ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

OPERATING SYSTEMS & PARALLEL COMPUTING

Processes and Threads

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

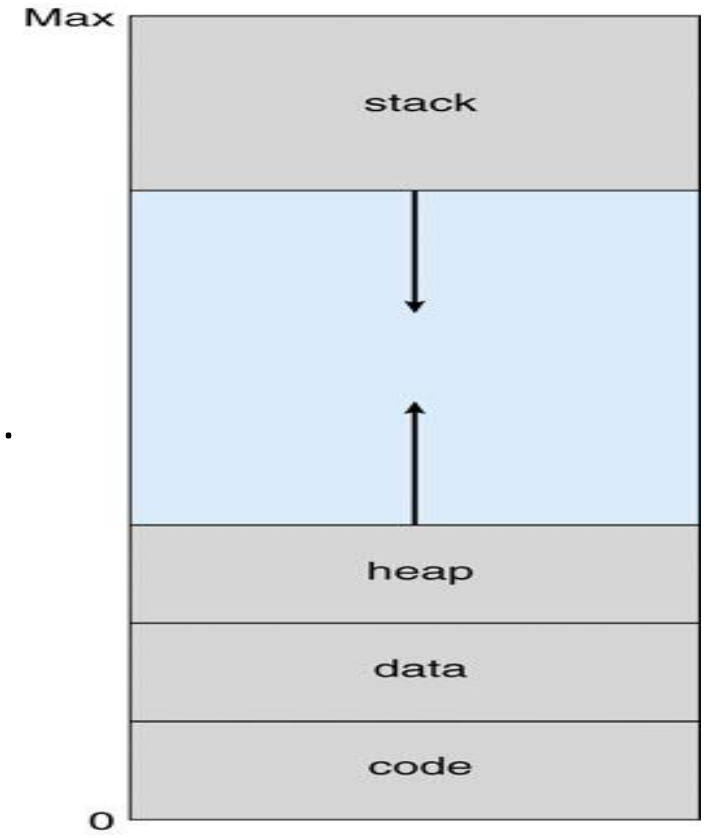
Introduction to Process

Process Concept

- Process is a program in execution; forms the basis of all computation; process execution must progress in sequential fashion.
- Program is a passive entity stored on disk (executable file), Process is an active entity; A program becomes a process when executable file is loaded into memory.
- Execution of program is started via CLI entry of its name, GUI mouse clicks, etc.
- A process is an instance of a running program; it can be assigned to, and executed on, a processor.
- Related terms for Process: Job, Step, Load Module, Task, Thread.

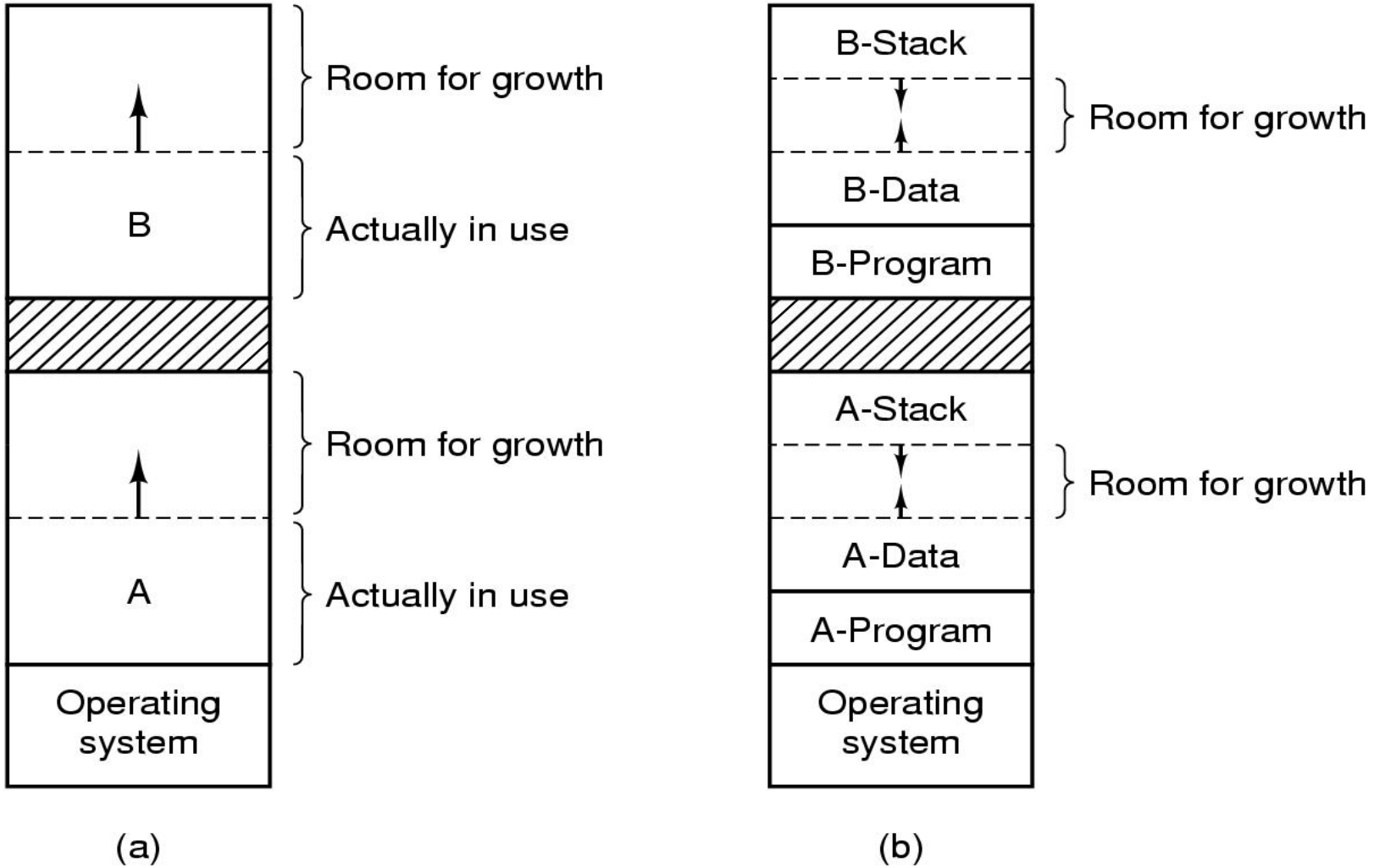
Process Parts

- A process includes three segments/sections:
  1. **Program**: code/text.
  2. **Data**: global variables and heap
     - Heap contains memory dynamically allocated during run time.
  3. **Stack**: temporary data
     - Procedure/Function parameters, return addresses, local variables.
- Current activity of a program includes its **Context**: program counter, state, processor registers, etc.
- One program can be several processes:
  - Multiple users executing the same Sequential program.
  - Concurrent program running several process.



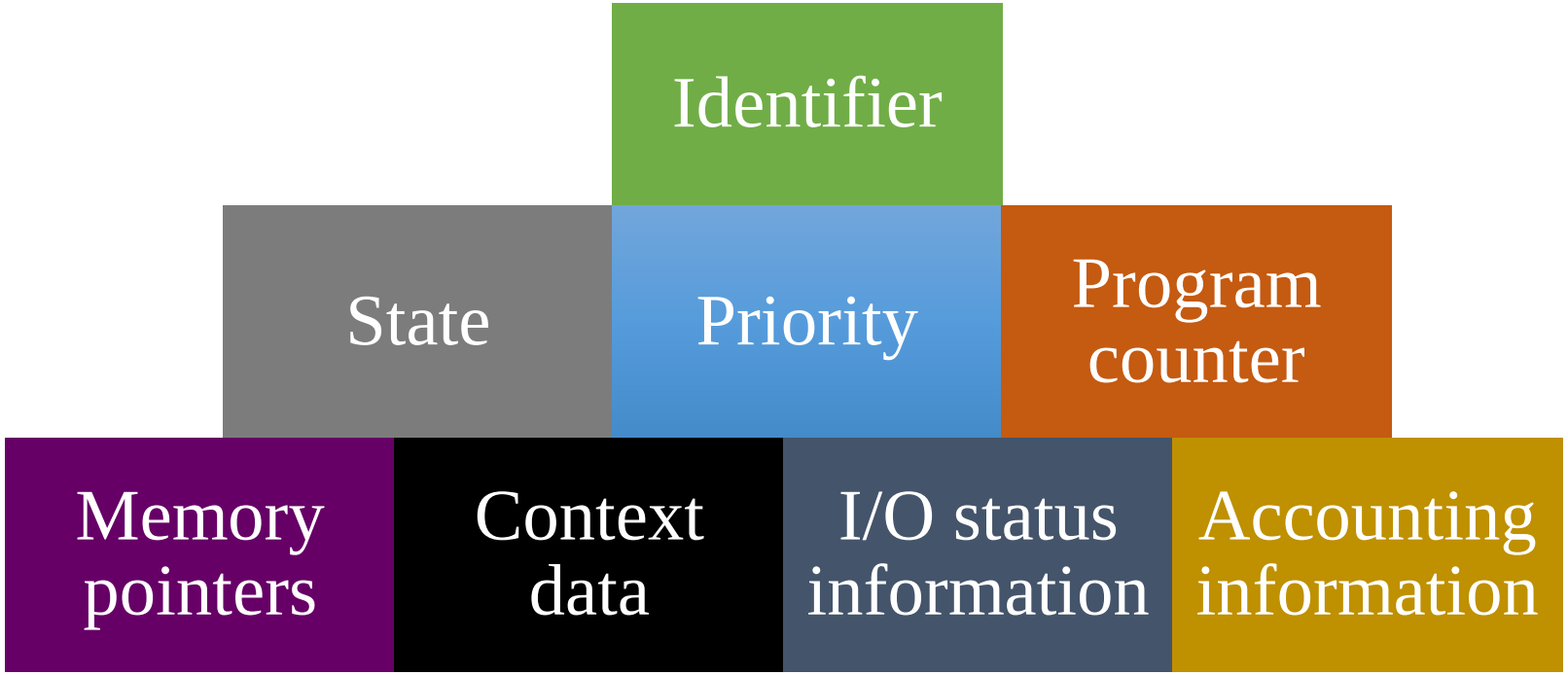
Process in Memory (1)

Multiple Processes in Memory

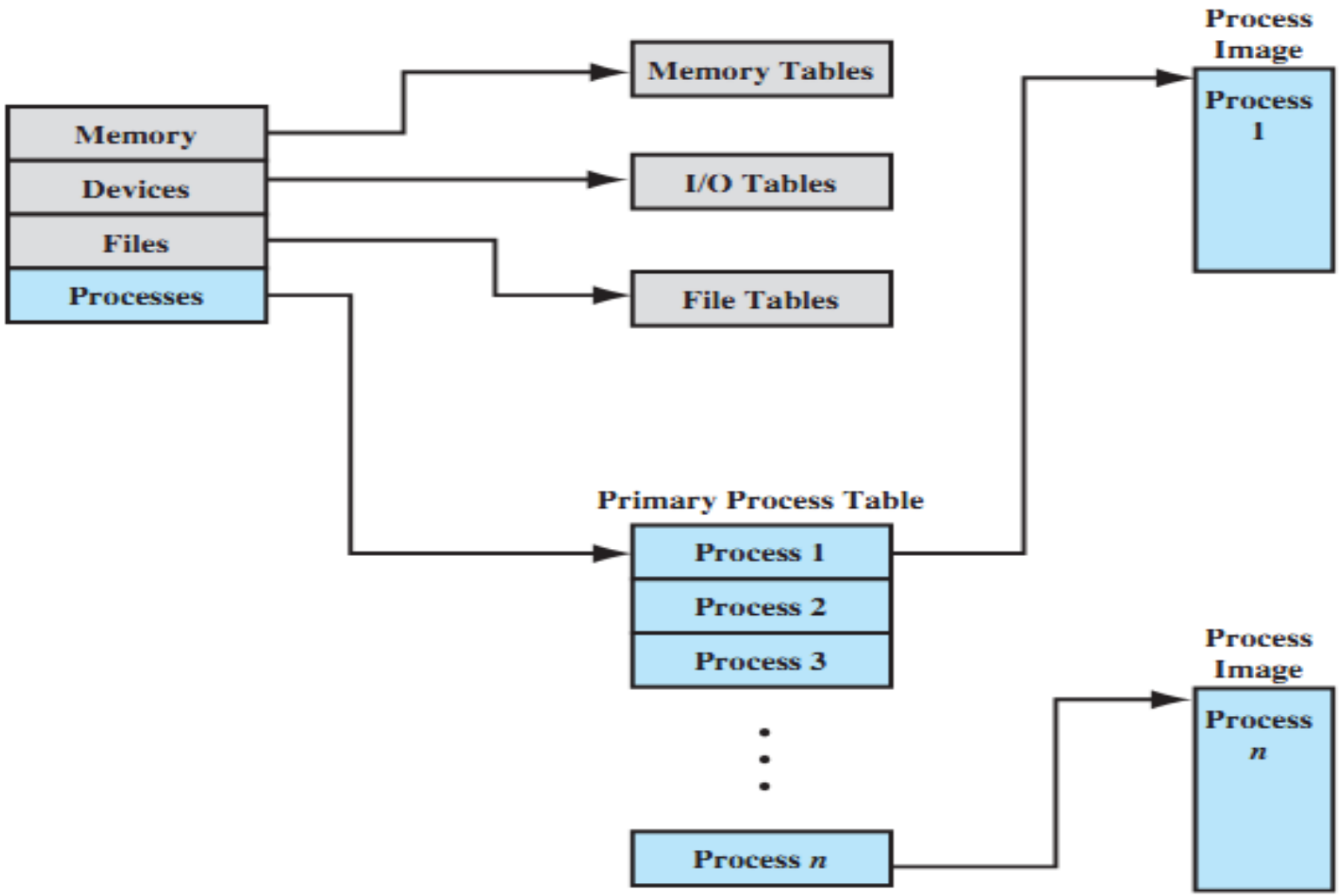


Process Elements

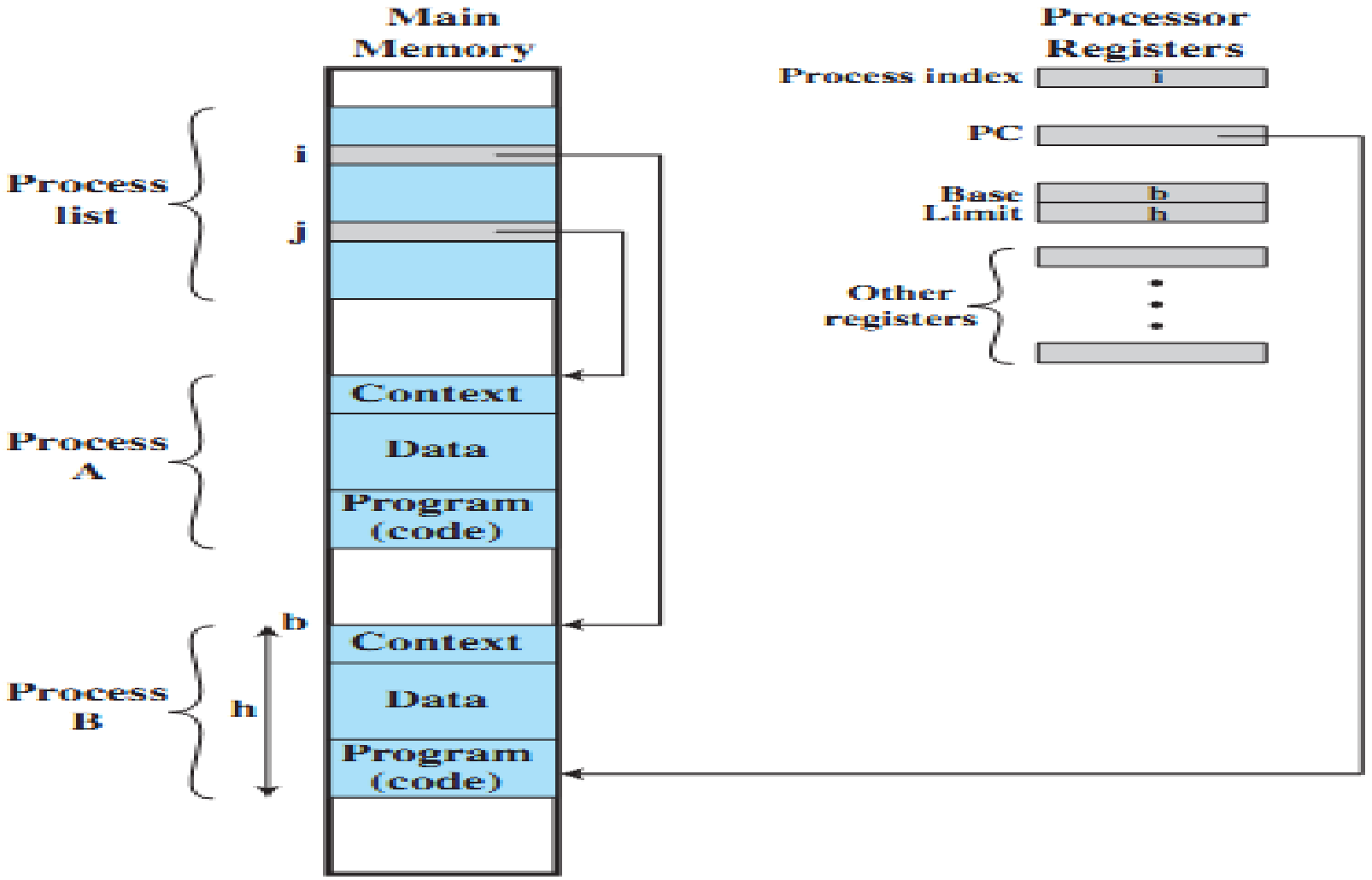
- While the program is executing, this process can be uniquely characterized by a number of elements, including:



General Structure of OS Control Tables



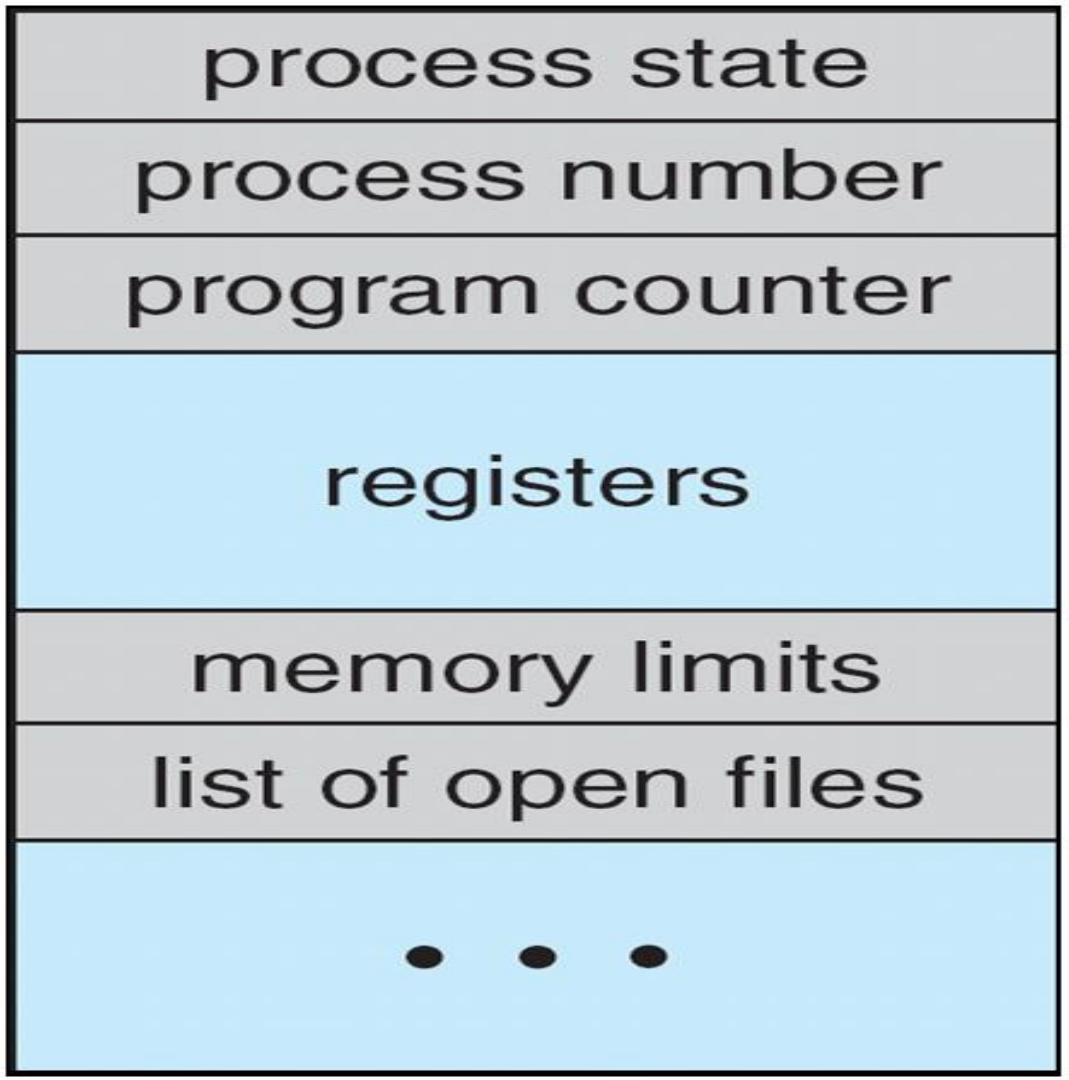
Typical Process Table Implementation



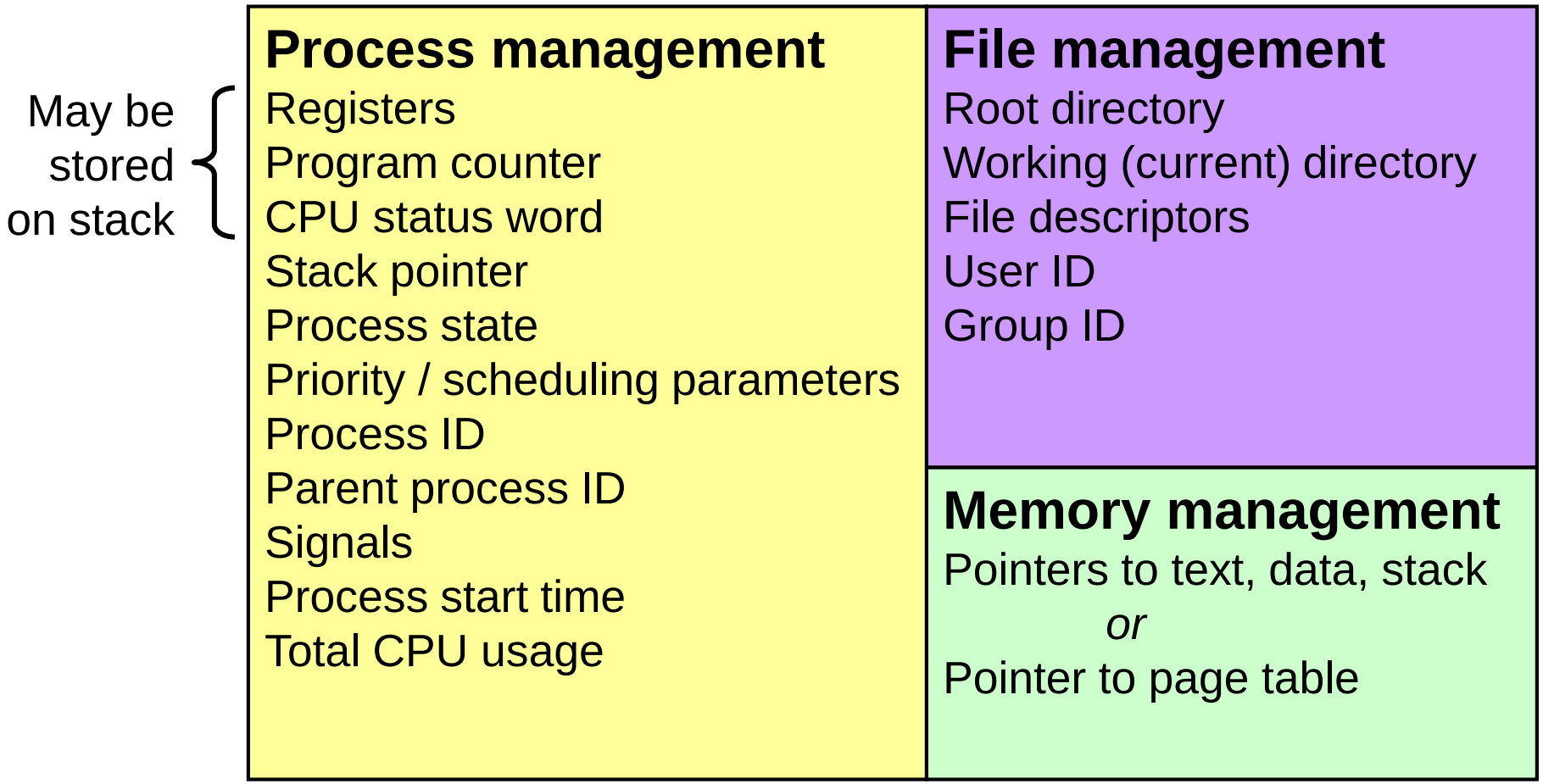
Process Attributes

- Process ID
- Parent process ID
- User ID
- Process state/priority
- Program counter
- CPU registers
- Memory management information
- I/O status information
- Access Control
- Accounting information

Typical process table entry



What's in a process table entry?



Fields of a typical process table entry

Components of Process Control Block (PCB)

- Process Control Block (PCB) – IBM name for information associated with each process – its context!
- PCB (execution context) is the data needed (process attributes) by OS to control process:
  1. Process location information
  2. Process identification information
  3. Processor state information
  4. Process control information

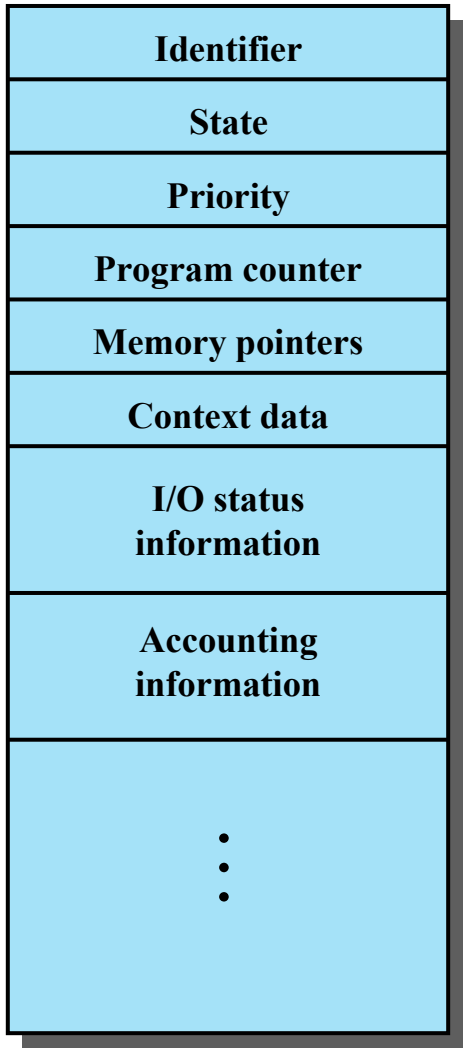


Figure 3.1 Simplified Process Control Block

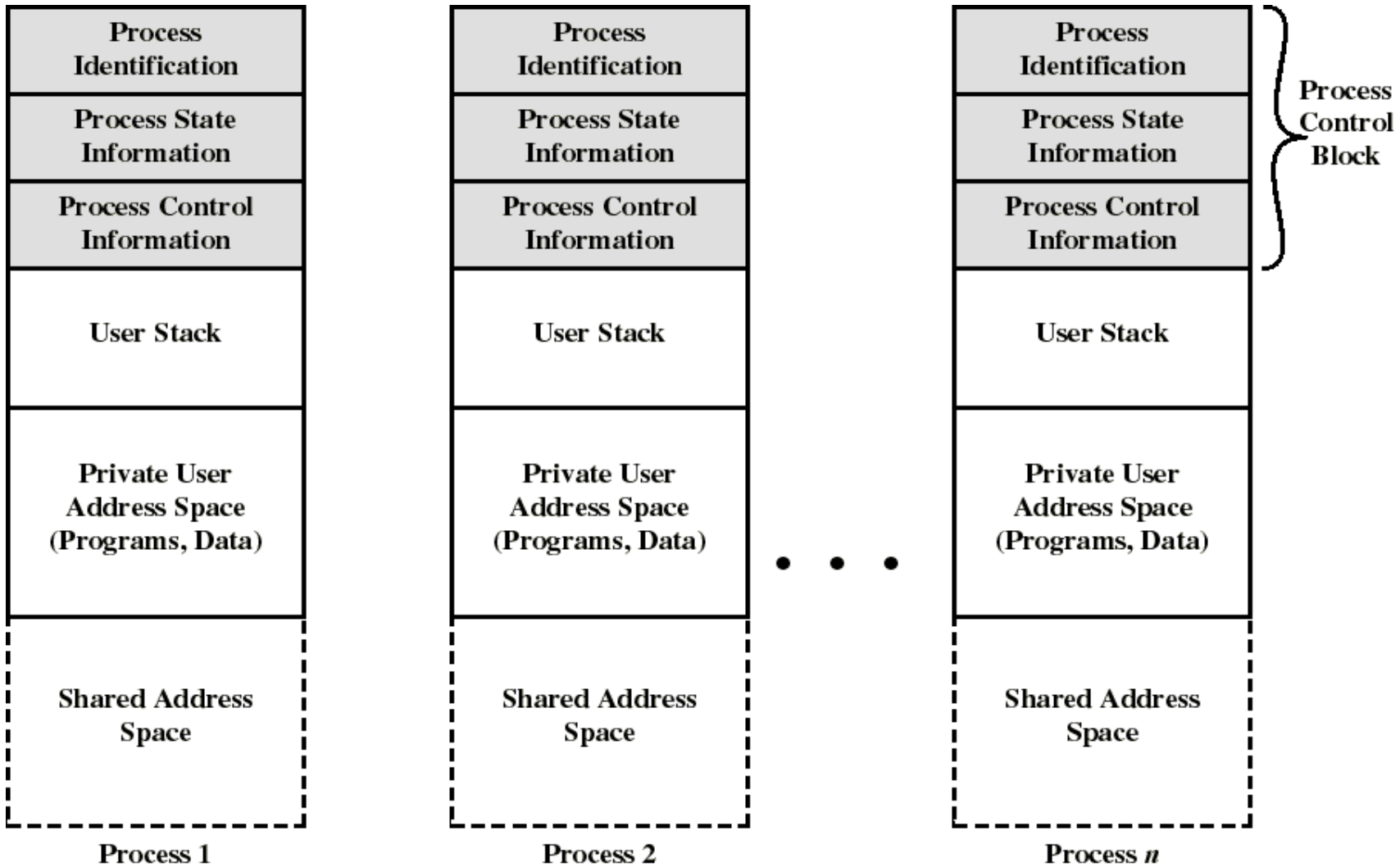
Role of the Process Control Block

- The most important data structure in an OS
  - Contains all of the information about a process that is needed by the OS
  - Blocks are read and/or modified by virtually every module in the OS
  - Defines the state of the OS
- Difficulty is not access, but protection
  - A bug in a single routine could damage process control blocks, which could destroy the system's ability to manage the affected processes
  - A design change in the structure or semantics of the process control block could affect a number of modules in the OS

Process Location Information

- Each process image in memory:
  - may not occupy a contiguous range of addresses (depends on memory management scheme used).
  - both a private and shared memory address space can be used.
- The location if each process image is pointed to by an entry in the process table.
- For the OS to manage the process, at least part of its image must be brought into main memory.

Process Images in Memory



Process Identification Information

- A few numeric identifiers may be used:
  - Unique process identifier (PID) –
    - indexes (directly or indirectly) into the process table.
  - User identifier (UID) –
    - the user who is responsible for the job.
  - Identifier of the process that created this process (PPID).
- Maybe symbolic names that are related to numeric identifiers.

Processor State Information

Consists of the contents of processor registers

- User-visible registers
- Control and status registers
- Stack pointers

Program status word (PSW)

- Contains condition codes plus other status information
- EFLAGS register is an example of a PSW used by any OS running on an x86 processor

Process Control Information

- scheduling and state information:
  - Process state (i.e., running, ready, blocked...)
  - Priority of the process
  - Event for which the process is waiting (if blocked).
- data structuring information
  - may hold pointers to other PCBs for process queues, parent-child relationships and other structures.

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

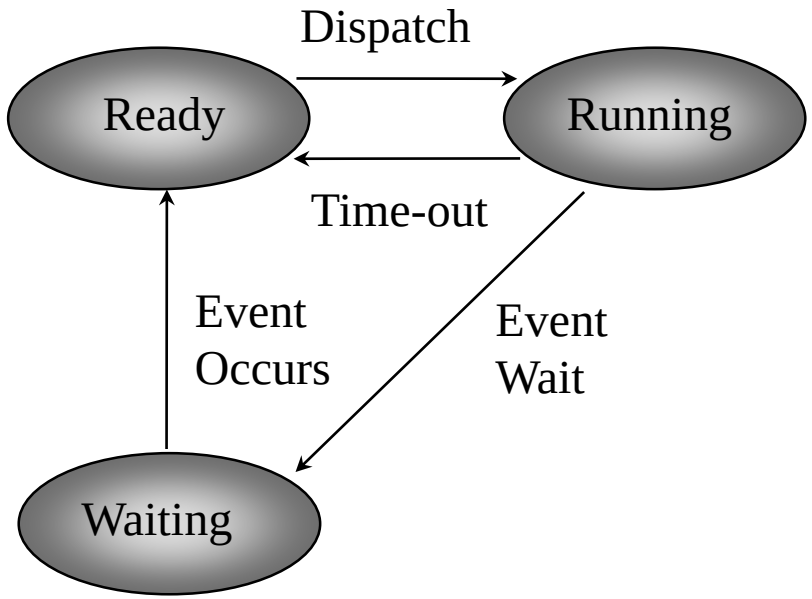
Process Control Information

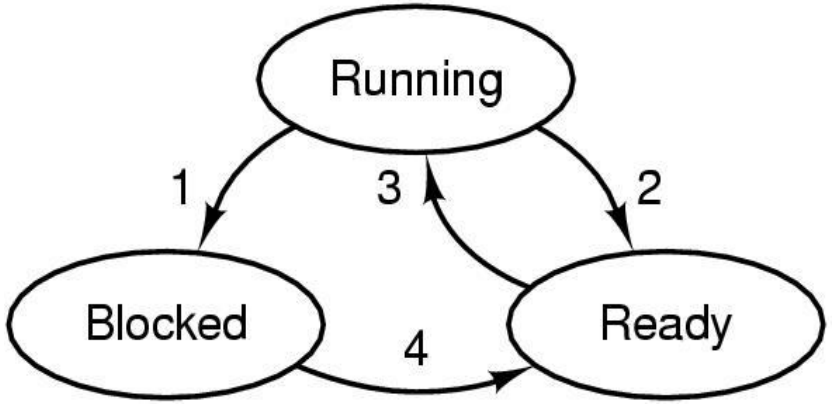
- Inter-process communication –
  - may hold flags and signals for IPC.
- Resource ownership and utilization –
  - resource in use: open files, I/O devices...
  - history of usage (of CPU time, I/O...).
- Process privileges (Access control) –
  - access to certain memory locations, to resources, etc…
- Memory management –
  - pointers to segment/page tables assigned to this process.

Process States

- Let us start with three states:
  1) Running state –
     - the process that gets executed (single CPU); its instructions are being executed.
  2) Ready state –
     - any process that is ready to be executed; the process is waiting to be assigned to a processor.
  3) Waiting/Blocked state –
     - when a process cannot execute until its I/O completes or some other event occurs.

A Three-state Process Model





1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

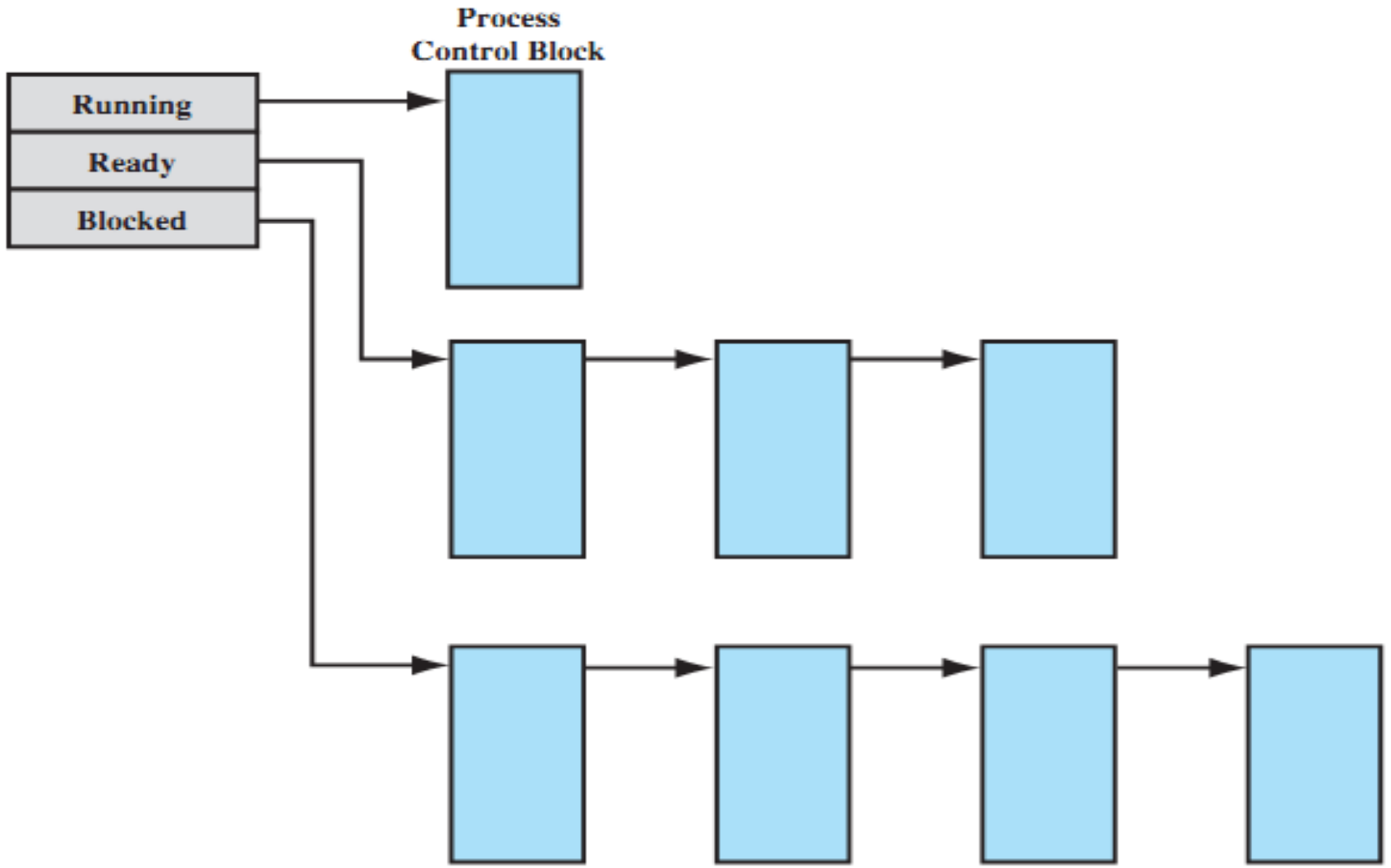
Process Transitions (1)

- Ready –> Running
  - When it is time, the dispatcher selects a new process to run.
- Running –> Ready
  - the running process has expired his time slot.
  - the running process gets interrupted because a higher priority process is in the ready state.

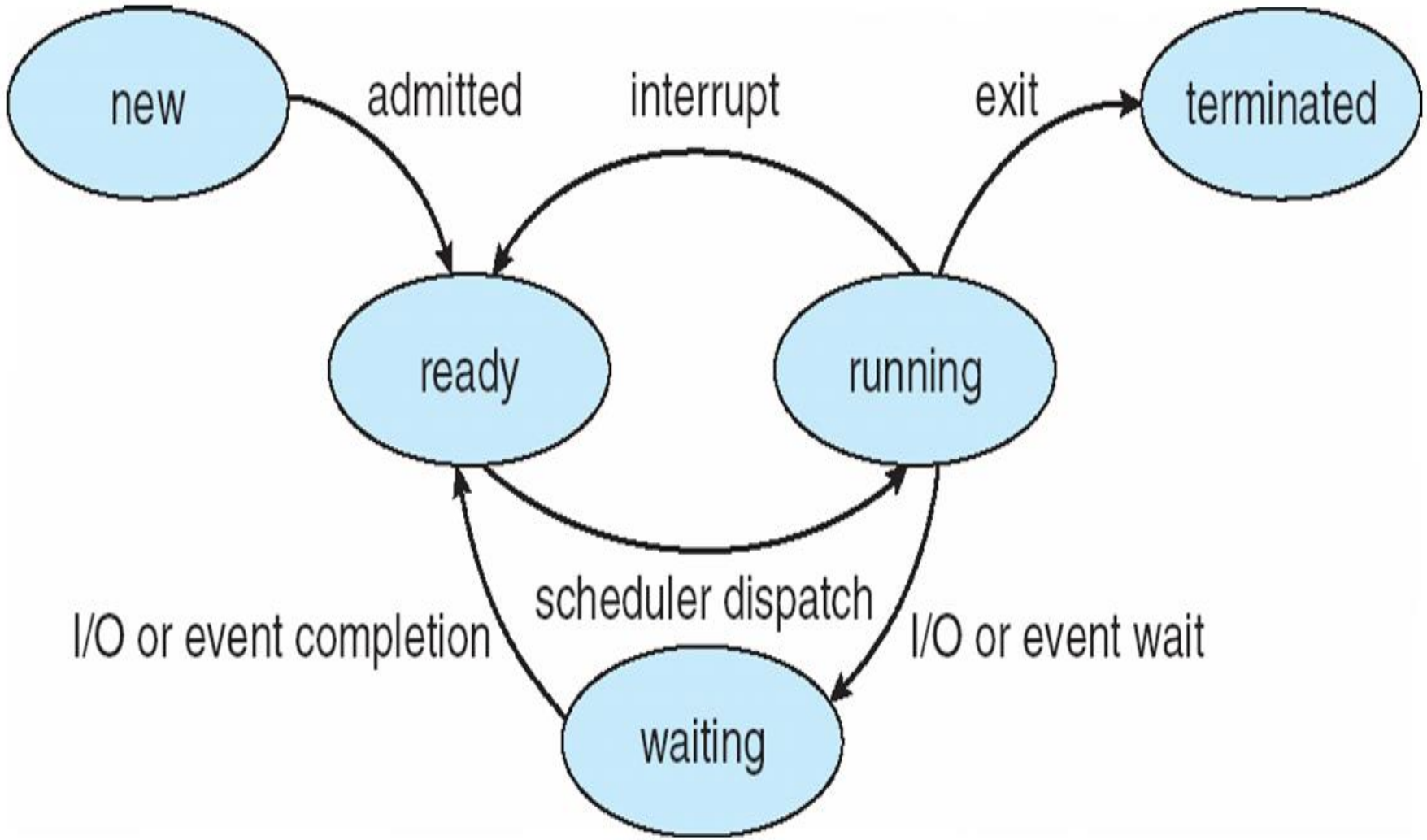
Process Transitions (2)

- Running –> Waiting
  - When a process requests something for which it must wait:
    - a service that the OS is not ready to perform.
    - an access to a resource not yet available.
    - initiates I/O and must wait for the result.
    - waiting for a process to provide input.
- Waiting –> Ready
  - When the event for which it was waiting occurs.

Process List Structures



Five-state Process Model



Other Useful States (1)

- New state –
  - OS has performed the necessary actions to create the process:
    - has created a process identifier.
    - has created tables needed to manage the process.
  - but has not yet committed to execute the process (not yet admitted):
    - because resources are limited.

Other Useful States (2)

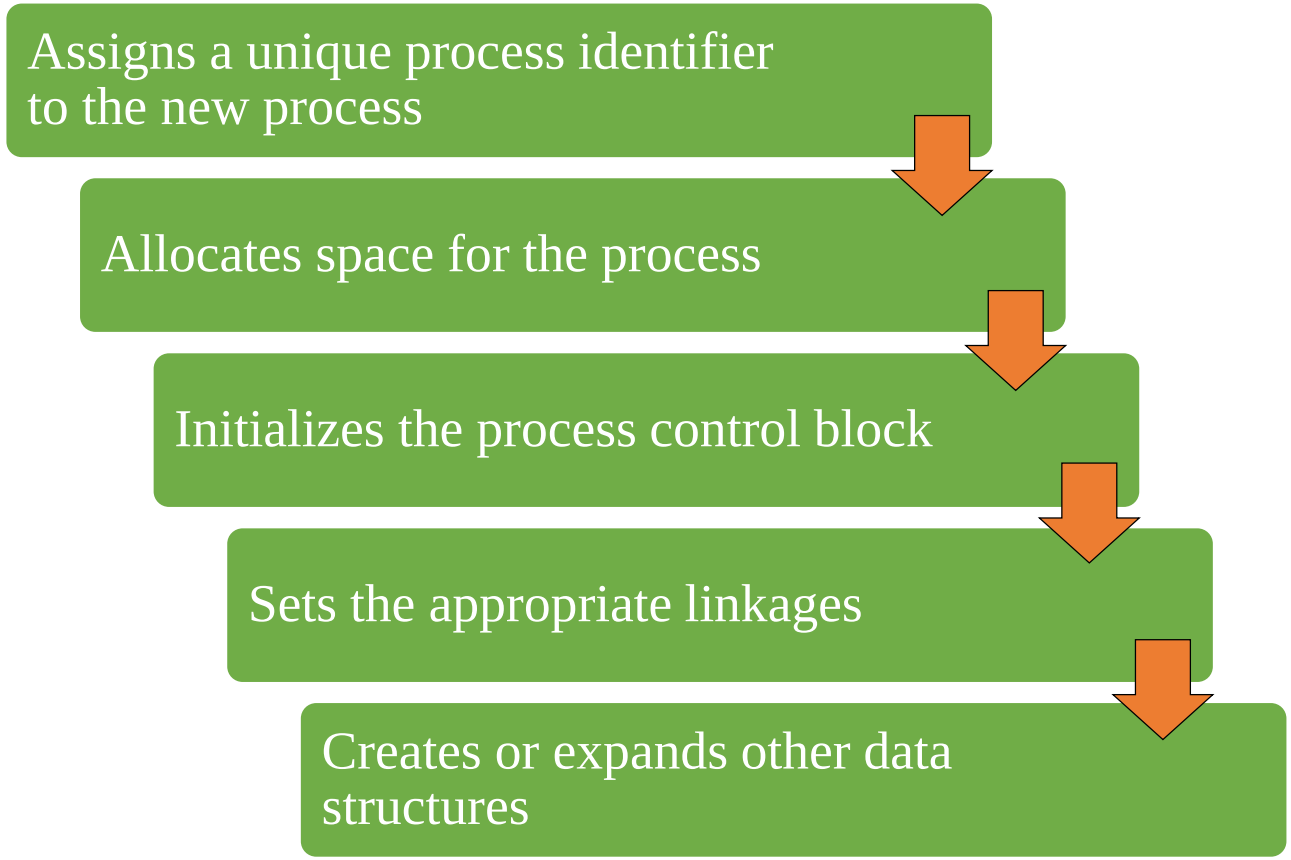
- Terminated state –
  - Program termination moves the process to this state.
  - It is no longer eligible for execution.
  - Tables and other info are temporarily preserved for auxiliary program –
    - Example: accounting program that cumulates resource usage for billing the users.
- The process (and its tables) gets deleted when the data is no more needed.

Reasons for Process Creation

- System initialization.
- Submission of a batch job.
- User logs on.
- Created by OS to provide a service to a user (e.g., printing a file).
- A user request to create a new process.
- Spawned by an existing process
  - a program can dictate the creation of a number of processes.

Process Creation

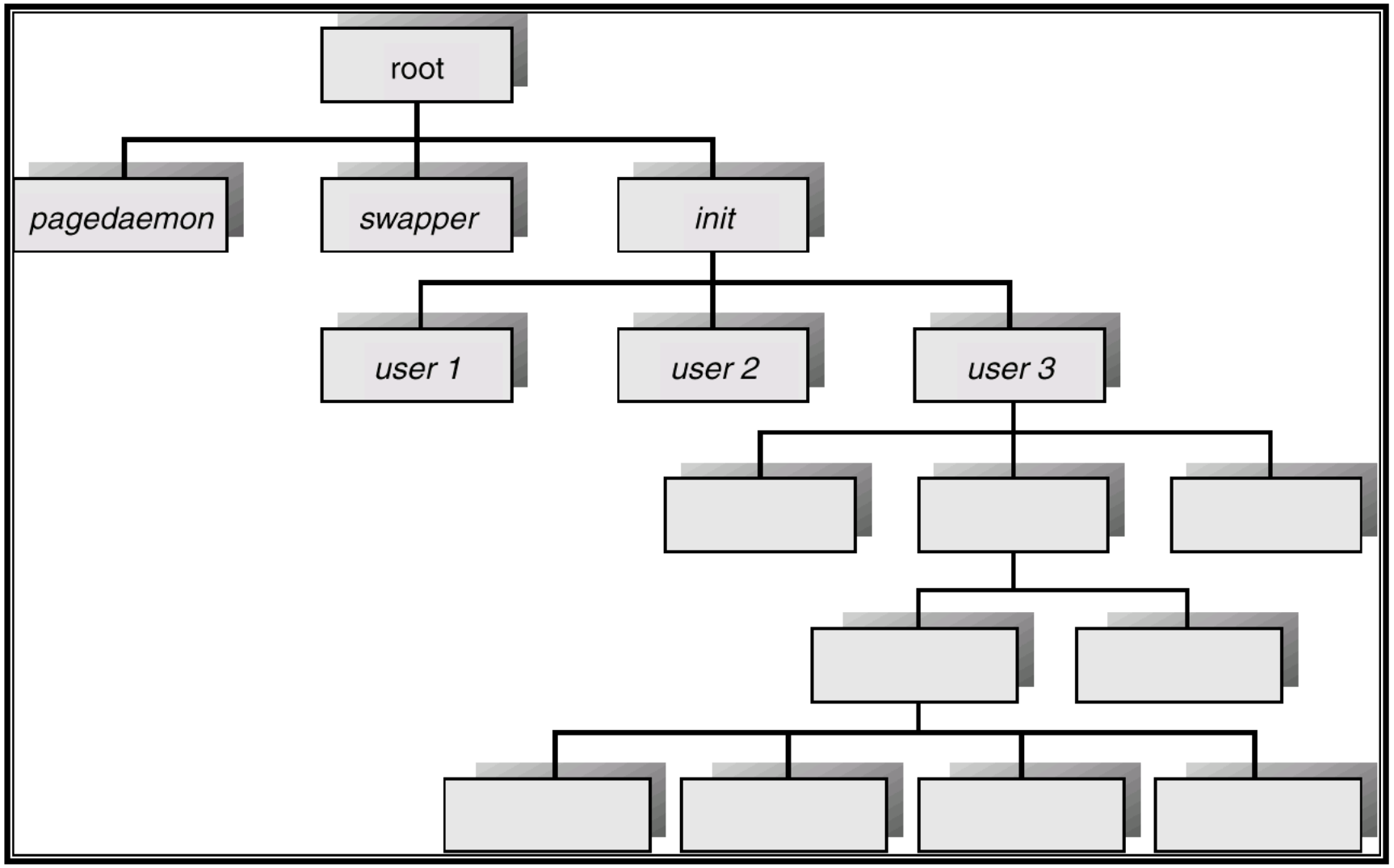
- Once the OS decides to create a new process it:



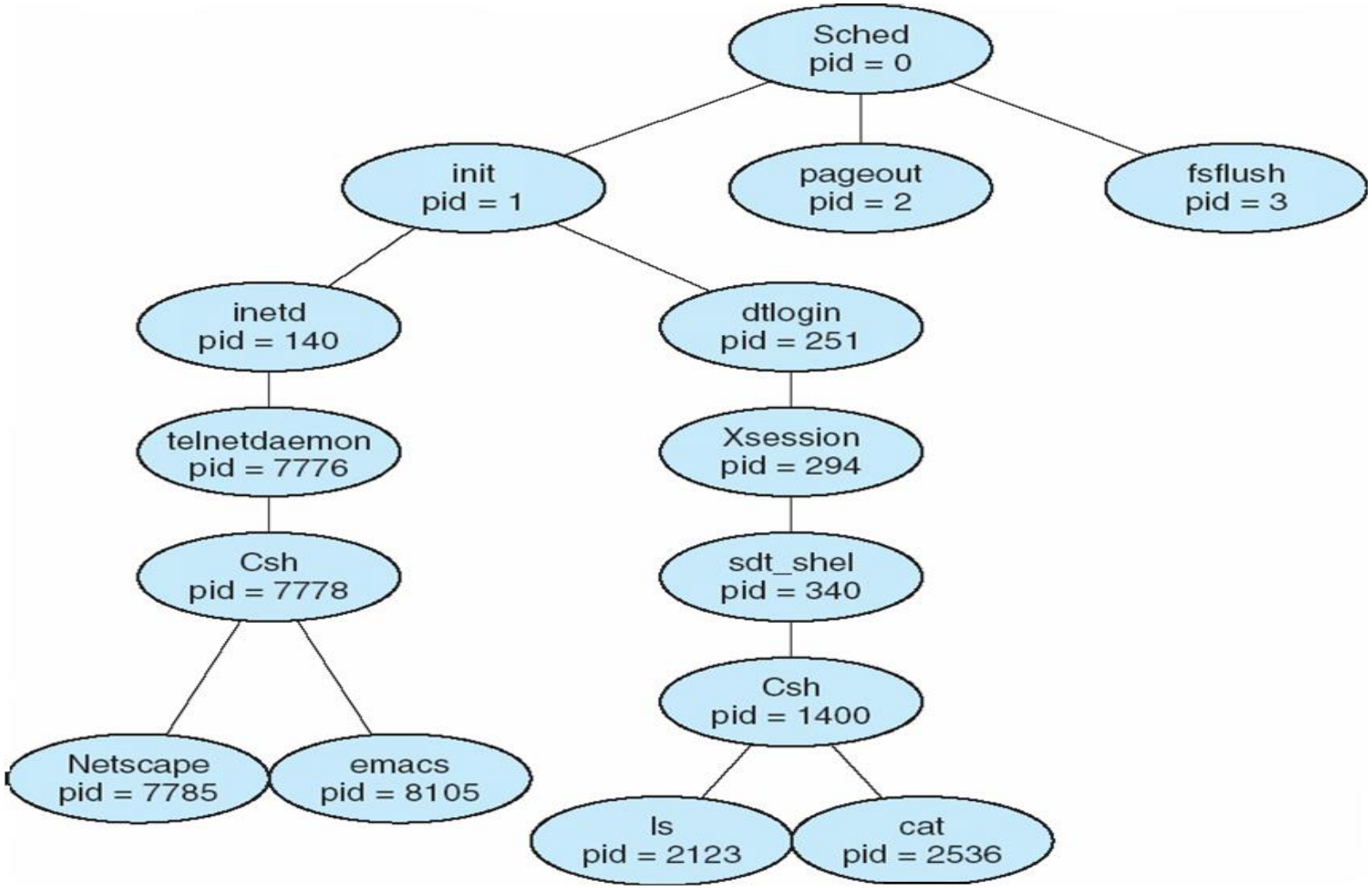
Process Creation (1)

- Parent process create children processes, which, in turn create other processes, forming a tree of processes.
- Possible resource sharing:
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- Possible execution:
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

A tree of processes on UNIX



A tree of processes on typical Solaris

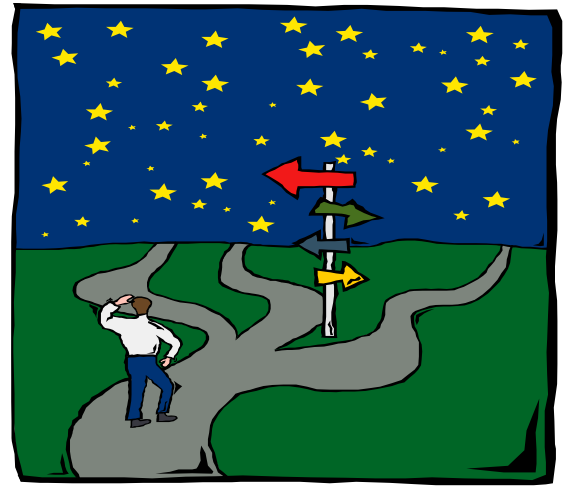


Process Creation (2)

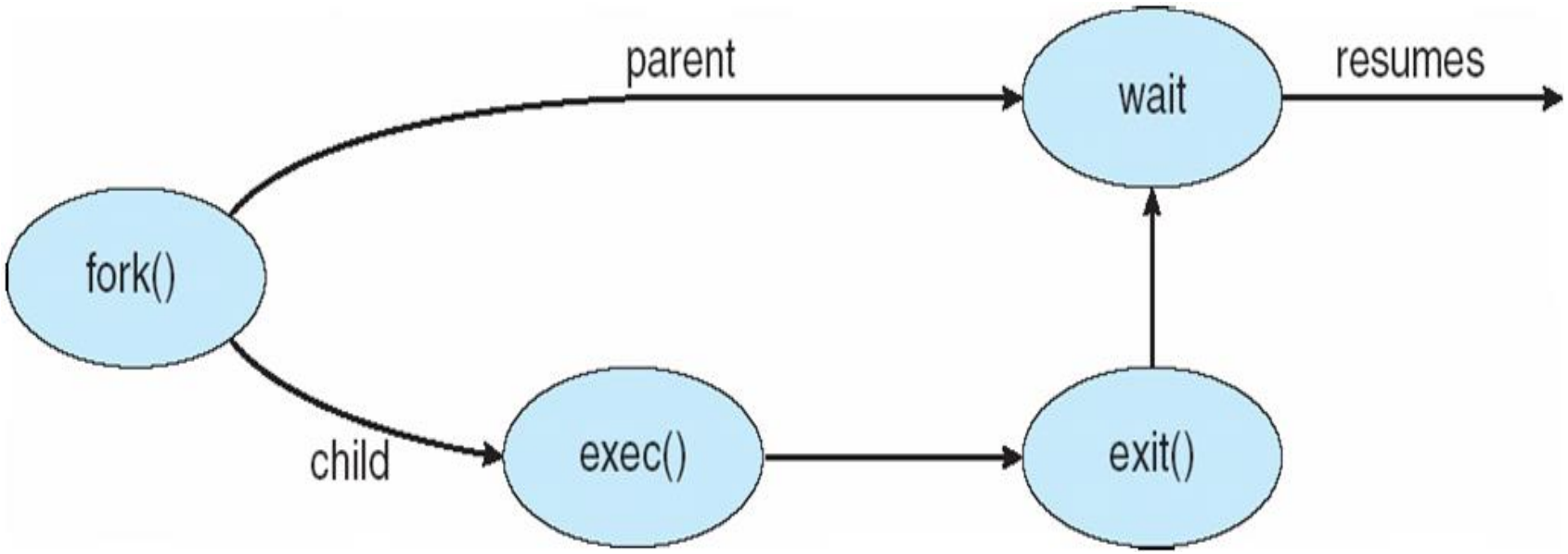
- Assign a unique process identifier (PID).
- Allocate space for the process image.
- Initialize process control block
  - many default values (e.g., state is New, no I/O devices or files...).
- Set up appropriate linkages
  - Ex: add new process to linked list used for the scheduling queue.

Process Creation (3)

- 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.



Process Creation (3)



C Program Forking Separate Process

```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
    }
    else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
    }
}
```

When does a process get terminated?

- Batch job issues *Halt* instruction.
- User logs off.
- Process executes a service request to terminate.
- Parent kills child process.
- Error and fault conditions.

Reasons for Process Termination (1)

- Normal/Error/Fatal exit.
- Time limit exceeded
- Memory unavailable
- Memory bounds violation
- Protection error
  - example: write to read-only file
- Arithmetic error
- Time overrun
  - process waited longer than a specified maximum for an event.

Reasons for Process Termination (2)

- I/O failure
- Invalid instruction
  - happens when trying to execute data.
- Privileged instruction
- Operating system intervention
  - such as when deadlock occurs.
- Parent request to terminate one child.
- Parent terminates so child processes terminate.

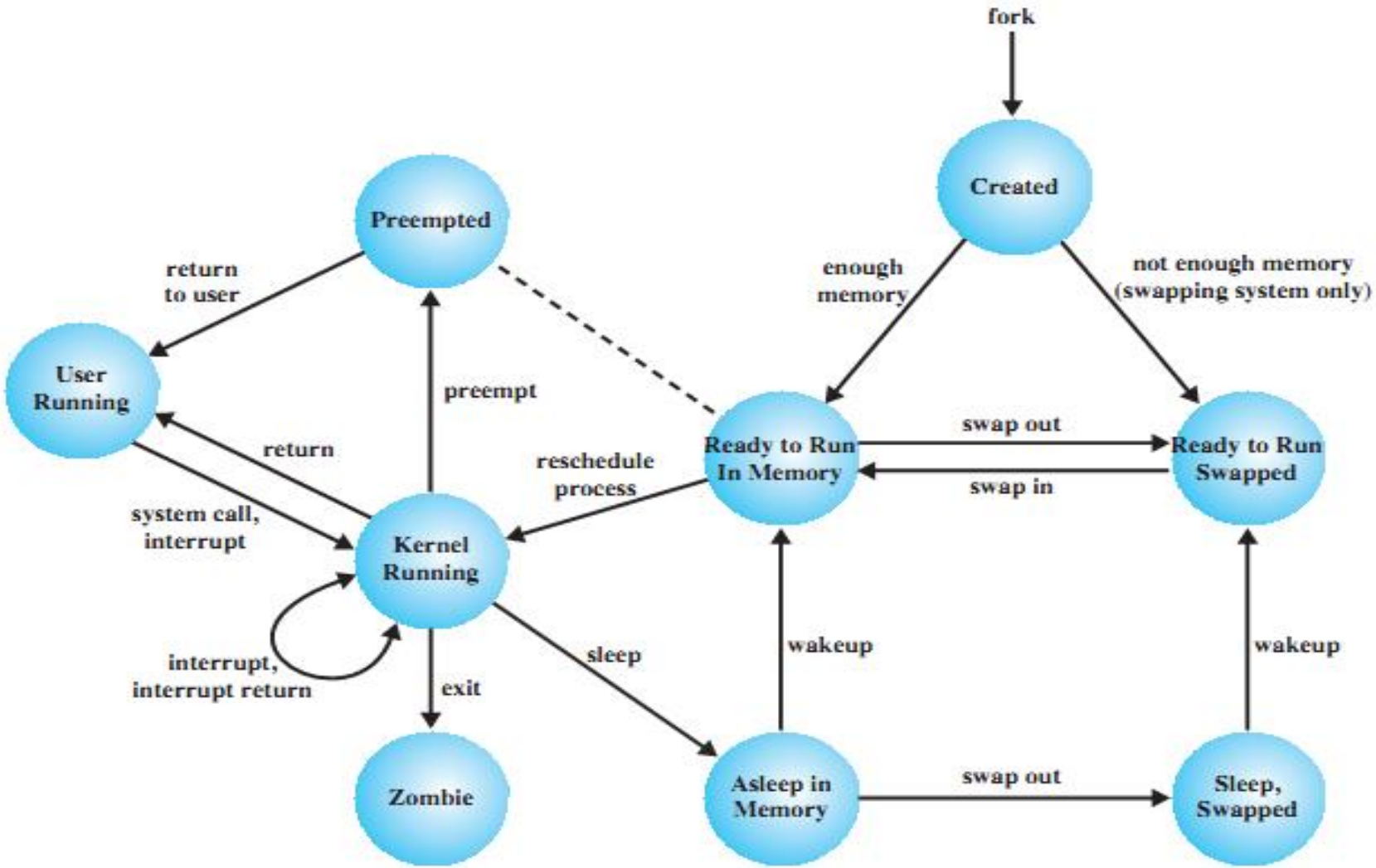
Process Termination

- Process executes last statement and asks the operating system to terminate it (**exit**):
  - Output data from child to parent (via **wait**).
  - Process' resources are deallocated by operating system.
- Parent may terminate execution of child processes (**abort**):
  - Child has exceeded allocated resources.
  - Mission assigned to child is no longer required.
  - If Parent is exiting:
    - Some OSs do not allow child to continue if its parent terminates.
    - Cascading termination – all children terminated.

UNIX SRV4 Process States

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

UNIX SVR4 States Process Model



ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Process Scheduling and Switching

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Processor Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing.
- Process scheduler selects among available processes for next execution on CPU.
- Maintains scheduling queues of processes:
  - Job queue – set of all processes in the system.
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute.
  - Device queues – set of processes waiting for an I/O device.
- Processes migrate among the various queues.

Types of Schedulers

1. Long-term scheduler (jobs scheduler) – selects which programs/processes should be brought into the ready queue.
2. Medium-term scheduler (emergency scheduler) – selects which job/process should be swapped out if system is loaded.
3. Short-term scheduler (CPU scheduler) – selects which process should be executed next and allocates CPU.

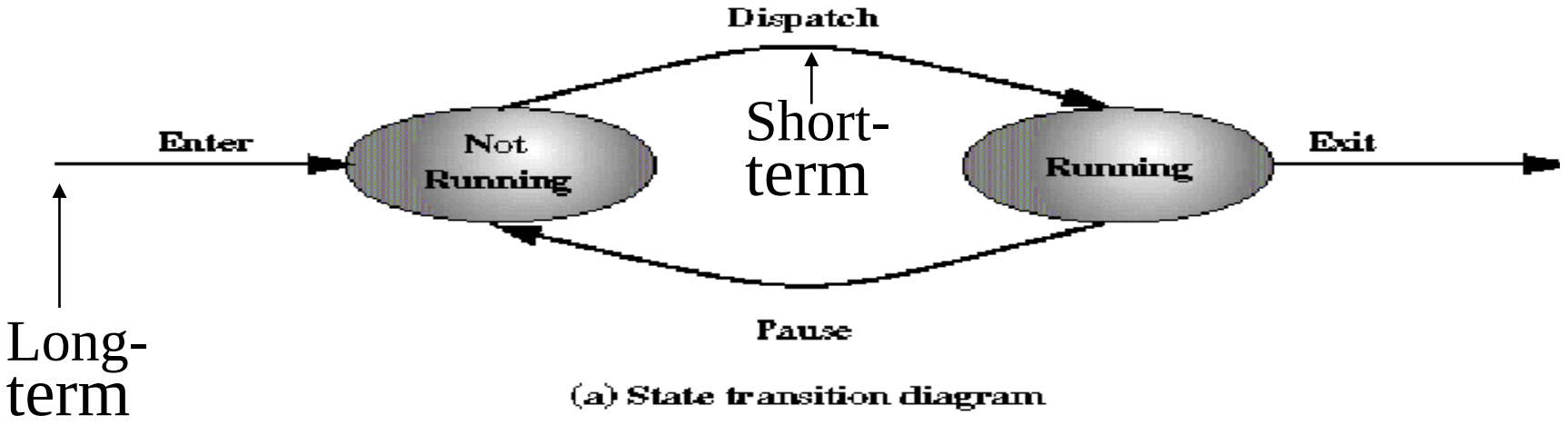
Long-Term Scheduling

- Determines which programs are admitted to the system for processing.
- Controls the degree of multiprogramming.
- If more processes are admitted:
  - less likely that all processes will be blocked – better CPU usage.
  - each process has less fraction of the CPU.
- Long-term scheduler strives for good process mix.

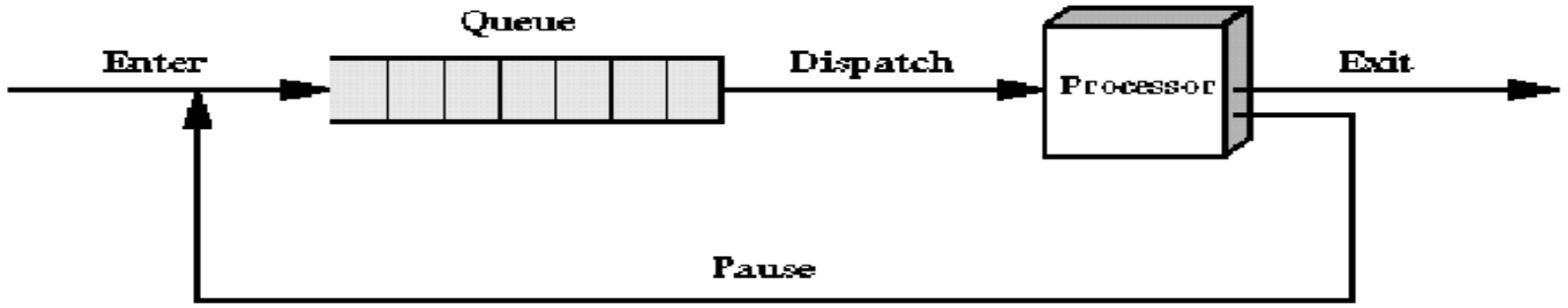
Short-Term Scheduling

- Determines which process is going to execute next (also called CPU scheduling).
- The short term scheduler is also known as the dispatcher (which is part of it).
- Is invoked on a event that may lead to choose another process for execution:
  - clock interrupts
  - I/O interrupts
  - operating system calls and traps
  - signals

Long/Short-Term Scheduling



(a) State transition diagram



(b) Queuing diagram

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

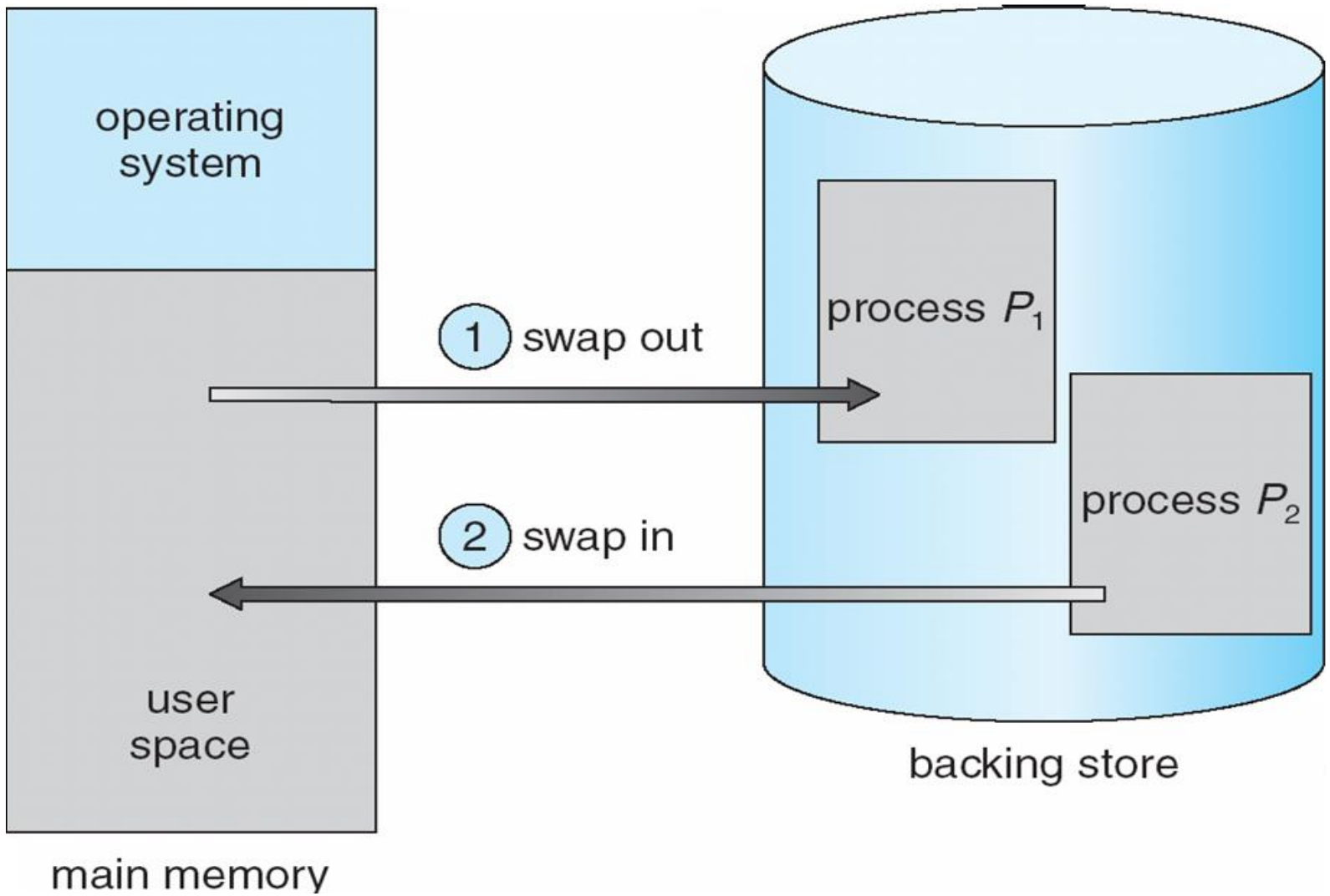
Aspects of Schedulers

- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).
- The long-term scheduler controls the degree of multiprogramming.
- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast).
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts.
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts.

Medium-Term Scheduling

- So far, all processes have to be (at least partly) in main memory.
- Even with virtual memory, keeping too many processes in main memory will deteriorate the system's performance.
- The OS may need to swap out some processes to disk, and then later swap them back in.
- Swapping decisions based on the need to manage multiprogramming.

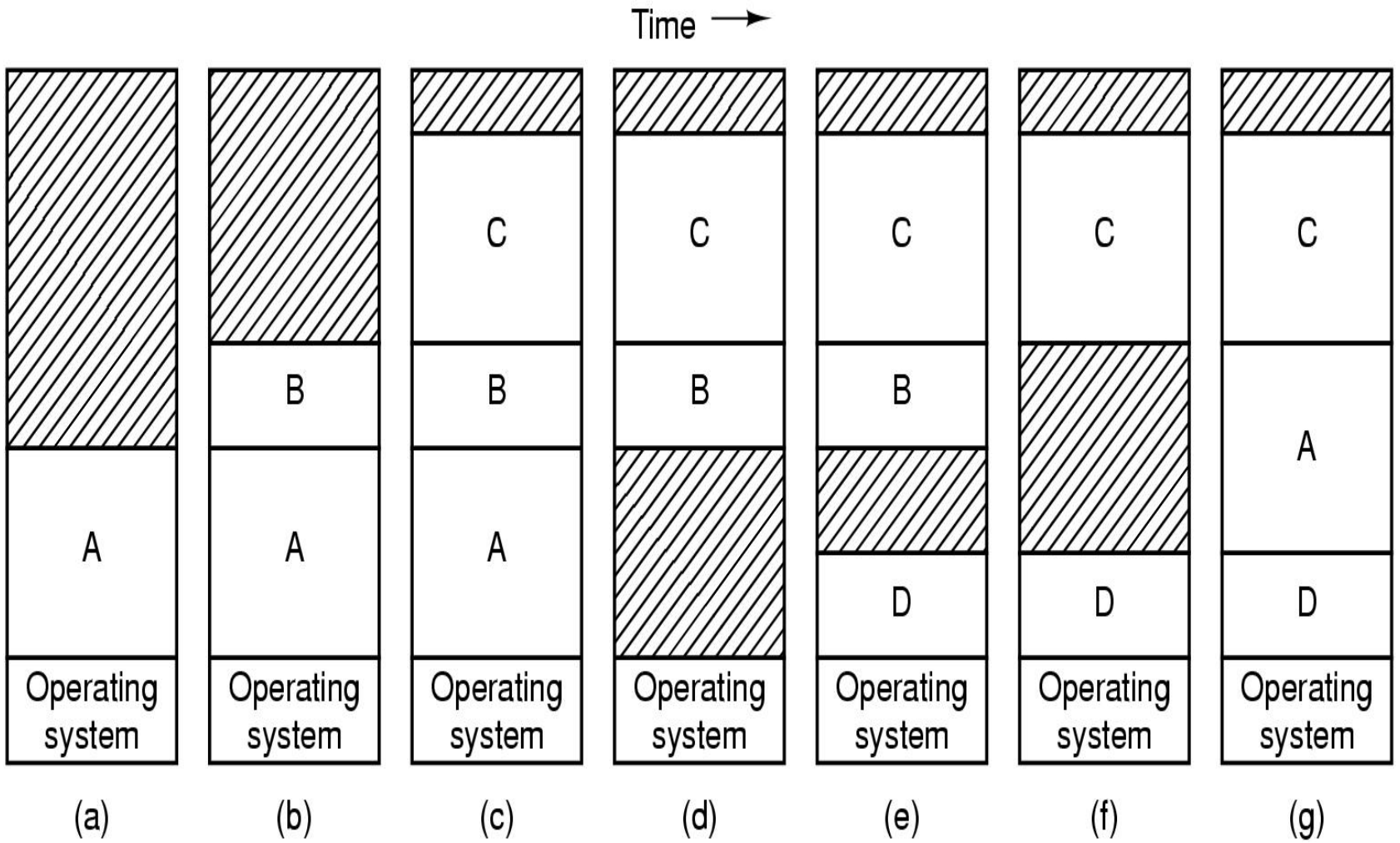
Schematic View of Swapping



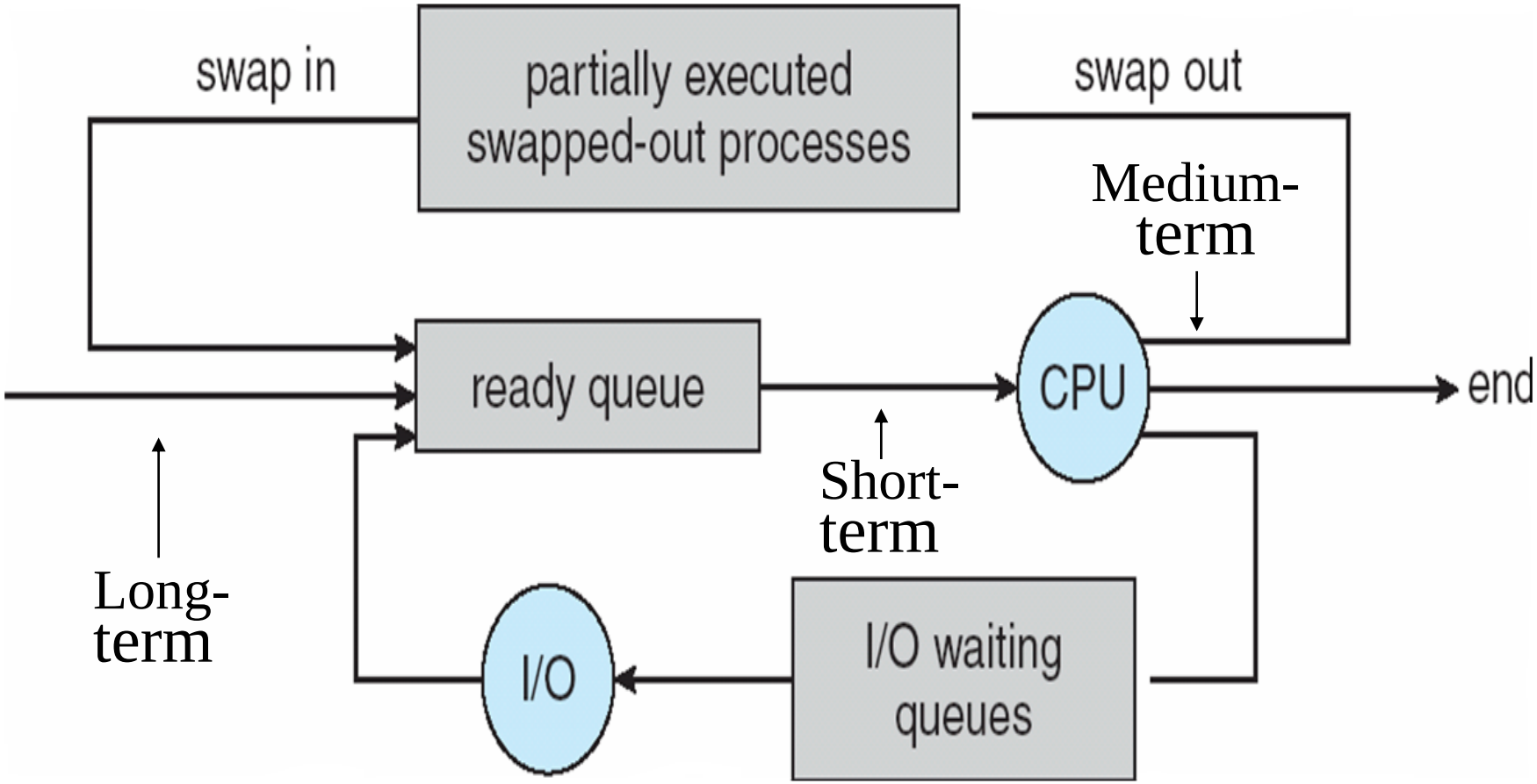
Dynamics of Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Swapping Example



Addition of Medium Term Scheduling

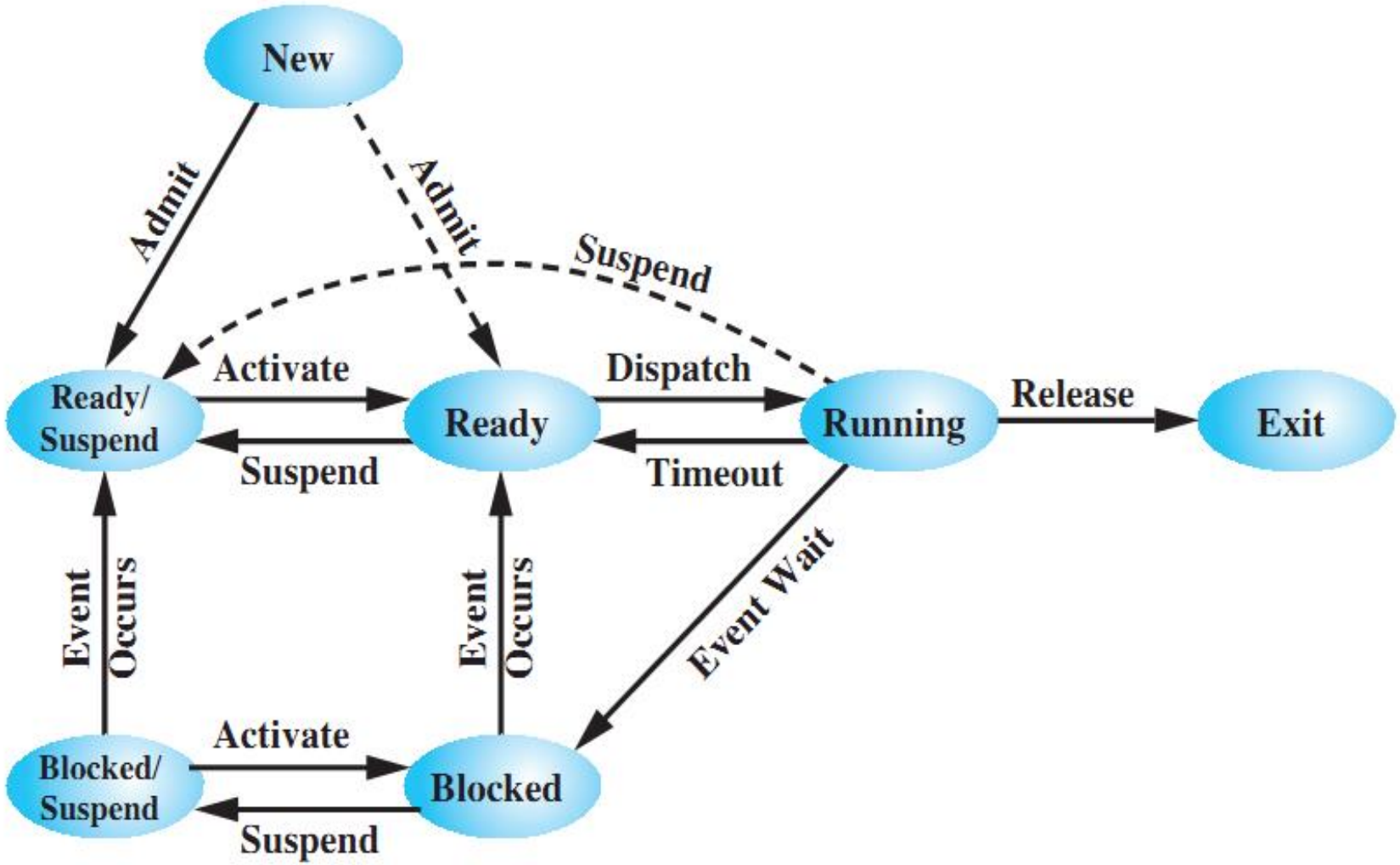


ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Support for Swapping

- The OS may need to suspend some processes, i.e., to swap them out to disk and then swap them back in.
- We add 2 new states:
  - Blocked Suspend: blocked processes which have been swapped out to disk.
  - Ready Suspend: ready processes which have been swapped out to disk.

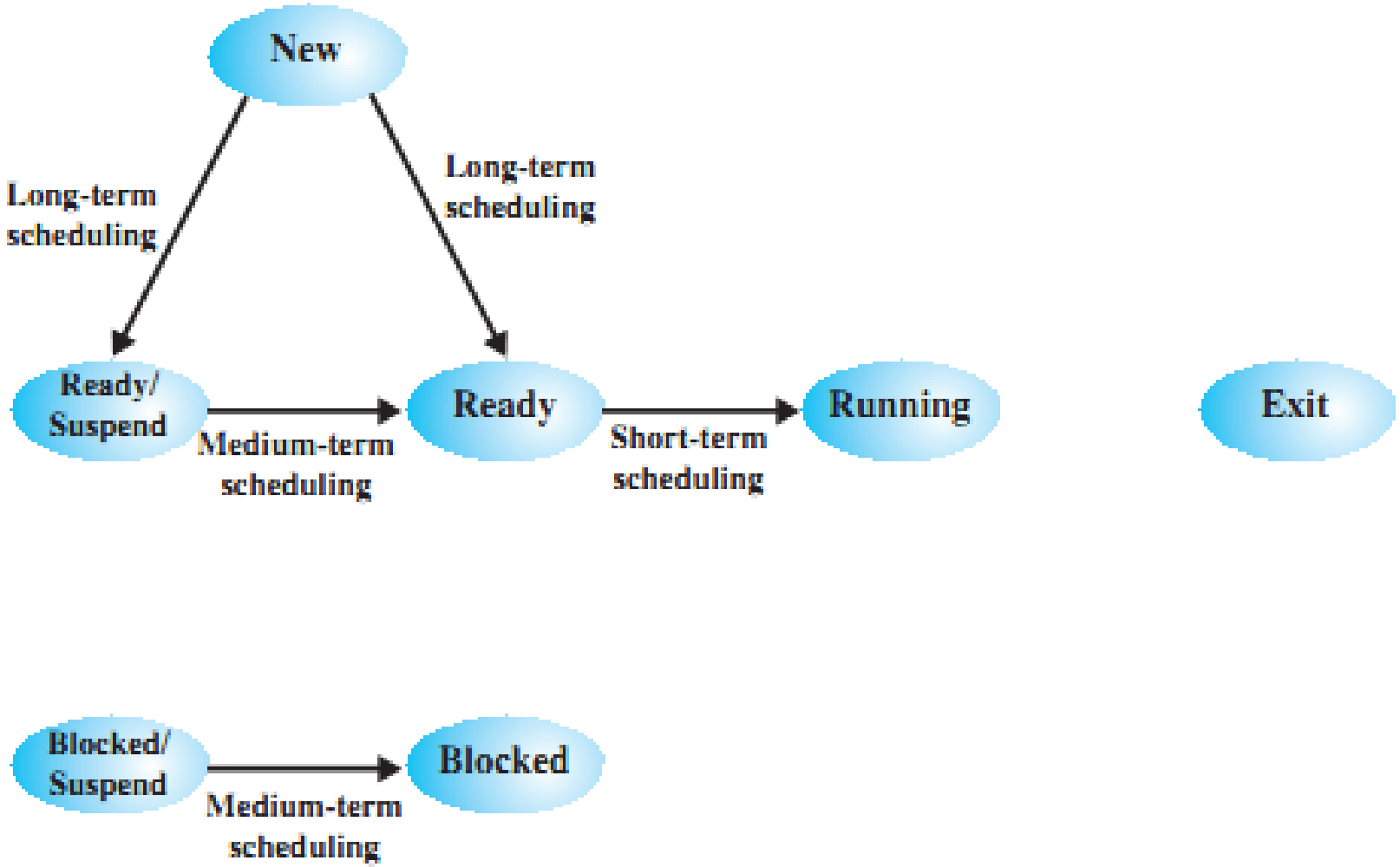
A Seven-state Process Model



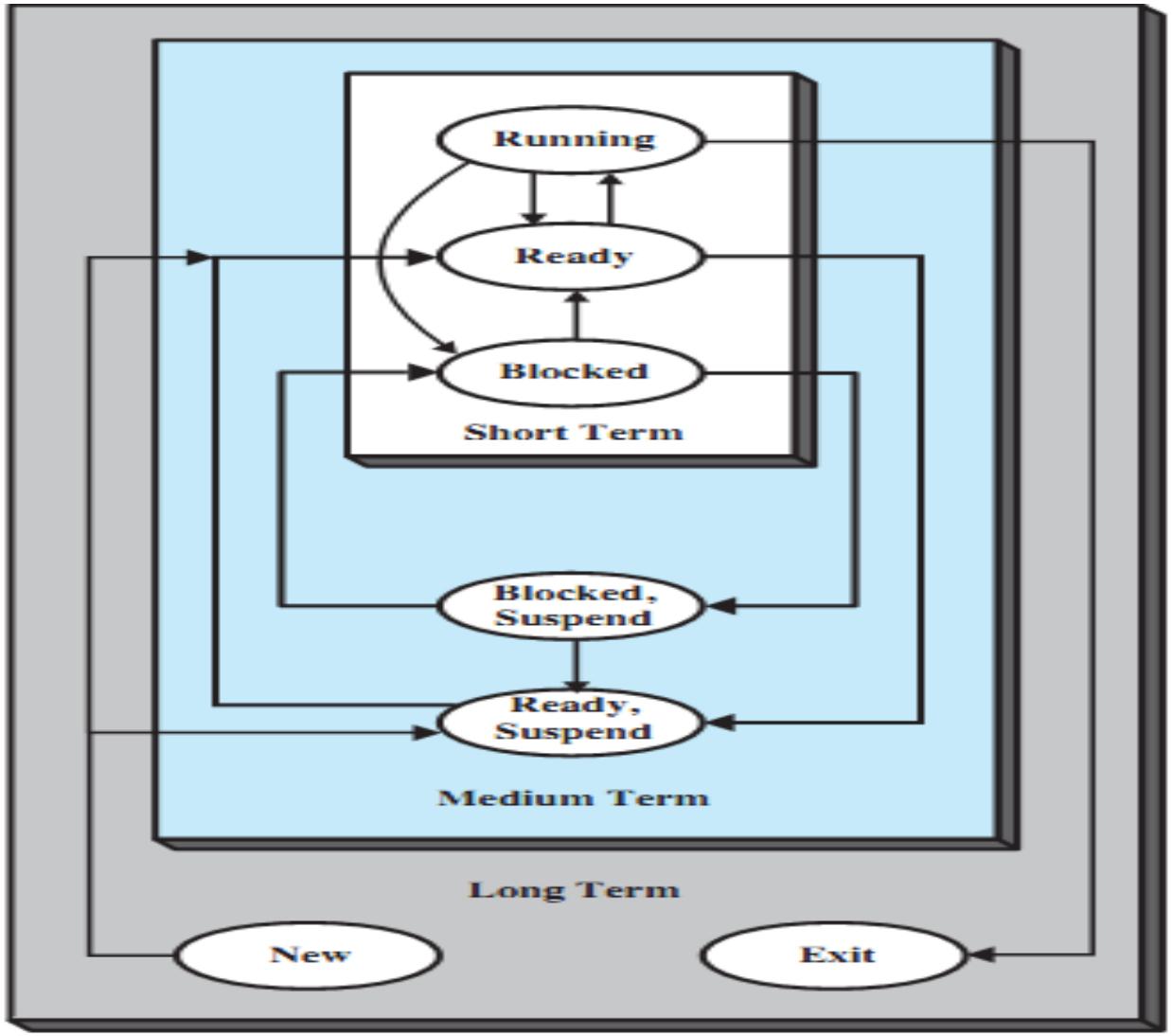
New state transitions

- Blocked —> Blocked Suspend
  - When all processes are blocked, the OS will make room to bring a ready process in memory.
- Blocked Suspend —> Ready Suspend
  - When the event for which it has been waiting occurs (state info is available to OS).
- Ready Suspend —> Ready
  - when no more ready processes in main memory.
- Ready —> Ready Suspend (unlikely)
  - When there are no blocked processes and must free memory for adequate performance.

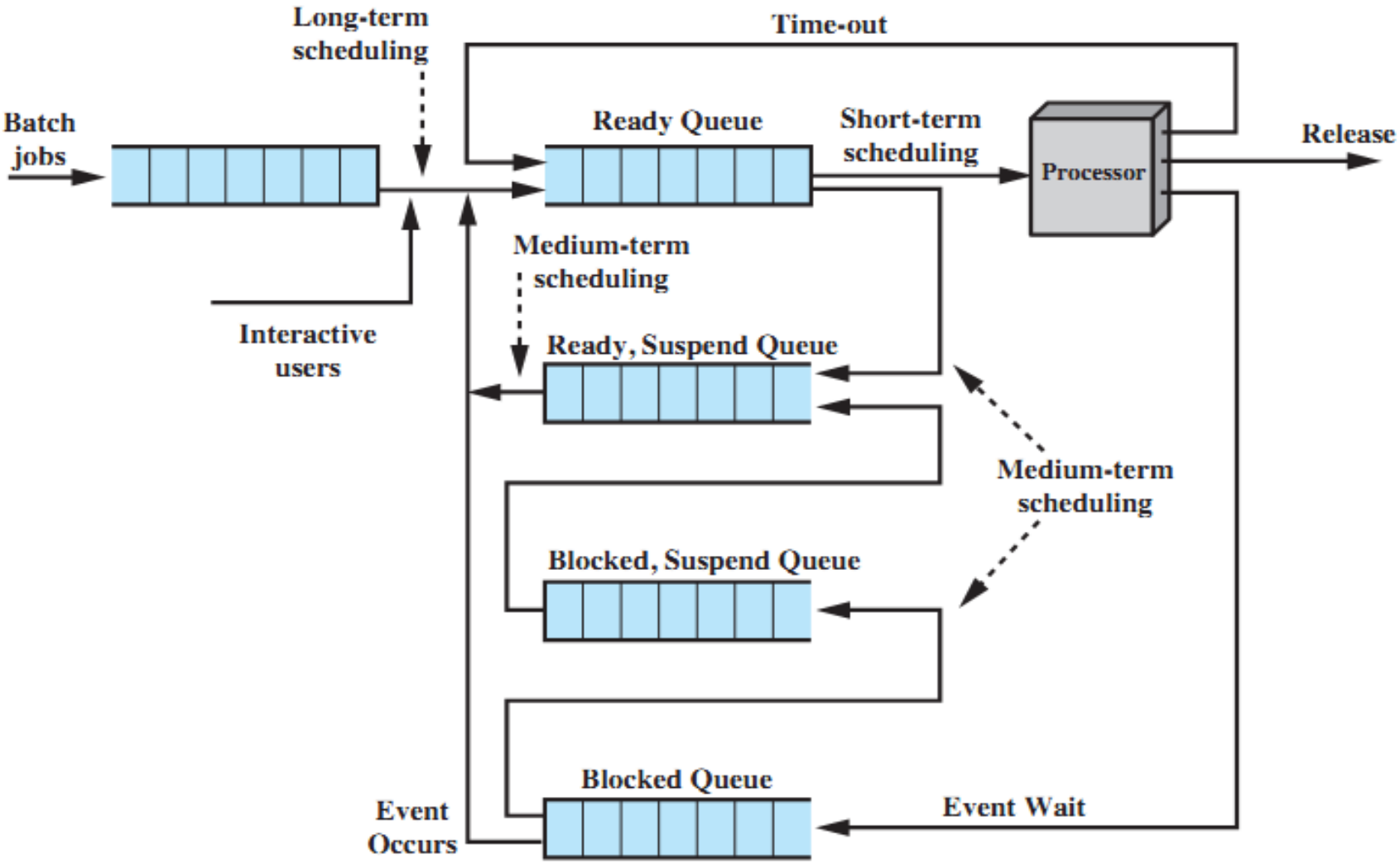
Classification of Scheduling Activity



Another view of the 3 levels of scheduling



Queuing Diagram for Scheduling



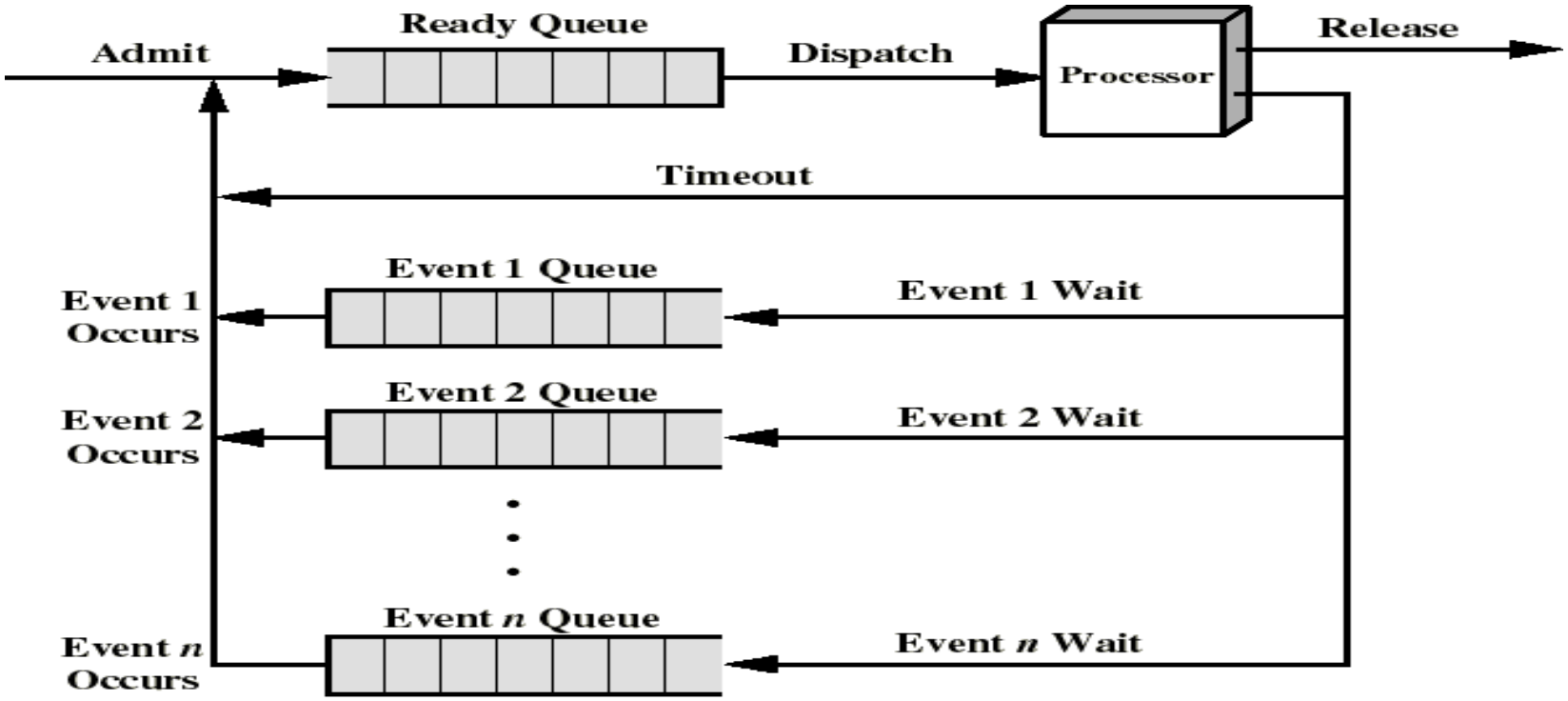
Dispatcher (short-term scheduler)

- Is an OS program that moves the processor from one process to another.
- It prevents a single process from monopolizing processor time.
- It decides who goes next according to a scheduling algorithm.
- The CPU will always execute instructions from the dispatcher while switching from process A to process B.

Process Scheduling Queues

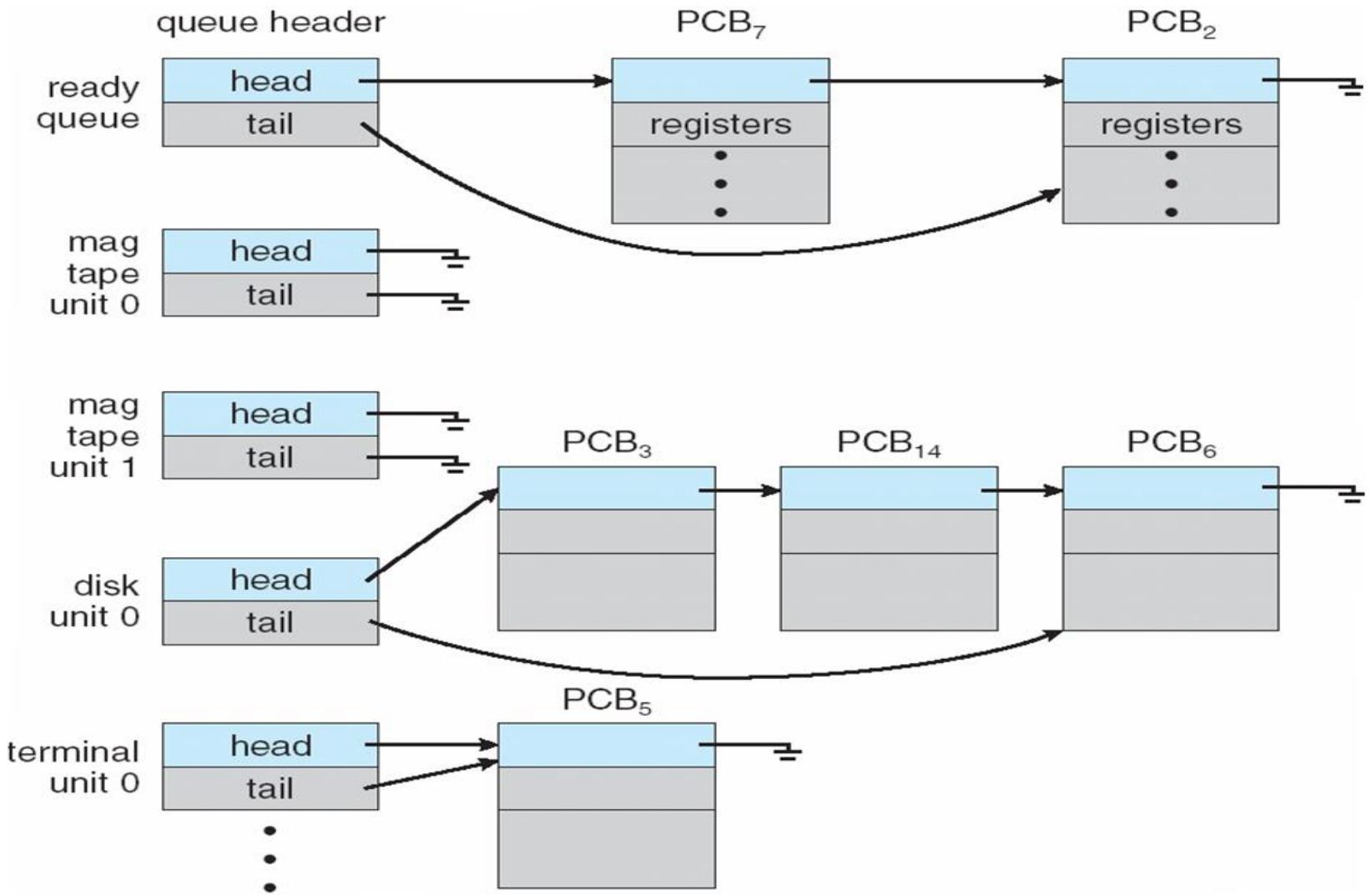
- Process queue – set of *all* processes in the system.
- Ready queue – set of processes residing in main memory, ready and waiting to execute.
- Device queues – set of processes waiting for an I/O device.
- Processes migrate among the various queues.

A Queuing Discipline

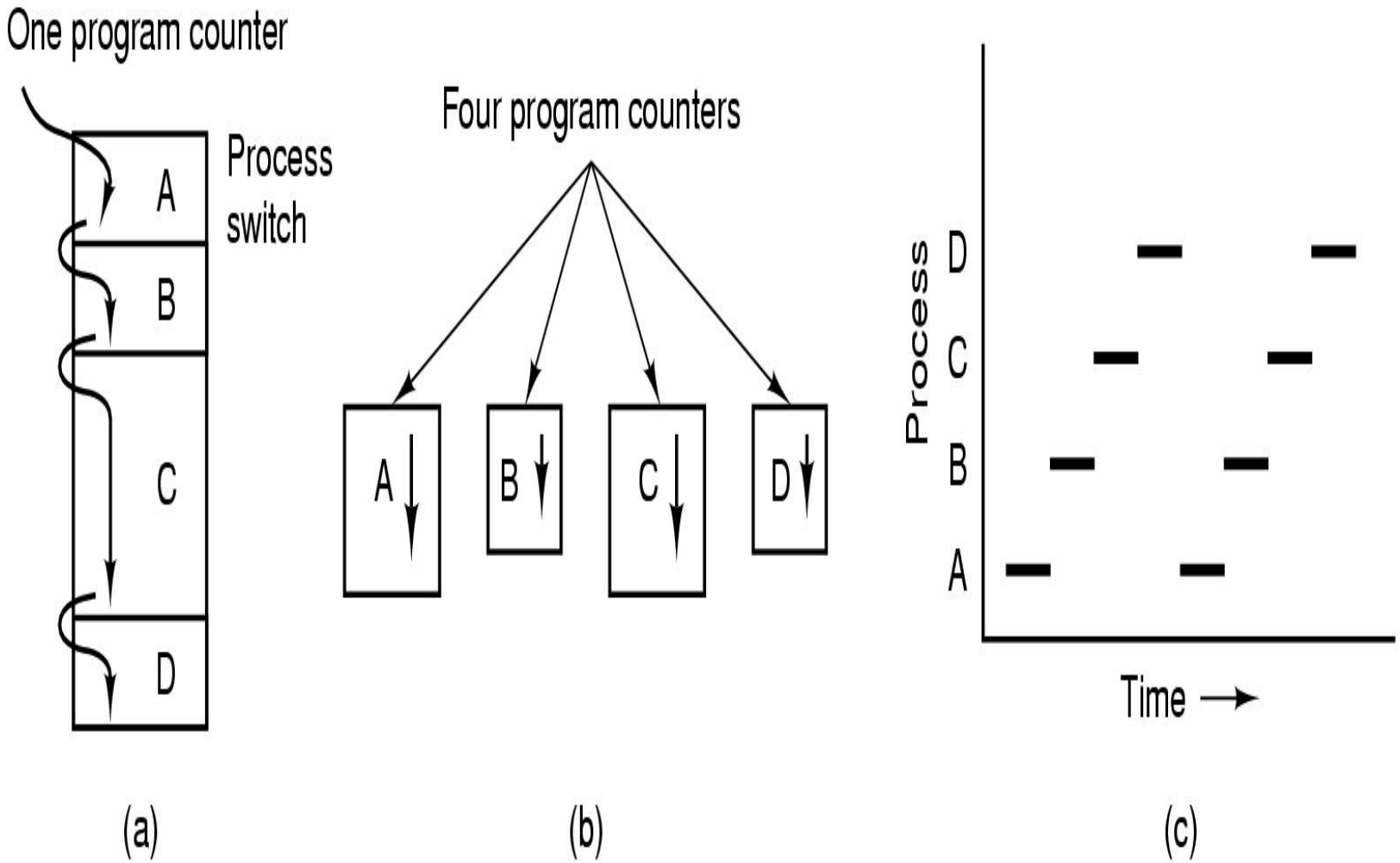


- **When event n occurs, the corresponding process is moved into the ready queue**

Ready Queue and various I/O Device Queues



Process Switch

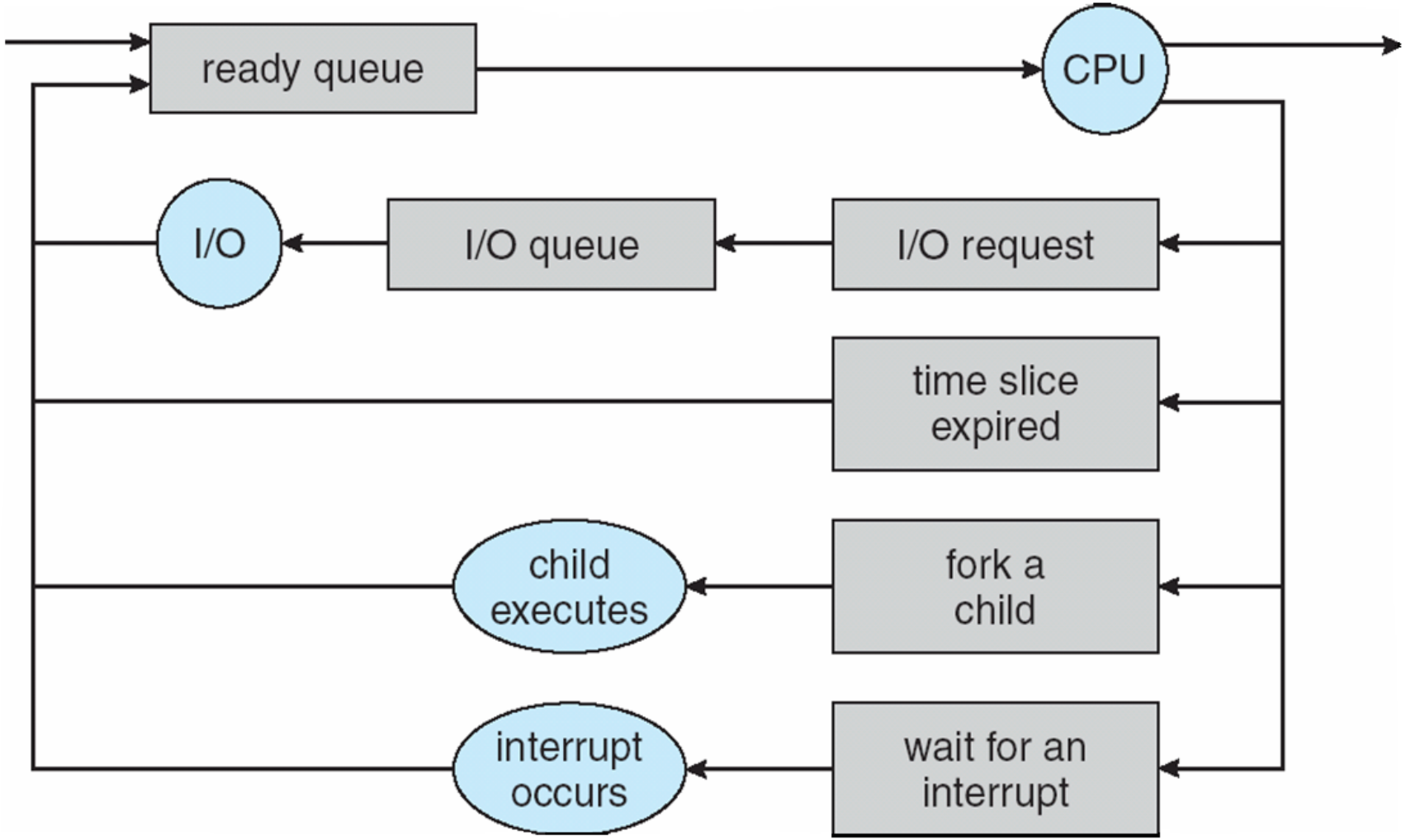


SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

When to Switch a Process?

- A process switch may occur whenever the OS has gained control of CPU. i.e., when:
  - Supervisor Call
    - explicit request by the program (example: file open) – the process will probably be blocked.
  - Trap
    - an error resulted from the last instruction – it may cause the process to be moved to terminated state.
  - Interrupt
    - the cause is external to the execution of the current instruction – control is transferred to Interrupt Handler.

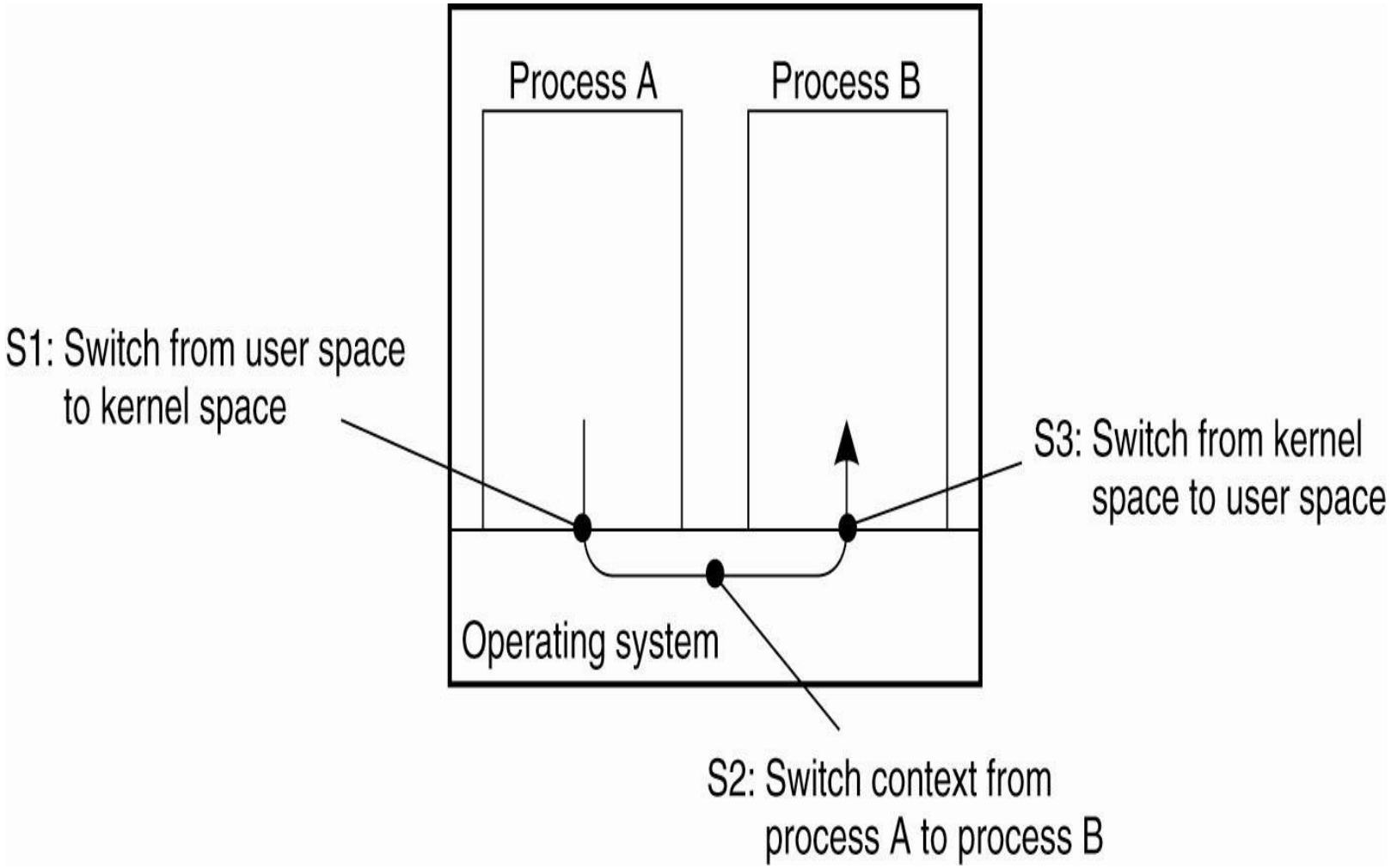
Reasons for Process Switch



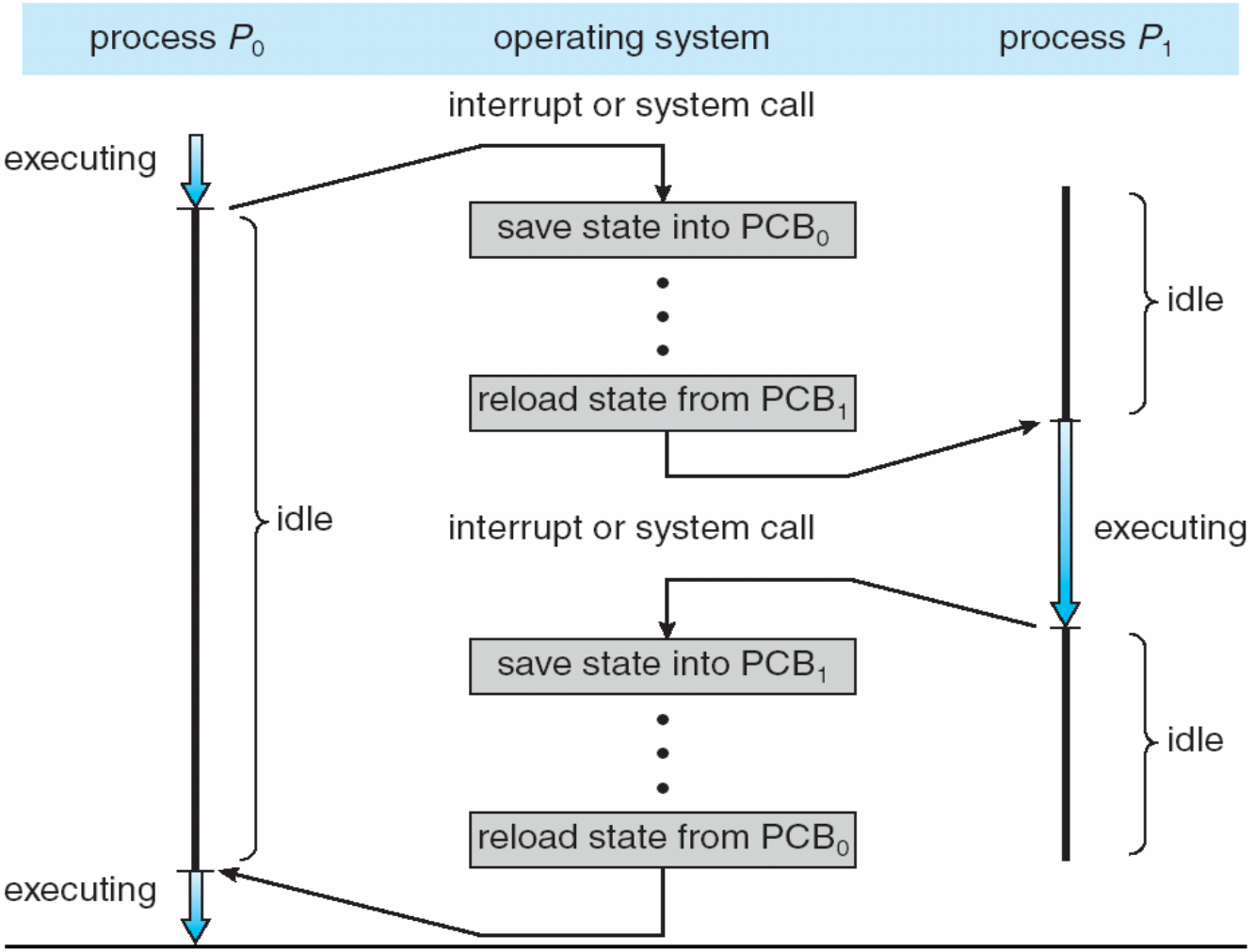
Context Switch

- 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.
- This is called context switch.
- Context of a process represented in the PCB.
- The time it takes is dependent on hardware support.
- Context-switch time is overhead; the system does no useful work while switching.

Context switch between processes (1)



Context switch between processes (2)

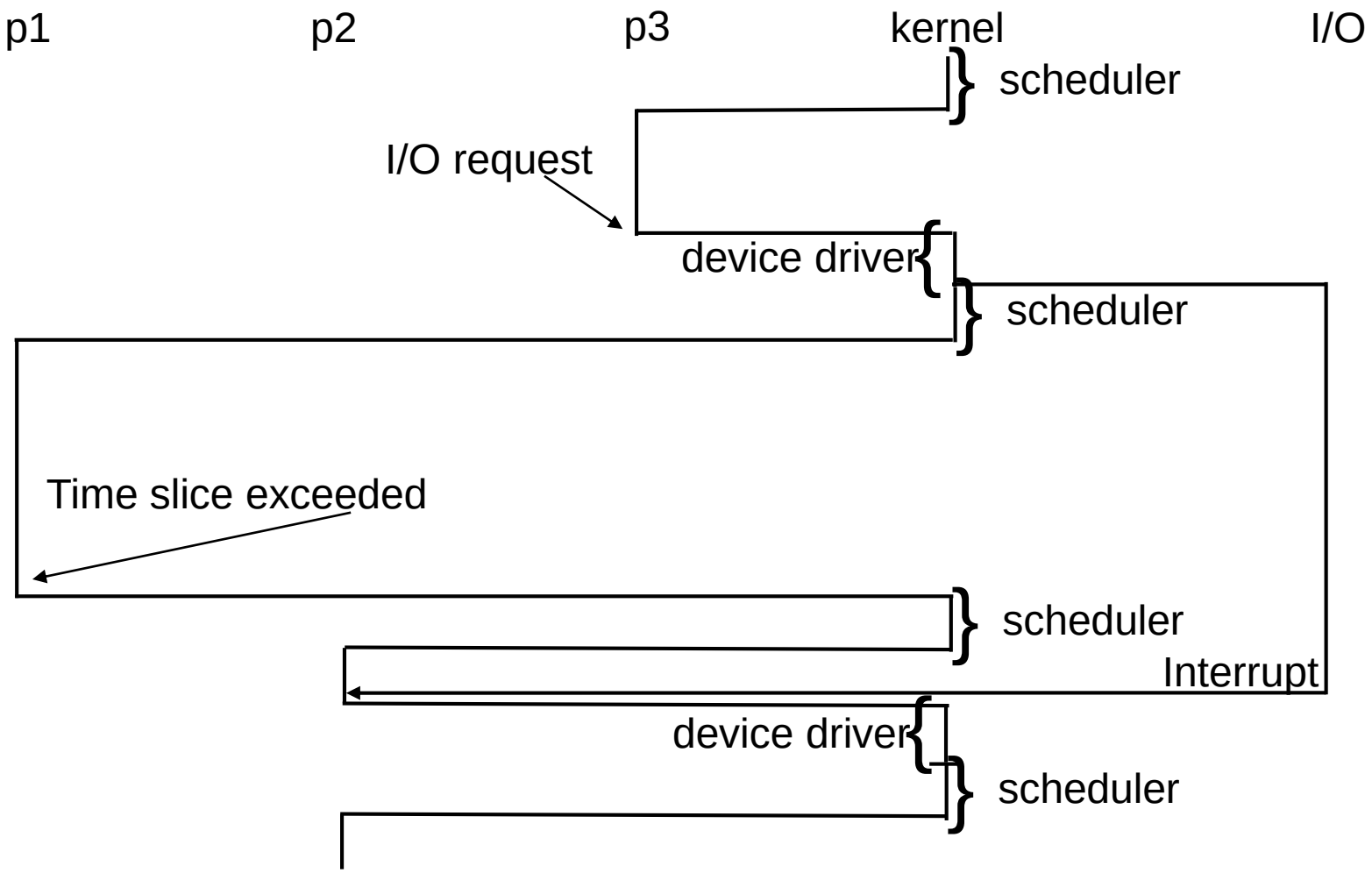


ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Steps in Context Switch

- Save context of processor including program counter and other registers.
- Update the PCB of the running process with its new state and other associate information.
- Move PCB to appropriate queue – ready, blocked,
- Select another process for execution.
- Update PCB of the selected process.
- Restore CPU context from that of the selected process.

Example of Context Switch



ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Mode Switch

- It may happen that an interrupt does not produce a context switch.
- The control can just return to the interrupted program.
- Then only the processor state information needs to be saved on stack.
- This is called mode switch (user to kernel mode when going into Interrupt Handler).
- Less overhead: no need to update the PCB like for context switch.

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Introduction to Task / Thread

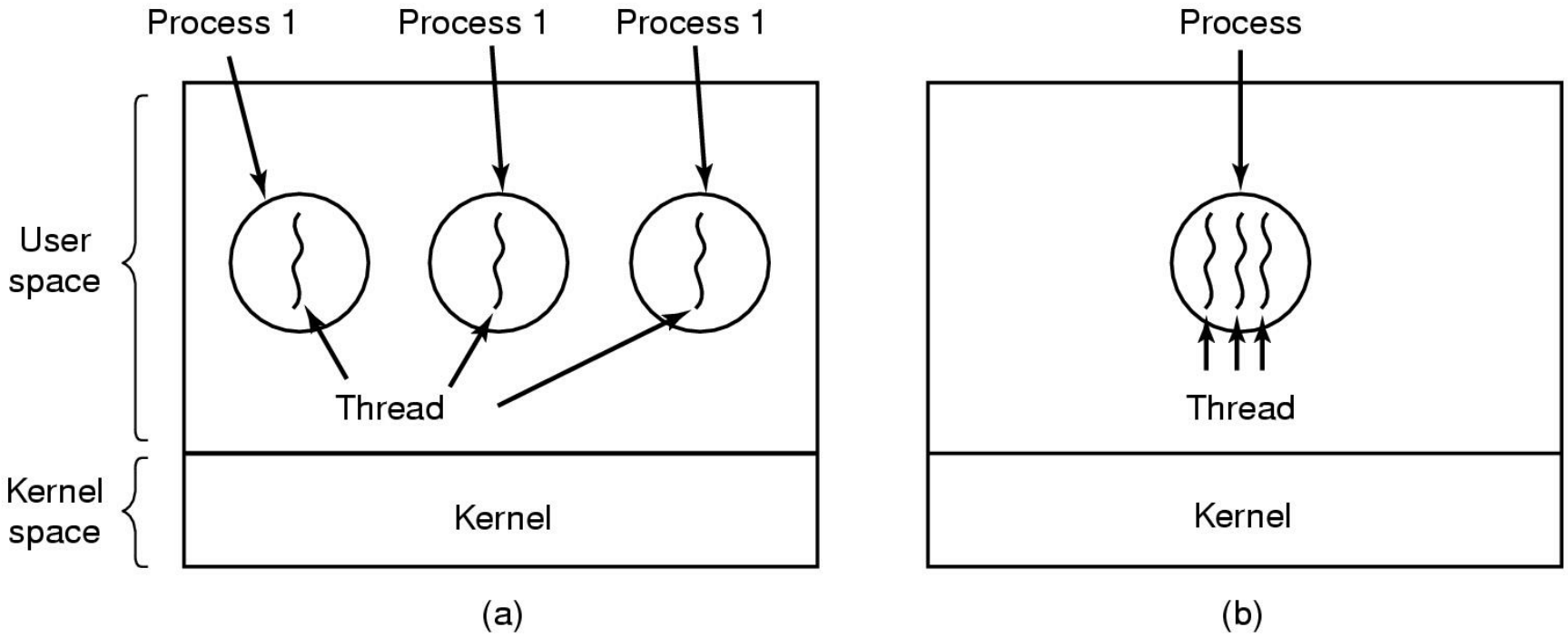
Process Characteristics (1)

- Unit of resource ownership – process is allocated:
  - an address space to hold the process image.
  - control of some resources (files, I/O devices...).
- Unit of dispatching – process is an execution path through one or more programs:
  - execution may be interleaved with other process.
  - the process has an execution state and a dispatching priority.

Process Characteristics (2)

- These two characteristics are treated independently by some recent OSs:
  1. The unit of resource ownership is usually referred to as a Task or (for historical reasons) also as a Process.
  2. The unit of dispatching is usually referred to a Thread or a Light-Weight Process (LWP).
- A traditional Heavy-Weight Process (HWP) is equal to a task with a single thread.
- Several threads can exist in the same task.
  - ***Multithreading*** **-** The ability of an OS to support multiple, concurrent paths of execution within a single process.

The Process vs. Thread Model



(a) Three processes each with one thread.

(b) One process with three threads.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

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.

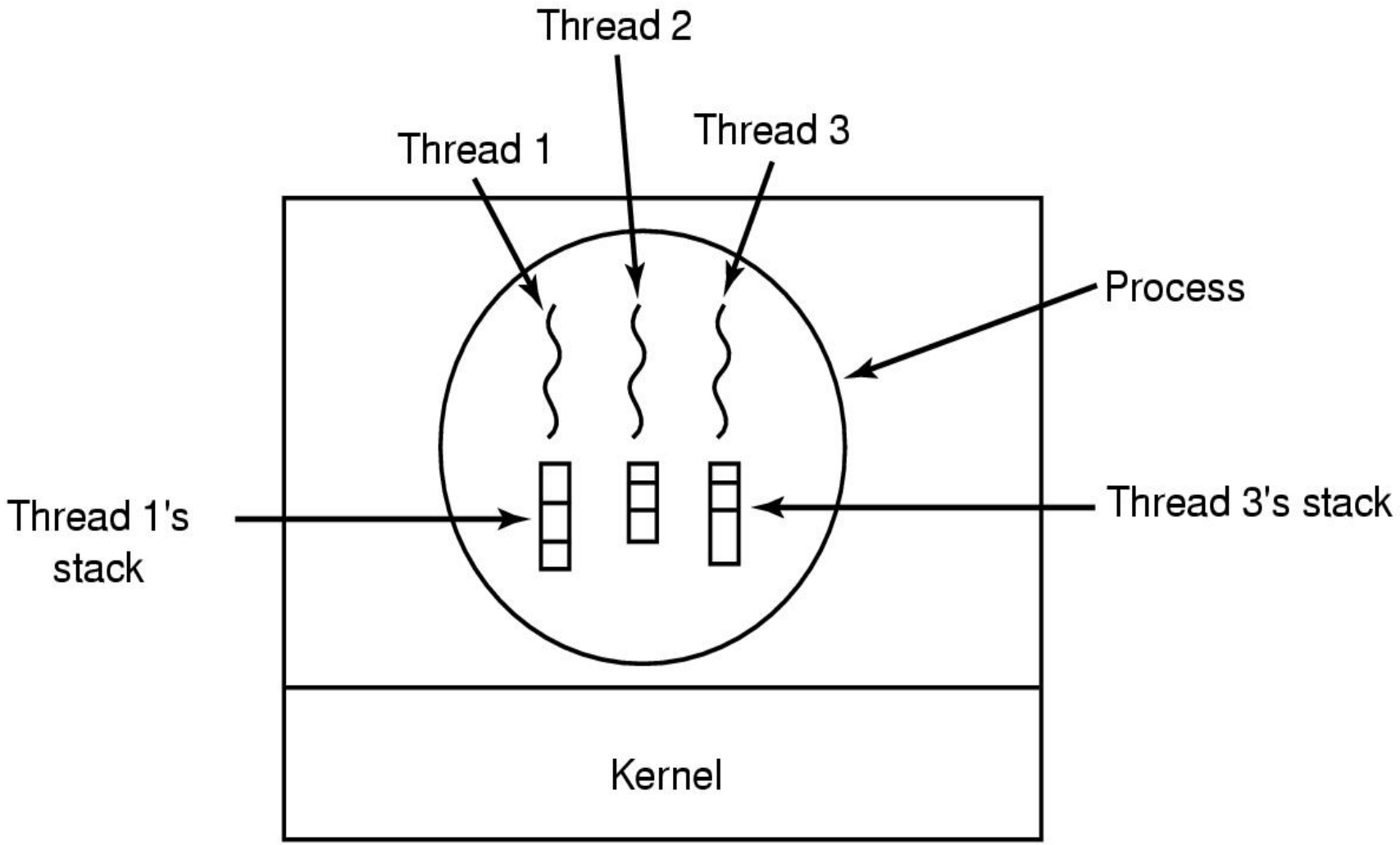
Tasks/Processes and Threads (1)

- 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

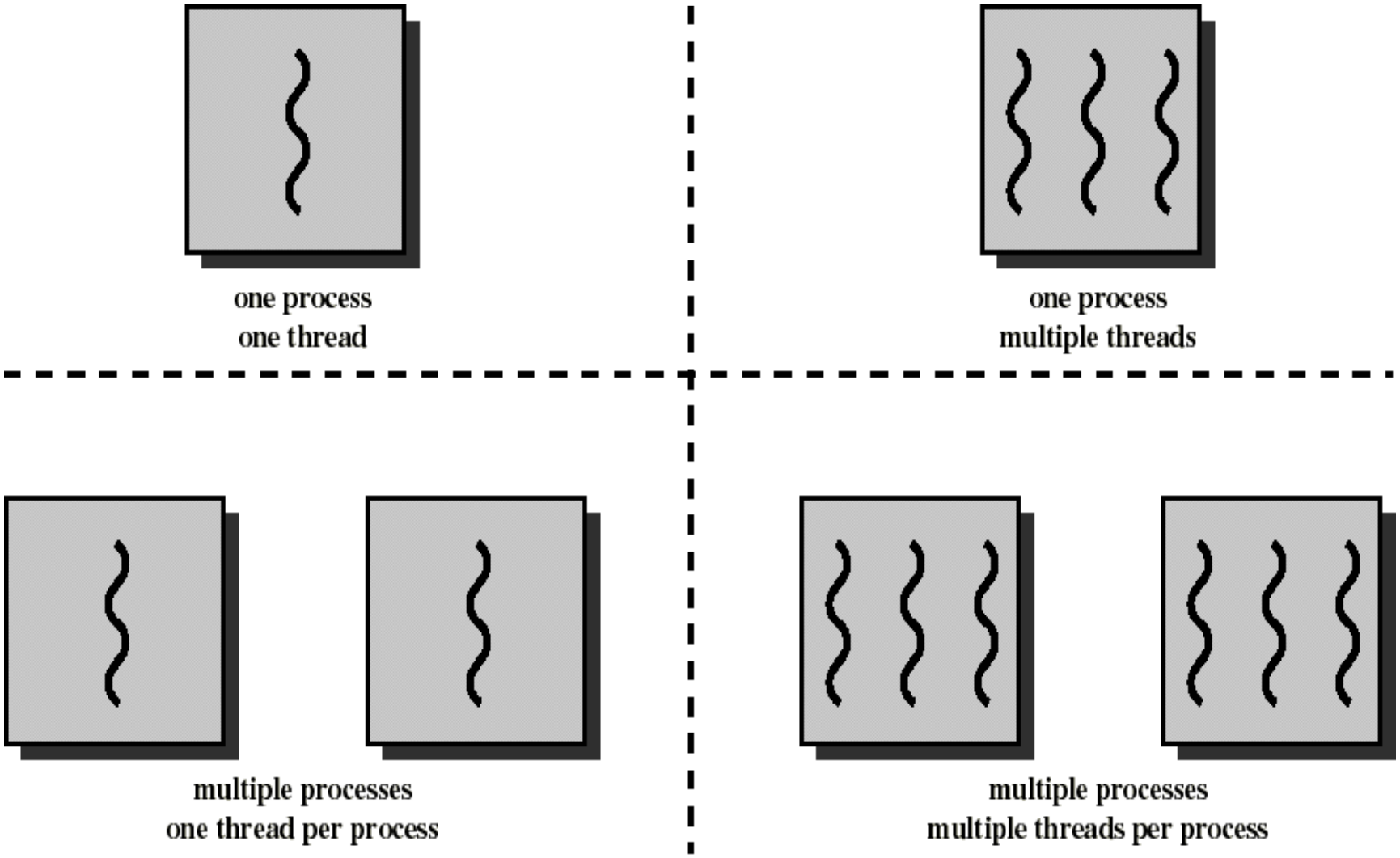
Tasks/Processes and Threads (2)

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	

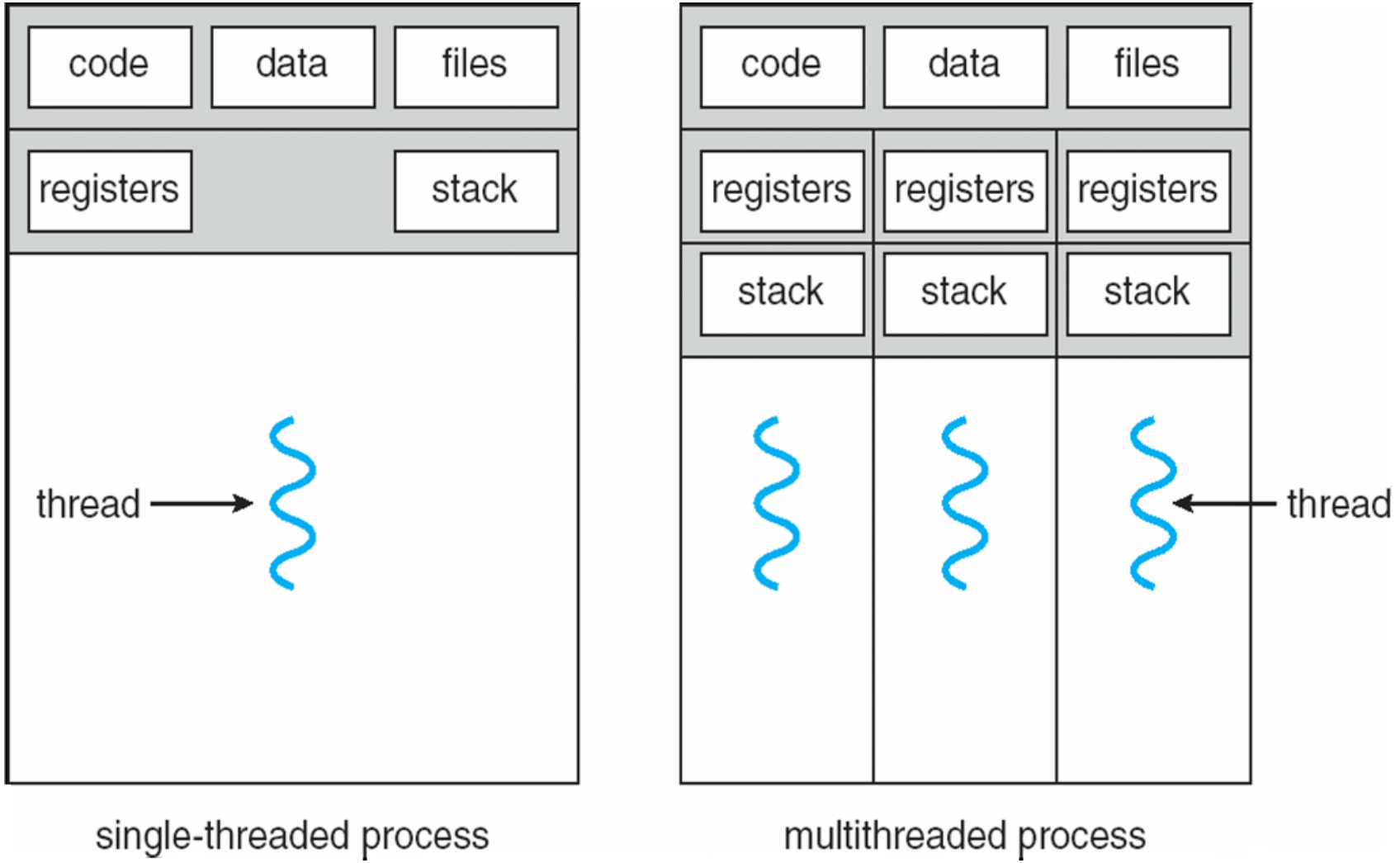
Each thread has its own stack



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.

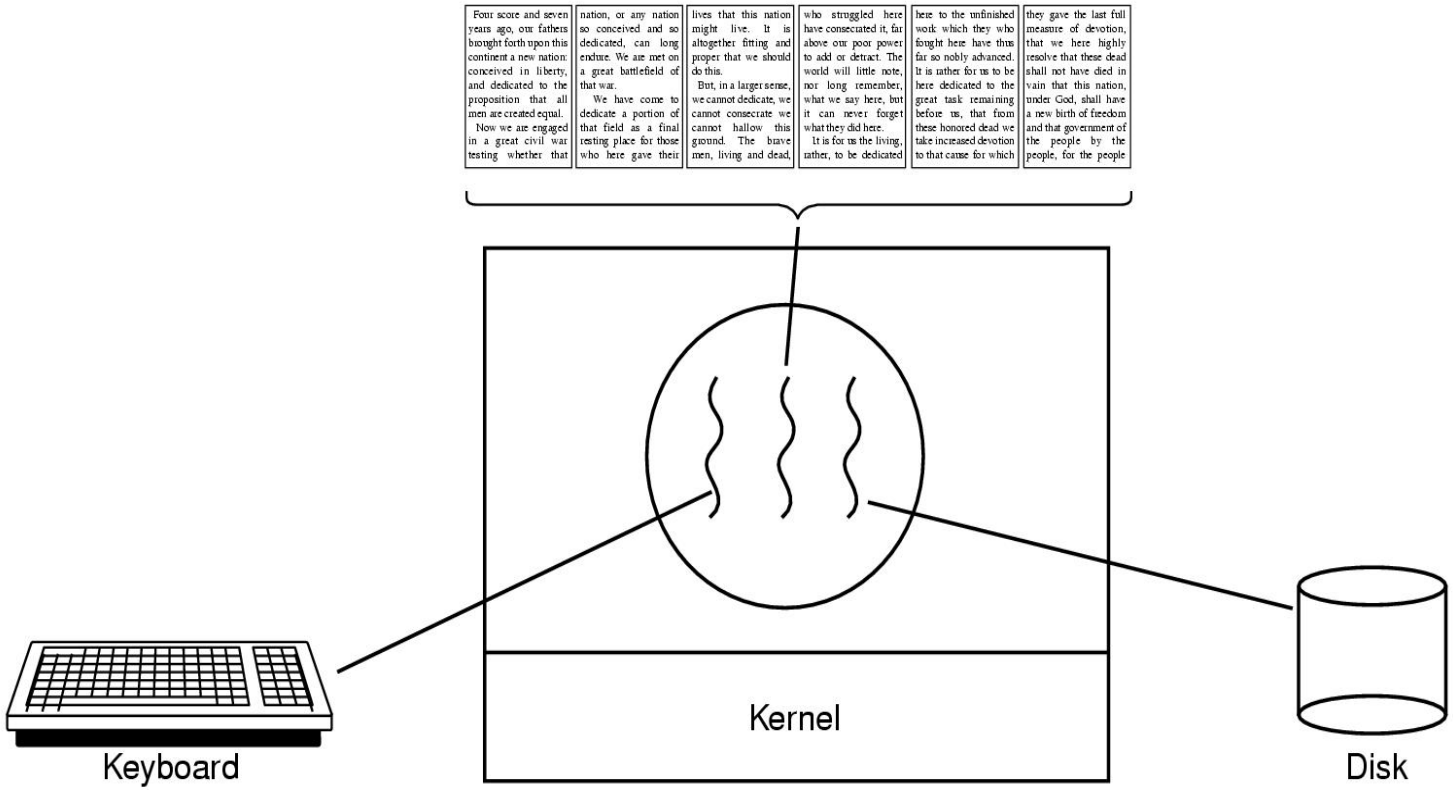
Benefits of Threads (2)

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.

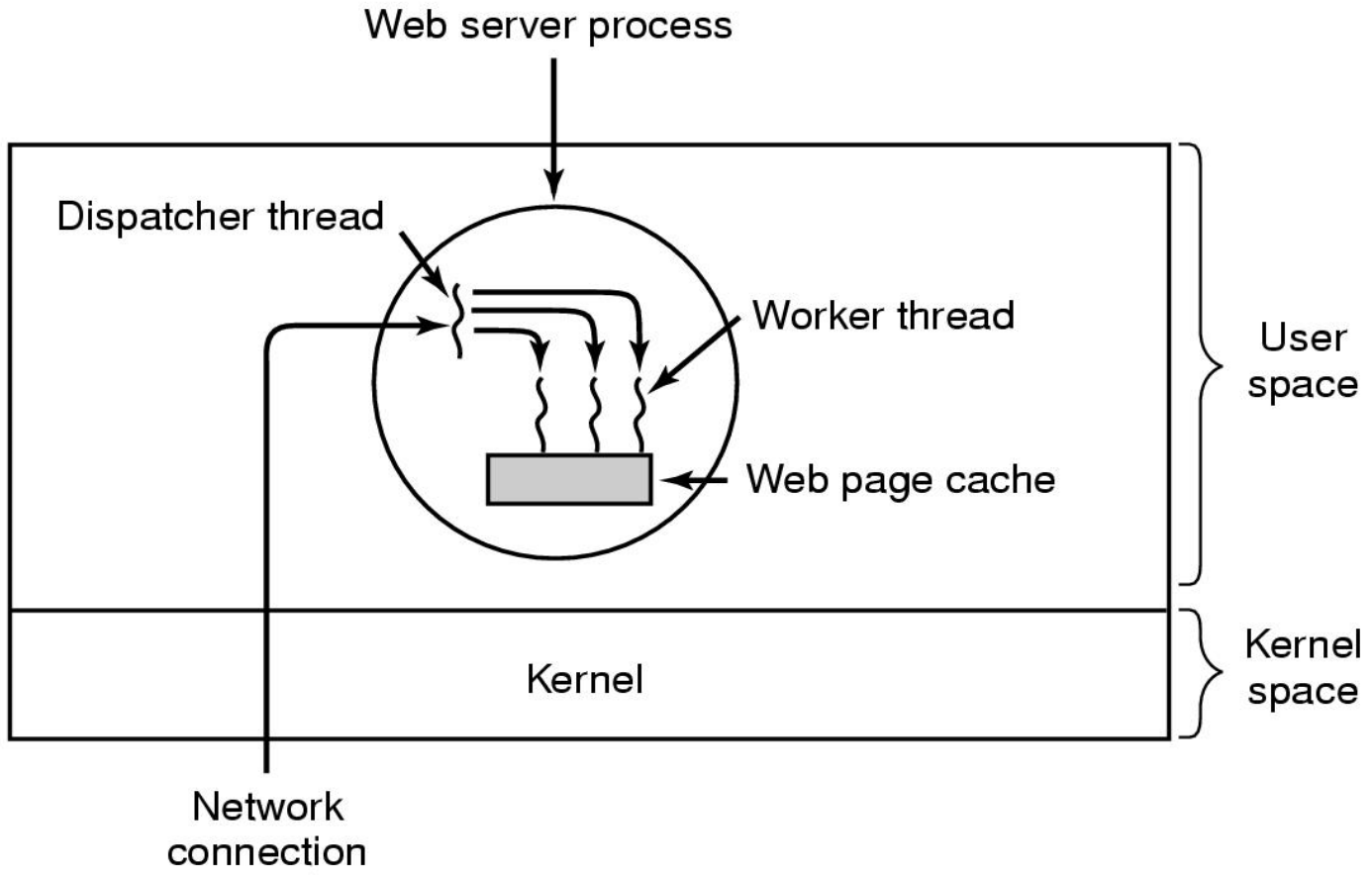
Examples of benefits of threads

- 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.

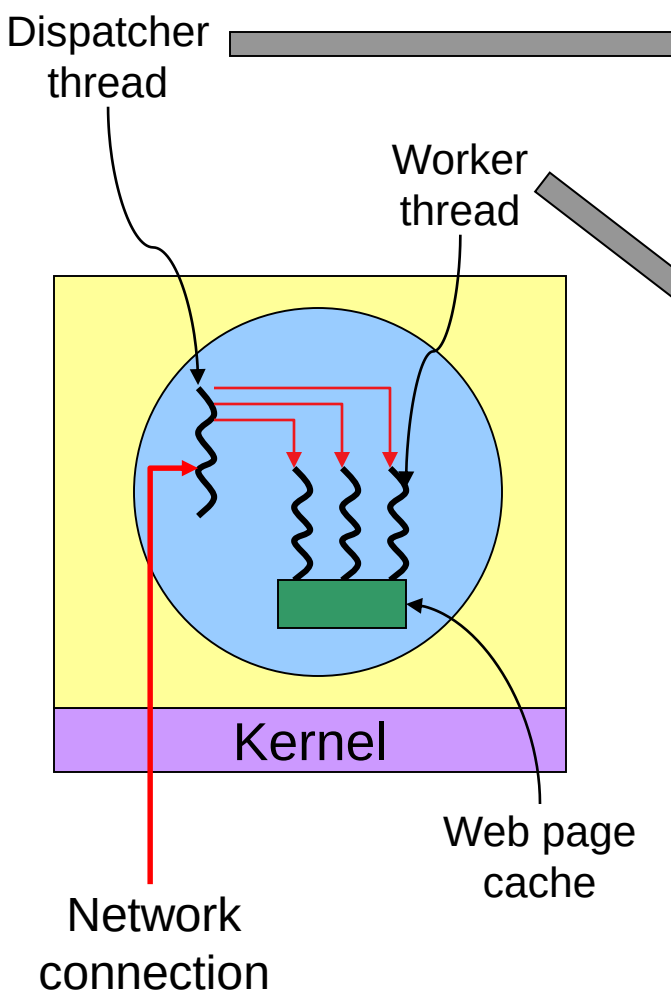


Examples of benefits of threads

- 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.



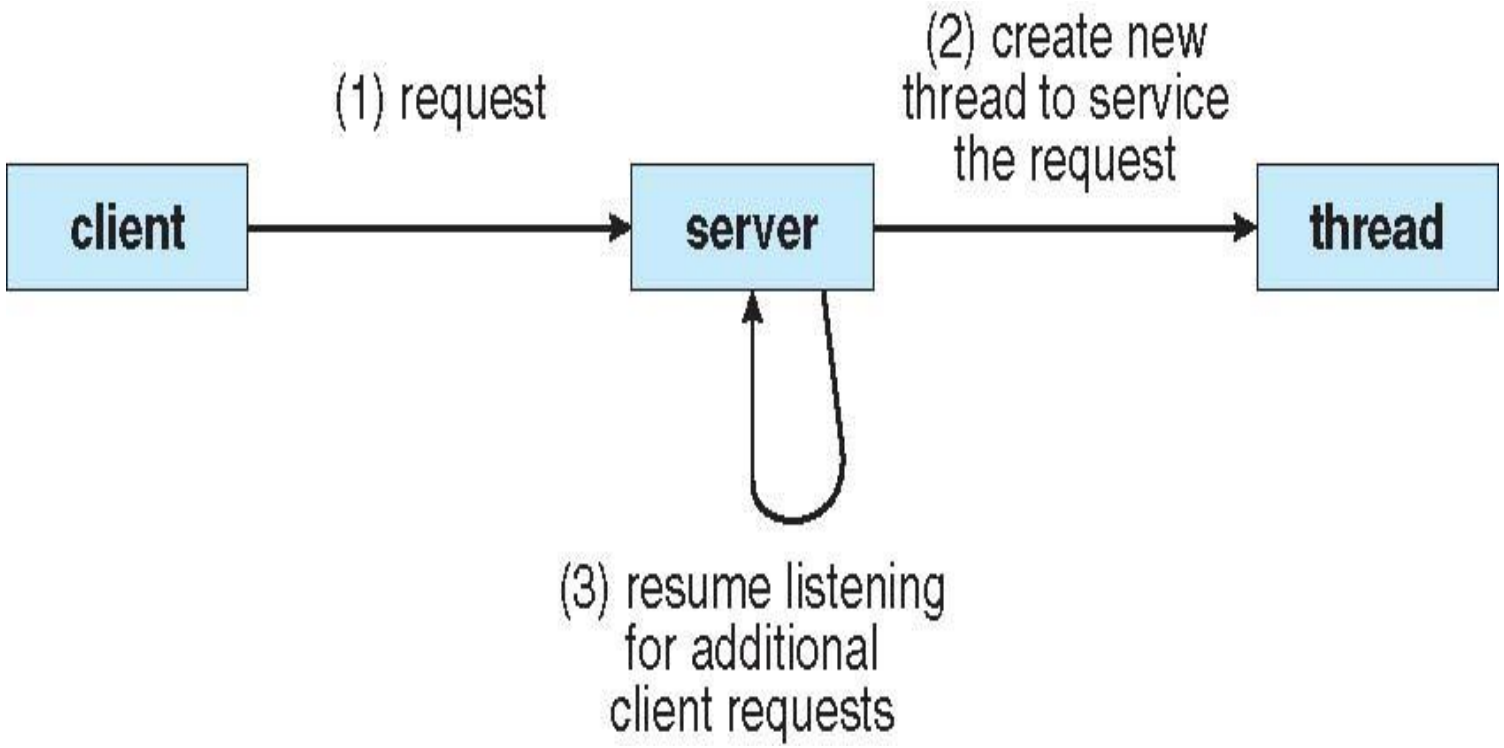
Multithreaded Web server




```
while(TRUE) {
  getNextRequest(&buf);
  handoffWork(&buf);
}
```

```
while(TRUE) {
  waitForWork(&buf);
  lookForPageInCache(&buf,&page);
  if(pageNotInCache(&page)) {
    readPageFromDisk(&buf,&page);
  }
  returnPage(&page);
}
```

Multithreaded Server Architecture



Important Implications

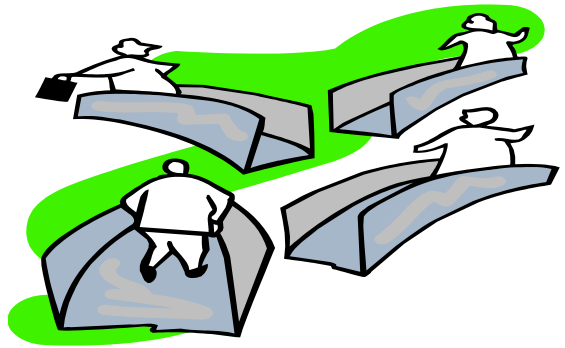
- Two Important Implications:
  1. Threaded applications often run faster than non-threaded applications (as context-switches between kernel and user-space are avoided).
  2. Threaded applications are harder to develop (although simple, clean designs can help here).
- Additionally, the assumption is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

Application benefits of threads (1)

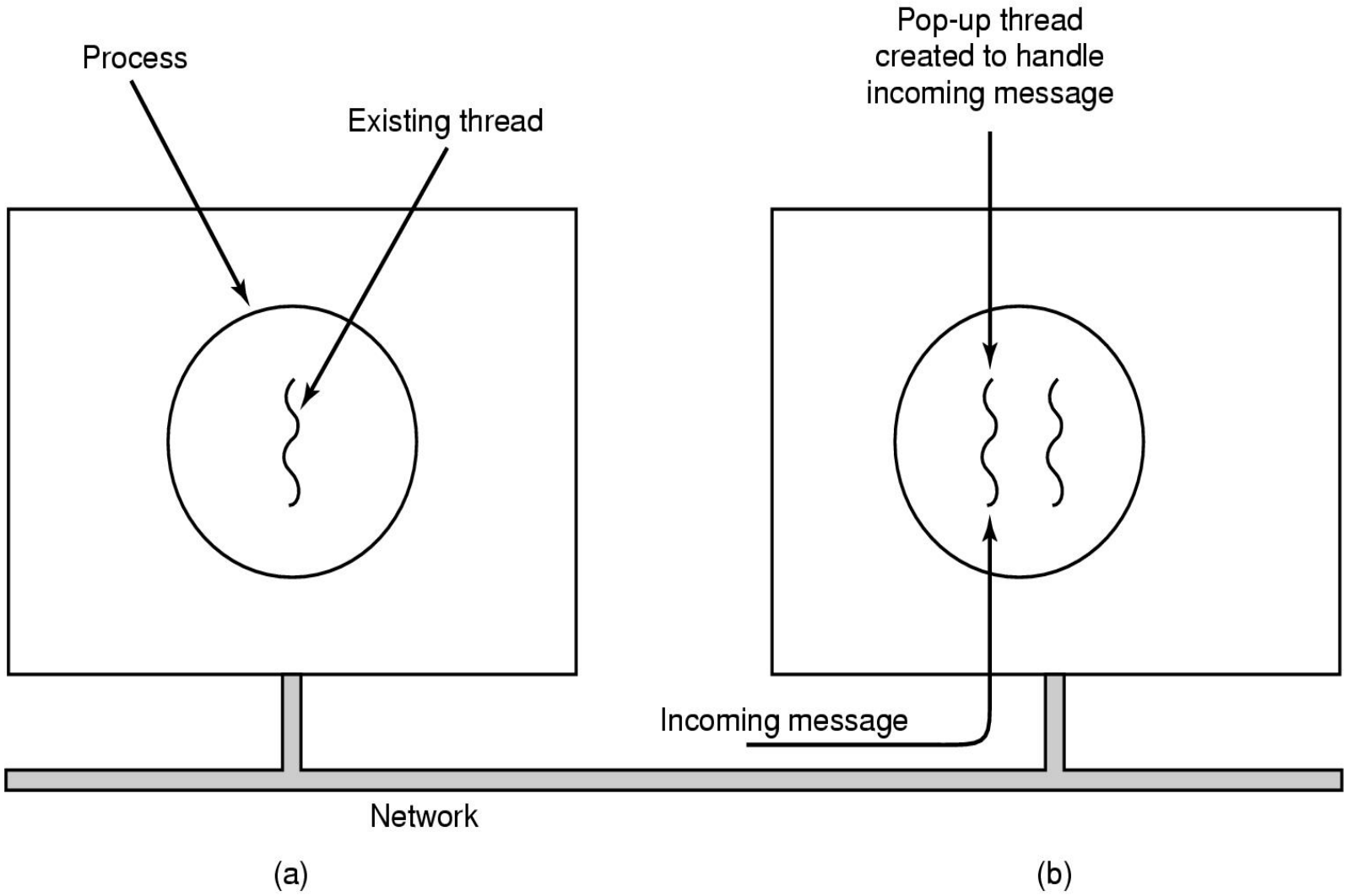
- Consider an application that consists of several independent parts that do not need to run in sequence.
- Each part can be implemented as a thread.
- Whenever one thread is blocked waiting for an I/O, execution could possibly switch to another thread of the same application (instead of blocking it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



Pop-Up Threads



ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

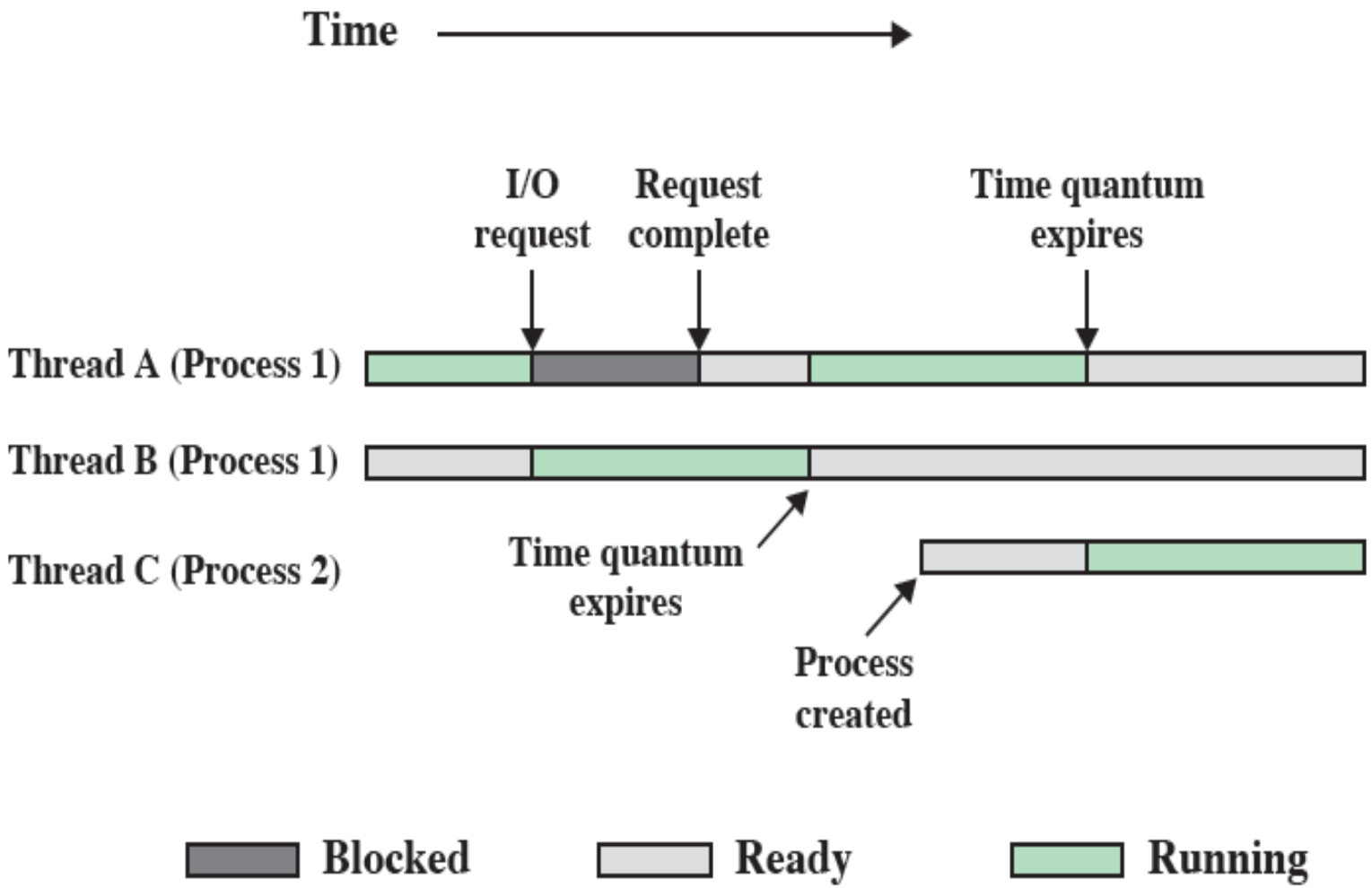
Thread Characteristics

- 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

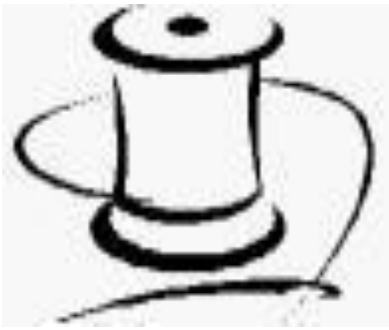


ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Threads Implementation

Contents

- Multithreading Levels
- Multithreading Models
- Threading Issues



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

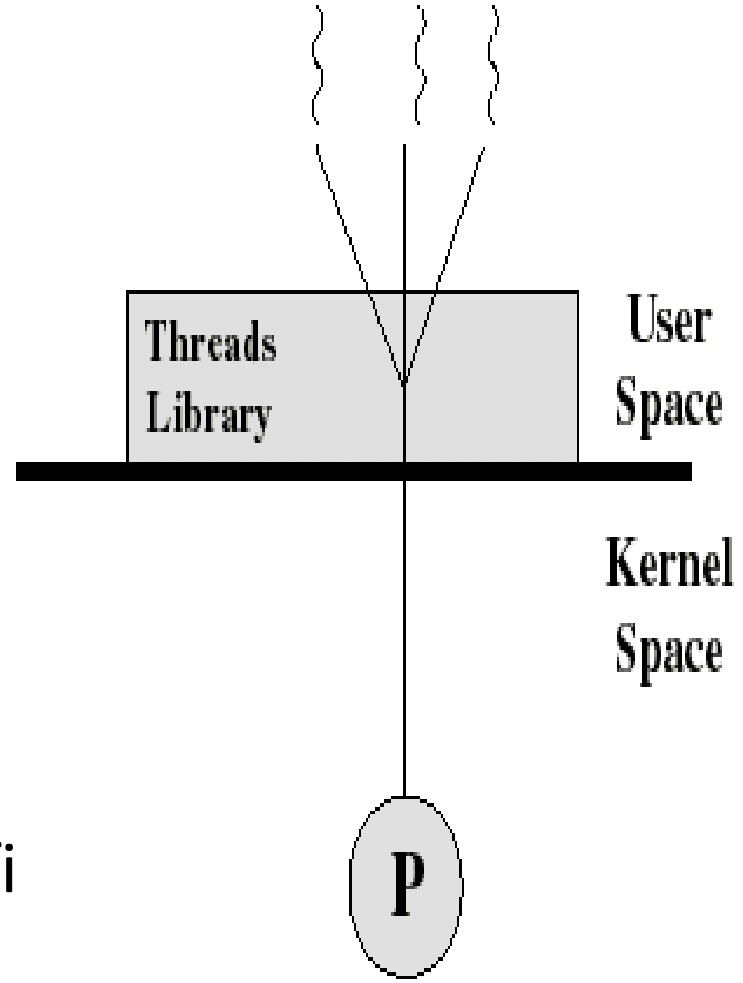
- Thread library provides programmer with API for creating and managing threads.
- Three multithreading levels:
  1) User-Level Threads (ULT)
     - Library entirely in user space.
  2) Kernel-Level Threads (KLT)
     - Kernel-level library supported by the OS.
  3) Hybrid ULT/KLT Approach

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

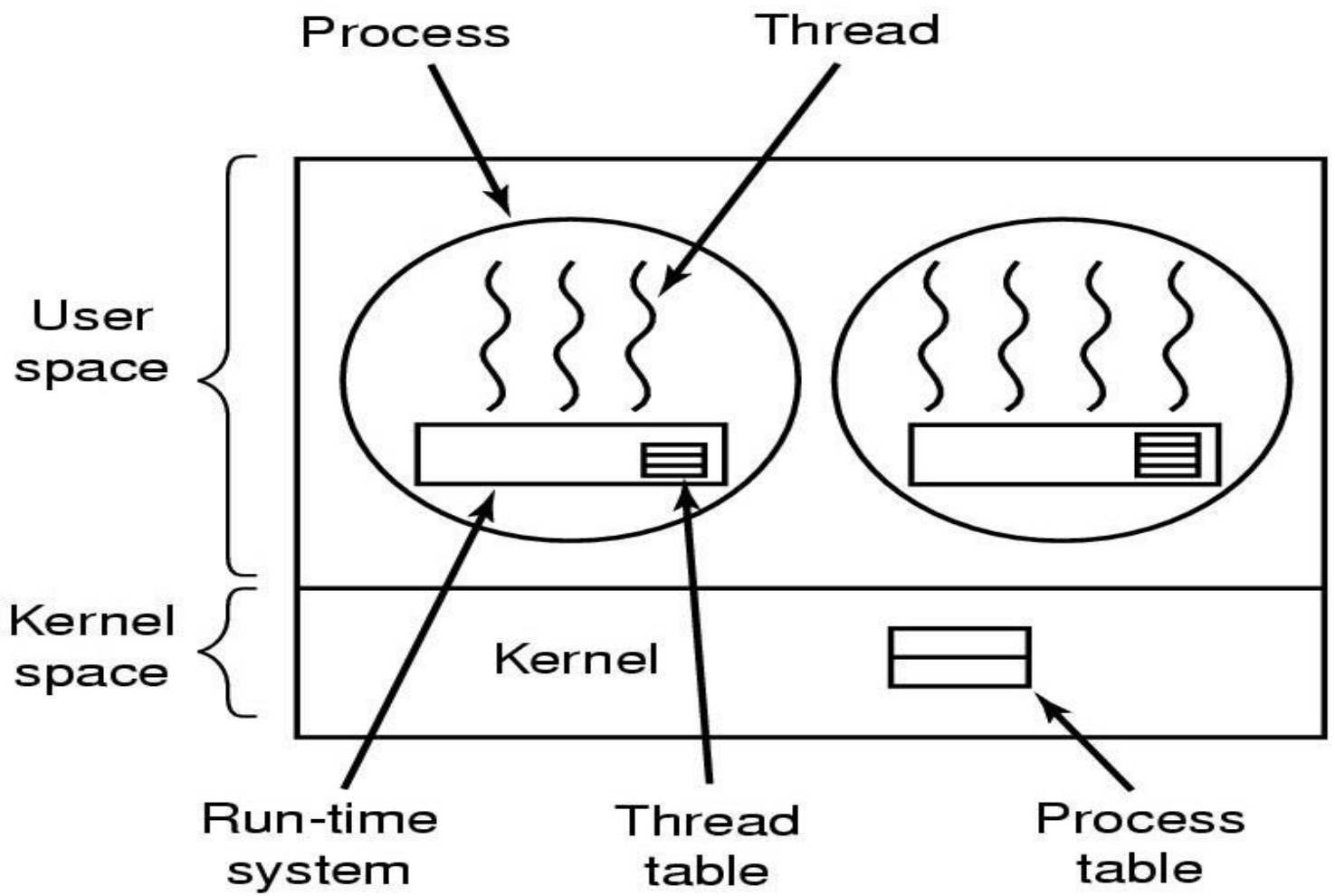
- User(-level) threads – management by user-level threads library.
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads
- Kernel(-level) threads – supported by the Kernel.
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris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

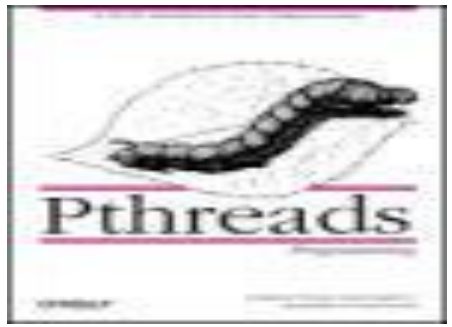


Implementing Threads in User Space



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

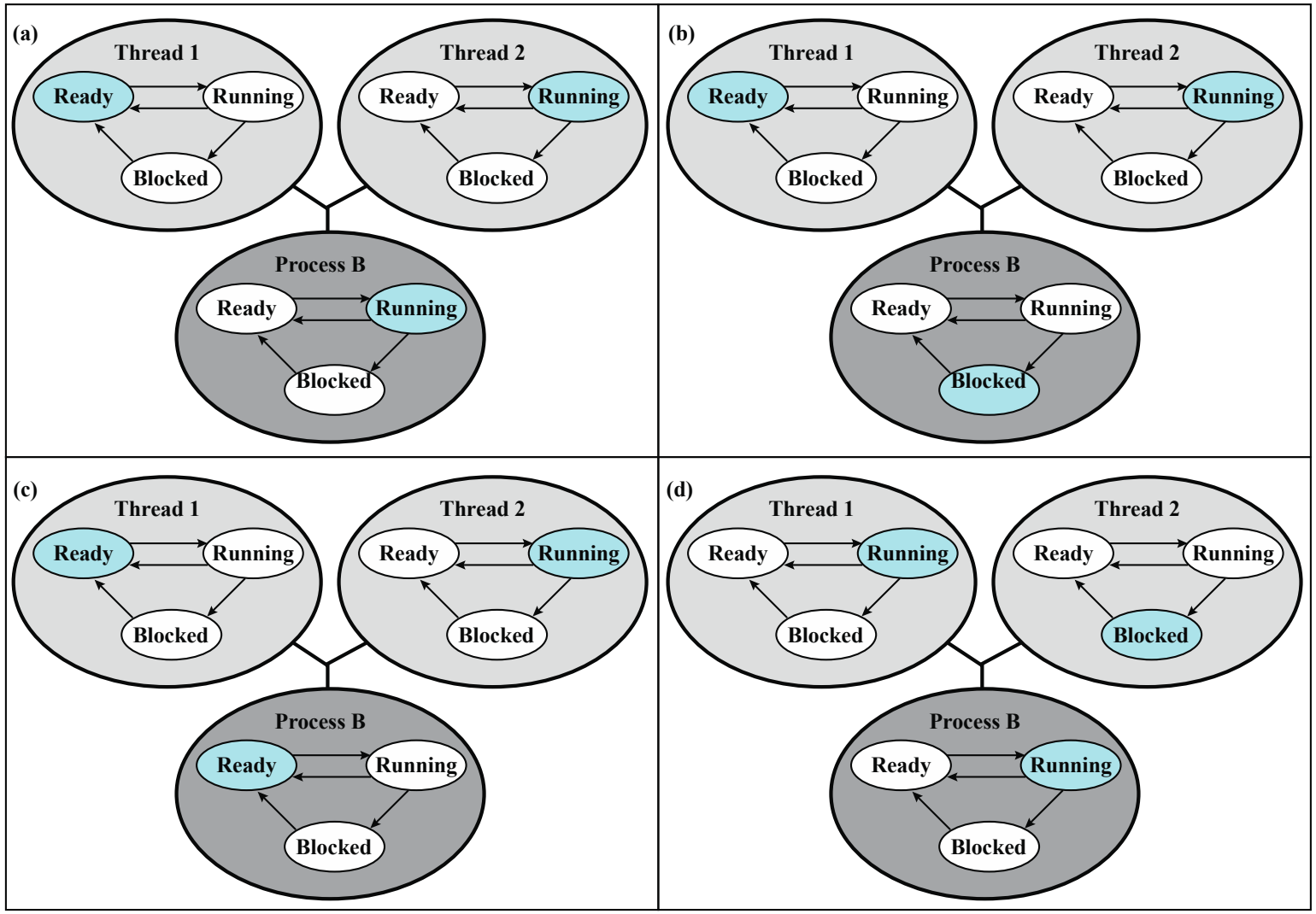
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- May be provided either as ULT or KLT.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX operating systems (Solaris, Linux, Mac OS X).

Some of the Pthreads function calls

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Kernel activity for ULTs

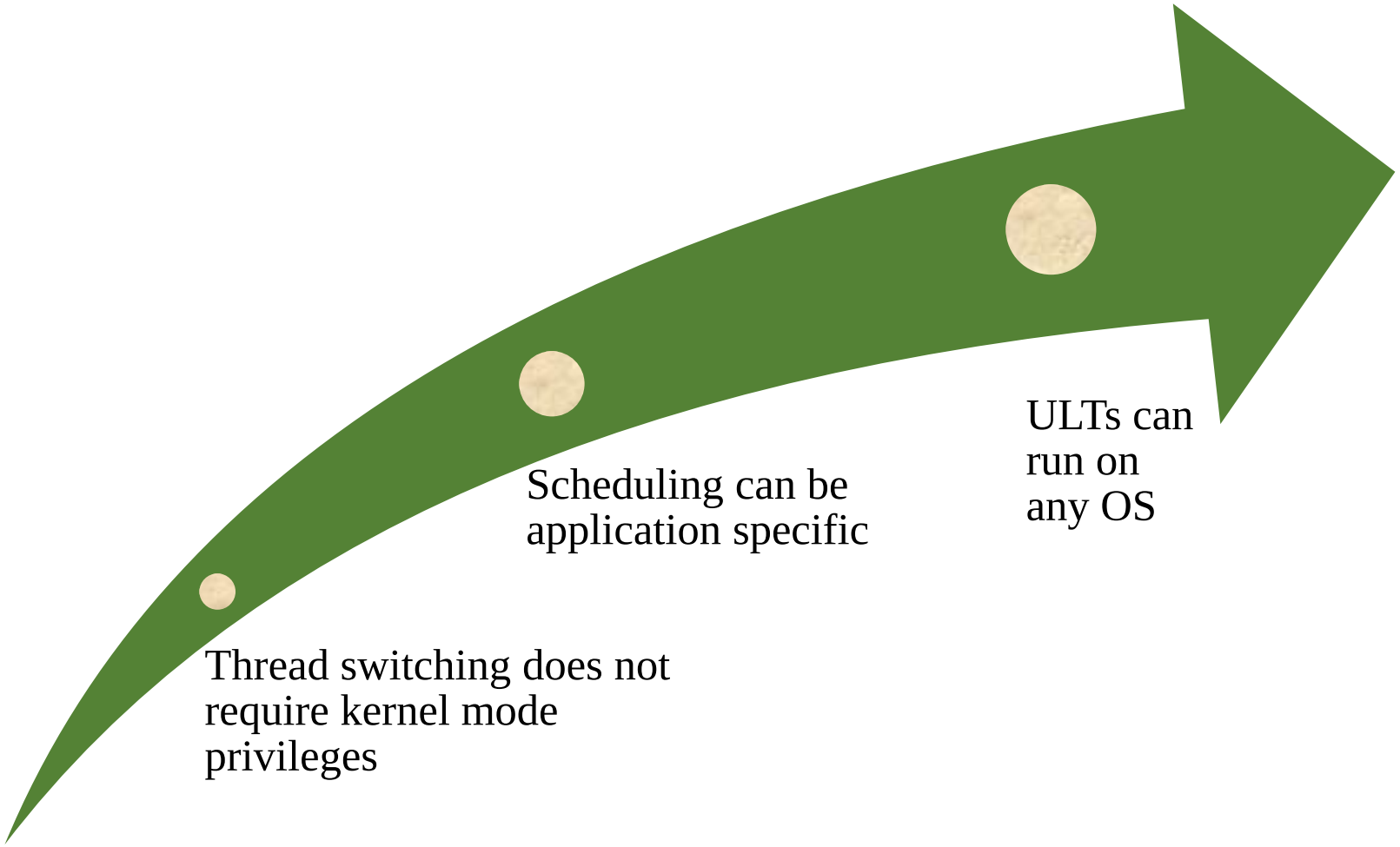
- The kernel is not aware of thread activity but it is still managing process activity.
- When a thread makes a system call, the whole task will be blocked.
- But for the thread library that thread is still in the running state.
- So thread states are independent of process states.



Colored state
is current state

Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

Advantages of ULTs

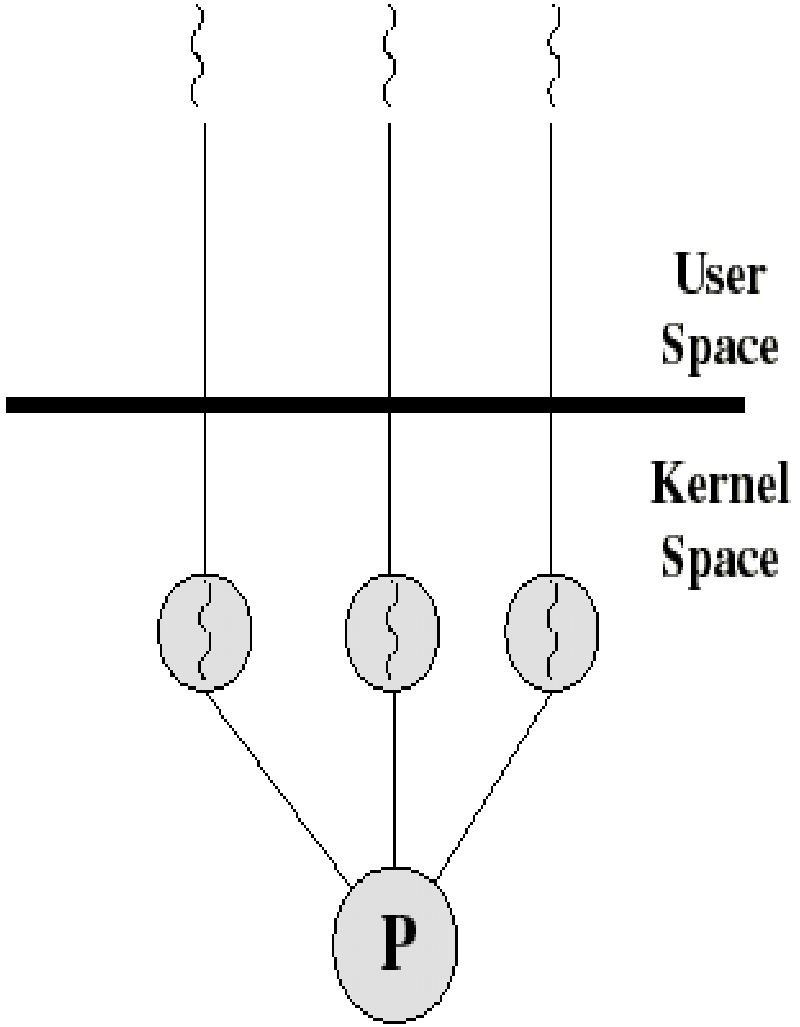


Advantages and inconveniences of ULT

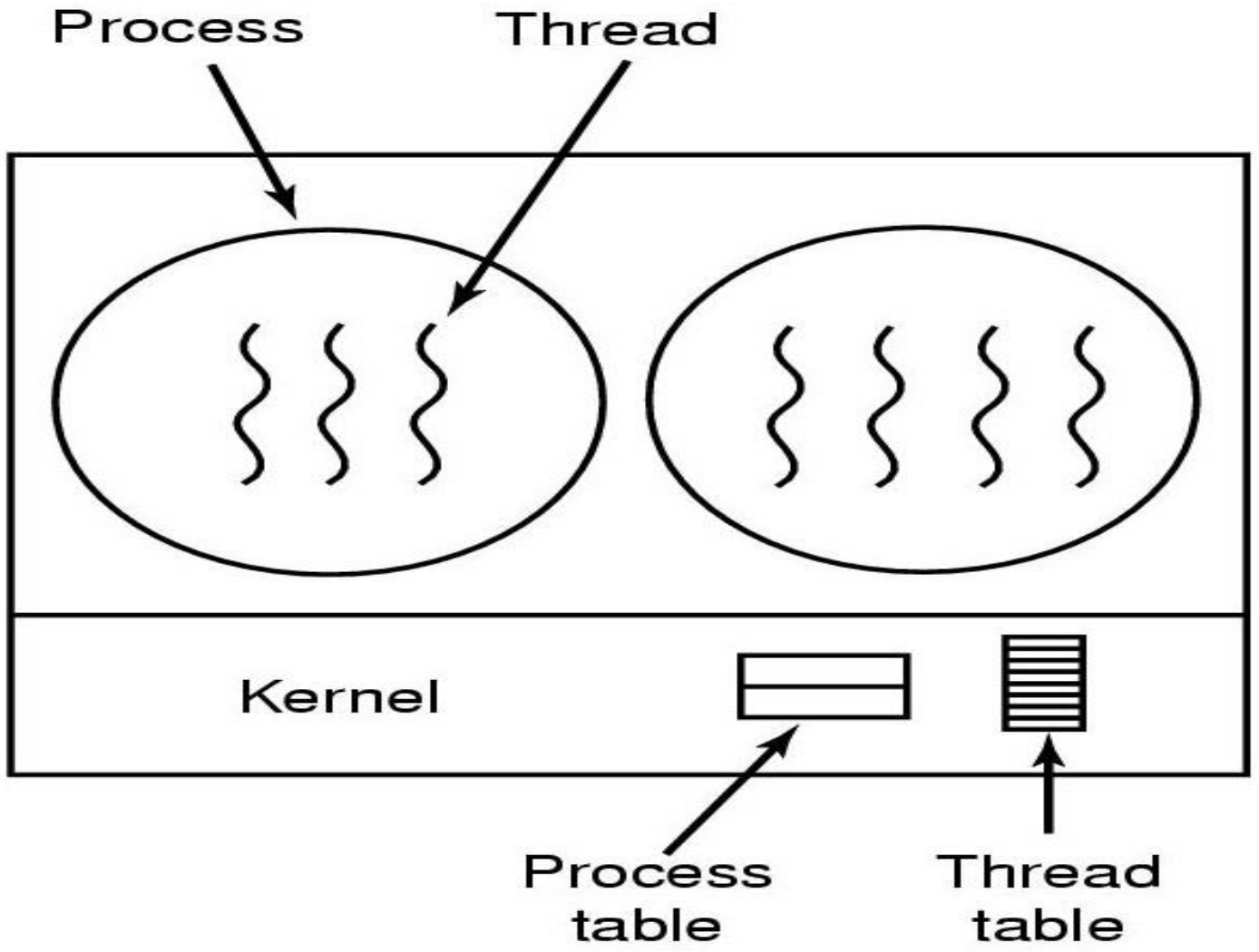
- Advantages
  - Thread switching does not involve the kernel: no mode switching.
  - Scheduling can be application specific: choose the best algorithm.
  - ULTs can run on any OS. Only needs a thread library.
- Inconveniences
  - Most system calls are blocking and the kernel blocks processes. So all 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.



Implementing Threads in the Kernel



KLT Idea

- Threads supported by the Kernel.
- Examples:
  - Windows 2000/XP
  - OS/2
  - Linux
  - Solaris
  - Tru64 UNIX
  - Mac OS X

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.

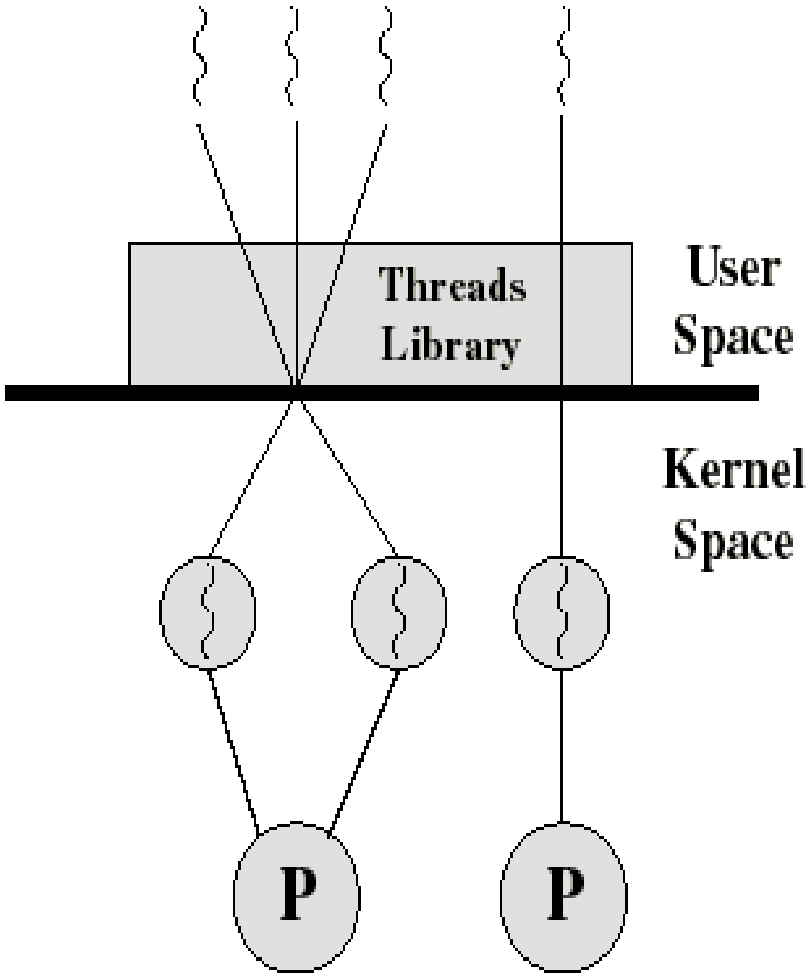
Thread operation latencies ( ) μs

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

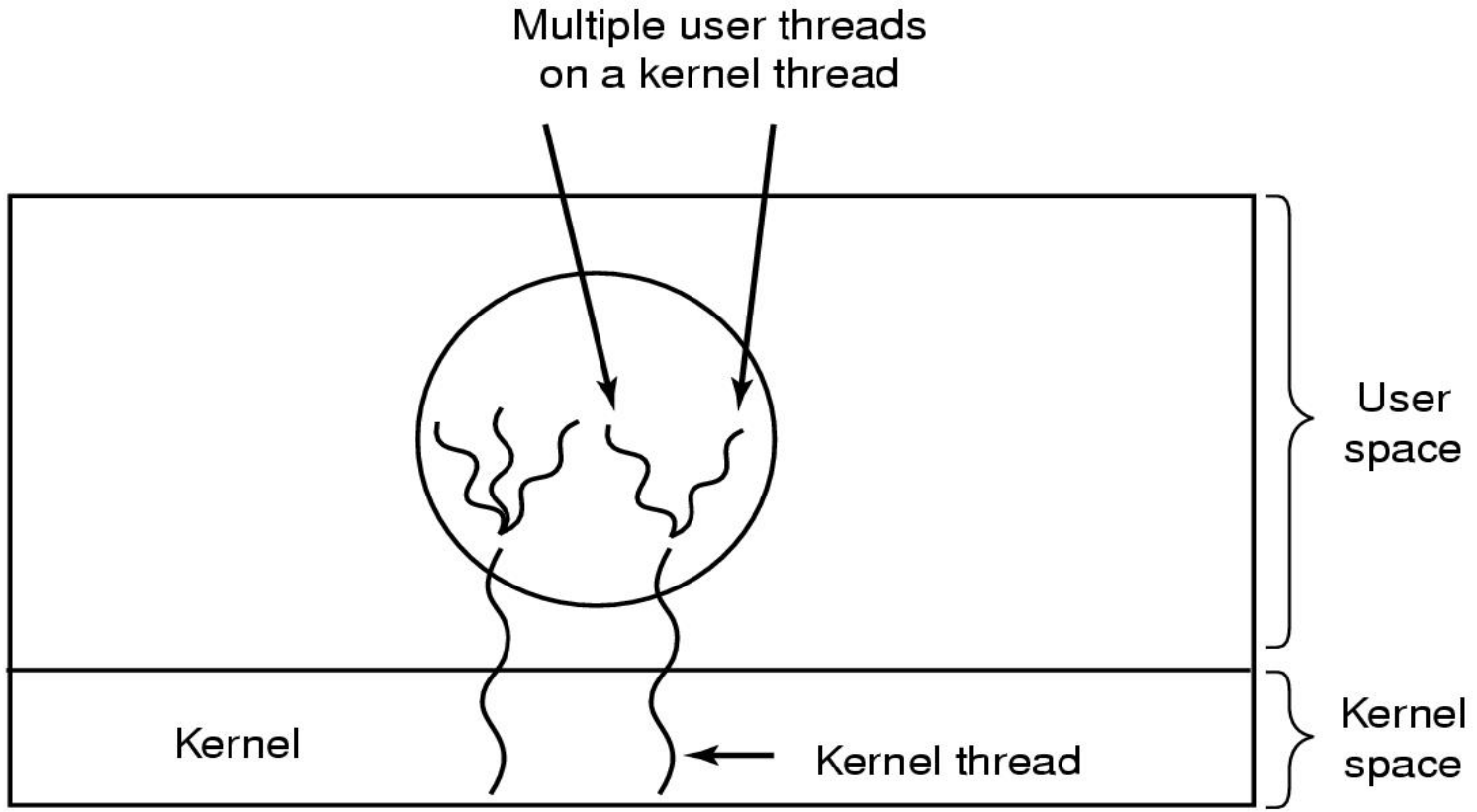
Source: Anderson, T. et al, “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”, ACM TOCS, February 1992.

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 prior to version 9.

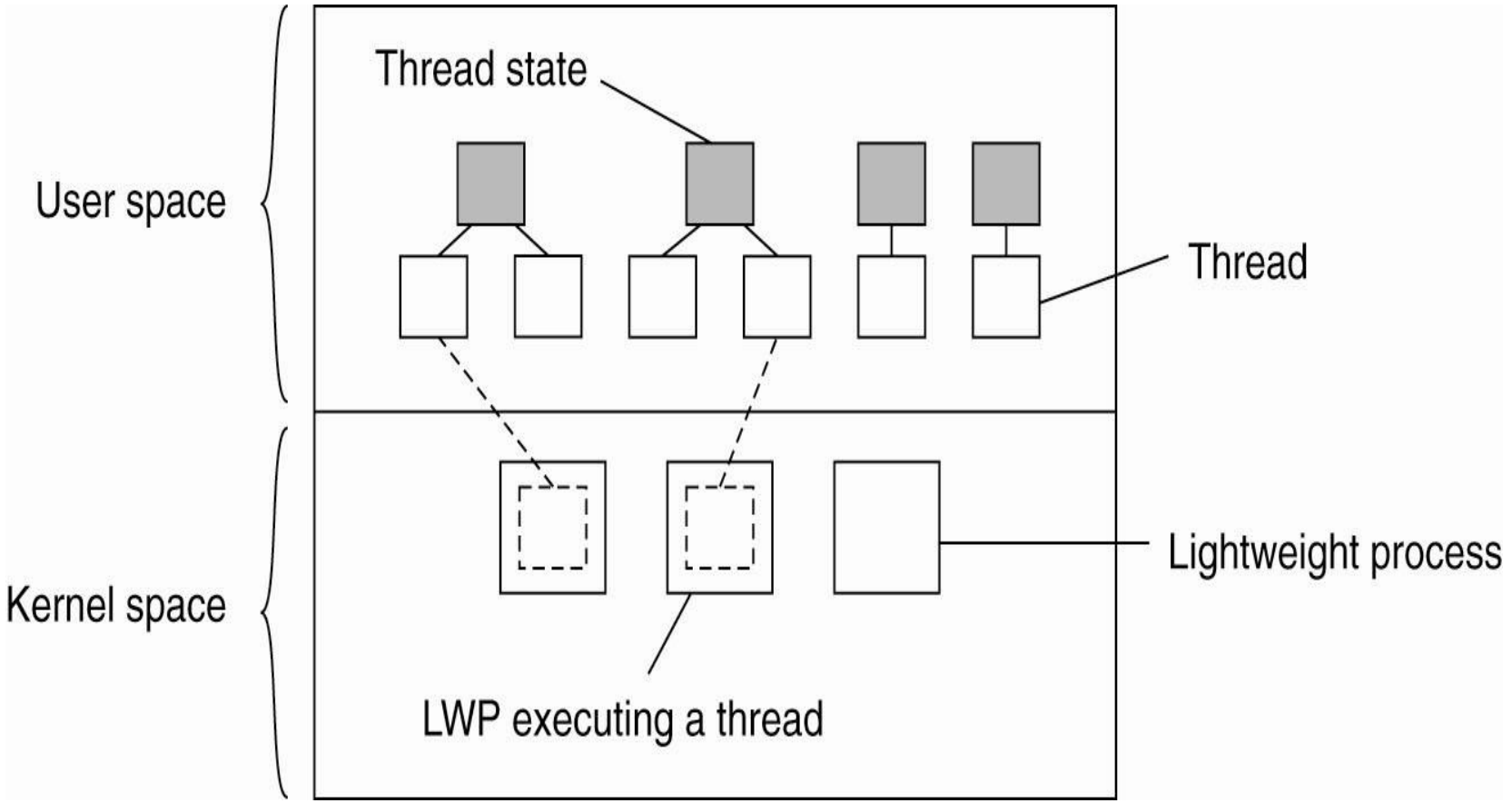


Hybrid Implementation (1)

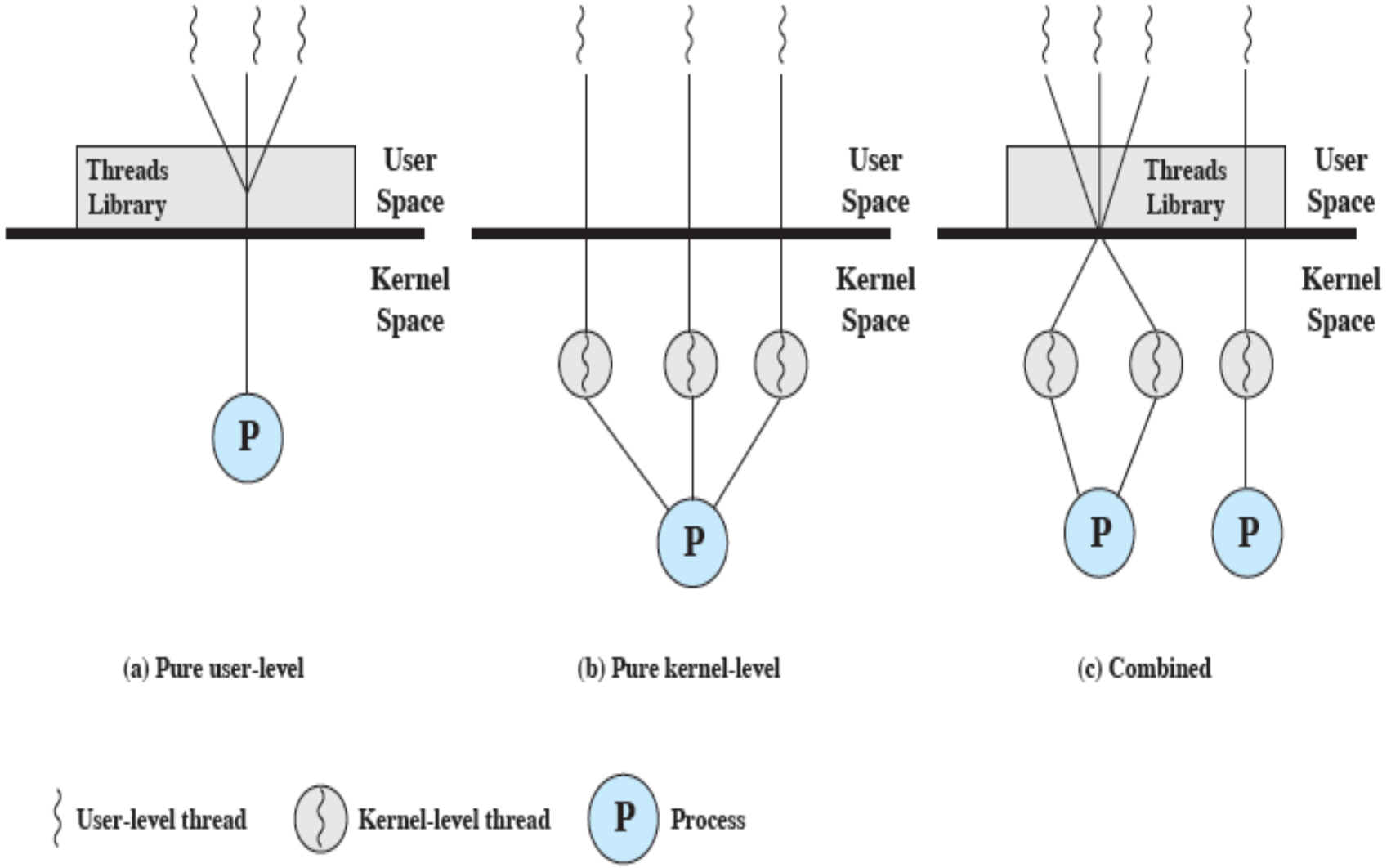


Multiplexing user-level threads onto kernel-level threads.

Hybrid Implementation (2)

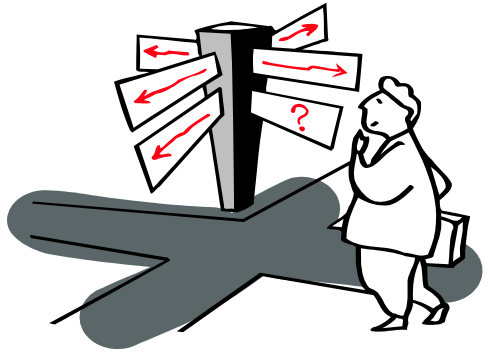


ULT, KLT and Combined Approaches



Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many
- Two-level Model

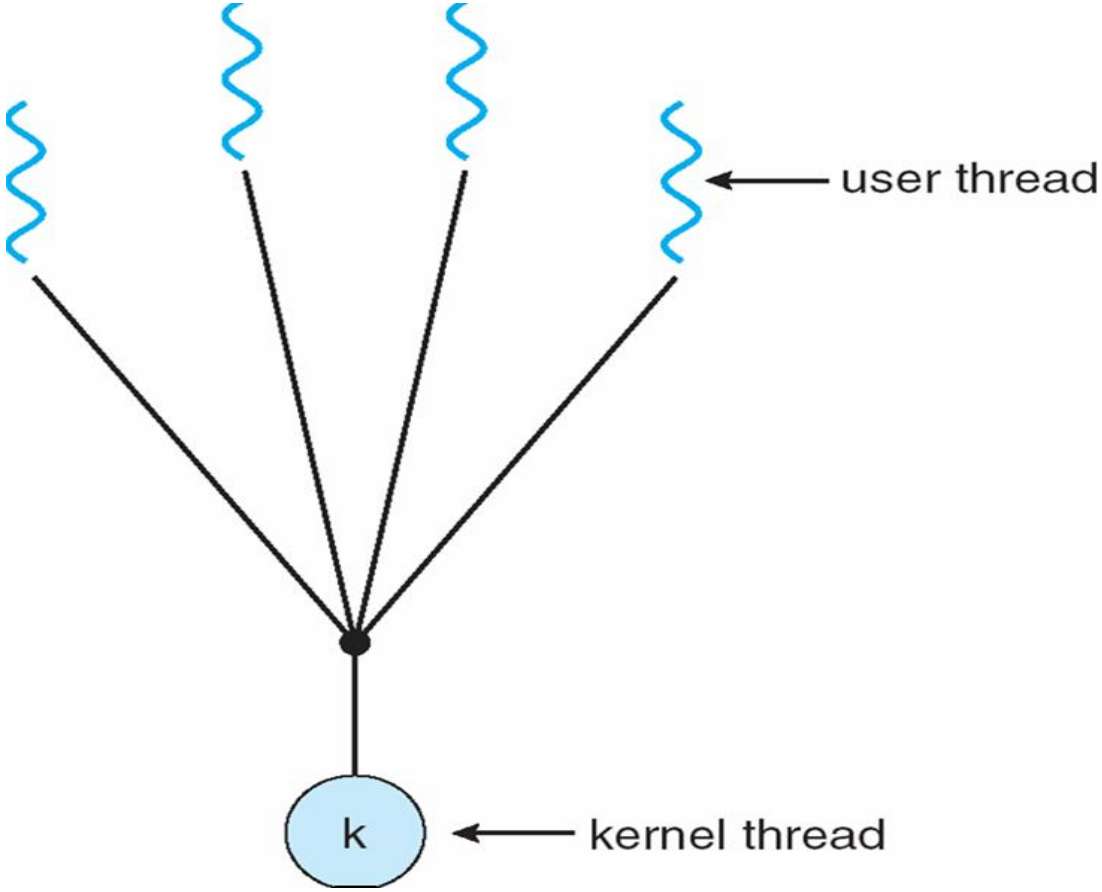


Relationship between Threads and Processes

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Many-to-One Model (1)

- Many user-level threads mapped to single kernel-level thread.

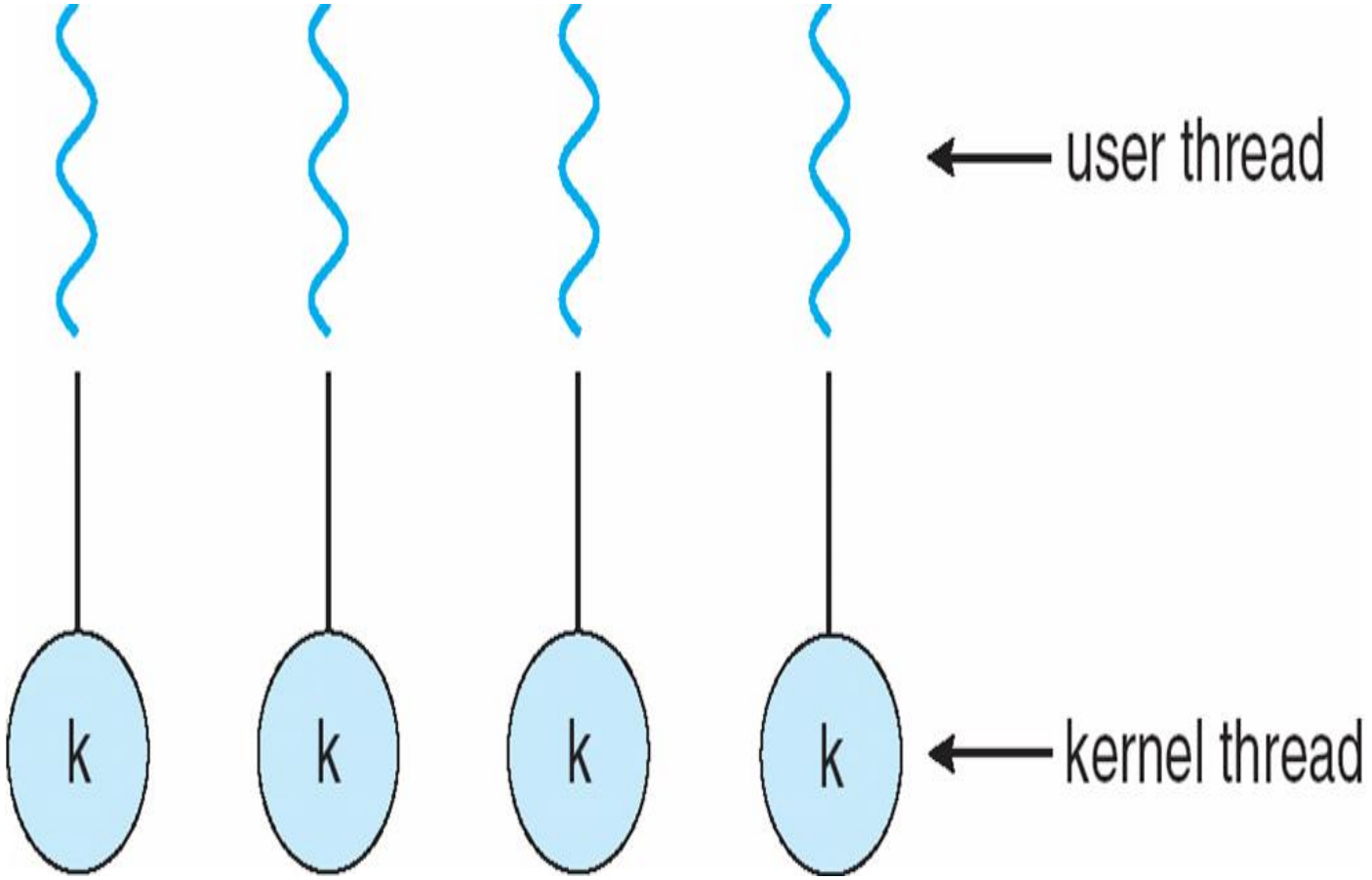


Many-to-One Model (2)

- One thread blocking causes all to block.
- Multiple threads may not run in parallel on multi-core 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 Model (1)

- Each user-level thread maps to kernel-level thread.

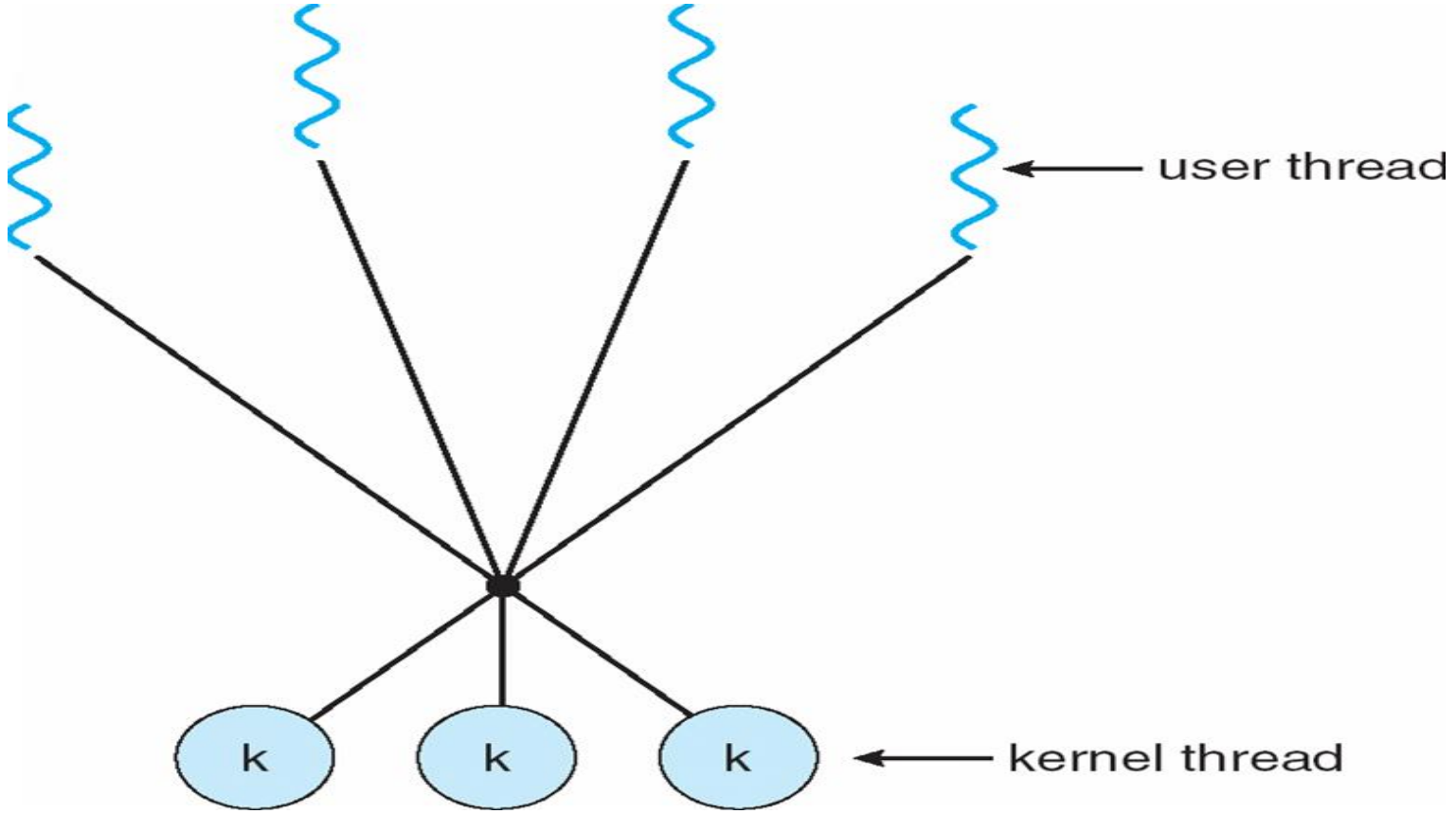


One-to-One Model (2)

- 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
  - Solaris 9 and later

Many-to-Many Model (1)

Allows many user level threads to be mapped to many kernel threads.

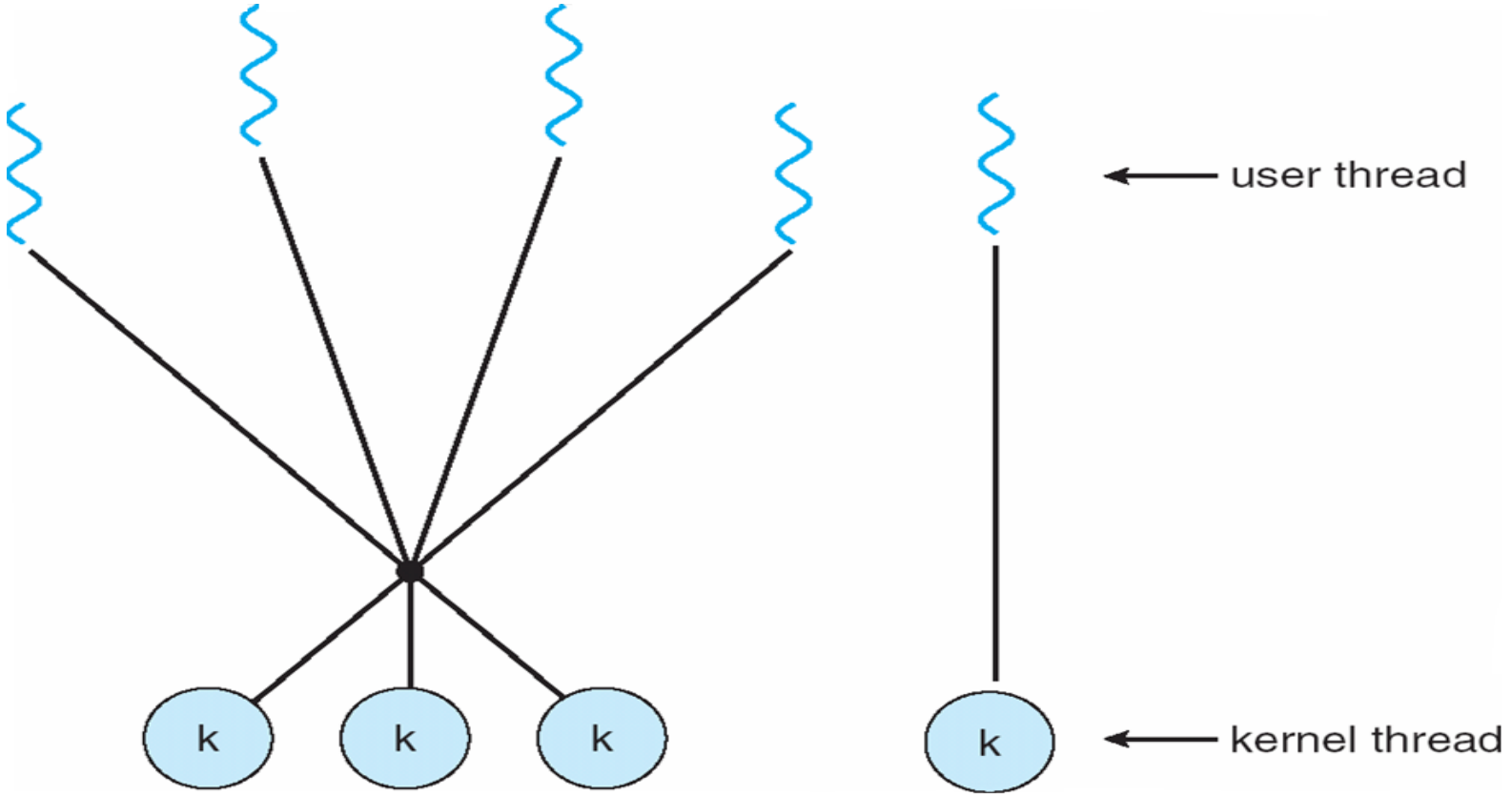


Many-to-Many Model (2)

- Allows the operating system to create a sufficient number of kernel threads.
- Solaris prior to version 9.
- Windows with the *ThreadFiber* package.

Two-level Model (1)

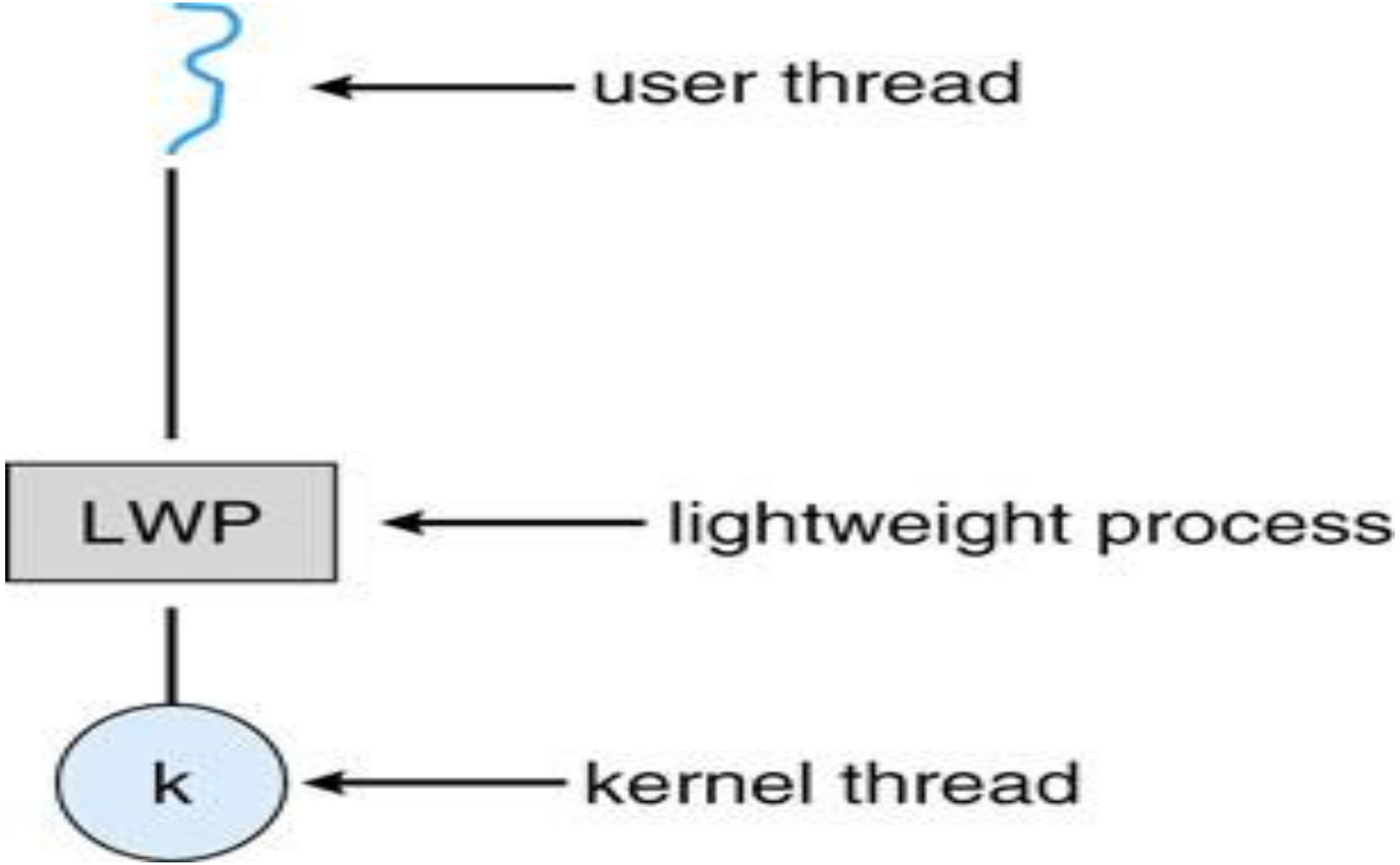
- Similar to Many-to-Many model, except that it allows a user thread to be **bound** to kernel thread.



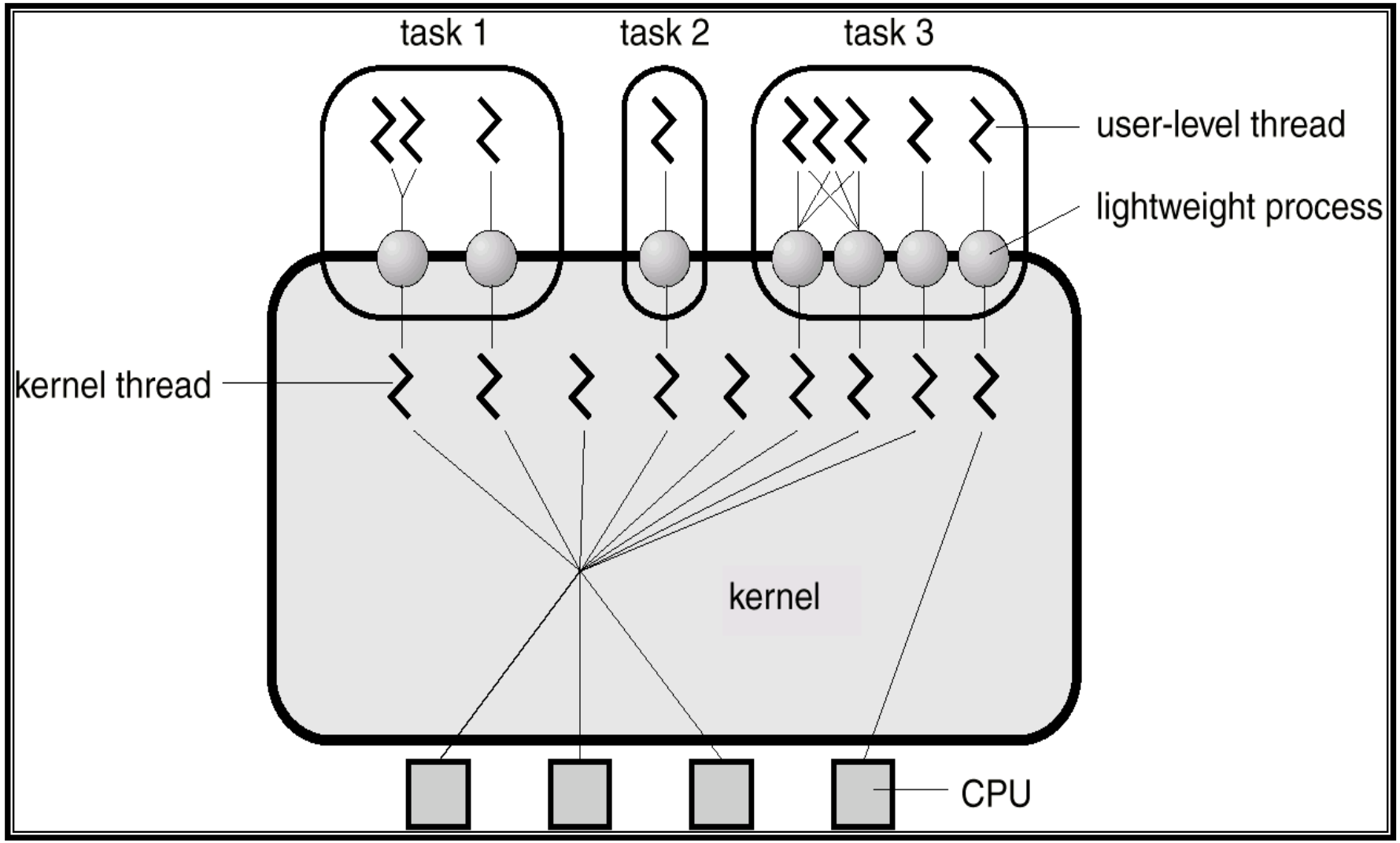
Two-level Model (2)

- Examples:
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Lightweight Process (LWP)

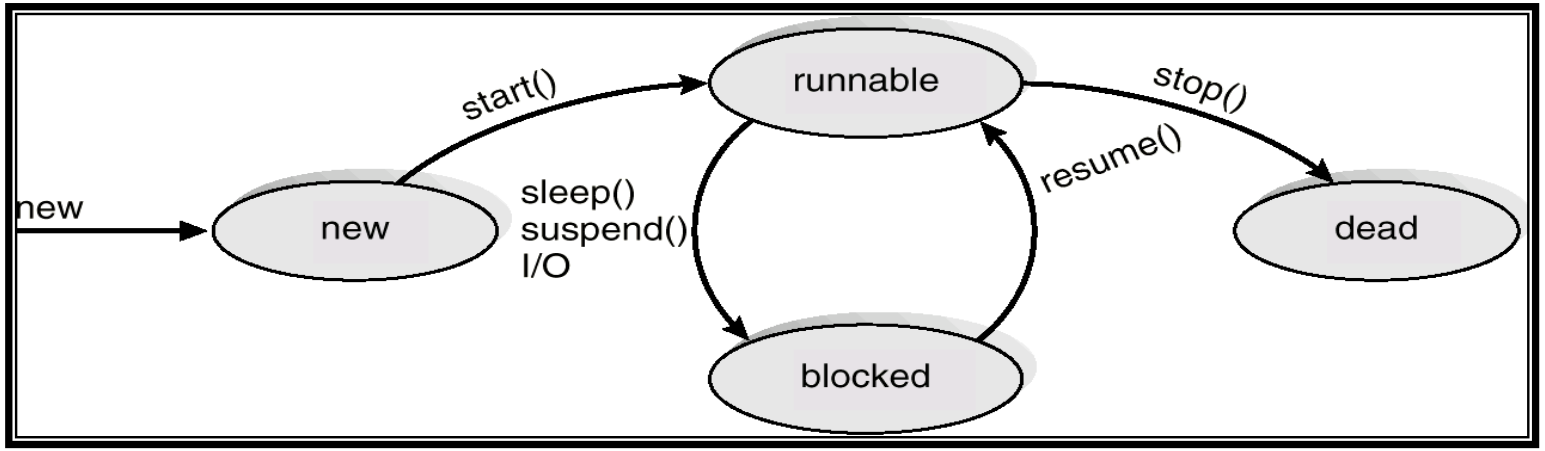


Solaris 2 Threads



Java Threads

- Java threads are managed by the JVM.
- Typically implemented using the threads model provided by underlying OS.
- Java threads may be created by:
  - Extending Thread class (at language-level)
  - Implementing the Runnable interface



Threading Issues

- Semantics of **fork()** and **exec()** system calls
  - Does **fork()** duplicate only the calling thread or all threads?
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-local storage
- Scheduler activations

Thread Cancellation

- Terminating a thread before it has finished.
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately.
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled.

Signal Handling (1)

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A signal handler is used to process signals:
  1. Signal is generated by particular event.
  2. Signal is delivered to a process.
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has default handler that kernel runs when handling signal:
  - User-defined signal handler can override default.
  - For single-threaded, signal delivered to process.

Signal Handling (2)

- Where should a signal be delivered for multi-threaded?
  - 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 Pools

- Create a number of threads in a pool where they await work.
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread.
  - Allows the number of threads in the application(s) to be bound to the size of the pool.

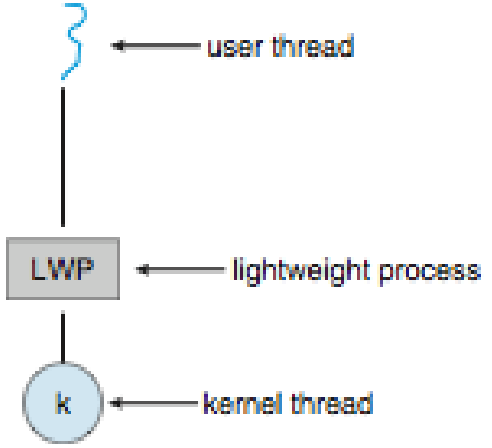
ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Thread-Local Storage

- Thread-local storage (TLS) allows each thread to have its own copy of data.
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool).
- Different from local variables:
  - Local variables visible only during single function invocation.
  - TLS visible across function invocations.
- Similar to **`static`** data:
  - TLS is unique to each thread.

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application/
- Typically use an intermediate data structure between user and kernel threads – lightweight process (LWP):
  - Appears to be a virtual processor on which process can schedule user thread to run.
  - Each LWP attached to kernel thread.
  - How many LWPs to create?
- Scheduler activations provide upcalls – a communication mechanism from the kernel to the upcall handler in the thread library.
- This communication allows an application to maintain the correct number kernel threads.



Operating System Examples

- Windows Threads
- Linux Threads

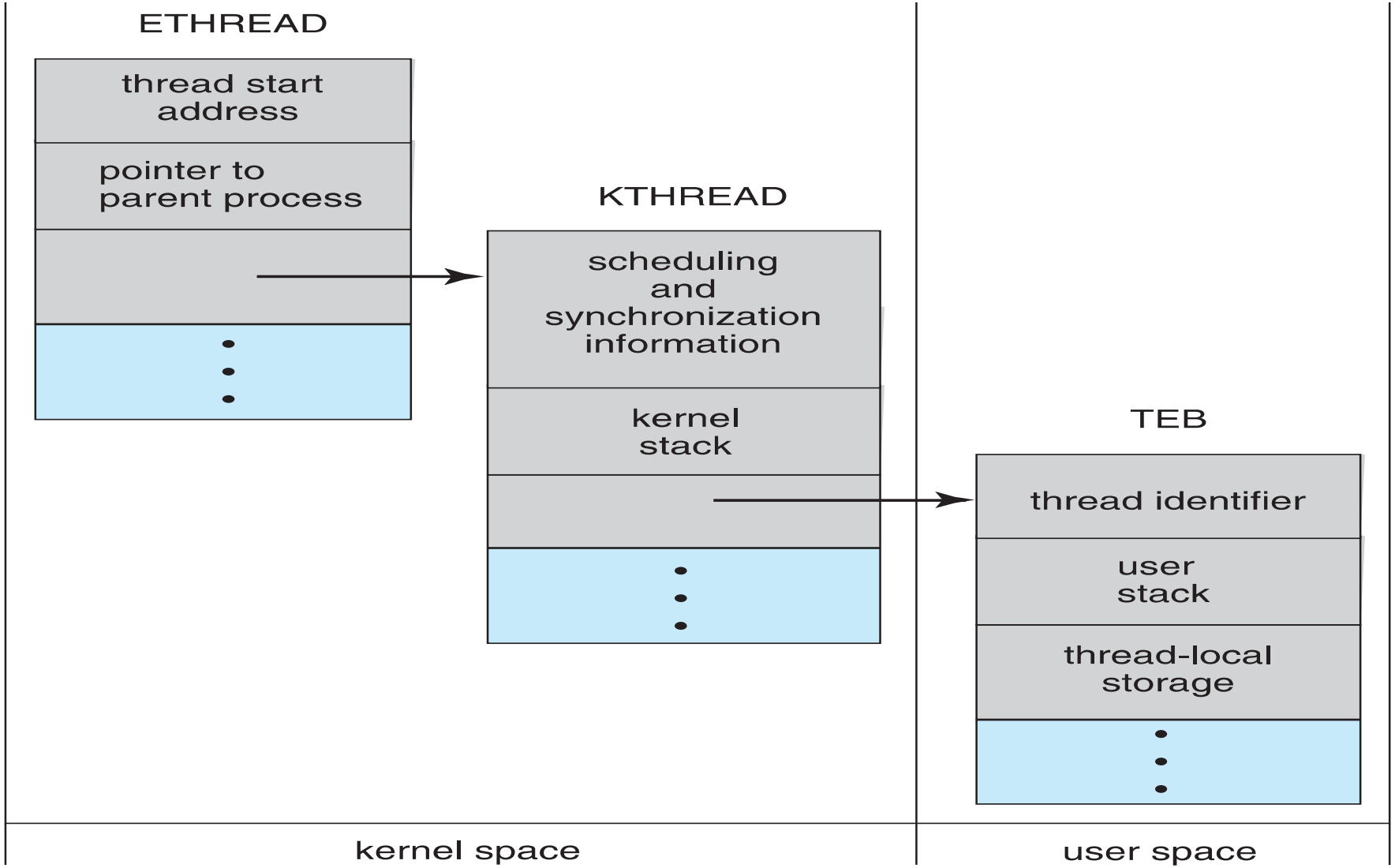
Windows Threads (1)

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7.
- 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 (2)

- 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).
  - Flags control behavior.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **`struct task_struct`** points to process data structures (shared or unique).

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Thread Python Implementation

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Starting a new Thread

EXAMPLE:

```
#!/usr/bin/python
import thread
import time
# Define a function for the thread
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print "%s: %s" % ( threadName, time.ctime(time.time()) )
# Create two threads as follows
try:
   thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print "Error: unable to start thread"
while 1:
   pass
```

Creating Thread using Threading Module

- To implement a new thread using the threading module, need to do the following:
  - Define a new subclass of the Thread class.
  - Override the __init__(self [,args]) method to add additional arguments.
  - override the run(self [,args]) method to implement what the thread should do when started.
- Once you have created the new Thread subclass, you can create an instance of it and then start a new thread by invoking the start(), which will in turn call run() method.

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

EXAMPLE:

```
#!/usr/bin/python
import threading
import time
exitFlag = 0
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter)
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name
def print_time(threadName, delay, counter):
     While counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName,
time.ctime(time.time()))
        counter -= 1
```

```
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
print "Exiting Main Thread"
```

When the above code is executed, it produces the following result:

```
Starting Thread-1
Starting Thread-2
Exiting Main Thread
Thread-1: Thu Mar 21 09:10:03 2013
Thread-1: Thu Mar 21 09:10:04 2013
Thread-2: Thu Mar 21 09:10:04 2013
Thread-1: Thu Mar 21 09:10:05 2013
Thread-1: Thu Mar 21 09:10:06 2013
Thread-2: Thu Mar 21 09:10:06 2013
Thread-1: Thu Mar 21 09:10:07 2013
Exiting Thread-1
Thread-2: Thu Mar 21 09:10:08 2013
Thread-2: Thu Mar 21 09:10:10 2013
Thread-2: Thu Mar 21 09:10:12 2013
Exiting Thread-2
```

Synchronizing Threads

- The threading module provided with Python includes a simple-toimplement
- locking mechanism that will allow you to synchronize threads
- ● A new lock is created by calling the Lock() method, which returns the new
- lock.
- ● The acquire(blocking) method of the new lock object would be used to
- force threads to run synchronously
- ● The optional blocking parameter enables you to control whether the
- thread will wait to acquire the lock.
- ● If blocking is set to 0, the thread will return immediately with a 0 value if
- the lock cannot be acquired and with a 1 if the lock was acquired. If
- blocking is set to 1, the thread will block and wait for the lock to be
- released.
- ● The release() method of the the new lock object would be used to release
- the lock when it is no longer required.

SCHOOL OF HUMAN SCIENCES

EXAMPLE:

```
#!/usr/bin/python
import threading
import time
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        # Get lock to synchronize threads
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # Free lock to release next thread
        threadLock.release()
def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName,
time.ctime(time.time()))
        counter -= 1
threadLock = threading.Lock()
threads = []
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
# Add threads to thread list
threads.append(thread1)
threads.append(thread2)
# Wait for all threads to complete
for t in threads:
    t.join()
print "Exiting Main Thread"
When the above code is executed, it produces the
following result:

Starting Thread-1
Starting Thread-2
Thread-1: Thu Mar 21 09:11:28 2013
Thread-1: Thu Mar 21 09:11:29 2013
Thread-1: Thu Mar 21 09:11:30 2013
Thread-2: Thu Mar 21 09:11:32 2013
Thread-2: Thu Mar 21 09:11:34 2013
Thread-2: Thu Mar 21 09:11:36 2013
Exiting Main Thread
```

MultiProcessing

```
1  import multiprocessing
2  def spawn():
3    print('test!')
4
5  if __name__ == '__main__':
6    for i in range(5):
7      p = multiprocessing.Process(target=spawn)
8      p.start()
```

If you have a shared database, you want to make sure that you're waiting for relevant processes to finish before starting new ones.

```
1  for i in range(5):
2    p = multiprocessing.Process(target=spawn)
3    p.start()
4    p.join() # this line allows you to wait for processes
```

MultiProcessing

```
1  import multiprocessing
2  def spawn(num):
3    print(num)
4
5  if __name__ == '__main__':
6    for i in range(25):
7      ## right here
8      p = multiprocessing.Process(target=spawn, args=(i,))
9      p.start()
```

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

R Implementation

Methods of Parallelization

- There are two main ways in which code can be parallelized, via *sockets* or via *forking*. These function slightly differently:
  - The *socket* approach launches a new version of R on each core. Technically this connection is done via networking (e.g. the same as if you connected to a remote server), but the connection is happening all on your own computer[3] I mention this because you may get a warning from your computer asking whether to allow R to accept incoming connections, you should allow it.
  - The *forking* approach copies the entire current version of R and moves it to a new core.

Methods of Parallelization

- Socket:
  - Pro: Works on any system (including Windows).
  - Pro: Each process on each node is unique so it can't cross-contaminate.
  - Con: Each process is unique so it will be slower
  - Con: Things such as package loading need to be done in each process separately. Variables defined on your main version of R don't exist on each core unless explicitly placed there.
  - Con: More complicated to implement.
- Forking:
  - Con: Only works on POSIX systems (Mac, Linux, Unix, BSD) and not Windows.
  - Con: Because processes are duplicates, it can cause issues specifically with random number generation (which should usually be handled by parallel in the background) or when running in a GUI (such as RStudio). This doesn't come up often, but if you get odd behavior, this may be the case.
  - Pro: Faster than sockets.
  - Pro: Because it copies the existing version of R, your entire workspace exists in each process.
  - Pro: Trivially easy to implement.

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Forking with `mclapply`

The most straightforward way to enable parallel processing is by switching from using `lapply` to `mclapply`. (Note I'm using `system.time` instead of `profvis` here because I only care about running time, not profiling.)

```
library(lme4)
```

```
## Loading required package: Matrix
```

```
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time(save1 <- lapply(1:100, f))
```

```
##    user  system elapsed
##   2.048   0.019   2.084
```

```
system.time(save2 <- mclapply(1:100, f))
```

```
##    user  system elapsed
##   1.295   0.150   1.471
```

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Using sockets with `parLapply`

As promised, the sockets approach to parallel processing is more complicated and a bit slower, but works on Windows systems. The general process we'll follow is

1. Start a cluster with n nodes.
2. Execute any pre-processing code necessary in each node (e.g. loading a package)
3. Use `par*apply` as a replacement for `*apply`. Note that unlike `mcapply`, this is *not* a drop-in replacement.
4. Destroy the cluster (not necessary, but best practices).

Starting a cluster

The function to start a cluster is `makeCluster` which takes in as an argument the number of cores:

```
numCores <- detectCores()
numCores
```

```
## [1] 4
```

```
cl <- makeCluster(numCores)
```

The function takes an argument `type` which can be either `PSOCK` (the socket version) or `FORK` (the fork version). Generally, `mclapply` should be used for the forking approach, so there's no need to change this.

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

```r
### lapply
library(parallel)
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time({
  library(lme4)
  save1 <- lapply(1:100, f)
})

### mclapply
library(parallel)
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time({
  library(lme4)
  save2 <- mclapply(1:100, f)
})


### mclapply
library(parallel)
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time({
  cl <- makeCluster(detectCores())
  clusterEvalQ(cl, library(lme4))
  save3 <- parLapply(cl, 1:100, f)
  stopCluster(cl)
})
```

lapply	mclapply	parLapply
4.237	4.087	6.954

```
library(parallel)
library(MASS)

starts <- rep(100, 40)
fx <- function(nstart) kmeans(Boston, 4, nstart=nstart)
numCores <- detectCores()
numCores
```

```
## [1] 8
```

```
system.time(
  results <- lapply(starts, fx)
)
```

```
##    user  system elapsed
##   1.346   0.024   1.372
```

```
system.time(
  results <- mclapply(starts, fx, mc.cores = numCores)
)
```

```
##    user  system elapsed
##   0.801   0.178   0.367
```

SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY