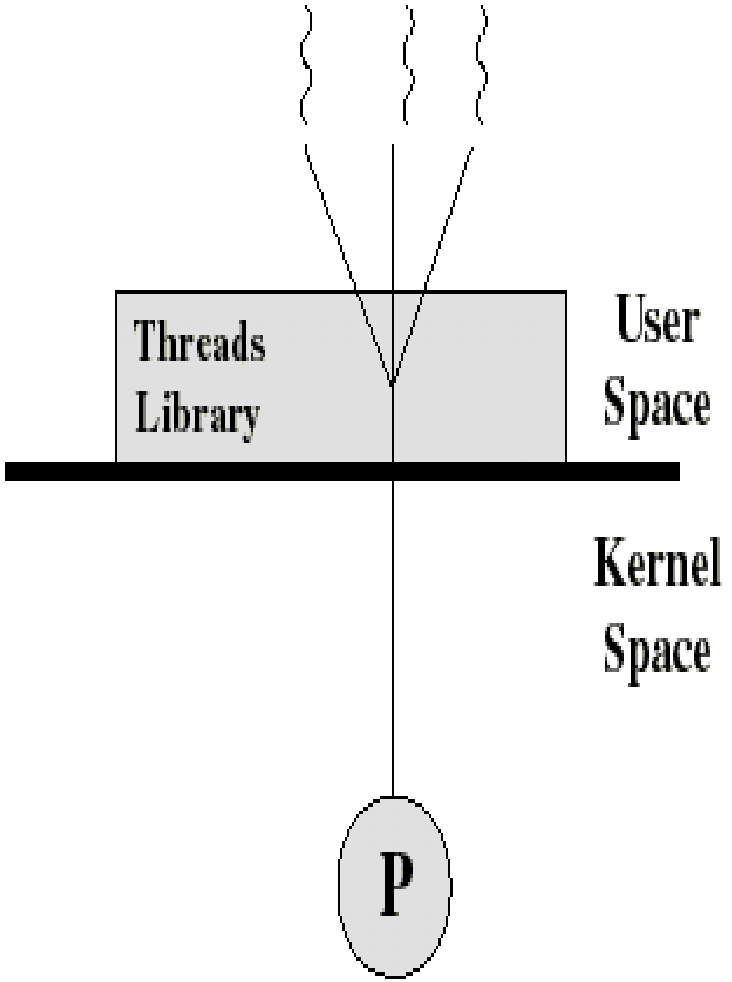
- **Thread library provides programmer with API for creating and managing threads.**
- **Three multithreading levels:**
  1) **User-Level Threads (ULT)**
     - Library entirely in user space.
  2) **Kernel-Level Threads (KLT)**
     - Kernel-level library supported by the OS.
  3) **Hybrid ULT/KLT Approach**

User(-level) and Kernel(-level) Threads

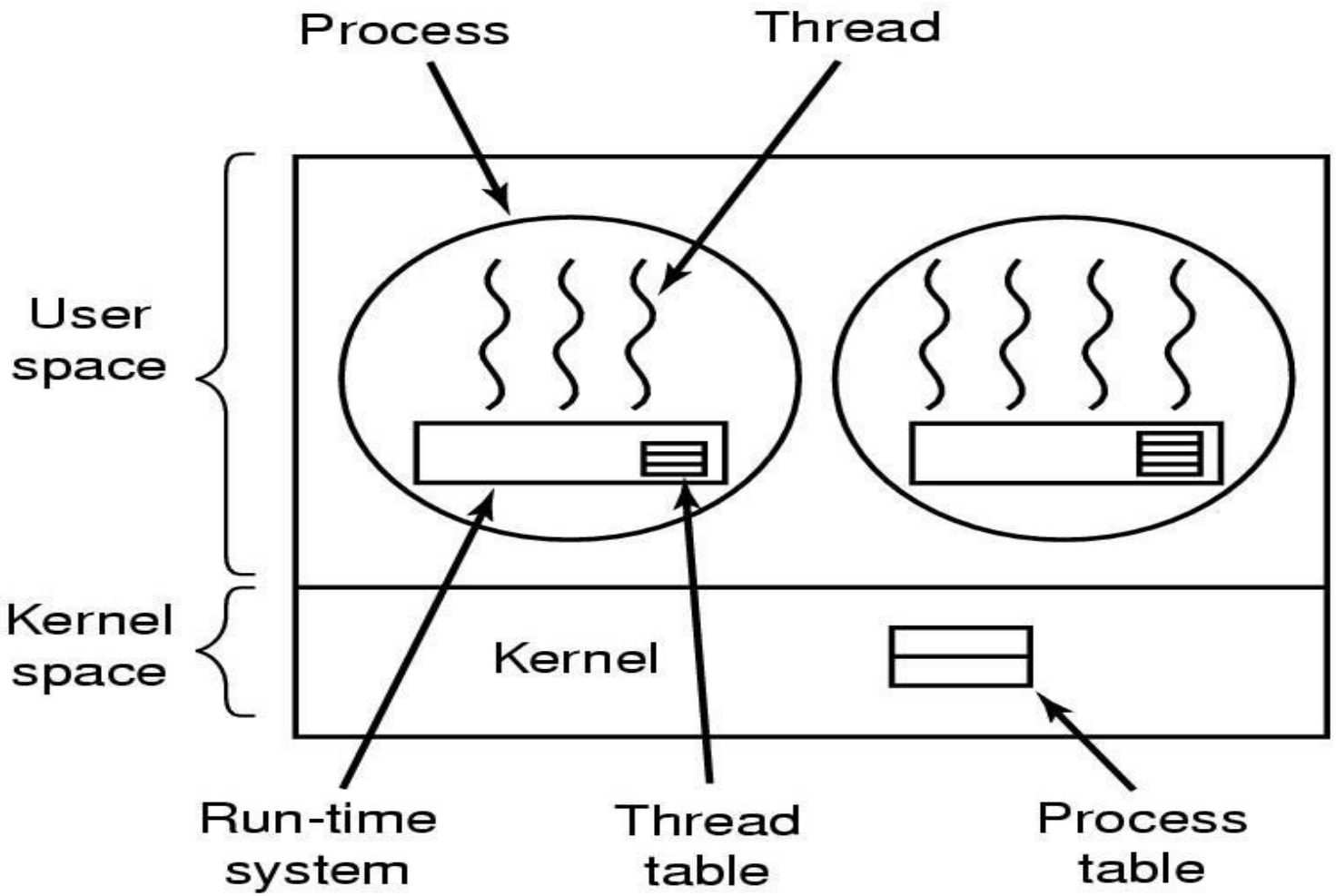
- User(-level) threads – management by user-level threads library.
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads
- Kernel(-level) threads – supported by the Kernel.
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

1) User-Level Threads (ULT)

- Thread management done by user-level threads library
- The kernel is not aware of the existence of threads.
- All thread management is done by the application by using a thread library.
- Thread switching does not require kernel mode privileges.
- Scheduling is application specific.



Implementing Threads in User Space



ULT Idea

- Thread management done by user-level threads library.
- Threads library contains code for:
  - creating and destroying threads.
  - passing messages and data between threads.
  - scheduling thread execution.
  - saving and restoring thread contexts.
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

POSIX Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- May be provided either as ULT or KLT.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX operating systems (Solaris, Linux, Mac OS X).

Kernel activity for ULTs

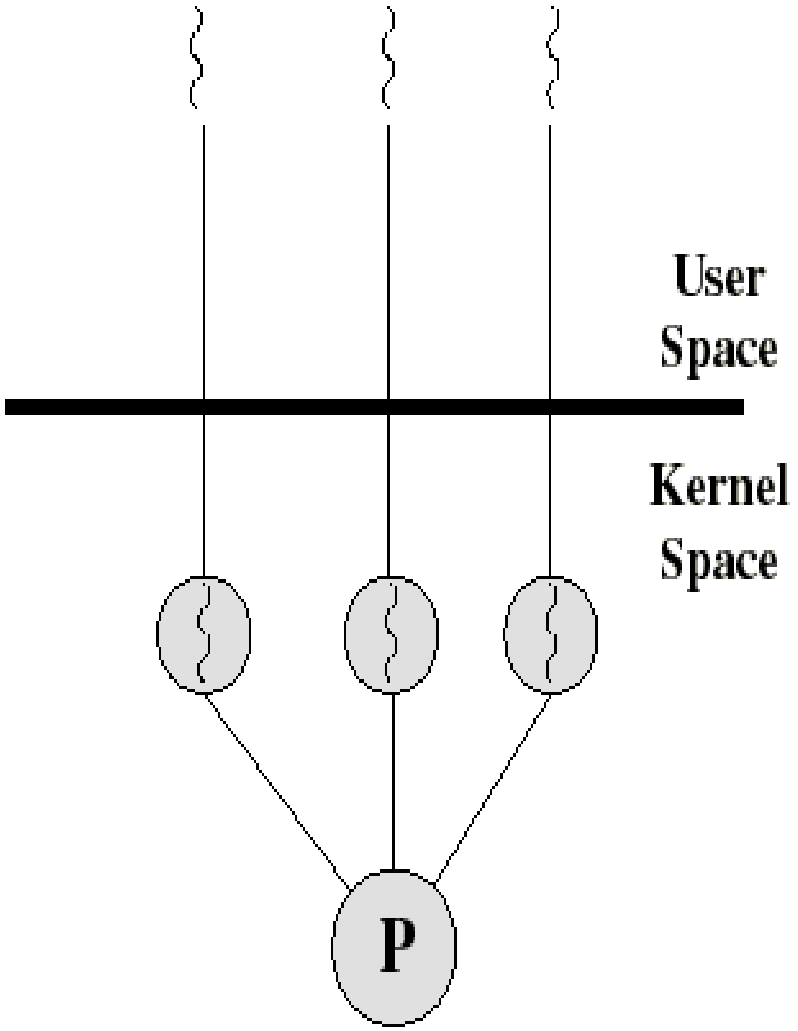
- The kernel is not aware of thread activity but it is still managing process activity.
- When a thread makes a system call, the whole task will be blocked.
- But for the thread library that thread is still in the running state.
- So thread states are independent of process states.

Advantages and inconveniences of ULT

- Advantages
  - Thread switching does not involve the kernel: no mode switching.
  - Scheduling can be application specific: choose the best algorithm.
  - ULTs can run on any OS. Only needs a thread library.
- Inconveniences
  - Most system calls are blocking and the kernel blocks processes. So all threads within the process will be blocked.
  - The kernel can only assign processes to processors. Two threads within the same process cannot run simultaneously on two processors.

2) Kernel-Level Threads (KLT)

- All thread management is done by kernel.
- No thread library but an API to the kernel thread facility.
- Kernel maintains context information for the process and the threads.
- Switching between threads requires the kernel.
- Scheduling on a thread basis.



KLT Idea

- Threads supported by the Kernel.
- Examples:
  - Windows
  - OS/2
  - Linux
  - Solaris
  - Tru64 UNIX
  - Mac OS X

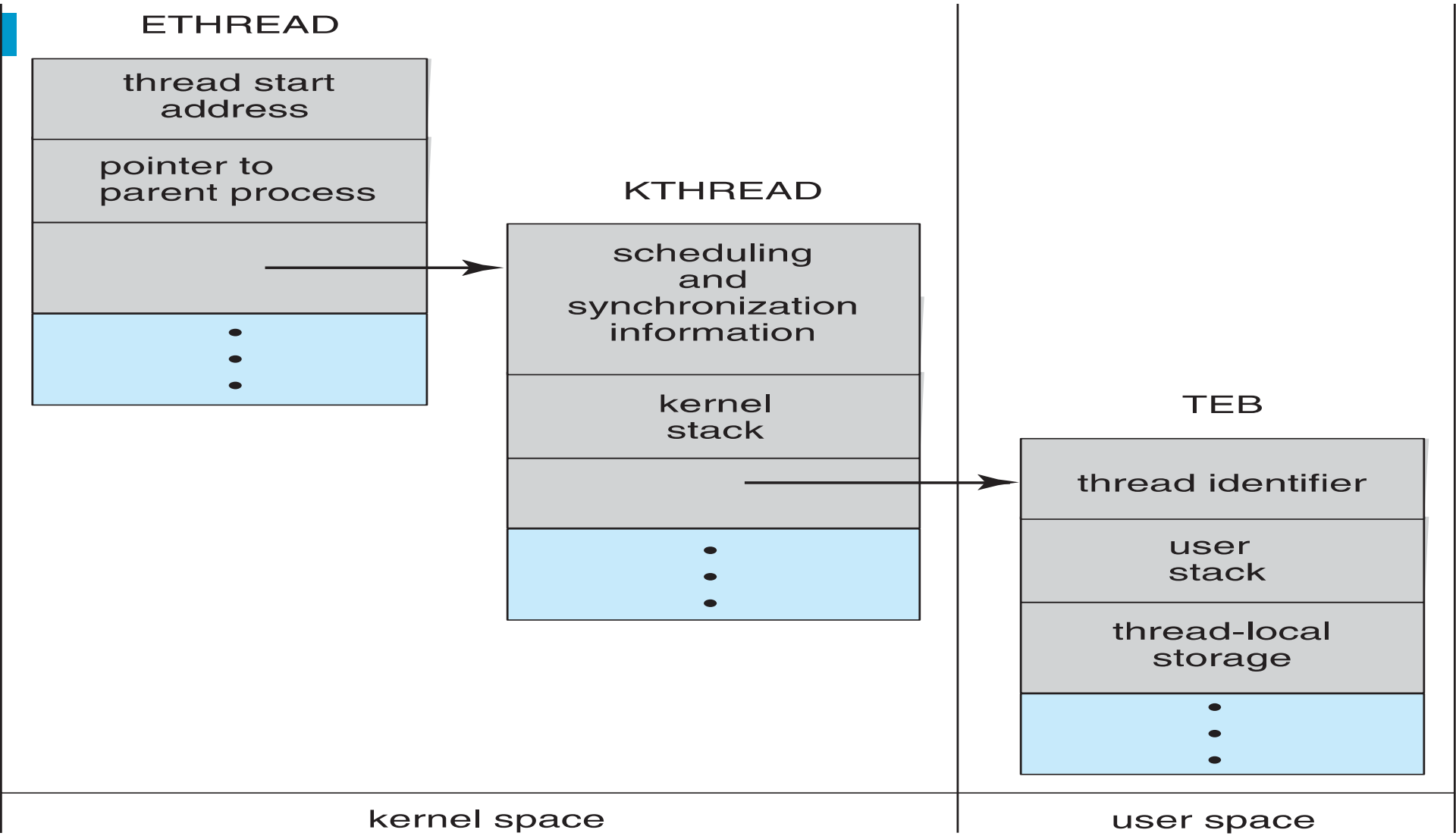
Windows Threads

- Windows implements the Windows API
- Implements the one-to-one mapping, kernel-level.
- Each thread contains:
  - A thread id.
  - Register set representing state of processor.
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode.
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs).
- The register set, stacks, and private storage area are known as the context of the thread.

Windows Threads

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space.
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space.
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space.

Windows Threads Data Structures



Linux Threads

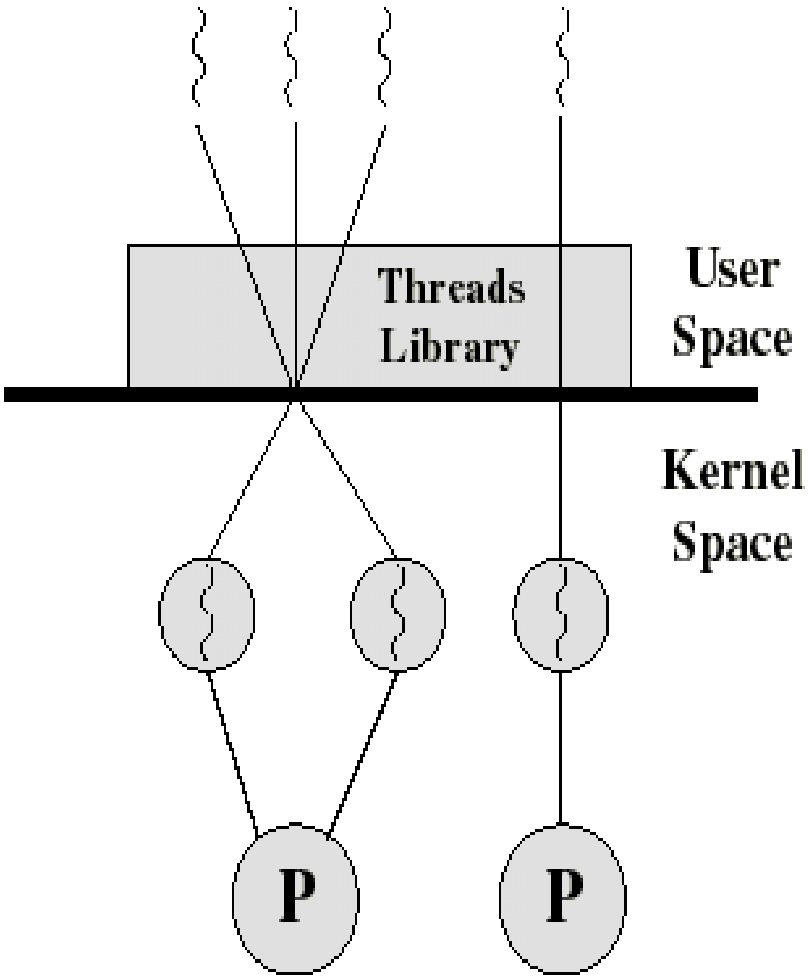
- Linux refers to them as tasks rather than threads.
- Thread creation is done through clone() system call.
- clone() allows a child task to share the address space of the parent task (process).
- This sharing of the address space allows the cloned child task to behave much like a separate thread.

Advantages and inconveniences of KLT

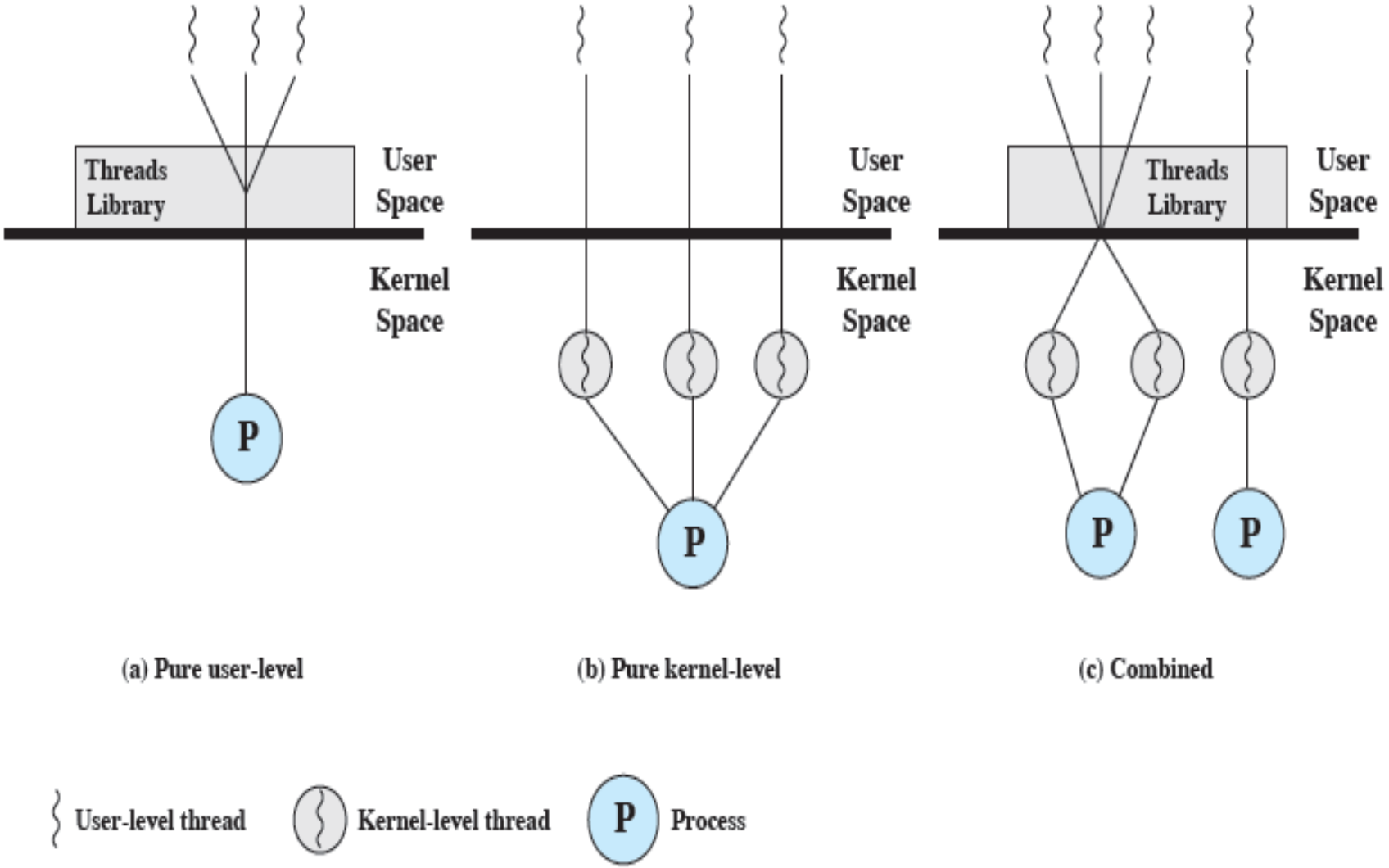
- Advantages
  - the kernel can simultaneously schedule many threads of the same process on many processors.
  - blocking is done on a thread level.
  - kernel routines can be multithreaded.
- Inconveniences
  - thread switching within the same process involves the kernel. We have 2 mode switches per thread switch.
  - this results in a significant slow down.

3) Hybrid ULT/KLT Approaches

- Thread creation done in the user space.
- Bulk of scheduling and synchronization of threads done in the user space.
- The programmer may adjust the number of KLTs.
- May combine the best of both approaches.
- Example is Solaris



ULT, KLT and Combined Approaches



Process

- A *Process* in Unix, Linux, or Windows comprises:
  - an *address space* – usually protected and virtual – mapped into memory
  - the *code* for the running program
  - the *data* for the running program
  - an *execution stack* and *stack pointer* (SP)
  - the *program counter* (PC)
  - a set of processor *registers* – general purpose and status
  - a set of system *resources*
    - files, network connections, pipes, …
    - privileges, (human) user association, …

Process – OS perspective

- The principal function of a processor is to execute machine instructions residing in main memory
  - Those instructions are provided in the form of programs
  - A processor may interleave the execution of a number of programs over time
- Program View
  - Its execution involves a sequence of instructions within that program
  - The behavior of individual process can be characterized by a list of the sequence of instructions – *trace* of the process
- Processor View
  - Executes instructions from main memory, as dictated by changing values in the program counter register
  - The behavior of the processor can be characterized by showing how the traces of various processes are interleaved

Process Scheduling Policies

- **Types of scheduling policies**
  - **Preemptive**
    - Used in time-sharing environments
    - Interrupts job processing
    - Transfers CPU to another job
  - **Nonpreemptive**
    - Functions without external interrupts

Process Scheduling Policies

- **Multiprogramming environment**
  - **More jobs than resources at any given time**
- **Operating system pre-scheduling task**
  - **Resolve three system limitations**
    - Finite number of resources (disk drives, printers, tape drives)
    - Some resources cannot be shared once allocated (printers)
    - Some resources require operator intervention (tape drives)

Process Scheduling Policies (continued)

- **Good process scheduling policy criteria**
  - **Maximize throughput**
    - Run as many jobs as possible in given amount of time
  - **Minimize response time**
    - Quickly turn around interactive requests
  - **Minimize turnaround time**
    - Move entire job in and out of system quickly
  - **Minimize waiting time**
    - Move job out of READY queue quickly

Process Scheduling Policies (continued)

- **Good process scheduling policy criteria (continued)**
  - **Maximize CPU efficiency**
    - Keep CPU busy
  - **Ensure fairness for all jobs**
    - Give every job equal CPU and I/O time
- **Final policy criteria decision in designer's hands**

Process Scheduling Policies (continued)

- **Problem**
  - **Job claims CPU for very long time before I/O request issued**
    - Builds up READY queue and empties I/O queues
    - Creates unacceptable system imbalance
- **Solution**
  - **Interrupt**
    - Used by Process Scheduler upon predetermined expiration of time slice
    - Current job activity suspended
    - Reschedules job into READY queue

Process Scheduling Algorithms

- Base on specific policy
  - Allocate CPU and move job through system
- Six algorithm types
  - First-come, first-served (FCFS)
  - Shortest job next (SJN)
  - Priority scheduling
  - Shortest remaining time (SRT)
  - Round robin
  - Multiple-level queues
- Current systems emphasize interactive use and response time (use preemptive policies)

First-Come, First-Served

- Nonpreemptive
- Job handled based on arrival time
  - Earlier job arrives, earlier served
- Simple algorithm implementation
  - Uses first-in, first-out (FIFO) queue
- Good for batch systems
- Unacceptable in interactive systems
  - Unpredictable turnaround time
- Disadvantages
  - Average turnaround time varies; seldom minimized

Shortest Job Next

- Non pre-emptive
- Also known as shortest job first (SJF)
- Job handled based on length of CPU cycle time
- Easy implementation in batch environment
  - CPU time requirement known in advance
- Does not work well in interactive systems
- Optimal algorithm
  - All jobs are available at same time
  - CPU estimates available and accurate

Priority Scheduling

- Non pre-emptive
- Preferential treatment for important jobs
  - Highest priority programs processed first
  - No interrupts until CPU cycles completed or natural wait occurs
- READY queue may contain two or more jobs with equal priority
  - Uses FCFS policy within priority
- System administrator or Processor Manager use different methods of assigning priorities

Priority Scheduling (continued)

- **Processor Manager priority assignment methods**
  - **Memory requirements**
    - Jobs requiring large amounts of memory allocated lower priorities (or vice versa)
  - **Number and type of peripheral devices**
    - Jobs requiring many peripheral devices allocated lower priorities (or vice versa)
  - **Total CPU time**
    - Jobs having a long CPU cycle given lower priorities (or vice versa)
  - **Amount of time already spent in the system (aging)**
    - Total time elapsed since job accepted for processing
    - Increase priority if job in system unusually long time

Shortest Remaining Time (SRT)

- Pre-emptive version of SJN
- CPU allocated to job closest to completion
  - Pre-emptive if newer job has shorter completion time
- Often used in batch environments
  - Short jobs given priority
- Cannot implement in interactive system
  - Requires advance CPU time knowledge
- Involves more overhead than SJN
  - System monitors CPU time for READY queue jobs
  - Performs context switching at pre-emption time.

Round Robin

- Pre-emptive
- Used extensively in interactive systems
- Based on predetermined slice of time
  - Each job assigned time quantum
- Time quantum size
  - Crucial to system performance
  - Varies from 100 ms to 1-2 seconds
- CPU equally shared among all active processes
  - Not monopolized by one job

Multiple-Level Queues

- **Works in conjunction with several other schemes**
- **Works well in systems with jobs grouped by common characteristic**
  - **Priority-based**
    - Different queues for each priority level
  - **CPU-bound jobs in one queue and I/O-bound jobs in another queue**
  - **Hybrid environment**
    - Batch jobs in background queue
    - Interactive jobs in foreground queue
- **Scheduling policy based on predetermined scheme**
- **Four primary methods**

UNIX Processes

- A program that has started is manifested in the context of a process.
- A process in the system is represented:
  - Process Identification Elements
  - Process State Information
  - Process Control Information
  - User Stack
  - Private User Address Space, Programs and Data
  - Shared Address Space

Process Control Block

- Process Information, Process State Information, and Process Control Information constitute the PCB.
- All Process State Information is stored in the Process Status Word (PSW).
- All information needed by the OS to manage the process is contained in the PCB.
- A UNIX process can be in a variety of states:

States of a UNIX Process

- User running: Process executes in user mode
- Kernel running: Process executes in kernel mode
- Ready to run in memory: process is waiting to be scheduled
- Asleep in memory: waiting for an event
- Ready to run swapped: ready to run but requires swapping in
- Preempted: Process is returning from kernel to user-mode but the system has scheduled another process instead
- Created: Process is newly created and not ready to run
- Zombie: Process no longer exists, but it leaves a record for its parent process to collect.

UNIX Process Management

- fork() – system call to create a copy of the current process, and start it running
  - No arguments!
- exec() – system call to change the program being run by the current process
- wait() – system call to wait for a process to finish
- signal() – system call to send a notification to another process

Creating a new process

- **In UNIX, a new process is created by means of the fork() - system call. The OS performs the following functions:**
  - It allocates a slot in the process table for the new process
  - It assigns a unique ID to the new process
  - It makes a copy of process image of the parent (except shared memory)
  - It assigns the child process to the Ready to Run State
  - It returns the ID of the child to the parent process, and 0 to the child.

- **Note, the fork() call actually is called once but returns twice - namely in the parent and the child process.**

Fork()

- Pid_t fork(void) is the prototype of the fork() call.
- Remember that fork() returns twice
  - in the newly created (child) process with return value 0
  - in the calling process (parent) with return value = pid of the new process.
  - A negative return value (-1) indicates that the call has failed

- Different return values are the key for distinguishing parent process from child process!
- The child process is an exact copy of the parent, yet, it is a copy i.e. an identical but separate process image.

ps and *kill*

- **Two of the shell commands that you will be using when working with processes are *ps* and *kill.***
  - ***ps* reports on active processes.**
    - -A lists all processes
    - -e includes the environment
    - -u username, lists all processes associated with username
  - **NOTE: the options may be differ on different systems!!**
  - **What if you get a listing that is too long?**
    - try ps -A | grep username

kill and *killall*

- In order to communicate with an executing process from the shell, you must send a signal to that process.
- The *kill* command sends a signal to the process.
- You can direct the signal to a particular process by using its *pid*

- The *kill* command has the following format:

  kill [options] pid

  - -l lists all the signals you can send
  - -p (on some systems prints process information)
  - -*signal* is a signal number

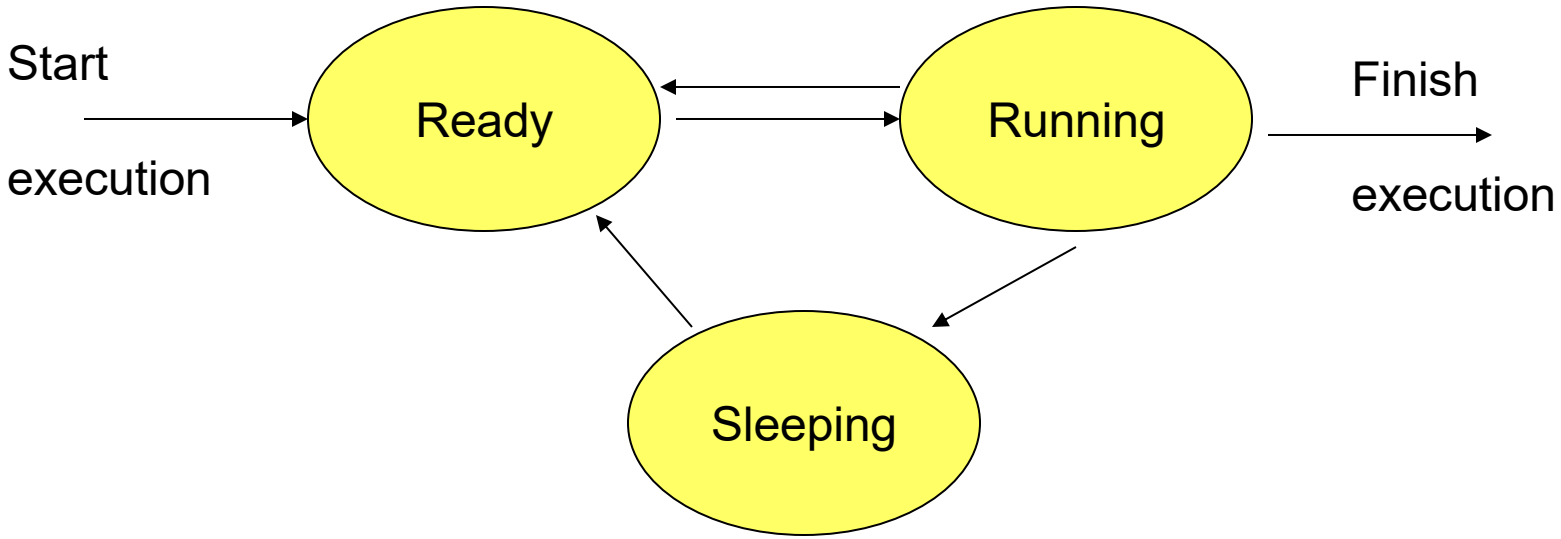
Linux Process Management

- Linux is a multitasking system
- Multiple programs can be executed at the same time
- Ultimately, a program needs to be executed by a CPU
- If there is only one CPU, how multiple programs can be executed at the same time?

  $\Rightarrow$ By time sharing
- That is, all programs are claimed to be executing. In fact, most of them are waiting for the CPU

Linux Process Management

- A program that is claimed to be executing is called a process
- For a multitasking system, a process has at least the following three states:



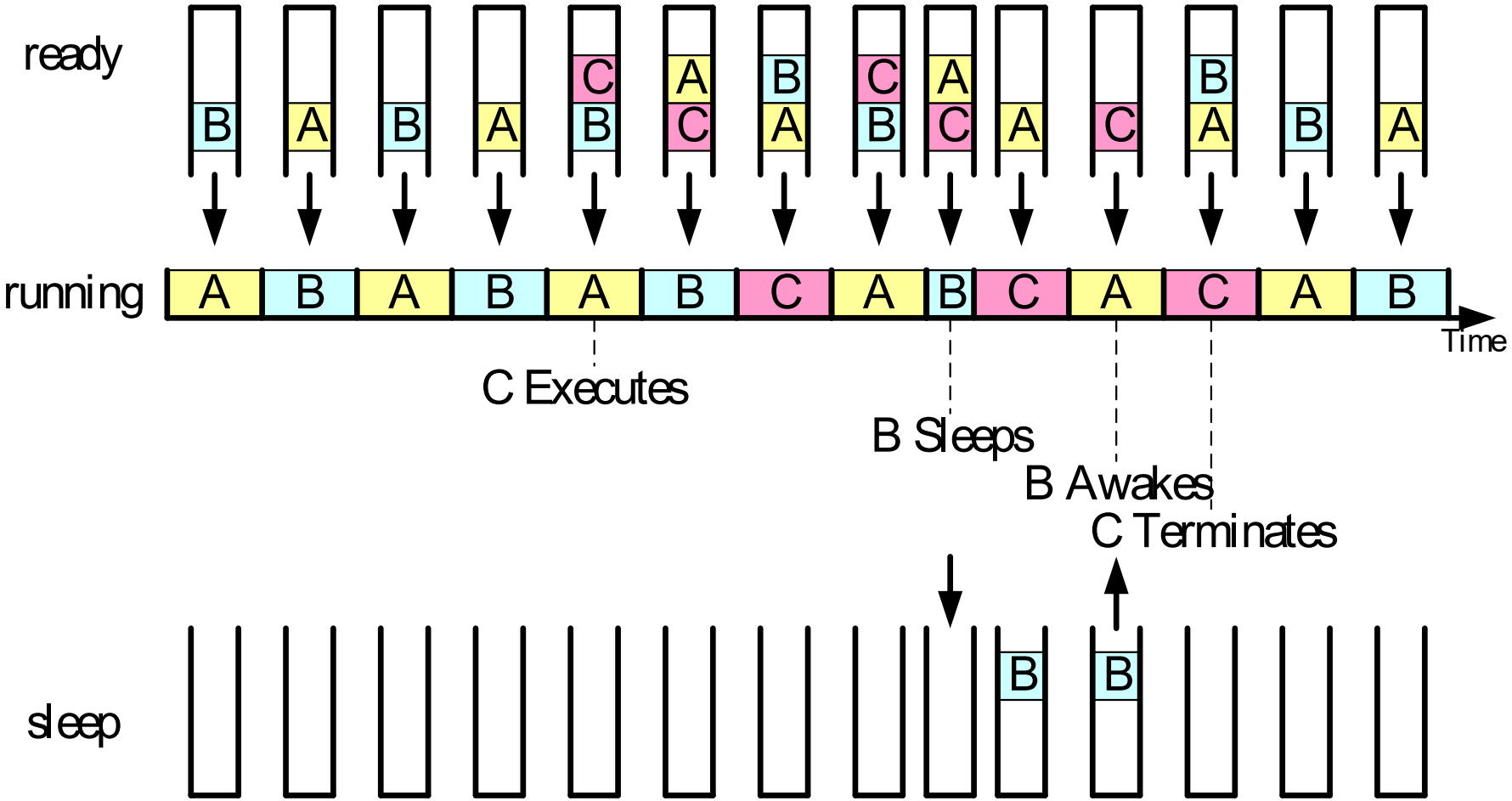
Linux Process Management

- Ready state
  - All processes that are ready to execute but without the CPU are at the ready state
  - If there is only 1 CPU in the system, all processes except one are at the ready state
- Running state
  - The process that actually possesses the CPU is at the running state
  - If there is only 1 CPU in the system, at most there is only one process is at the running state
- Sleeping state
  - The process that is waiting for other resources, e.g. I/O, is at the sleeping state

Linux Process Management

- Processes will alternatively get into the CPU one after the other (called the round robin scheme)
- A process will be “in” a CPU for a very short time (quantum)
  - For Linux, each quantum is about 100msec
- At the time that a process is selected to be “in” the CPU
  - It goes from ready state to running state
- After that, it will be swapped out
  - It goes from running state back to ready state
- Or it may due to the waiting of an I/O device, e.g. mouse
  - It goes from running state to sleeping state
- When obtaining the required resource
  - It goes from sleeping state to ready state

Linux Process Management



Organization of Process Table

- **Descriptor: referencing process**
  - **Contains approximately 70 fields: describe process attributes**
  - **Includes information needed to manage process**
  - **Dynamically allocated by kernel**
    - Process execution time
  - **Organized by doubly linked lists**
    - "Next run" field
    - "Previously run" field
  - **Scheduler manages and updates descriptors using macros**

Process Synchronization

- **Wait queues and semaphores**
  - **Synchronize two processes with each other**
- **Wait queue**
  - **Linked circular list of process descriptors**
  - **Problems solved**
    - Mutual exclusion and producers and consumers
- **Semaphore structure**
  - **Three fields (semaphore counter, number of waiting processes, list of processes waiting for semaphore)**
    - Counter contains only binary values: except if several units of one resource available

Process Management

- **Linux scheduler**
  - **Scans processes list in READY state**
  - **Chooses process to execute**
    - Using predefined criteria
- **Three scheduling types**
  - **Real-time processes (two)**
  - **Normal processes (one)**
- **Process scheduling policy determination**
  - **Combination of type and priority**

Process Management (cont'd.)

Name	Scheduling Policy	Priority Level	Process Type
SCHED_FIFO	First in first out	Highest	For non-preemptible real-time processes
SCHED_RR	Round Robin and priority	Medium	For preemptible real-time processes
SCHED_OTHER	Priority only	Lowest	For normal processes

Linux has three process types, each signaling a different level of priority.

Process Management (cont'd.)

- **First type**
  - **Highest priority (SCHED_FIFO)**
    - First in, first out algorithm
  - **Cannot be preempted**
  - **Runs to completion unless:**
    - Process goes into WAIT state
    - Process relinquishes processor voluntarily
  - **All FIFO processes complete**
    - Scheduler processes lower priority types

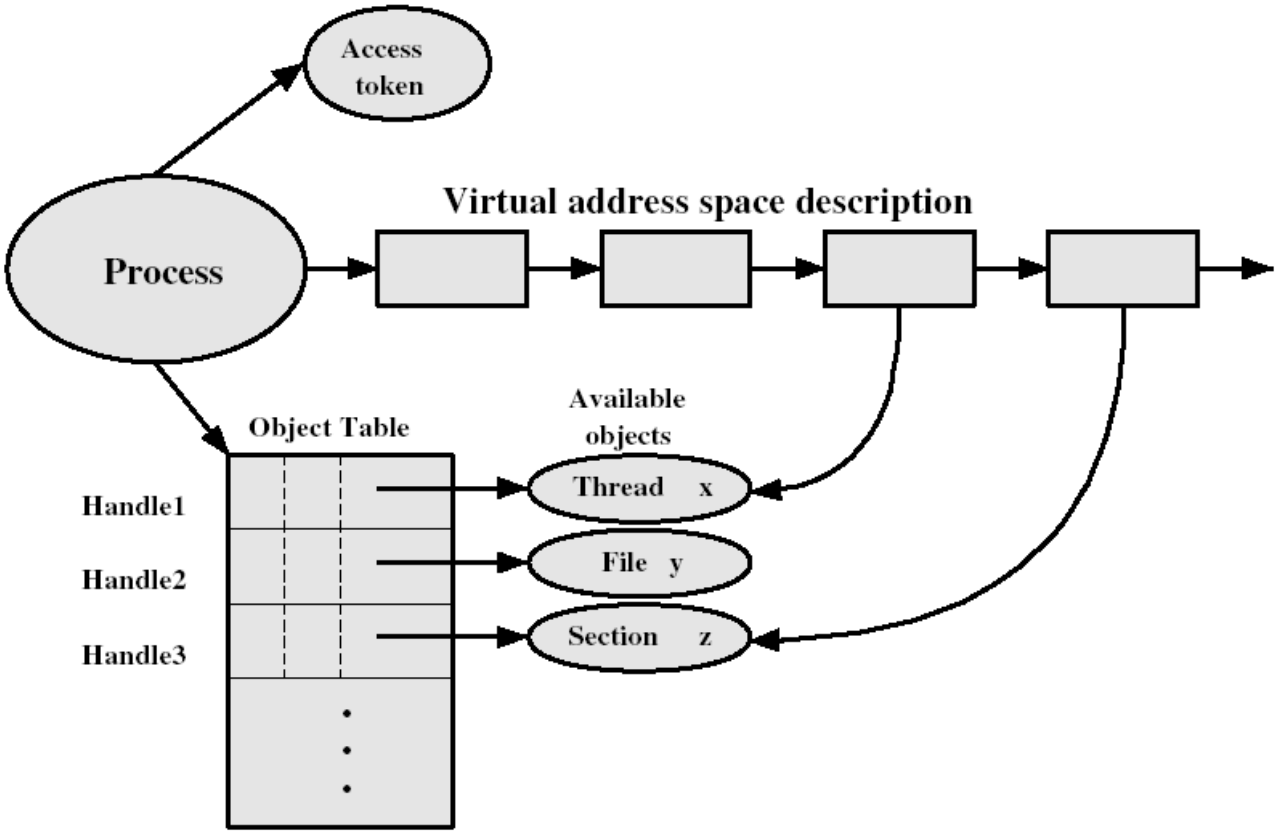
Process Management (cont'd.)

- **Second type**
  - **Medium priority (SCHED_RR)**
    - Round robin algorithm with small time quantum
  - **Time quantum expires**
    - Other higher priority processes (FIFO, RR) selected and executed: before first process allowed to complete
- **Third type**
  - **Low priority (SCHED_OTHER)**
  - **Executed if no higher priority processes in READY queue**

Running Processes in the Background

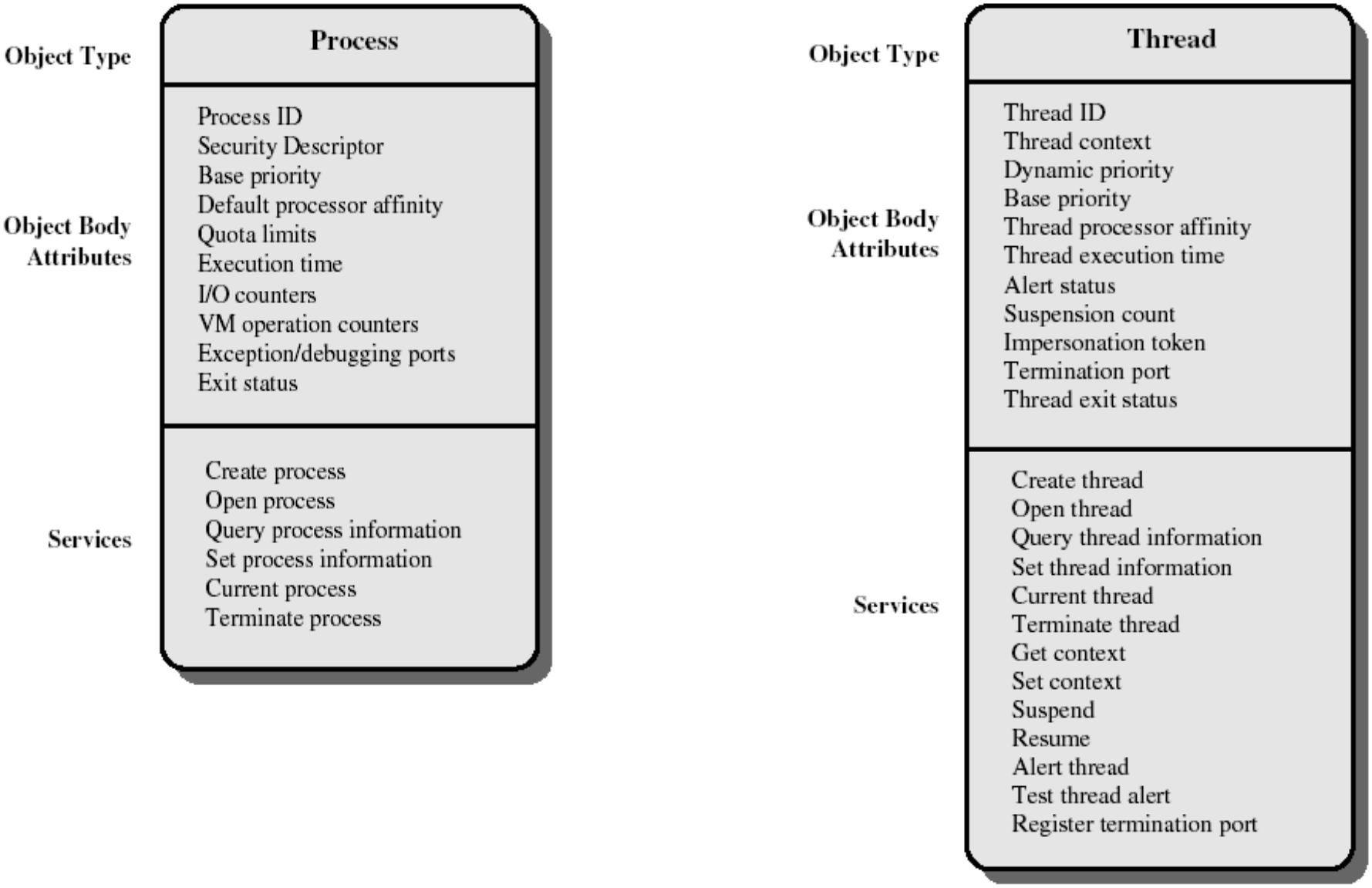
- Can run a process in the background while working with another program in the foreground
- To run a program in the background, append the & character to end of the startup command, e.g., top&

Windows Processes



- **Characteristics of Processes**
  - Implemented as objects
  - May contain one or more threads
  - Both processes and threads have built-in synchronization capabilities

Windows Process and Thread Objects



Windows Process and Thread Management