Operating Systems
COMP2006

Processes and Threads
Lecture 2

Process and Thread

References: Silberschatz, Galvin, and Gagne, *Operating System Concepts*, Chapters 3, 4

Topics:

- Process concept.
- Operations on processes
- Inter-process communication.
- Threads.

Process Concept

- Multiprogramming allows multiple programs to be loaded into memory and to be executed concurrently
  - Need stricter control on the various programs.
- In a batch system, we call the program as *jobs*;
  - In a time-shared system, we call them *user programs* or *tasks*;
  - Textbook uses the terms *job* and *process*.
- *Process* is a program in execution;
  - A program (executable file containing list of instructions) is passive
    - It does nothing unless its instructions are executed.
  - A process has a Program Counter (PC)
    - It specifies the next instruction to execute
    - Process execution progresses sequentially – the CPU executes one instruction of the process after another until the process completes.
  - A process needs resources to accomplish its task
    - Resources, e.g. CPU time, memory, files, and I/O devices.
    - Several processes may execute concurrently by multiplexing CPU among them.

Process Concept (cont.)

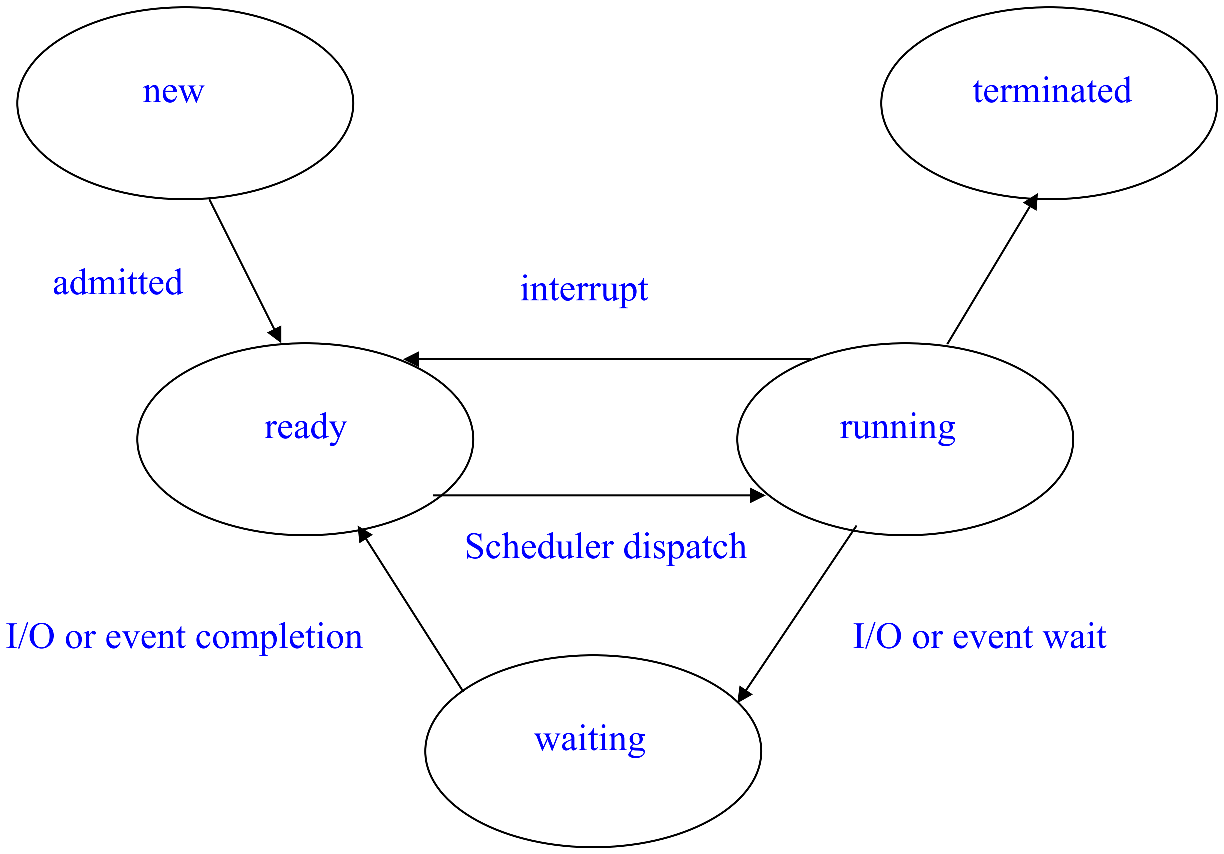
- A process includes:
  - Text section: the program code
  - Program counter: contains a pointer to the next instruction to execute
  - Contents of processor registers.
  - Stack: contains temporary data; e.g., subroutine parameters, return addresses, temporary variables.
  - Data section: contains global variables.
  - Heap: memory that is dynamically allocated during run time
- Two processes may be associated with the same program
  - They are considered two separate execution sequences.
- Process Management of an OS is responsible for the following:
  - Creation and deletion of user and system processes.
  - Suspension and resumption of processes.
  - Provision of mechanisms for process synchronisation.
  - Provision of mechanisms for process communications.
  - Provision of mechanisms for deadlock handling.

Process Concept (cont.)

- A process can be **user** level or **kernel** level
- As a process executes, it changes state.
- Each process may be in one of the following process states:
  - *New* – the process is being created.
  - *Running* – instructions are being executed.
  - *Waiting* – the process is waiting for some event to occur.
  - *Ready* – the process is waiting to be assigned to a processor.
  - *Terminated* – the process has finished execution

Process Concept (cont.)

Transition diagram of process state:



Process Concept (cont.)

- Each process is represented by a data structure
  - called Process Control Block (PCB) or Task Control Block
- Each PCB contains:
  - Process number (pid).
  - Process state: new, ready, running, waiting, etc.
  - Program counter: shows the address of next instruction to be executed.
  - CPU registers: the registers vary in number and type, depending on the computer architecture.
  - CPU scheduling information: process priority, pointers to schedule queues, etc.
  - Memory management information: base and limit register, page tables, etc.
  - Accounting information: amount of CPU and time used, etc.
  - I/O status information: the list of I/O devices allocated to this process, a list of open files, etc.

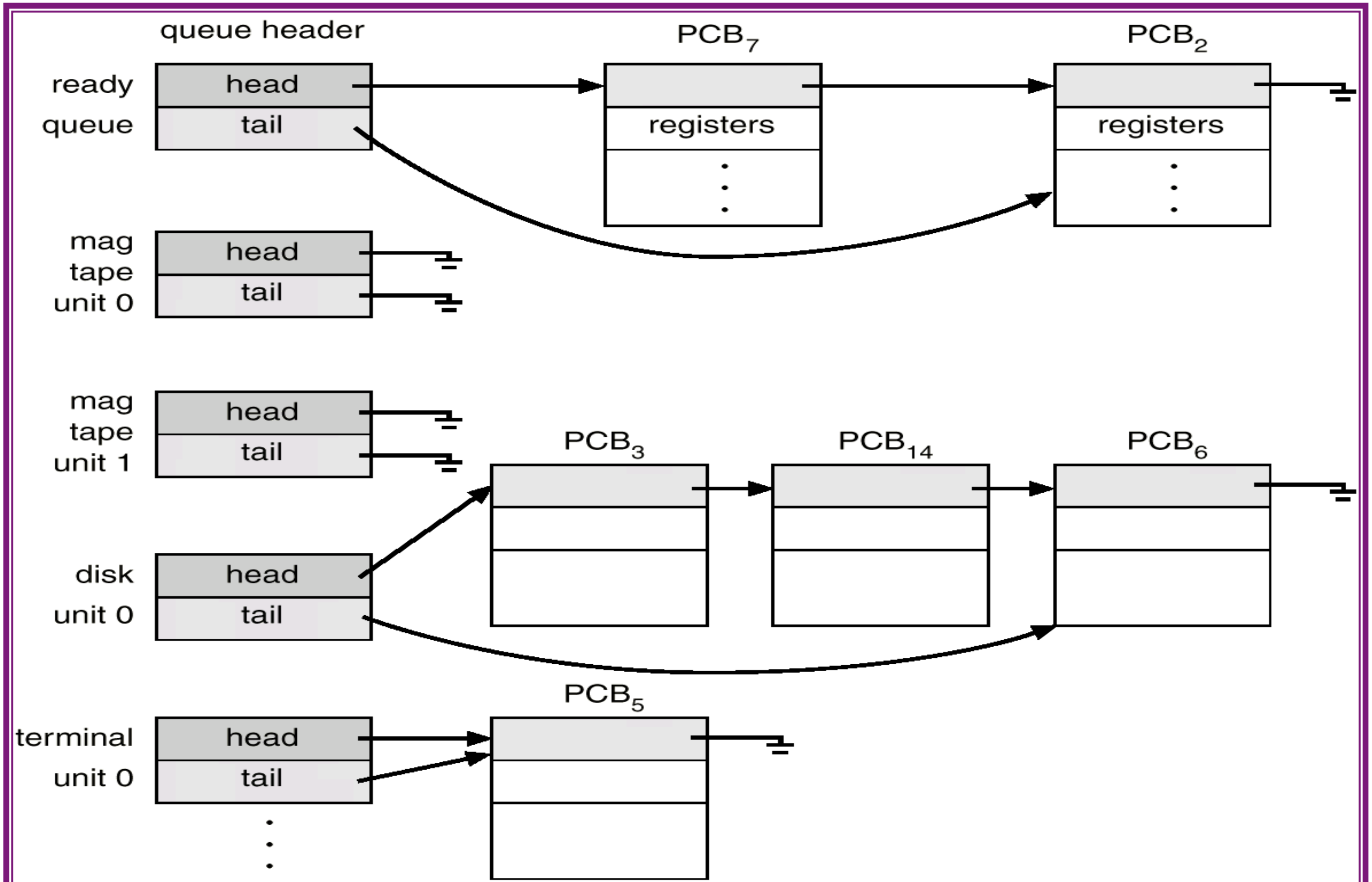
Process Concept (cont.)

- PCB in Linux is a C structure *task_struct* that contains:
  - long state; // *state of the process*
  - struct sched_entity se; // *scheduling information*
  - struct task_struct *parent; // *the parent of this process*
  - struct task_struct *children; // *the children of this process*
  - struct file_struct *files; // *list of open files*
  - struct mm_struct *mm // *this process's address space*
- **Parent** of a process is a process that creates it
  - Its **children** are processes that it creates
  - **Siblings** are children with the same parent
- All active processes are represented by a double linked list of *task_struct*
  - A variable *current* is used as a pointer to currently executing process.

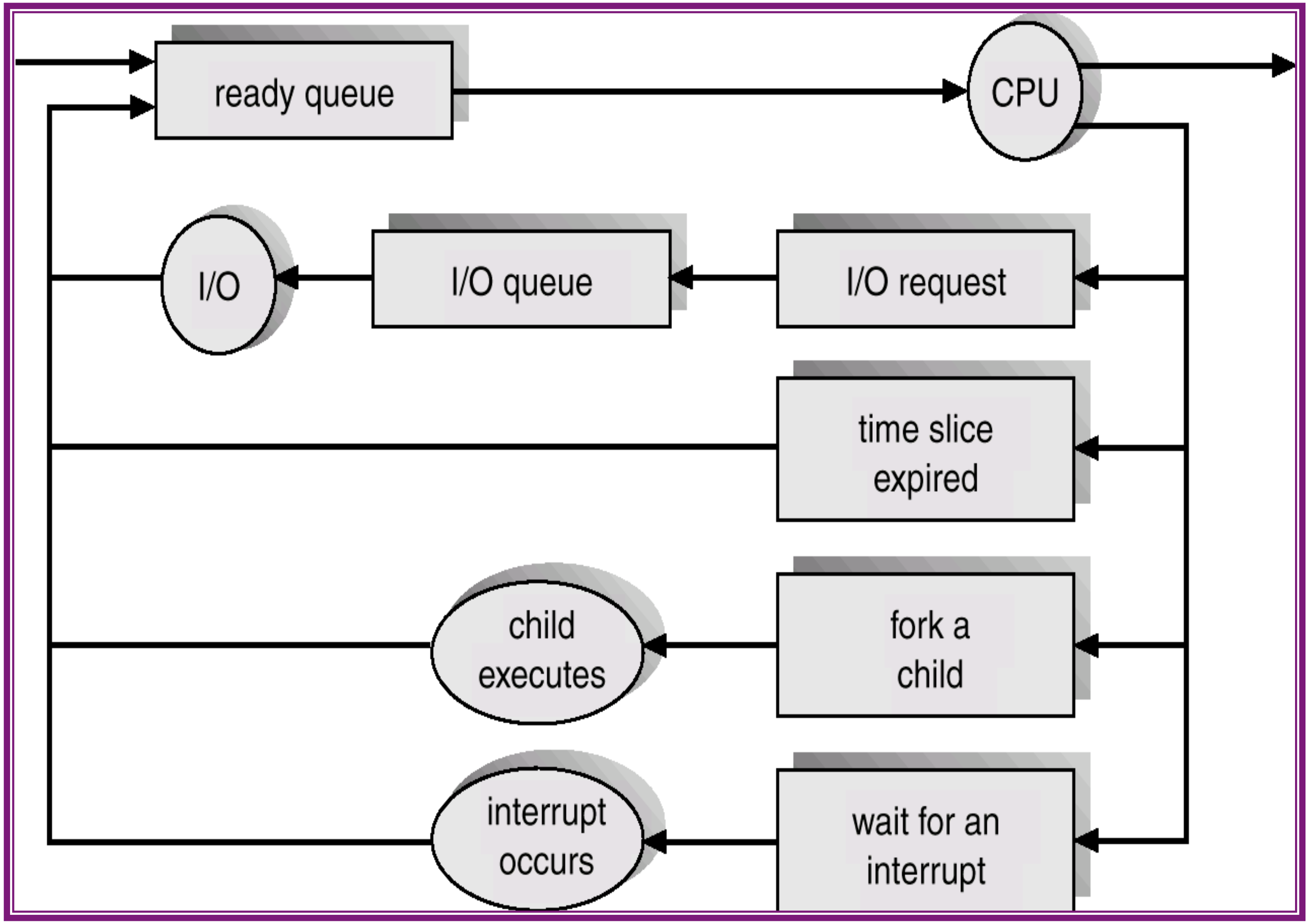
Process Scheduling

- OS uses process/CPU scheduler to select one of available processes to be executed by CPU
  - The scheduler is used to meet system objective
    - Multiprogramming aims to maximize CPU utilization
    - Timesharing aims to switch processes frequently so that users can interact with their running programs
- OS keeps several queues (implemented as link lists):
  - ***Job queue*** – All processes entering the system are put in a job queue.
  - ***Ready queue*** – A set of all processes residing in main memory, ready and waiting to execute.
  - ***Device queues*** – A set of processes waiting for an I/O device
    - Each device has its own device queue.
- Process migrates between the various queues
  - Each queue is a list of PCBs
  - When a process terminates, the OS removes its PCB from the queue and de-allocates its resources.

Ready Queue and Various I/O Device Queues

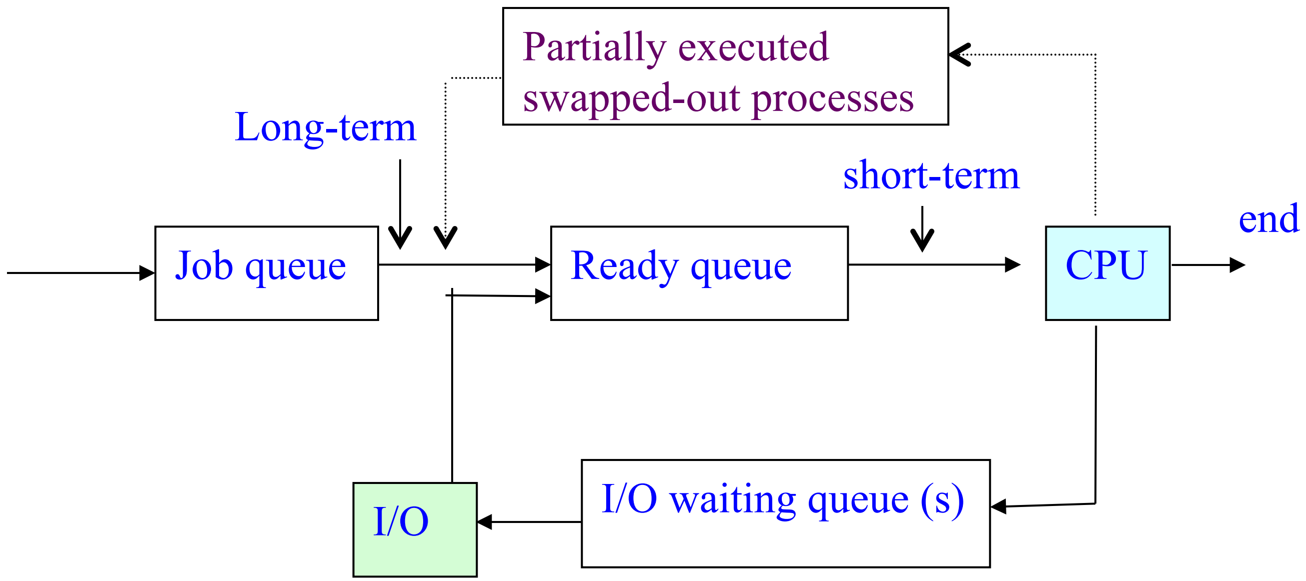


Representation of Process Scheduling



Schedulers

- A process migrates between various scheduling queues.
  - Selection on which process to migrate is done by the appropriate scheduler.
- Long-term scheduler (or job scheduler) selects which processes should be brought into the ready queue.
  - It is invoked very infrequently (seconds, minutes) → the scheduler can be slow
  - It controls the degree of multiprogramming (the number of processes in memory).
  - A time sharing system often has no long-term scheduler.



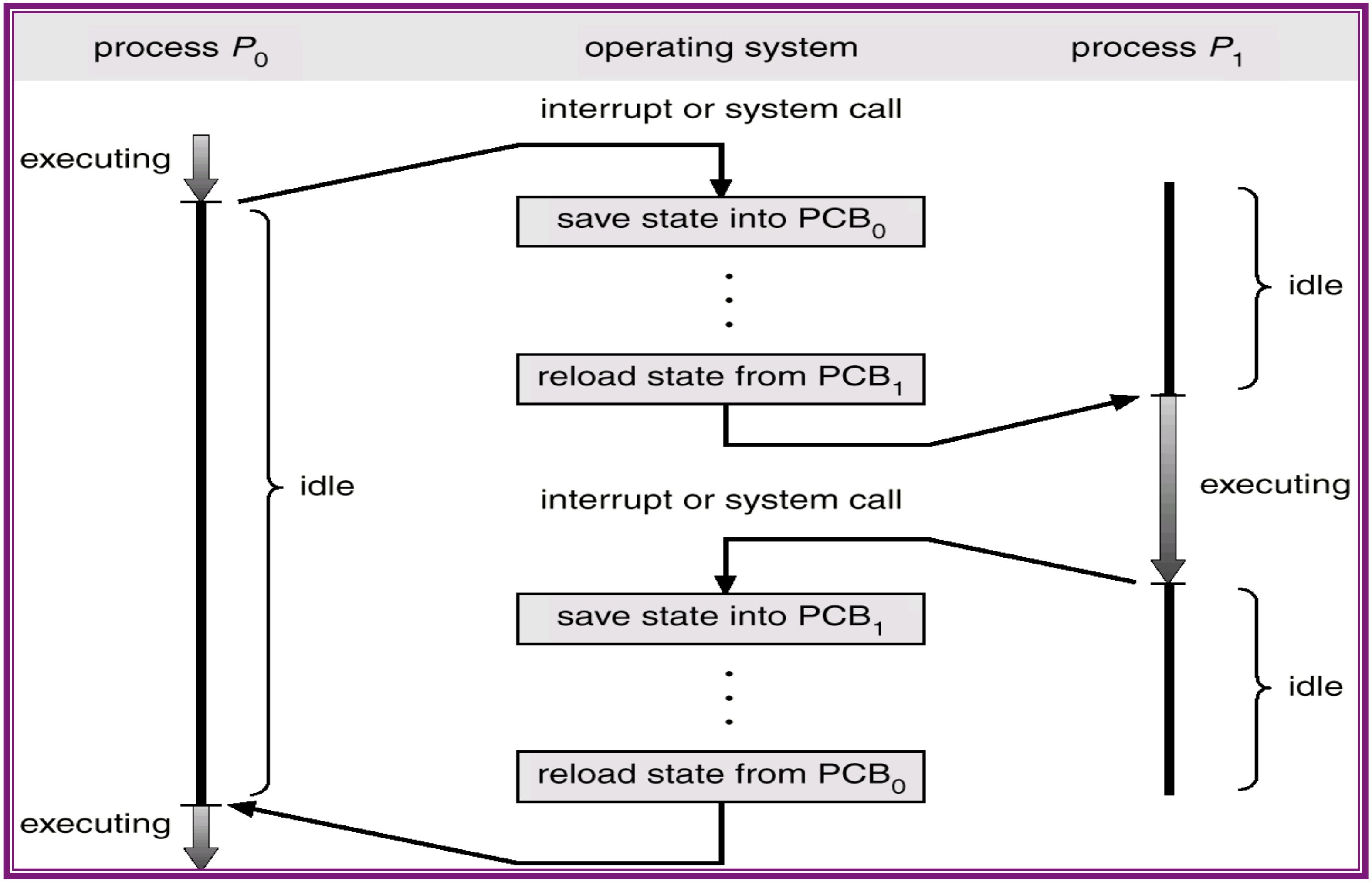
Schedulers (cont.)

- Short-term scheduler (or CPU scheduler) selects or **dispatches** a process to be executed next and allocated CPU
  - It is invoked very frequently (milliseconds); thus the scheduler must be fast.
- A process can be described as either:
  - I/O-bound process
    - It spends more time doing I/O than computations
    - It has many short CPU bursts.
  - CPU-bound process
    - It spends more time doing computations
    - It has few very long CPU bursts.
- Long-term scheduler needs to select a good mix of I/O bound and CPU bound processes.
- Medium-term scheduler is sometimes introduced to remove processes from memory to reduce the degree of multiprogramming.

Context Switch

- When CPU switches to another process, the system must save the *context* of the old process and load the *context* for the new process
  - The OS suspends the old process and runs the new one
  - The saved context is needed when the old process resume its execution
- Context-switch time is overhead
  - The system does no useful work while switching
  - Performance bottleneck.
- Context-switch-time is dependent on hardware support
  - Typically from a few milliseconds.
- When do we switch CPU?
  - Job voluntarily waits (system calls).
  - Interrupt: Higher priority event/job needs attention.
  - Interrupt: Timer.

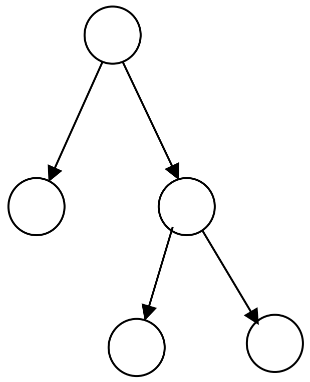
CPU Switch From Process to Process



Operations on processes

Process Creation

- Parent process creates child processes, which in turn can create other processes, forming a tree of processes
  - in Unix use system call fork( ).
  - Each process is known by its process ID (an integer value)
    - The init process has PID = 1; it is the root parent process for all user processes
    - Use *ps –el* command to list all processes active in the system
  - A process needs resources (CPU, memory, files, I/O, etc.).



- When a process creates a sub-process:

  For resources
  - Parent and child processes may share all resources.
  - Child processes may share subset of parent’s resources.
  - Parent and child processes may share no resources.

  For execution
  - Parent and child processes may execute concurrently → Linux
  - Parent may wait until child process terminates
    - parent calls wait() to move itself out from ready queue

  For address space
  - Child process address space is duplicate of parent’s.
  - Child has a program loaded into it → in UNIX: use exec ().

UNIX example

- **fork ()** system call creates a new process.

printf(“one \n”);	
pid = fork ();	PC
printf (“two \n”);	

After executing fork ():

printf(“one \n”);	
pid = fork ();	
printf (“two \n”);	PC

printf(“one \n”);	
pid = fork ();	
printf (“two \n”);	PC

Operations (cont.)

```
int main ()
{
    pid_t pid;

    pid = fork ();   // create a child process

    if (pid < 0) {  // error occurs
        return 1;
    }
    else if (pid == 0) { // child process; OS returns 0 to the child process
        execute_function_for_child();  // alternatively, use execlp ("/bin/ls", "ls",
                                          NULL) to execute the command "ls"
    }
    else { // pid > 0 is the child's PID returned to the parent process
        wait (NULL);  // the parent process waits for its child's termination
    }
}
```

Operations (cont.)

Process termination

- A process terminates when it executes last statement and/or calls **exit ()**;
  - OS will deallocate the child's resources
  - the child can return its status value to parent that calls **pid = wait (&status)**
    - If the parent does not call wait(), the OS does not remove the child from process table
      - The child process becomes a '*zombie*'
      - When the parent terminates, the child becomes *orphan*, and *init* will automatically be its parent.
        - *init* process periodically calls wait() removing the orphans from process table
- Parent may terminate execution of child processes **(abort);** Reasons for termination:
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting
    - Some OS does not allow child to continue if its parent terminates.
    - Linux does not terminate the child when its parent terminates

Cooperating Processes

- Concurrent processes executing in OS may be either independent or cooperating processes.
  - *Independent processes* cannot affect or be affected by the execution of another process.
  - *Cooperating processes* can affect or be affected by the execution of another process.
- Advantages of process cooperation:
  - Information sharing
    - Several users may need the same piece of information.
  - Computation speed-up
    - Break a task into subtasks and execute in parallel.
  - Modularity
    - Divide the system function into separate processes.
  - Convenience
    - A user may want to do editing, printing in parallel.
- Cooperating processes can communicate via:
  - **shared-memory** (Read Section 3.5.1 POSIX Shared memory)
  - **message passing**

Shared memory vs. message passing

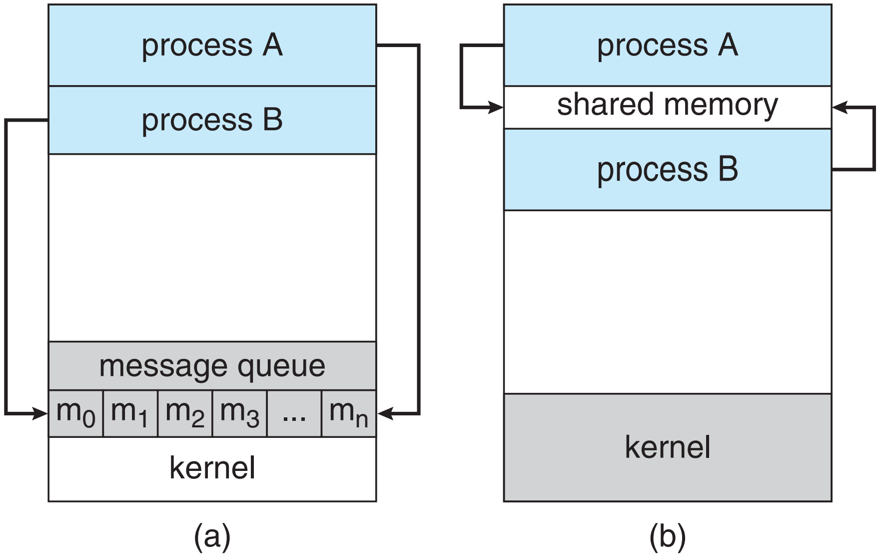


Fig. 3.12 (textbook)

Shared memory

- A process uses a system call to create a shared memory region
  - Other process that wants to communicate via the memory region must attach to it to its address space (use a system call)
  - Once established, all accesses to the shared memory are as if assessing normal memory area (no system call)
    - However, all accesses must be synchronized

Message Passing

- Use a system call to exchange message between processes
  - More time consuming
  - OS handles synchronization between processes
  - Good for distributed system with small amount of data exchanged
  - Good for multicore system
    - Shared memory suffers from cache coherency issues.

Example of concurrent processes

Producer-consumer problem

- Paradigm for cooperating processes: *producer* process produces information that is consumed by a *consumer* process.
- To allow producer and consumer processes to run concurrently, there is a buffer of items that can be filled by the producer, and emptied by the consumer.
  - *Unbounded-buffer* places no practical limit on the size of the buffer.
  - *Bounded-buffer* assumes that there is a fixed buffer size.
- The buffer can be in a form of shared memory, or provided by the OS via message passing.

Producer-consumer (cont.)

Bounded-buffer with shared memory solution

/* Shared data */

var *buffer*: **array** [0..n-1] of *item*;

in, out: 0..n-1; /* initially *in* = *out* = 0; *in* is the pointer to the next free position; *out* is the pointer to the first full position */

Producer process

repeat

produce an item in *nextp;*

…

// while buffer is full

while (*in*+1 mod *n* == *out*) **do** *no-op*;

buffer[*in*] = *nextp*;

in = *in*+1 **mod** *n*;

until false;

Consumer process

repeat

// while buffer is empty

while (*in* == *out*) **do** *no-op*;

nextc = *buffer*[*out*];

out = *out*+1 **mod** *n*;

…

consume the item in *nextc*;

…

until false;

- Solution is correct, but can only fill up n-1 buffer;
  - How to fill up n items in the buffer?

Message Passing

- Message passing facility provides at least two operations:
  - **Send** (message) – message size fixed or variable.
  - **Receive** (message).
- If P and Q wish to communicate, they need to:
  - Establish a *communication link* between them.
  - Exchange messages via *send/receive.*
- Several methods for logically implementing a link and *send/receive* operations:
  - Direct or indirect communication.
  - Synchronous or asynchronous communication.
  - Automatic or explicit buffering.

Message Passing (cont.)

Direct communication

- In the direct-communication, each process must explicitly name the recipient or sender of the communication.
- Primitives used:
  - **Send** (P, message) – send a message to process P.
  - **Receive** (Q, message) – receive a message from process Q.
- Properties of communication link for this scheme:
  - Links are established automatically.
  - A link is associated with exactly one pair of communicating processes.
  - Between each pair there exists exactly one link.
  - The link may be uni-directional, but is usually bi-directional.

Example

Producer process

repeat

…

produce an item in nextp;

…

send (*consumer, nextp***)**

until *false*;

Consumer process

repeat

receive (*producer, nextc*);

…

consume the item in *nextc;*

…

until *false*;

Message passing (cont.)

Indirect communication

- Messages are sent to and received from mailboxes.
  - Each mailbox has a unique id.
  - Processes can communicate only if they share a mailbox.
- Primitives used:
  - **Send** (A, message): send a message to mailbox A.
  - **Receive** (A, message): receive a message from mailbox A.
- Properties of communication link for this scheme:
  - Link established only if processes share a common mailbox.
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link may be uni-directional or bi-directional.
- Operations:
  - Create a new mailbox.
  - Send and receive messages through mailbox.
  - Destroy a mailbox.

Indirect communication(cont.)

Mailbox sharing

- Consider processes P1, P2, and P3 share a mailbox A, and process P1 sends a message to mailbox A while processes P2 and P3 execute a **receive** from A,
  - Which process will receive the message sent by P1?
- Three possible solutions:
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select the receiver arbitrarily. Sender is notified who the receiver was.

Synchronization

- Message passing can be either *blocking* (*synchronous*) or *nonblocking* (asynchronous).
- Blocking send: The sender is blocked until the message is received.
- Nonblocking send: the sender resumes operation after sending the message.
- Blocking receive: the receiver blocks until message is available.
- Nonblocking receive: receiver retrieves either a valid message or a NULL.

Buffering

- A link has some capacity that determines the number of messages that can reside in it temporarily
- This can be viewed as queue of messages attached to the link; implemented in one of three ways:
  - zero capacity – 0 messages: sender must wait for the receiver (rendezvous); called as no-buffering.
  - bounded capacity – finite length of *n* messages: sender must wait if link is full.
  - unbounded capacity – infinite length: sender never waits.
- For the non-zero capacity (automatic buffering), the sender does not know if a message has arrived at the destination.
  - may use asynchronous communication; the receiver sends back an acknowledgement to the sender.

Exception conditions

- Error may happen during process communication;
- When a failure occurs either in centralized (single machine) or distributed (processes reside in different machine), some error recovery must take place.
  - Process terminates; e.g., process P is waiting for a message from a terminated process Q.

    Solution: OS terminates P or notify P that Q has terminated.
  - Lost messages; e.g., message from P to Q become lost due to hardware error;
    - The most common lost detection method is by using timeout
    - An acknowledgement must be received by sender.

    Solution:
    - OS detects it and resends the message.
    - Sender detects it and resends.
    - OS detects it and lets the sender know.
  - Scrambled messages – message is received but in error
    - It is detected by parity checking, CRC.

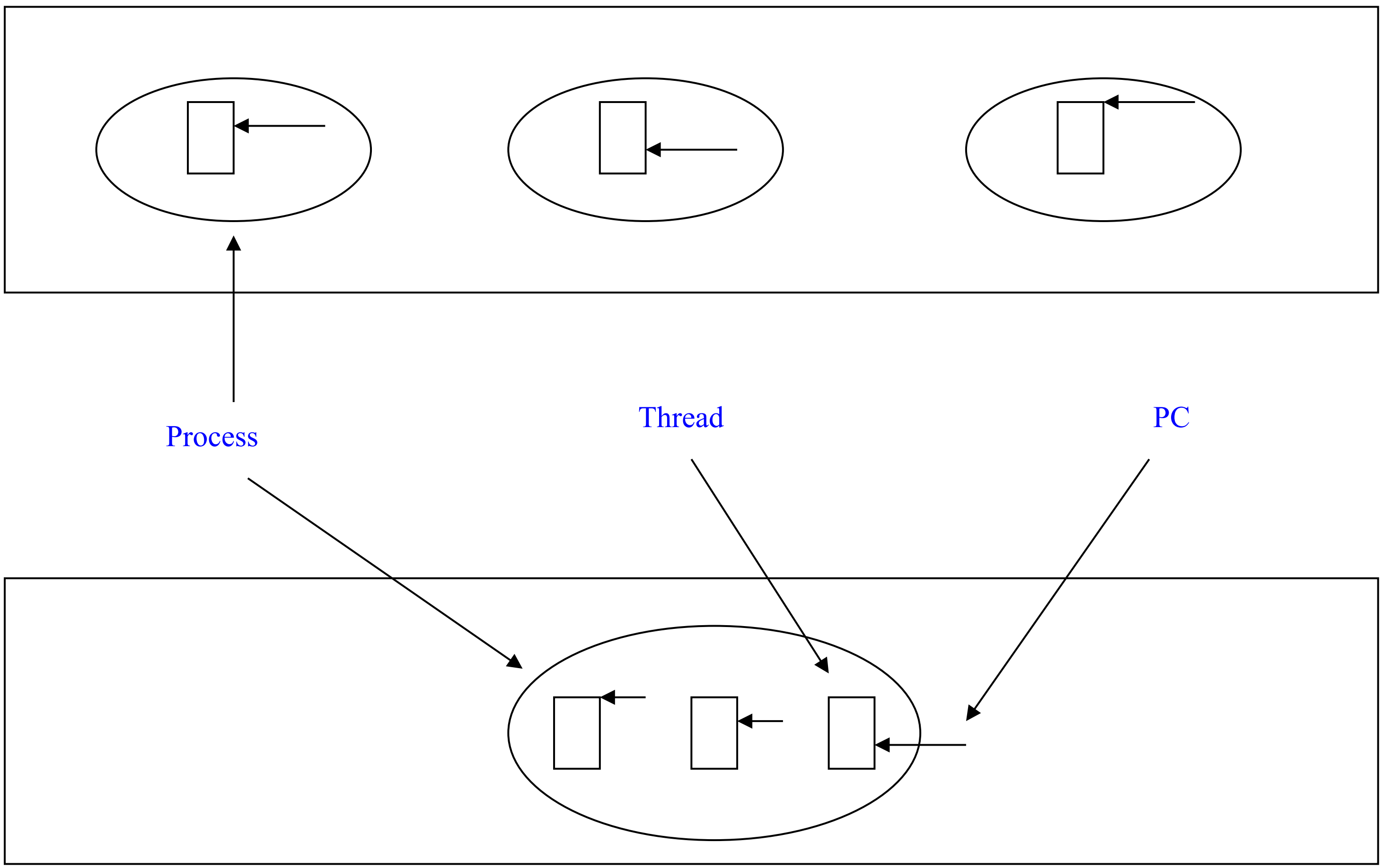
    Solution: resend it.

Threads

- A *thread* of control is an independent sequence of execution of program code in a process.
- A thread (or lightweight process) is a basic unit of CPU utilization.
  - A traditional process (or heavyweight process) is equal to a task with one thread.
- A traditional process has a single thread that has sole possession of the process' s memory and other resources
  - Context switch becomes performance bottleneck
  - Threads are used to avoid the bottleneck.
  - Threads share all the process memory, and other resources.
- Threads within a process:
  - are generally invisible from outside the process.
  - are scheduled and executed independently in the same way as different single-threaded processes.
- On a multiprocessor, different threads may execute on different processors
  - On a uni-processor, threads may interleave their execution arbitrarily.

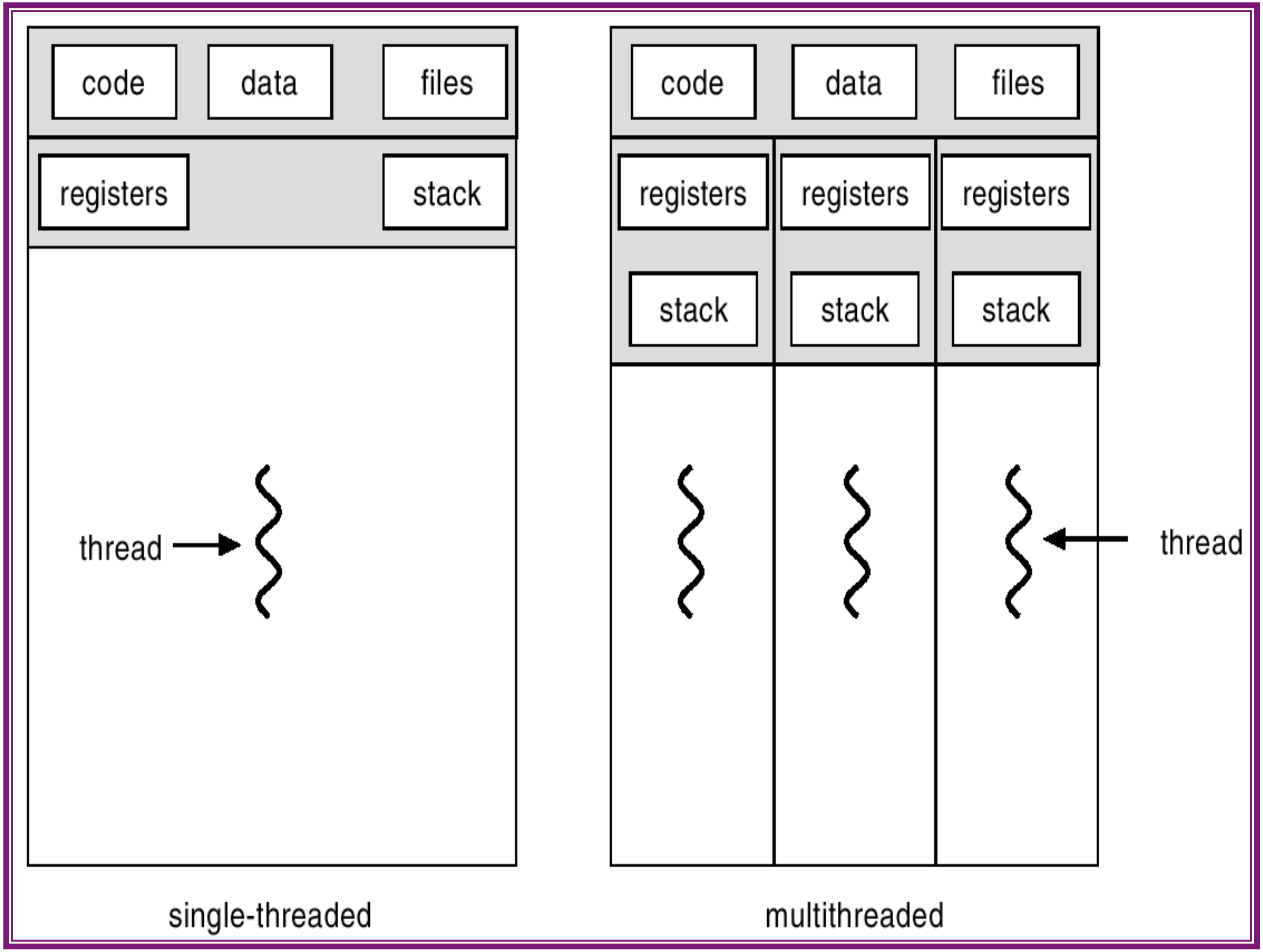
Threads (cont.)

Three processes with one thread each



One process with three threads

Single and Multithreaded Processes



Threads (cont.)

- Threads operate, in many respects, in the same manner as processes:
  - Threads can be in one of several states: ready, blocked, running, or terminated, etc.
  - Threads share CPU
    - Only one thread at a time is running.
  - A thread within a process executes sequentially, and each thread has its own PC and Stack.
  - Thread can create child threads, can block waiting for system calls to complete
    - If one thread is blocked, another can run.
- One major different with process: threads are not independent of one another
  - All threads can access every address in the task
    - A thread can read or write any other thread's stack.
  - There is no protection between threads (within a process)
    - However this should not be necessary since processes may originate from different users and may hostile to one another while threads (within a process) should be designed (by same programmer) to assist one another.

Threads (cont.)

Benefits

- **Responsiveness:** For interactive application, when one thread in the program is blocked and waiting, a second thread in the same task can run.
- **Resource Sharing:** By default, threads share the memory and resources within a process
  - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.
- **Economy:** It is more economical to create and context switch threads than processes;
  - in Solaris 2, creating a process is about 30 times slower than creating a thread, and context switch is about five times slower.
- **Utilization of multiprocessor architectures:** Each thread may run in parallel on a different processor.

Threads (cont.)

User and Kernel Threads

- Threads can be:
  - Kernel-supported threads.
  - User-level threads.
- In kernel-supported threads, a set of system calls similar to those for processes are provided.
  - Supported by the Kernel;
    - Kernel does context switch from one thread to another → time consuming.
  - Windows XP, Mac OS X, Solaris 2, Tru64 UNIX, BeOS, Linux.

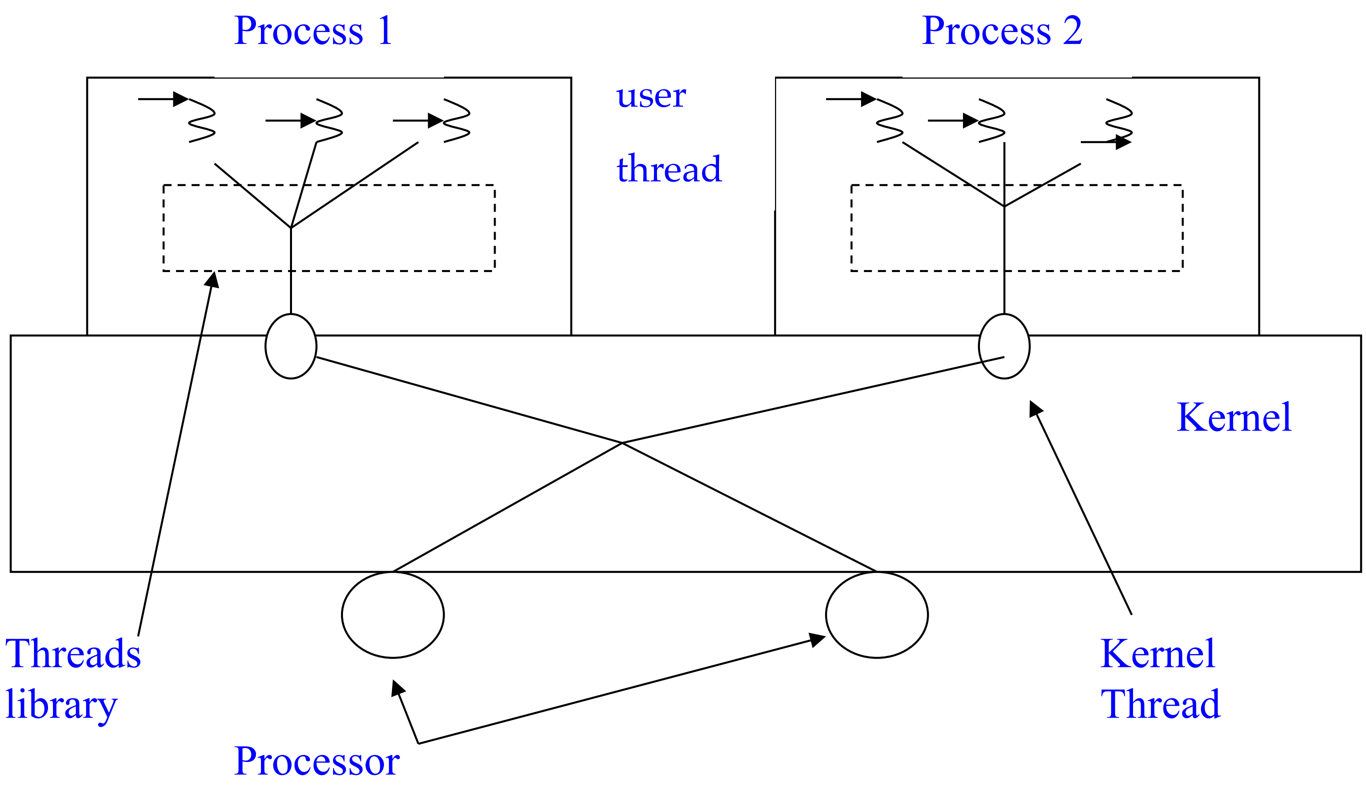
Threads (cont.)

- User-level threads: supported above the kernel, via a set of library calls at the user level.
  - Kernel does not know that there are user-level threads.
  - Thread management is done by user-level threads library.
  - Faster than kernel-level.
  - Drawback: Kernel considers a set of user-level threads as a single thread;
    - if any user level thread in the set is blocked (e.g., I/O), then other threads in the set cannot run.
  - POSIX Pthreads, Mach C-threads, Solaris 2 UI-threads.
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

User to Kernel Threads Mapping

Many-to-one Model:

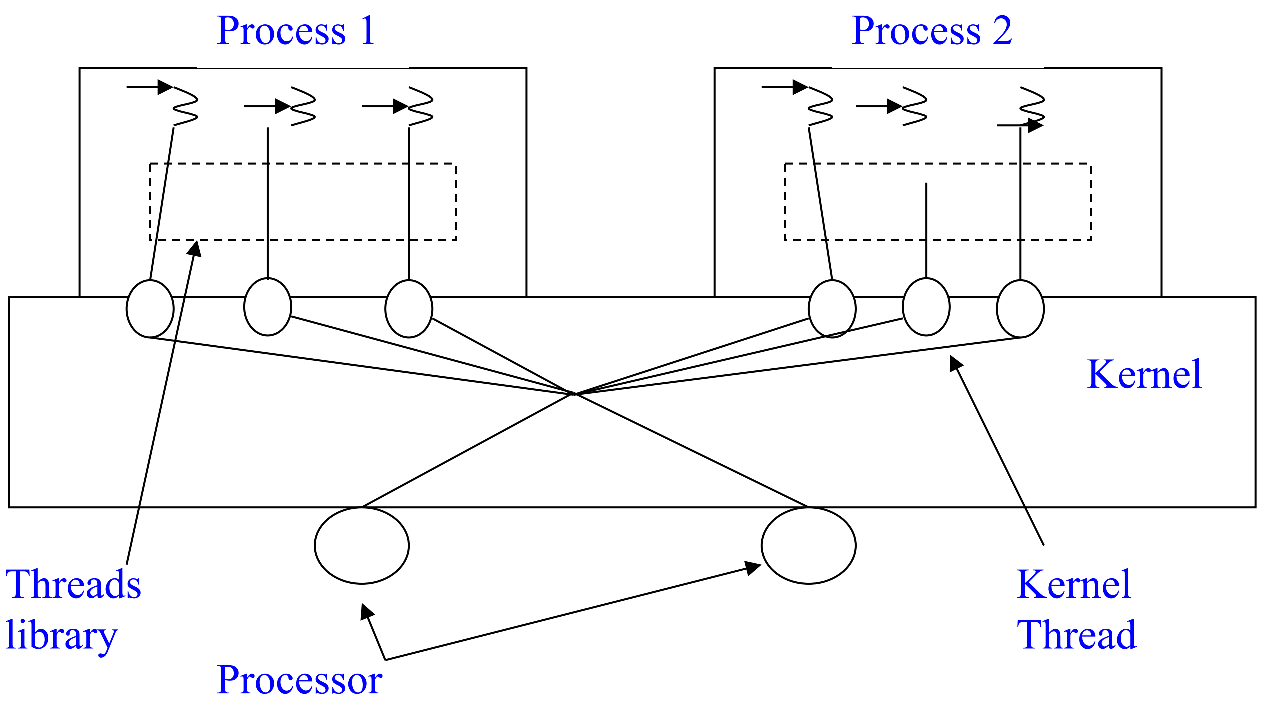
- Many user-level threads are mapped to one kernel thread.
- Multiple threads within a process cannot run in parallel on multiprocessors.
- Used by Solaris 2 (Green threads), and threads libraries for systems with no kernel-threads.



Mapping (cont.)

One-to-one Model:

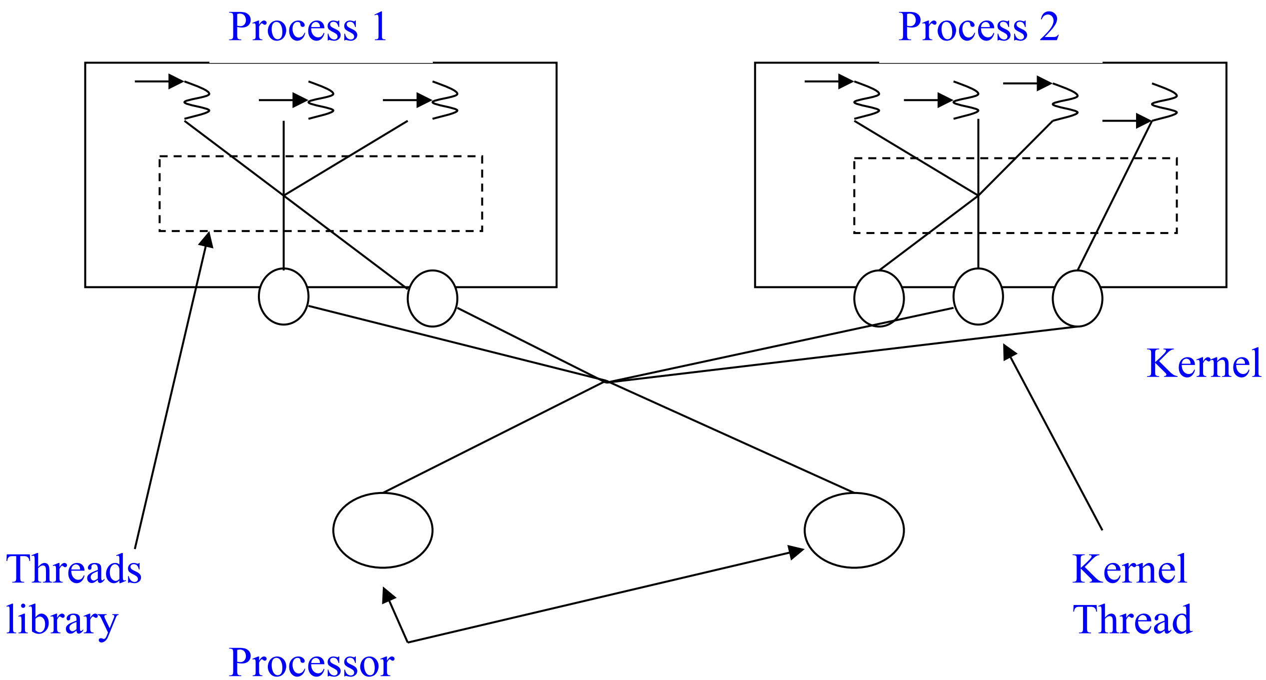
- Each user thread is mapped into a kernel thread
  - allows more concurrency; allowing another thread to run when a thread is blocked.
  - Allows multiple threads running in parallel on multiprocessors.
- Examples: Windows OS, Linux



Mapping (cont.)

Many-to-many Model:

- Allows *M* user-level threads to be mapped to *N* kernel threads.
- Allows the OS to create a sufficient number of kernel threads.
- Many user threads can be created as necessary, and their corresponding kernel threads can run in parallel on multiprocessors.
- Solaris 2, IRIX, HP-UX, Tru64 UNIX, Windows NT/2000 with the *ThreadFiber* package



Thread Libraries

- Programmers use API of a thread library to create and manage threads
- Thread library can be implemented as
  – User level threads
  – Kernel level threads
- Three main thread libraries:
  – POSIX Pthreads → may be user or kernel threads
  – Win32 → kernel threads.
  – Java → implemented using the thread library of the host system
    - In Windows: it uses Win32
    - In Linux: use Pthreads
- Read Textbook for example thread programs

Thread Libraries (cont.)

Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- Primary goal: To allow multithreaded programs to be portable across multiple OS platforms.
- API specifies behavior of the thread library, implementation is up to development of the library
  - can be kernel or user-level threads.
  - Common in UNIX operating systems.
- POSIX: each thread maintains processor registers, stack, and signal mask
  - other resources must be globally accessible to all threads in the process.

Threading Issues

Semantics of fork() and exec() system calls:

- If one thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process, including all threads and LWPs.
- If one thread in a program calls fork(), does the new process duplicate all threads or is the new process single-threaded (only the thread that invoked the fork)?
  - Duplicate only the thread that invoked the fork system call
    - Appropriate if exec is called immediately.
  - Duplicate all threads
    - if the new process does not call exec.

Threading Issues (cont.)

Thread cancellation:

- A thread (target thread) is terminated before it has completed.
- Two different scenarios:
  - Asynchronous cancellation – One thread immediately terminates the target thread.
  - Deferred cancellation – the target thread periodically checks if it should terminate.
- Problems with cancellation:
  - If resources have been allocated to a cancelled thread → OS reclaims system resources, (usually) not all resources;
    - asynchronous cancellation may not reclaim all resources.
  - A thread was cancelled while in the middle of updating data shared with other threads.

Threading Issues (cont.)

Signal handling:

- A signal is used in Unix to notify a process that a particular event has occurred.
- Synchronous signals: delivered to the same process that performs the operation causing the signal;
  - Operations can be divide by zero, illegal memory access, etc.
- Asynchronous signals: generated by an event external to a running process; <control> <C>, timer expired.
- All signals follow the same pattern:
  - Signal generation → occurrence of a particular event.
  - Delivering the signal to a process.
  - Signal handling.
- Two possible signal handlers:
  - By a default signal handler → every signal has a default handler which is run by the kernel.
  - By a user-defined signal handler → override the default.

Threading Issues (cont.)

Signal handling (cont.):

- When a process has more than one threads, where should the signal be delivered? Options:
  - Deliver the signal to the thread to which the signal applies → for synchronous signals.
  - Deliver the signal to every thread in the process → for some asynchronous signals such as <control><C>.
  - Deliver the signal to certain threads in the process → options in some versions of UNIX.
  - Assign a specific thread to receive all signals for the process → implemented in Solaris 2.
  - POSIX: signals are sent to process, and each thread has a signal mask to allow the thread to disable signals of a particular type
    - A signal will be delivered to all threads in the process that are not masking signals of that type.

Threading Issues (cont.)

Thread pools

- In a multithreaded web server, when the server receives a request, a separate thread is created; Disadvantages:
  - Time consuming for creating and destroying a thread for each request.
  - No limit (bound) on the number of threads created
    - May exhaust system resources.
  - One solution: use thread pools
    - Create a number of threads at process startup, and place them into a pool waiting for work.
- The benefits of thread pools are:
  - Faster to service a request.
  - Limits the number of threads that exist at any one point.
- The number of threads in the pool can be set:
  - Heuristically
    - Based on system resources, such as CPU, memory or based on number of requests.
  - Dynamically
    - Adjusted based on usage patterns.

Threading Issues (cont.)

Thread specific data:

- By default, threads within a process share data of the process.
- For some applications, each thread might need its own copy of certain data
  - They need thread-specific data.
- Win32, Pthreads, Java support thread-specific data.

OS Examples

Windows Threads

- Implement Windows API
- Use one-to-one mapping
- A process contains one or more threads
- Each thread contains
  - a thread id
  - register set
  - separate user and kernel stacks
  - private data storage area
- Context of a thread: register set, stacks, and private storage area

OS Examples

Linux Threads

- Linux does not differentiate processes and threads
  - It refers to them as *tasks* rather than *threads* or *processes*
- Linux provides both fork () and clone () system calls
  - Fork () is used to create a *traditional* process
  - Clone () is used to create a thread
    - Can set flags to determine how much resource sharing between the parent and child tasks
    - Clone() allows a child task to share the address space of the parent task (process).
- Kernel version 2.6: one-to-one thread mapping.