

# CS 471 - Lecture 2

## Processes and Threads

George Mason University

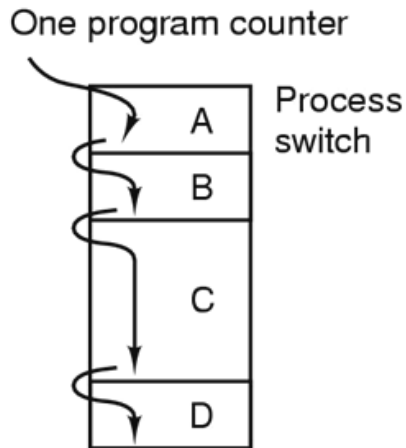
Fall 2012

# Processes

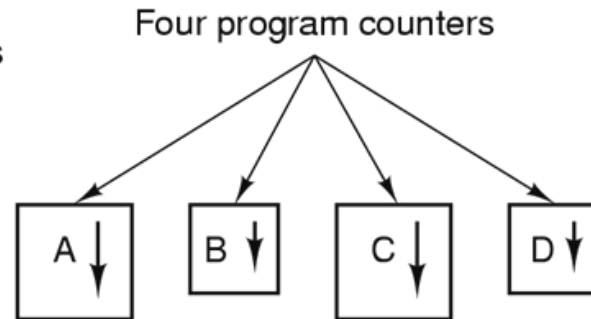
- Process Concept
- Process States
- Process Creation and Termination
- Process Scheduling
- Process Communication

# Process Concept

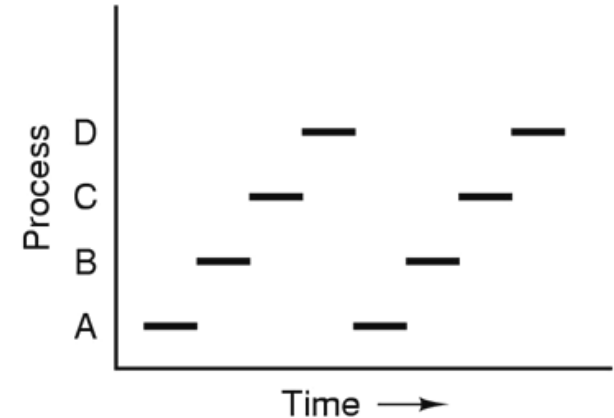
- Process: *a program in execution*
  - process execution must progress in sequential fashion.
- A *program* is a passive entity, whereas a process is an *active entity* with a program counter and a set of associated resources.



(a)



(b)



(c)

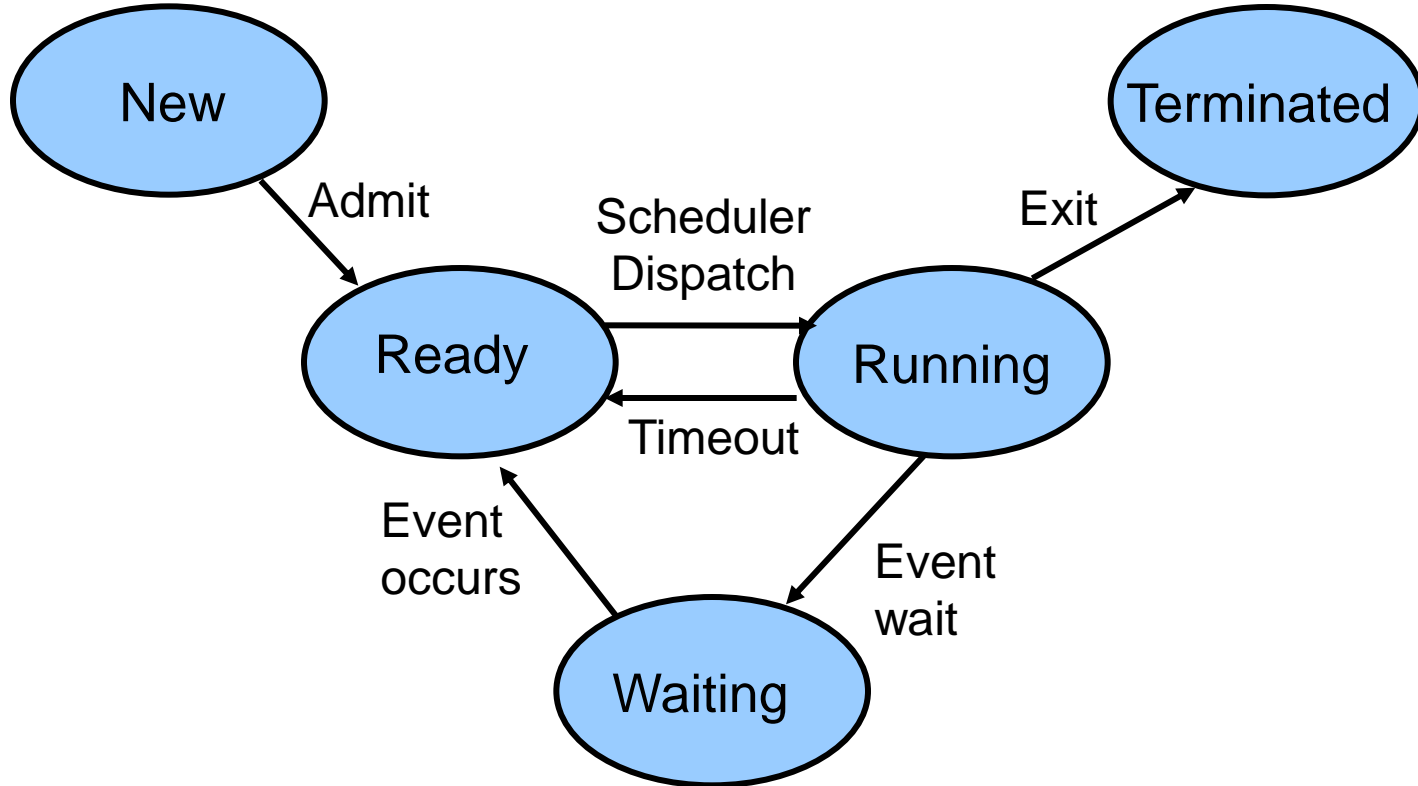
# The Process (Cont.)

- Each process has its own *address space*:
  - Text section (text segment) contains the executable code
  - Data section (data segment) contains the global variables
  - Stack contains temporary data (local variables, return addresses..)
  
- A process may contain a heap, which contains memory that is dynamically allocated at run-time.
  
- ✓ The program counter and CPU registers are part of the *process context*.

# Process States

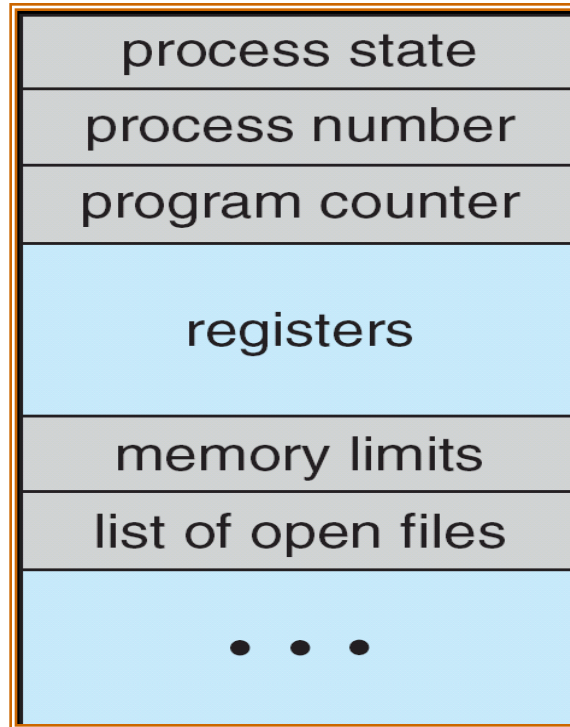
- As a process executes, it changes *state*
  - new: The process is being created.
  - running: Instructions are being executed.
  - waiting (blocked): The process is waiting for some event to occur (such as I/O completion or receipt of a signal).
  - ready: The process is waiting to be assigned to the CPU.
  - terminated: The process has finished execution.

# Simple State Transition Model



# Process Control Block (PCB)

- To implement the process model, the operating system maintains the *process table*, with one *process control block* per process.



# Process Control Block in Unix

## ■ Process Management

- Registers
- Program Counter
- Program Status Word
- Stack Pointer
- Process State
- Priority
- Scheduling Parameters
- Process ID
- Parent process
- Process group
- Time when process started
- CPU time used

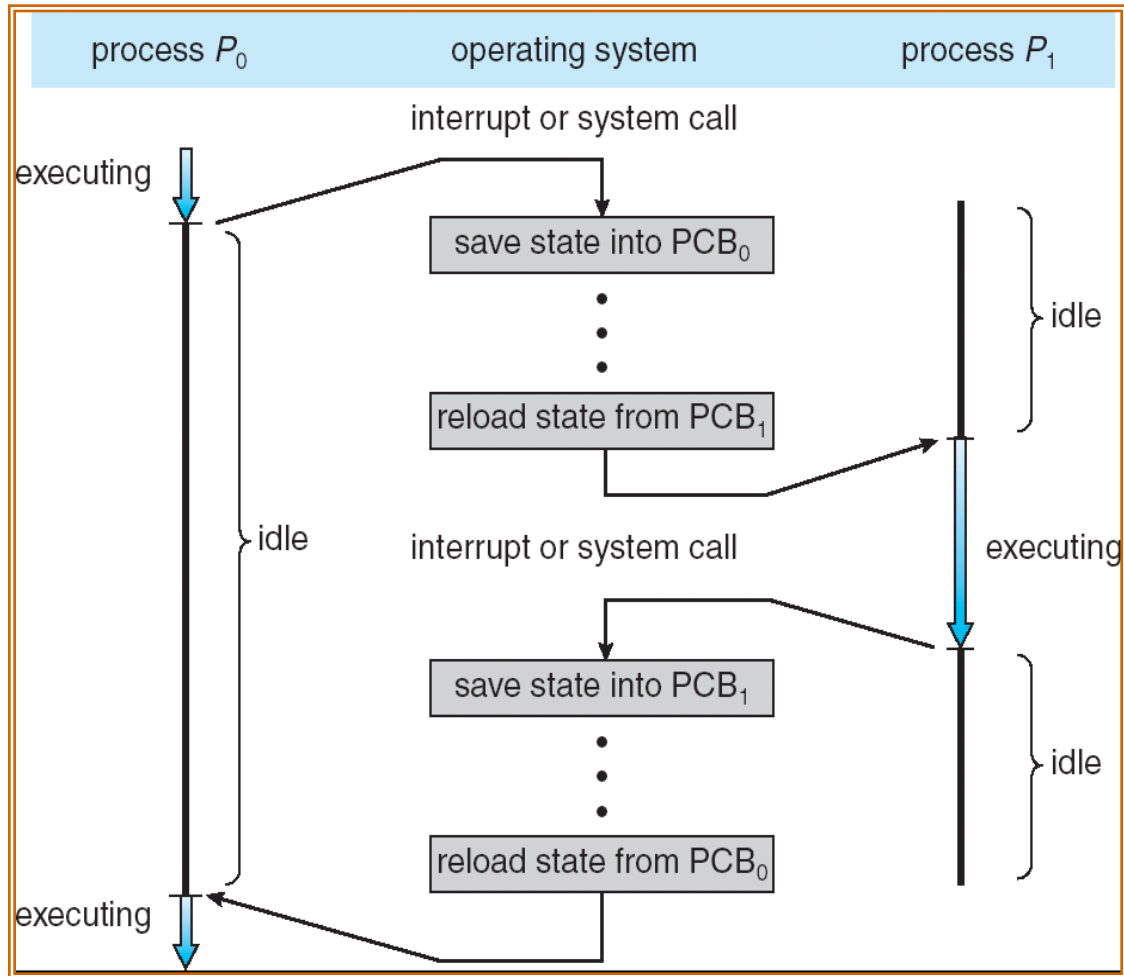
## ■ Memory Management

- Pointer to text (code) segment
- Pointer to data segment
- Pointer to stack segment

## ■ File Management

- Root directory
- Working directory
- User Id
- Group Id

# CPU Switch From Process to Process



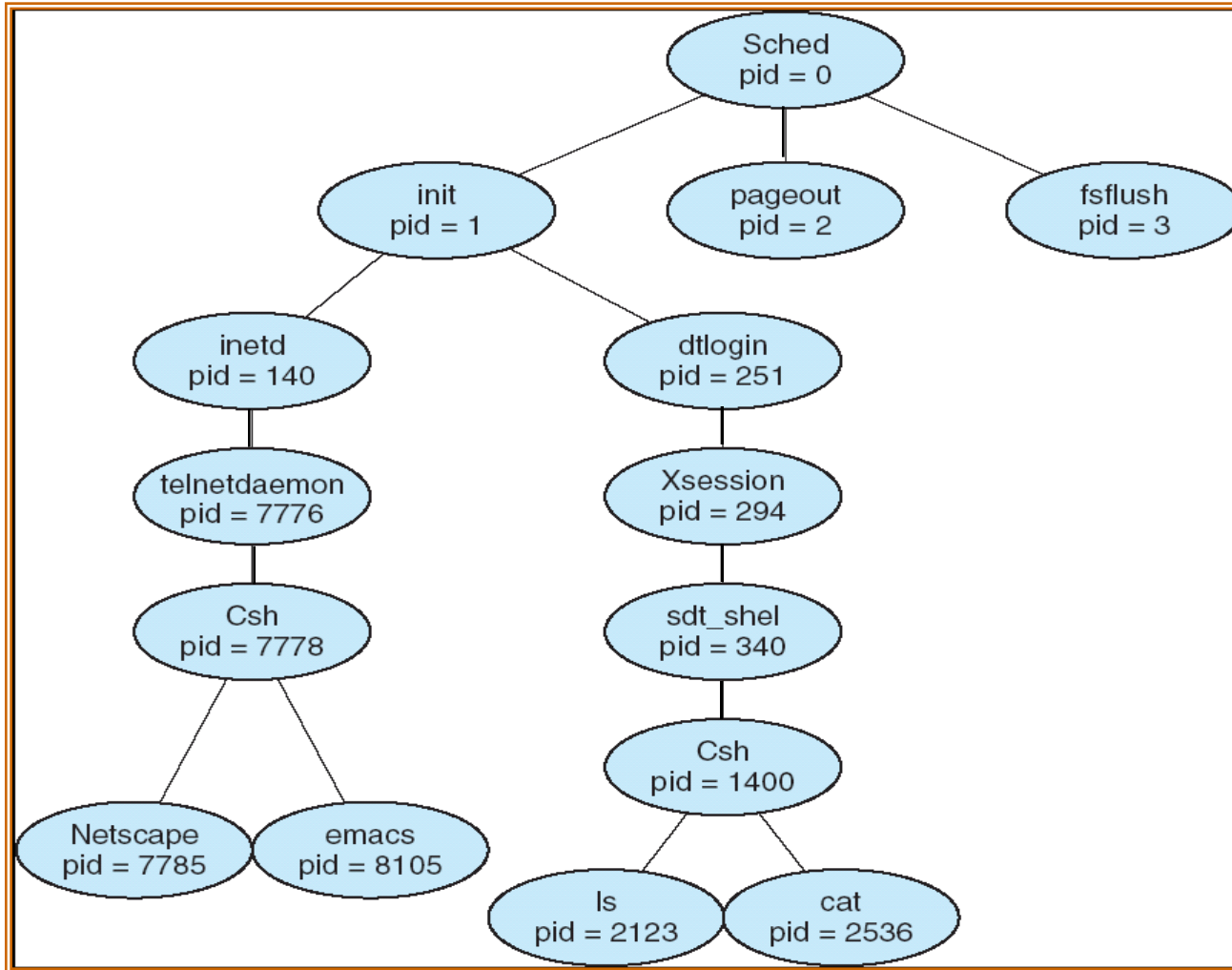
# Context Switch

- 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.
- Context-switch time is pure overhead; the system does no useful work while switching.
- Overhead dependent on hardware support (typically 1-1000 microseconds).

# Process Creation

- Principal events that cause process creation
  - System initialization
  - Execution of a process creation system call by a running process
  - User request to create a new process
  
- Parent process creates child processes, which, in turn create other processes, forming a tree (hierarchy) of processes.

# An example process tree in Solaris



# Process Creation in Unix

- Each process has a *process identifier (pid)*
- The parent executes *fork* system call to spawn a child.
- The child process has a *separate copy* of the parent's address space.
- Both the parent and the child continue execution at the instruction following the *fork* system call.
  
- The return code for the *fork* system call is
  - zero for the new (child) process
  - the (nonzero) pid of the child for the parent.
- Typically, the child executes a system call like *exec/p* to load a binary file into memory.

## Example program with “fork”

```
void main ()
{
int p;

p = fork();
if (p < 0) {error_msg}

else if (p == 0) {/* child process */
                execlp("/bin/lS", "lS", NULL);
                }
else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    exit(0);
}
}
```

# A very simple shell

```
while (1)
{
    type_prompt();
    read_command(com);
    p = fork();
    if (p < 0) {error_msg}

    else if (p == 0) { /* child process */
        execute_command(com);
    }
    else { /* parent process */
        wait(NULL);
    }
}
```

# Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait or waitpid).
  - Process' resources (virtual memory resources, locks, etc. ) are deallocated by operating system.
  - The OS records the process status and resource usage, notifying the parent in response to a wait call.
- Parent may terminate the execution of child processes (e.g. TerminateProcess() in Win32)
- Process may also terminate due to errors

# Process Termination in Unix

- In Unix, a process willing to terminate invokes the exit system call, passing an argument that is its *return code*.
- The return code is passed to the parent that has issued the *wait* system call.
- If a parent is not waiting (but keeps running) when the child process terminates, the child process becomes a *zombie*
  - Its process ID and return code are preserved by the kernel.
- If a parent terminates without waiting for children, the child processes become *orphans*.
- When a process terminates, its orphaned children and zombies are adopted by a special system process *init* (with *pid = 1*), that periodically waits for children.

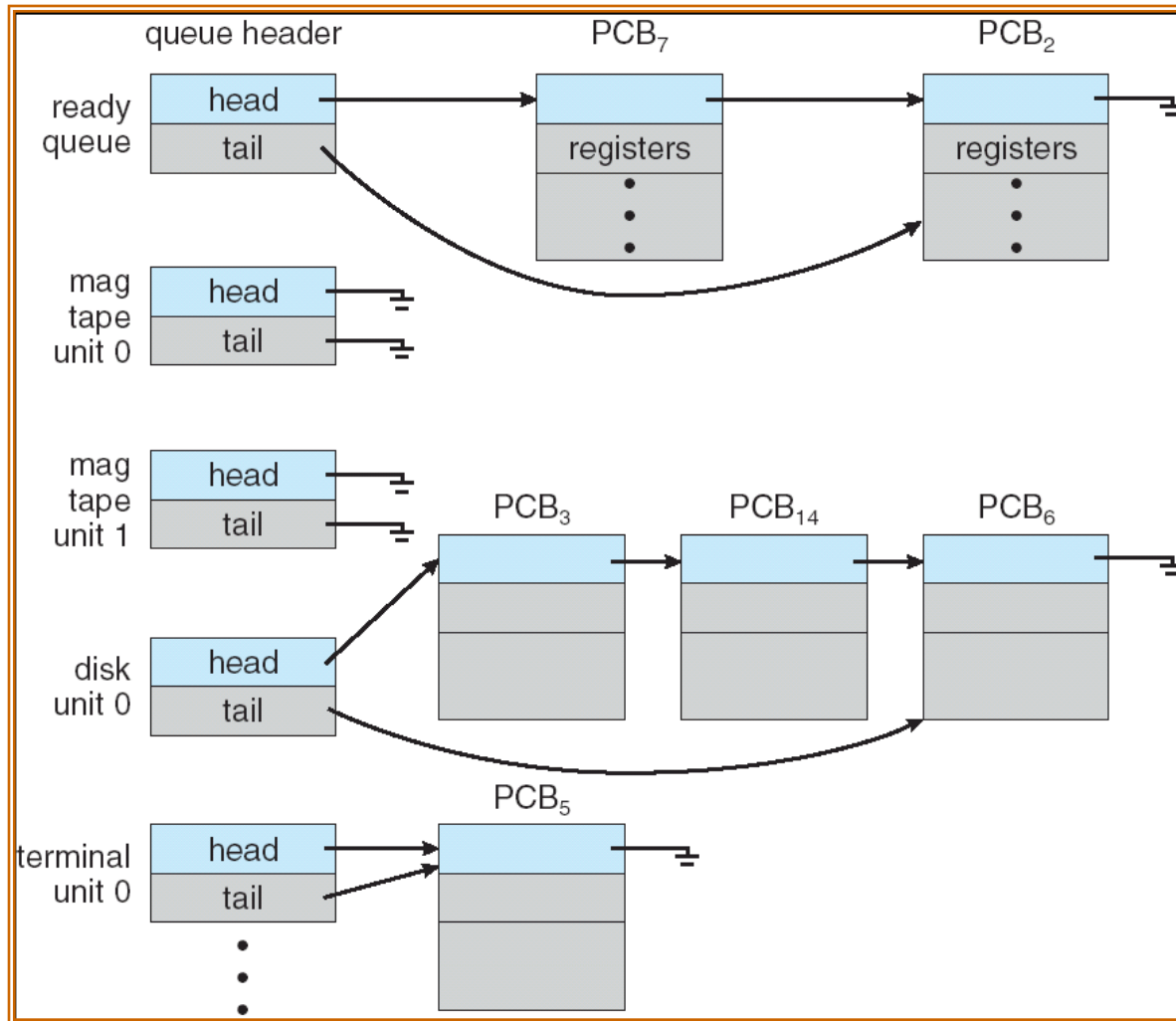
# Process Scheduling

- The operating system is responsible for managing the *scheduling* activities.
  - A uniprocessor system can have only one running process at a time
  - The main memory cannot always accommodate all processes at run-time
  - The operating system will need to decide on which process to execute next (CPU scheduling), and which processes will be brought to the main memory (job scheduling)

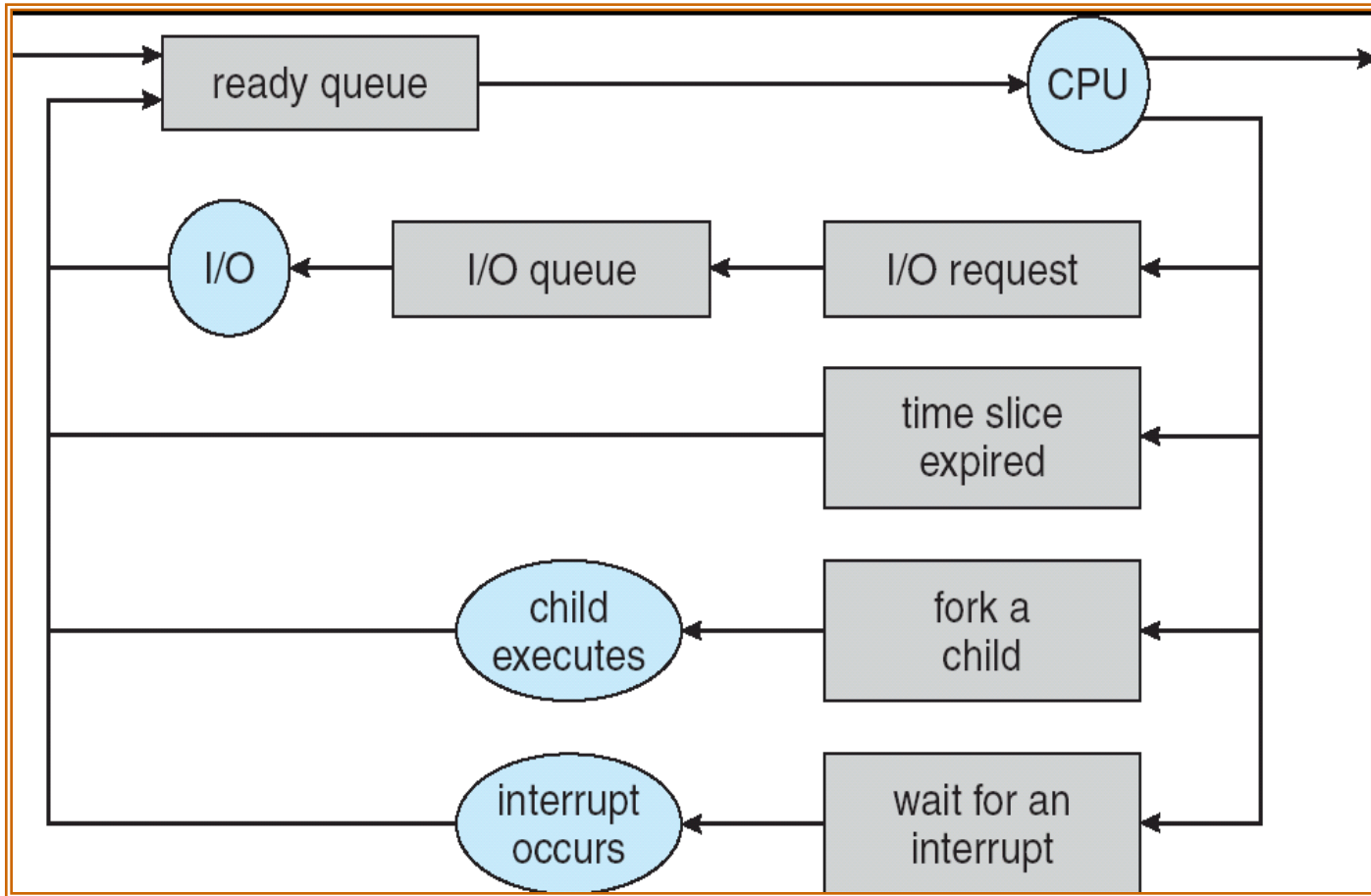
# Process Scheduling Queues

- Job queue – set of all processes in the system.
- Ready queue – set of all processes residing in main memory, ready and waiting for CPU.
- Device queues – set of processes waiting for an I/O device.
- Process migration is possible among these queues.

# Ready Queue and I/O Device Queues



# Process Lifecycle

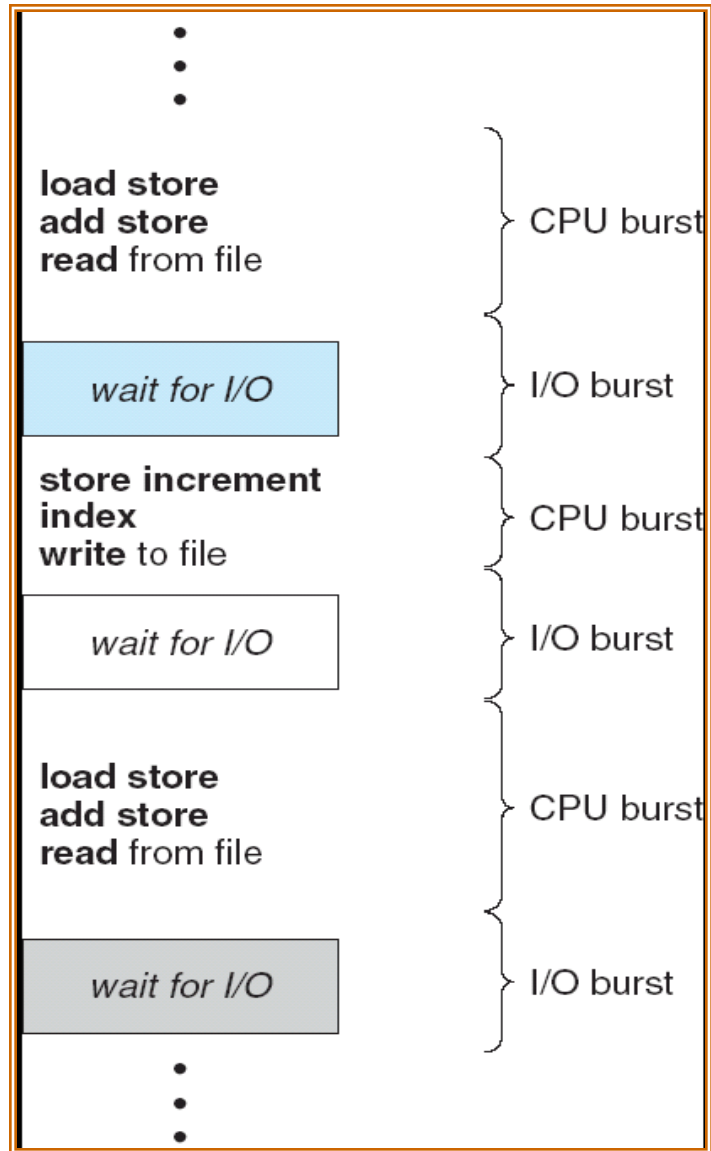


# Schedulers

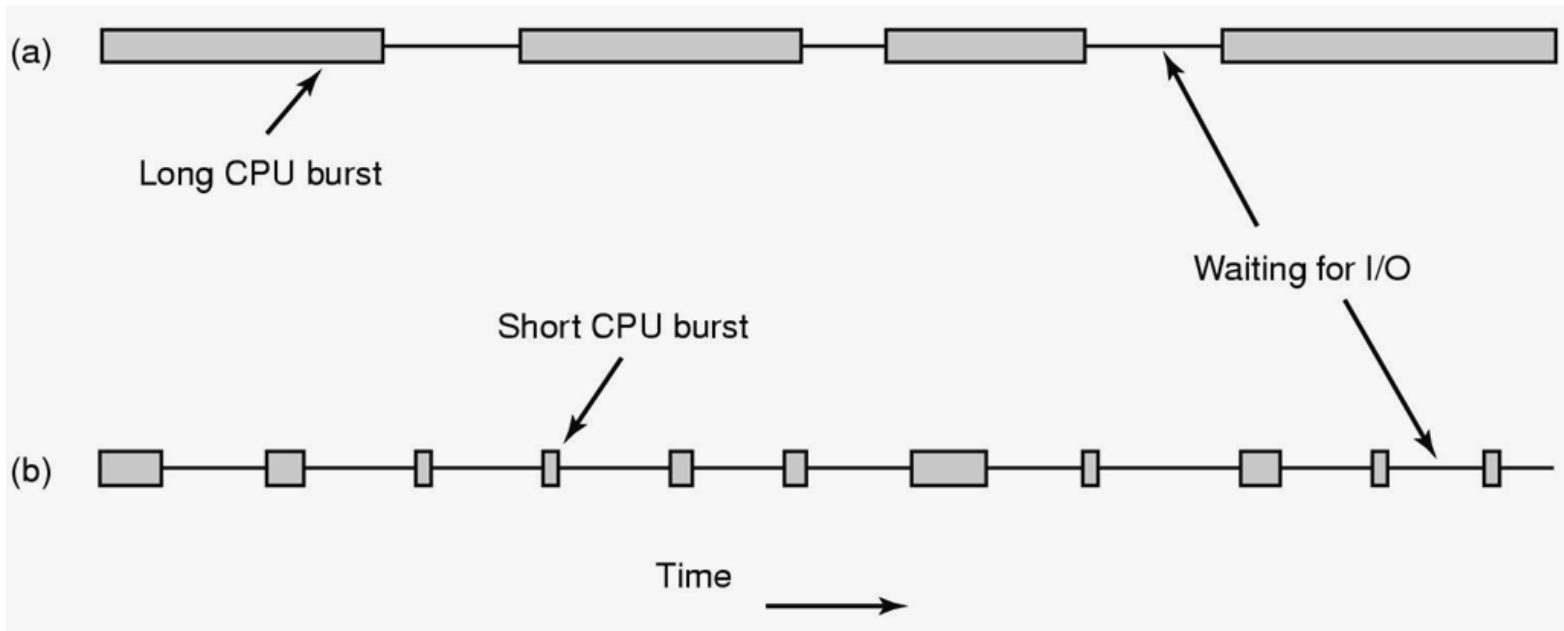
- The processes may be first spooled to a mass-storage system, where they are kept for later execution.
  
- The *long-term scheduler (or job scheduler)* – selects processes from this pool and loads them into memory for execution.
  - The long term scheduler, if it exists, will control the *degree of multiprogramming*
  
- The *short-term scheduler (or CPU scheduler)* – selects from among the *ready* processes, and allocates the CPU to one of them.
  - Unlike the long-term scheduler, the short-term scheduler is invoked very frequently.

# CPU and I/O Bursts

- CPU–I/O Burst Cycle –
  - Process execution consists of a *cycle* of CPU execution and I/O wait.
- *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.



# CPU-bound and I/O-bound Processes



(a) A CPU-bound process

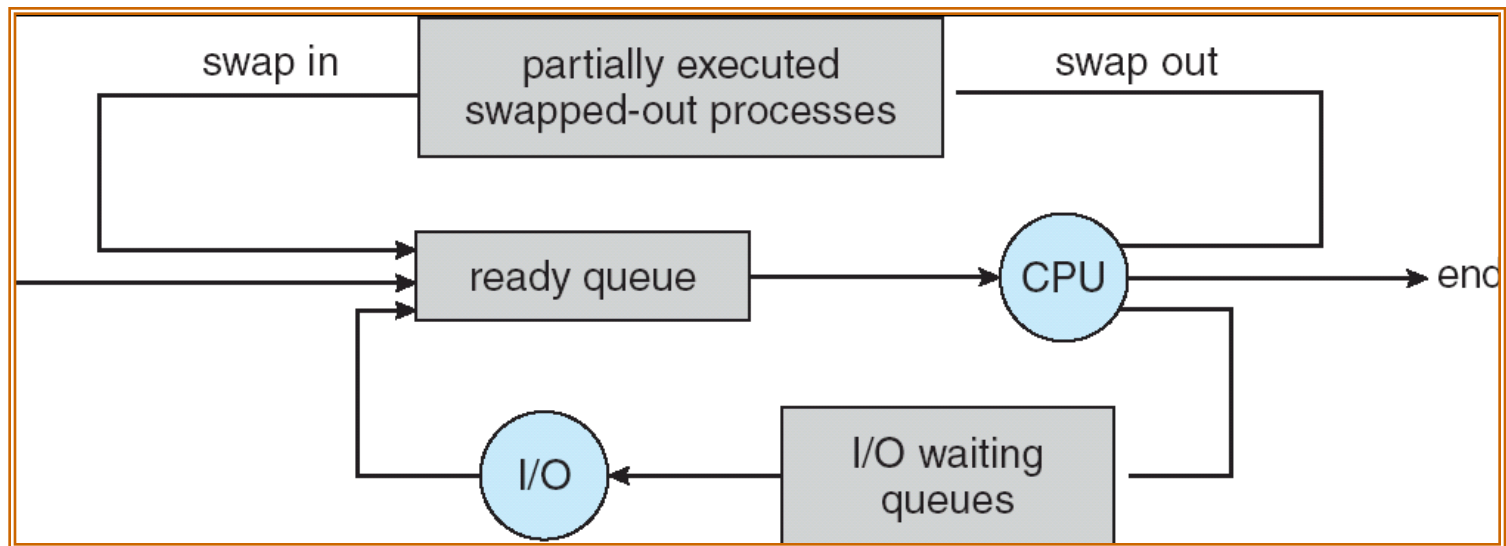
(b) An I/O-bound process

# Scheduler Impact

- Consequences of using I/O-bound and CPU-bound process information
  - Long-term (job) scheduler decisions
  - Short-term (CPU) scheduler decisions

# Addition of Medium-Term Scheduler

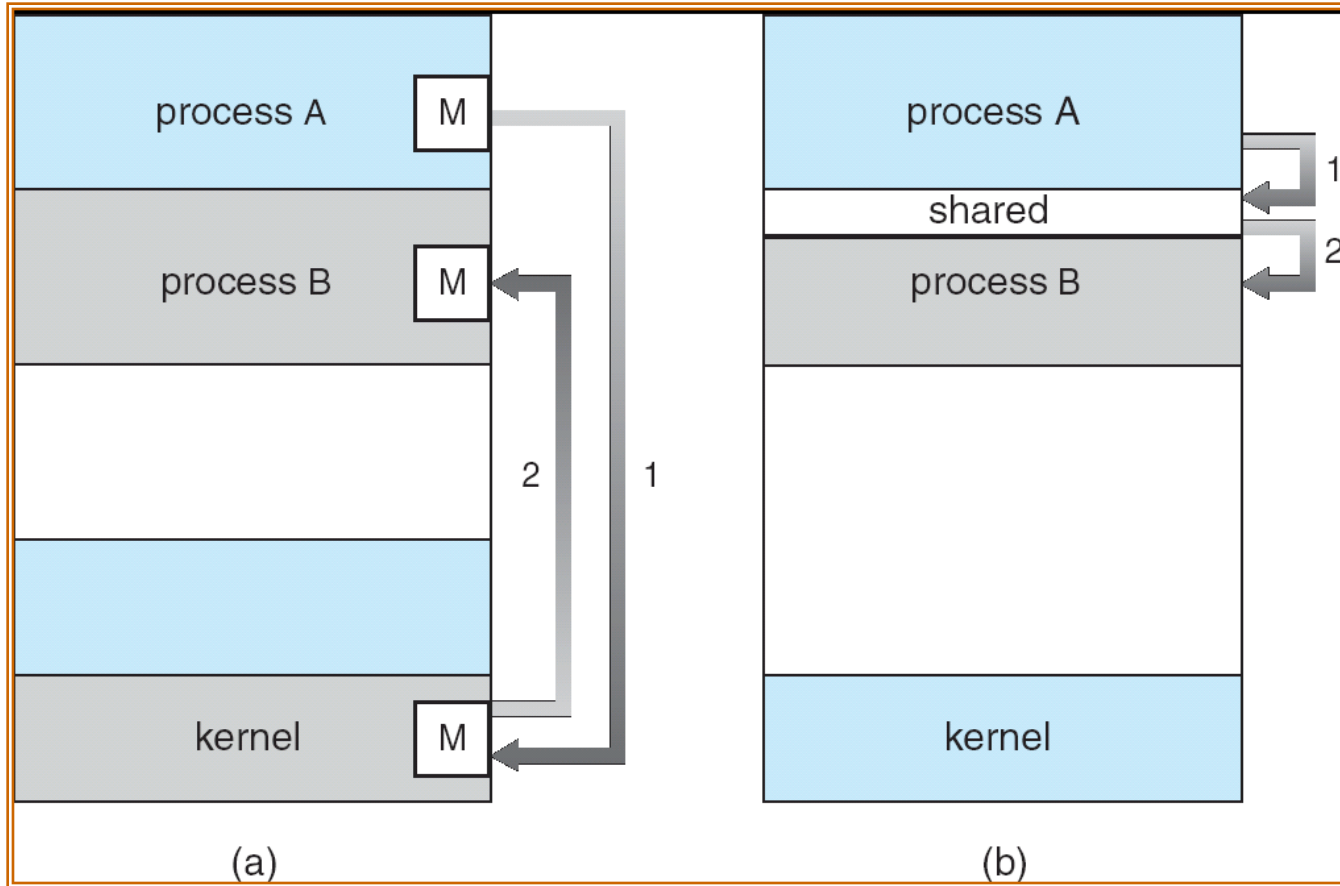
- The medium-term scheduler can reduce the degree of multiprogramming by removing processes from memory.
- At some later time, the process can be re-introduced into memory (*swapping*).



# Process Communication

- Mechanism for processes to communicate and to synchronize their actions.
- Two models
  - Communication through a shared memory region
  - Communication through message passing

# Communication Models



Message Passing

Shared Memory

- ✓ Observe: in a distributed system, message-passing is the only possible communication model.

# Communication through message passing

- Message system – processes communicate with each other *without resorting to shared variables*.
  
- A message-passing facility must provide at least two operations:
  - *send(message, recipient)*
  - *receive(message, recipient)*
  
- With *indirect communication*, the messages are sent to and received from **mailboxes** (or, **ports**).
  - **send (A, message)** /\* A is a mailbox \*/
  - **receive (A, message)**

# Communication through Message Passing

- Message passing can be either *blocking (synchronous)* or *non-blocking (asynchronous)*
  - Blocking Send: The sending process is blocked until the message is received by the receiving process or by the mailbox
  - Non-blocking Send: The sending process resumes the operation as soon as the message is received by the kernel
  - Blocking Receive: The receiver blocks until the message is available
  - Non-blocking Receive: “Receive” operation does not block; it either returns a valid message or a default value (null) to indicate a non-existing message

# Communication through Shared Memory

- The memory region to be shared must be explicitly defined
- Using system calls – in Unix:
  - *Shmget* creates a shared memory block
  - *Shmat* maps an existing shared memory block into a process's address space
  - *Shmdt* removes (“unmaps”) a shared memory block from the process's address space
  - *Shmctl* is a general-purpose function allowing various operations on the shared block (receive information about the block, set the permissions, lock in memory, ...)
- Problems with simultaneous access to the shared variables
- Compilers for *concurrent programming languages* can provide direct support when declaring variables (e.g. “shared int buffer”)

## An example (From Fig. 3.16 of the textbook)

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the identifier for the shared memory segment */
    int segment_id;
    /* a pointer to the shared memory segment */
    char *shared_memory;
    /* the size (in bytes) of the shared memory segment */
    const int segment_size = 4096;

    /** allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR |
        S_IWUSR);

    /** attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);
    printf("shared memory segment %d attached at address
        %p\n", segment_id, shared_memory);
}
```

## An Example (2)

```
/** write a message to the shared memory segment */
    sprintf(shared_memory, "Hi there!");

/** now print out the string from shared memory */
    printf("**%s*\n", shared_memory);

/** now detach the shared memory segment */
    if ( shmdt(shared_memory) == -1) {
        fprintf(stderr, "Unable to detach\n");
    }

/** now remove the shared memory segment */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

# Threads

- Overview
- Multithreading
- Example Applications
- User-level Threads
- Kernel-level Threads
- Hybrid Implementations

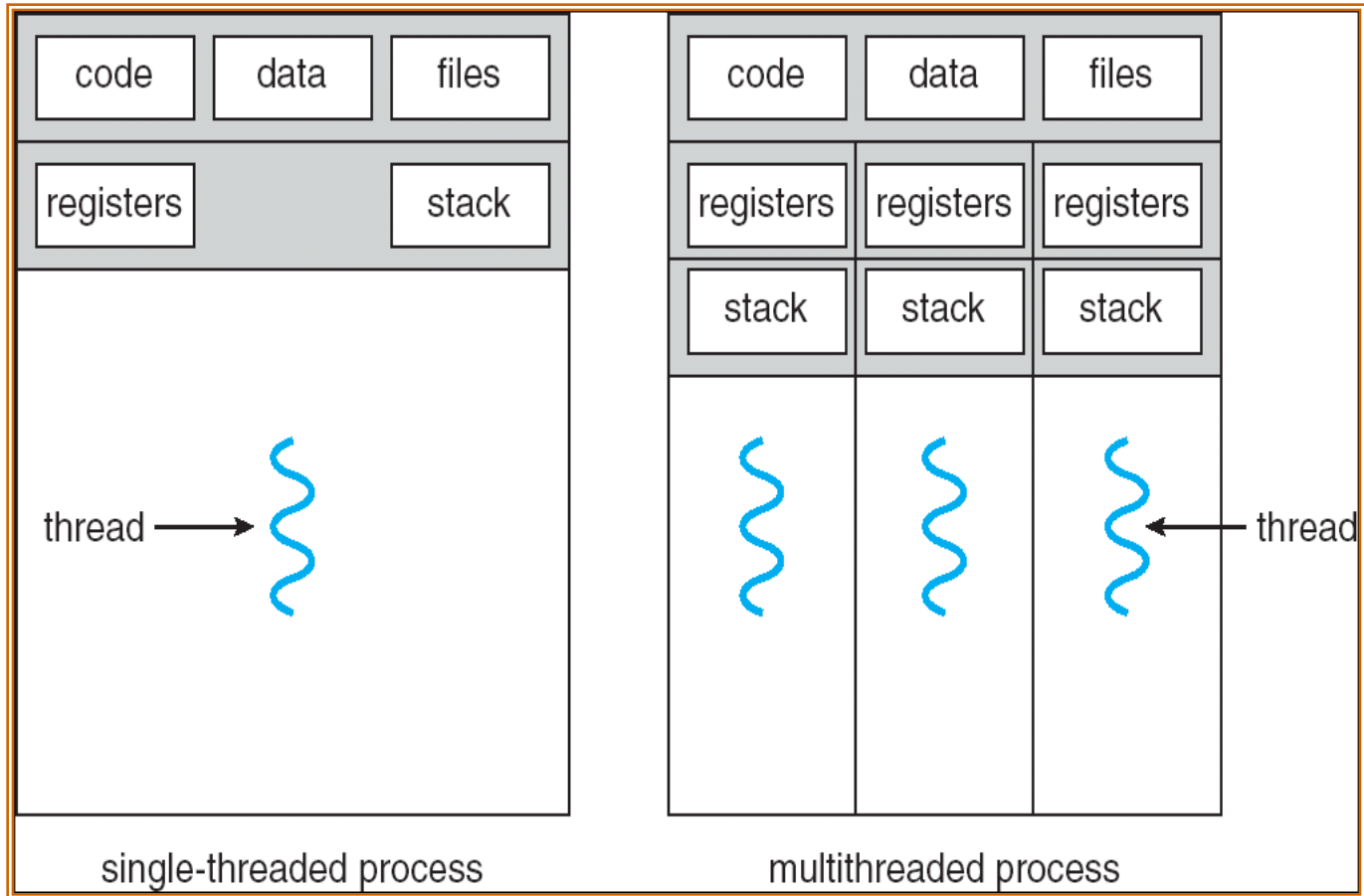
# Threads

- A process, as defined so far, has only one *thread of execution*.
- Idea: Allow multiple threads of execution within the same process environment, to a large degree independent of each other.
- Multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer.

# Threads (Cont.)

- Multiple threads within a process will share
  - The address space
  - Open files
  - Other resources
  
- Potential for efficient and close cooperation

# Single and Multithreaded Processes



# Multithreading

- When a multithreaded process is run on a single CPU system, the threads take turns running.
- All threads in the process have exactly the same address space.

## Per Process Items

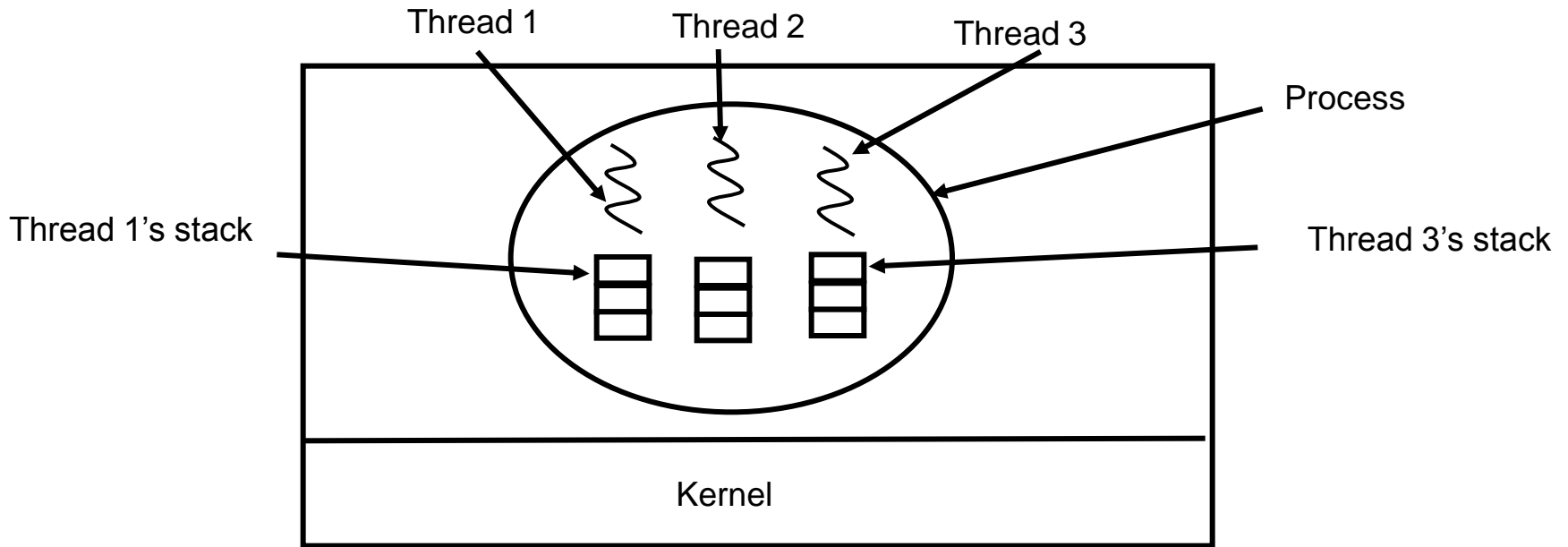
Address Space  
Global Variables  
Open Files  
Accounting Information

## Per Thread Items

Program Counter  
Registers  
Stack  
State

# Multithreading (Cont.)

- Each thread can be in any one of the several states, just like processes.
- Each thread has its own stack.



# Benefits

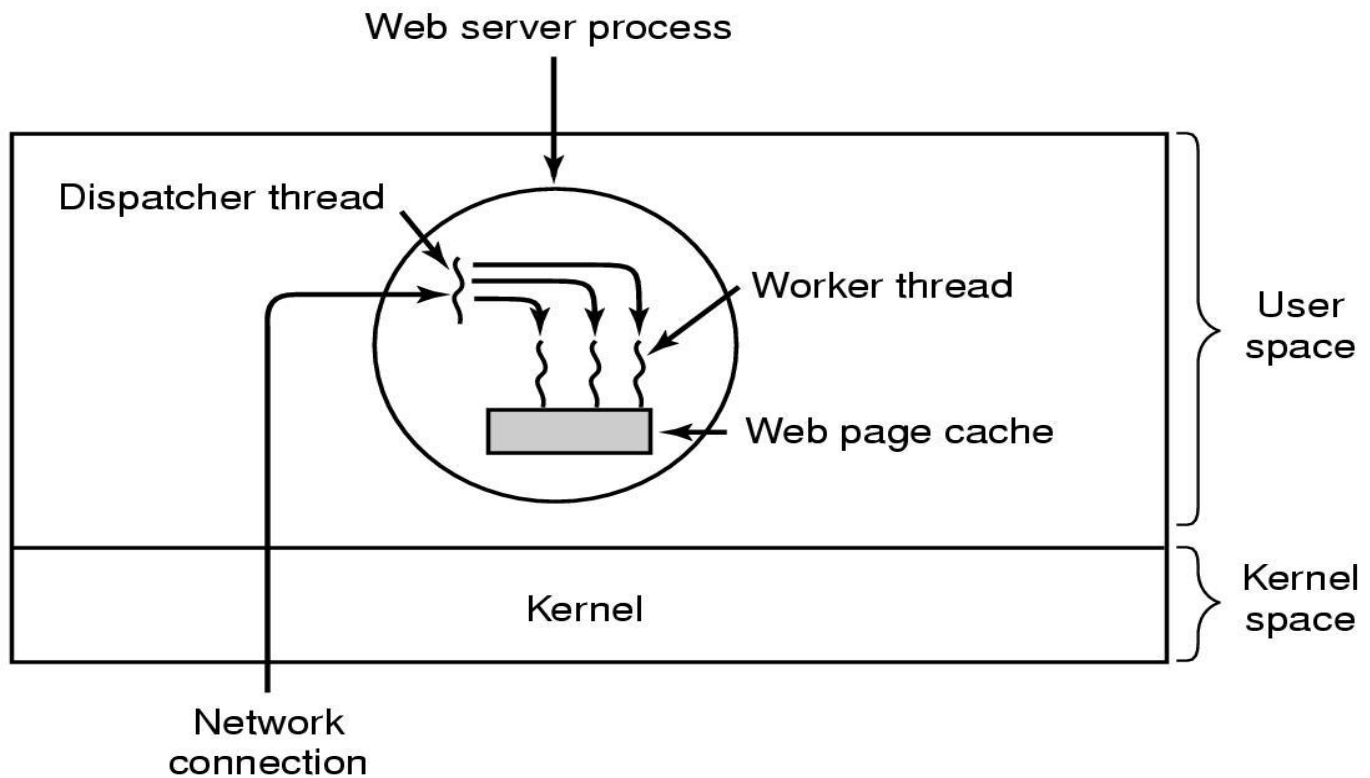
- Responsiveness
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or performing a lengthy operation.
- Resource Sharing
  - Sharing the address space and other resources may result in high degree of cooperation
- Economy
  - Creating / managing processes is much more time consuming than managing threads.
- Better Utilization of Multiprocessor Architectures

# Example Multithreaded Applications

- A word-processor with three threads
  - Re-formatting
  - Interacting with user
  - Disk back-up
  
- What would happen with a single-threaded program?

# Example Multithreaded Applications

- A multithreaded web server



# Example Multithreaded Applications

- The outline of the code for the dispatcher thread (a), and the worker thread (b).

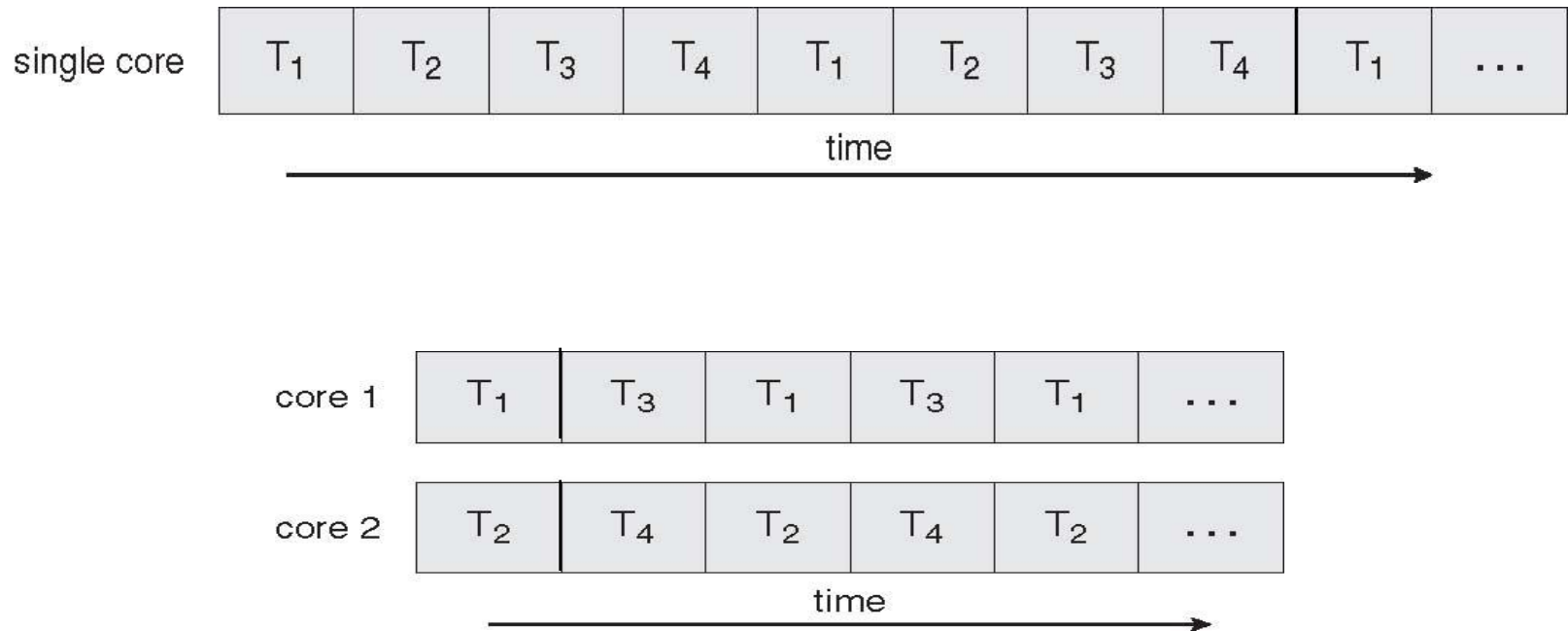
```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a)

```
while(TRUE) {  
    wait_for_work(&buf);  
    check_cache(&buf; &page);  
    if_not_in_cache(&page)  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b)

# Threads in Multicore Platforms



- Concurrent and parallel execution of threads

# Threads in Multicore Platforms (Cont.)

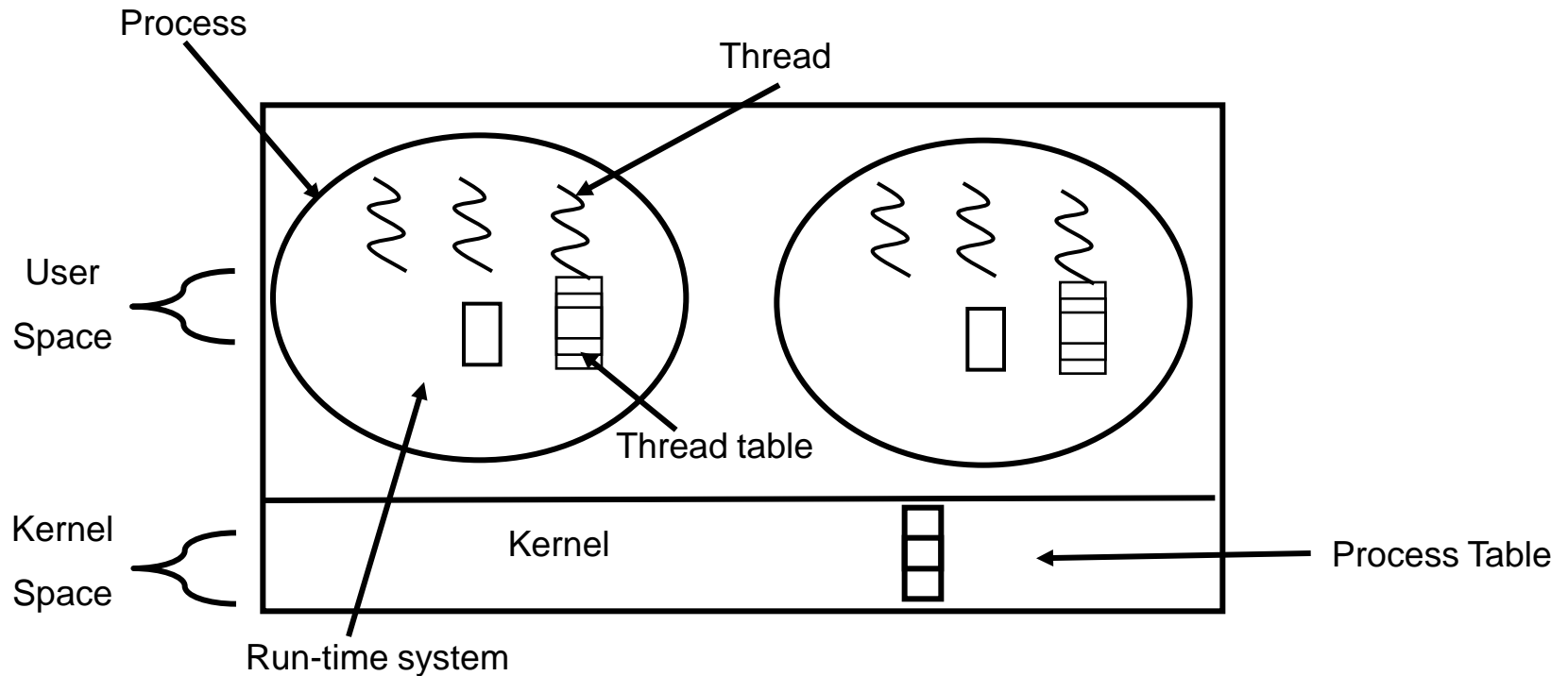
- **Challenge:** modify old programs and design new programs that are multithreaded
  
- **Issues:**
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

# Implementing Threads

- Processes usually start with a single thread
- Usually, library procedures are invoked to manage threads
  - *Thread\_create*: typically specifies the name of the procedure for the new thread to run
  - *Thread\_exit*
  - *Thread\_join*: blocks the calling thread until another (specific) thread has exited
  - *Thread\_yield*: voluntarily gives up the CPU to let another thread run
- Threads may be implemented in the *user space* or in the *kernel space*

# User-level Threads

- User threads are supported above the kernel and are implemented by a thread library at the user level.
- The library (or run-time system) provides support for thread creation, scheduling and management with no support from the kernel.



## User-level Threads (Cont.)

- When threads are managed in user space, each process needs its own private *thread table* to keep track of the threads in that process.
- The thread-table keeps track only of the per-thread items (program counter, stack pointer, register, state..)
- When a thread does something that *may* cause it to become blocked *locally* (e.g. wait for another thread), it calls a run-time system procedure.
- If the thread must be put into blocked state, the procedure performs *thread switching*.

# User-level Threads: Advantages

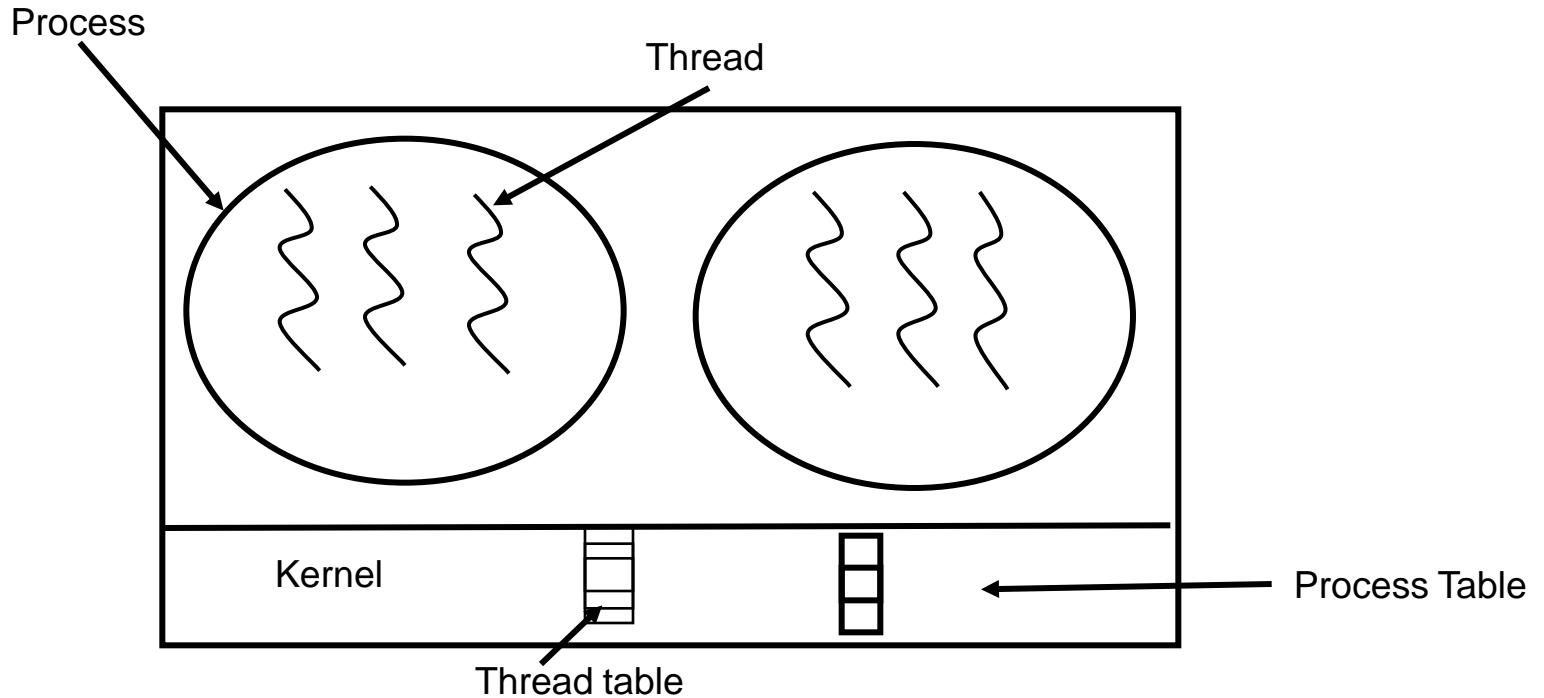
- The operating system does not need to support multi-threading.
- Since the kernel is not involved, thread switching may be very fast.
- Each process may have its own customized thread scheduling algorithm.
- Thread scheduler may be implemented in the user space very efficiently.

# User-level Threads: Problems

- The implementation of *blocking system calls* is highly problematic (e.g. read from the keyboard). *All* the threads in the process risk being blocked!
- Possible Solutions:
  - Change all system calls to non-blocking
  - Sometimes it may be possible to tell in advance if a call will block (e.g. *select* system call in some versions of Unix) → “jacket code” around system calls
- How to deal with page faults?

# Kernel-level threads

- Kernel threads are supported directly by the OS: The kernel performs thread creation, scheduling and management in the kernel space

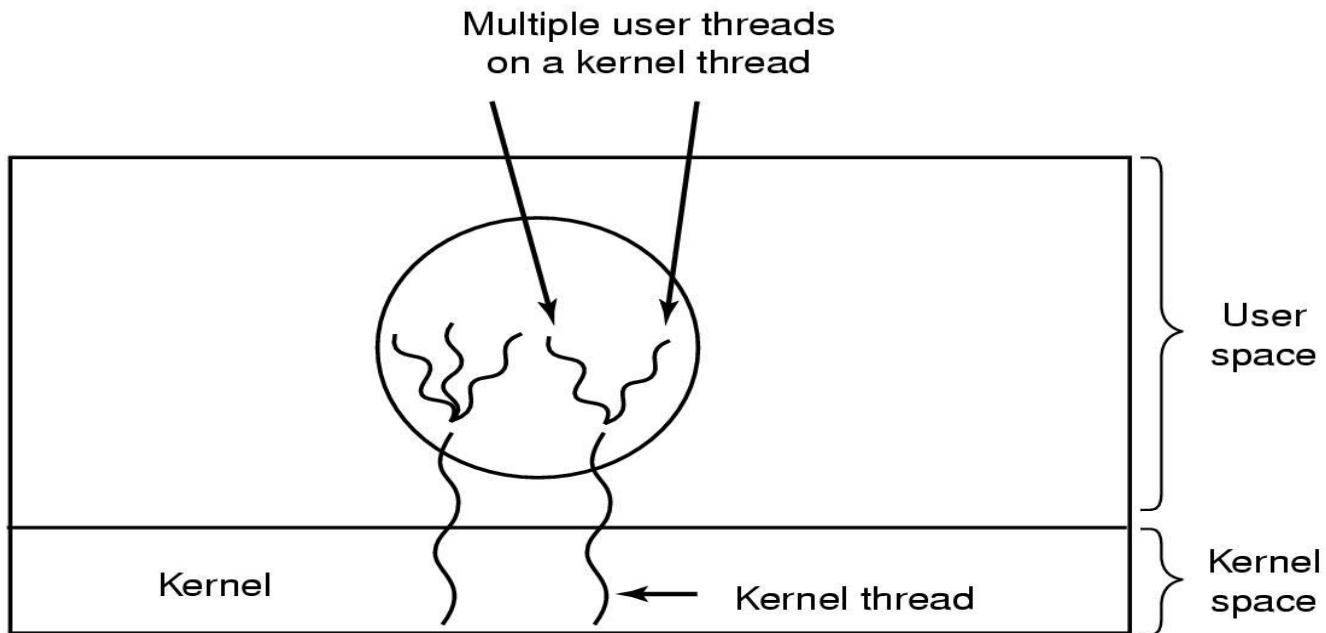


# Kernel-level threads

- The kernel has a thread table that keeps track of all threads in the system.
- All calls that *might* block a thread are implemented as system calls (greater cost).
- When a thread blocks, the kernel may choose another thread from the same process, or a thread from a different process.

# Hybrid Implementations

- An alternative solution is to use kernel-level threads, and then multiplex user-level threads onto some or all of the kernel threads.
- A kernel-level thread has some set of user-level threads that take turns using it.



# Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX operating systems
- Pthread programs use various statements to manage threads: *pthread\_create*, *pthread\_join*, *pthread\_exit*, *pthread\_attr\_init*, ...

# Thread Calls in POSIX

Thread Call	Description
<i>pthread_create</i>	Create a new thread in the caller's address space
<i>pthread_exit</i>	Terminate the calling thread
<i>pthread_join</i>	Wait for a thread to terminate
<i>pthread_mutex_init</i>	Create a new mutex
<i>pthread_mutex_destroy</i>	Destroy a mutex
<i>pthread_mutex_lock</i>	Lock a mutex
<i>pthread_mutex_unlock</i>	Unlock a mutex
<i>pthread_cond_init</i>	Create a condition variable
<i>pthread_cond_destroy</i>	Destroy a condition variable
<i>pthread_cond_wait</i>	Wait on a condition variable
<i>pthread_cond_signal</i>	Release one thread waiting on a condition variable

# Windows XP Threads

- Windows XP supports kernel-level threads
- The primary data structures of a thread are:
  - ETHREAD (executive thread block)
    - Thread start address
    - Pointer to parent process
    - Pointer to the corresponding KTHREAD
  - KTHREAD (kernel thread block)
    - Scheduling and synchronization information
    - Kernel stack (used when the thread is running in kernel mode)
    - Pointer to TEB
  - TEB (thread environment block)
    - Thread identifier
    - User-mode stack
    - Thread-local storage

# Linux Threads

- In addition to *fork()* system call, Linux provides the *clone()* system call, which may be used to create threads
- Linux uses the term *task* (rather than process or thread) when referring to a flow of control
- A set of flags, passed as arguments to the *clone()* system call determine how much sharing is involved (e.g. open files, memory space, etc.)