Chapter 2

Processes and Threads

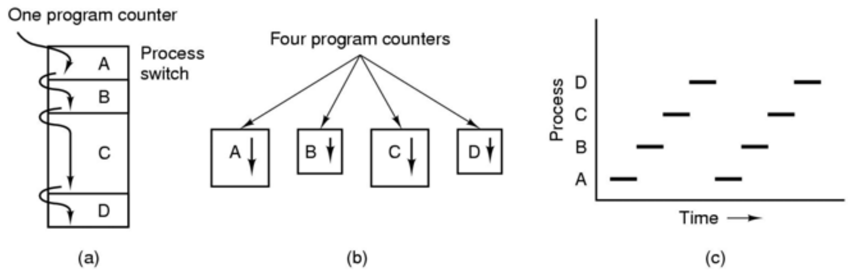
Today
  2.1 Processes
  2.2 Threads
Next week
  2.3 Inter-process communication
  2.4 Classical IPC problems
Week 3
  2.5 Scheduling

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Process – a program in execution

Processes

The Process Model

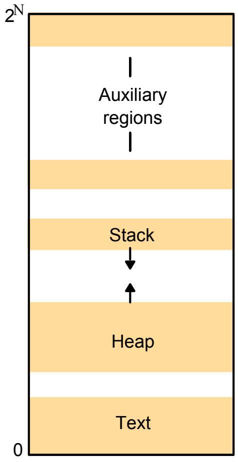


(a) (b) (c)

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

Process Concept

- A process includes:
  - program counter
  - code segment
  - stack segment
  - data segment
- Process = Address Space + One thread of control



Address space

Process Creation

Principal events that cause process creation

1. System initialization
2. Execution of a process creation system call
3. User request to create a new process
4. Initiation of a batch job

Process Termination

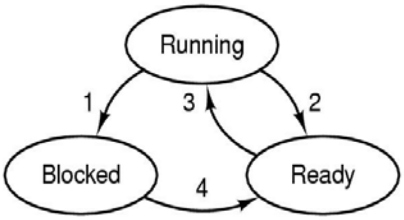
Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal

Process States (1)



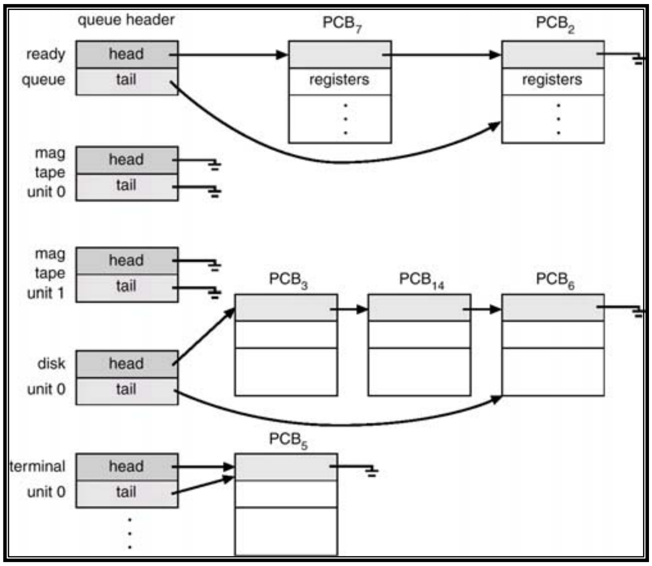
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - running
  - blocked
  - ready
- Transitions between states shown

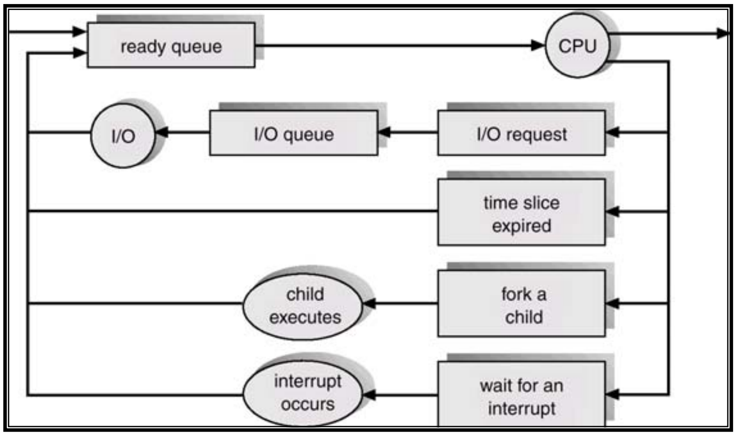
Process Scheduling Queues

- 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 between the various queues during their lifetime.

Ready Queue And Various I/O Device Queues



Processes migrate between queues



Implementation of Processes (1)

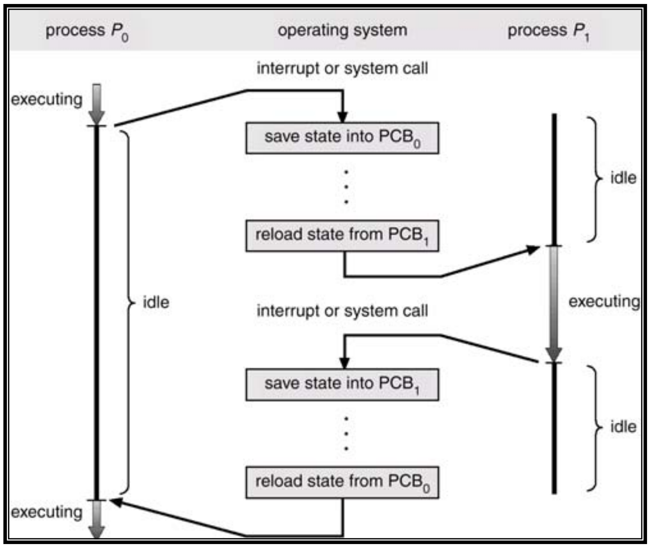
Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Fields of a process table entry (also called PCB – Process Control Block)

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 overhead; the system does no useful work while switching.
- Time dependent on hardware support.

CPU Switch From Process to Process



Cooperating Processes

- Sequential programs consist of a single process
- Concurrent applications consist of multiple cooperating processes that execute concurrently
- Advantages
  - Can exploit multiple CPUs (hardware concurrency) for speeding up application
  - Application can benefit from software concurrency, e.g. web servers, window systems

Cooperating processes cont'd

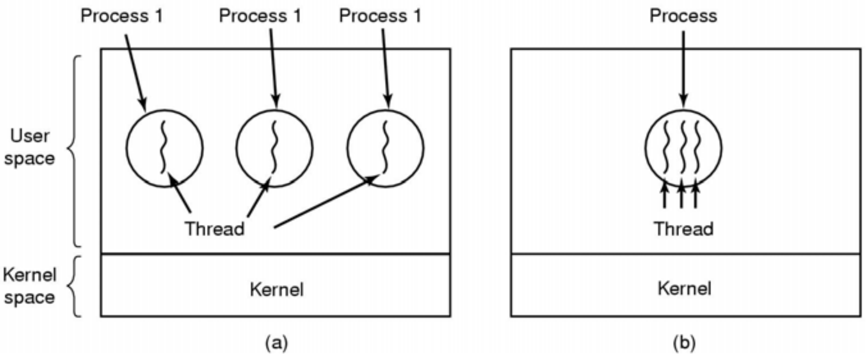
- Cooperating processes need to share information (data)
- Since each process has its own address space, operating system mechanisms are needed to let processes exchange information
- Two paradigms for cooperating processes
  - Shared Memory
    - OS enables two independent processes to have a shared memory segment in their address spaces
  - Message-passing
    - OS provides mechanisms for processes to send and receive messages
- Next class will focus on concurrent programming

Threads: Motivation

- Traditional processes created and managed by the OS kernel
- Process creation expensive – e.g., fork system call
- Context switching expensive
- Cooperating processes - no need for memory protection, i.e., separate address spaces

Threads

The Thread Model (1)



(a) Three processes each with one thread

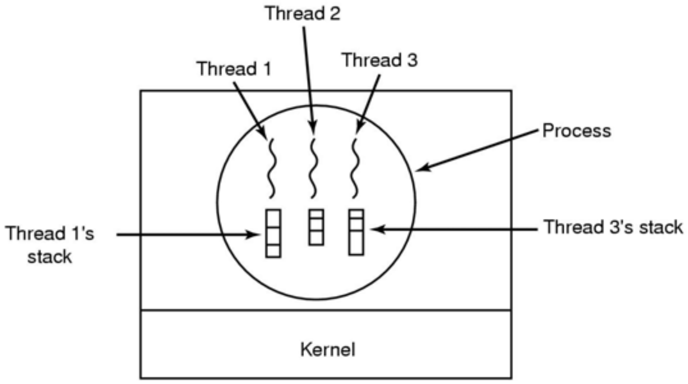
(b) One process with three threads

The Thread Model (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	

- Items shared by all threads in a process
- Items private to each thread

The Thread Model (3)

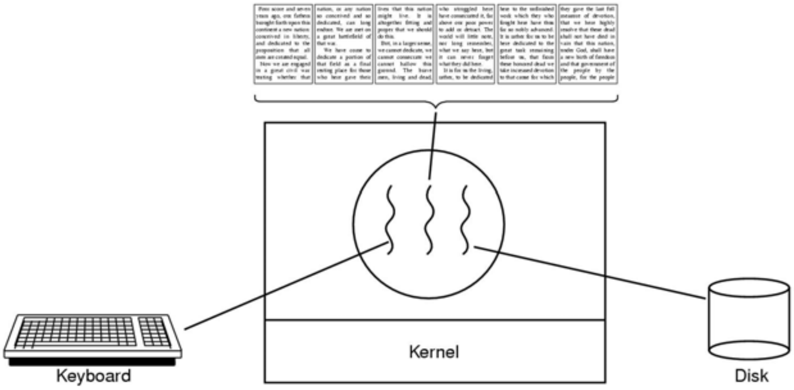


Each thread has its own stack

Threads

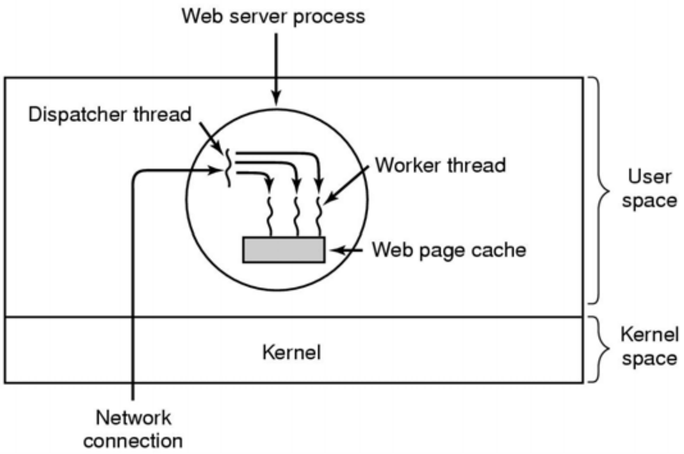
- Execute in same address space
  - separate execution stack, share access to code and (global) data
- Smaller creation and context-switch time
- Can exploit fine-grain concurrency
- Easier to write programs that use asynchronous I/O or communication

Thread Usage (1)



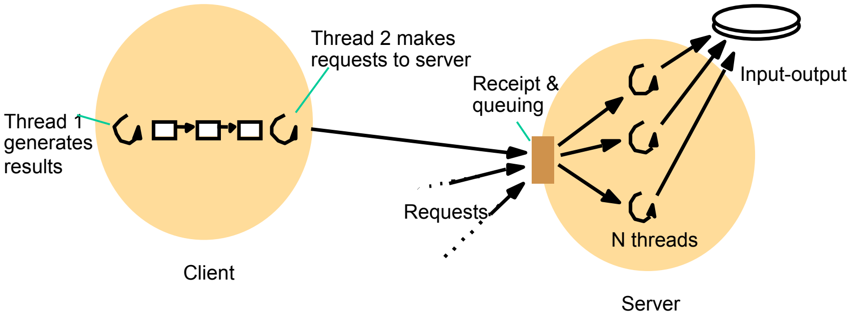
A word processor with three threads

Thread Usage (2)



A multithreaded Web server

Client and server with threads



Threads cont’d

- User-level vs kernel-level threads
  - kernel not aware of threads created by user-level thread package (e.g. Pthreads), language (e.g. Java)
  - user-level threads typically multiplexed on top of kernel level threads in a user-transparent fashion

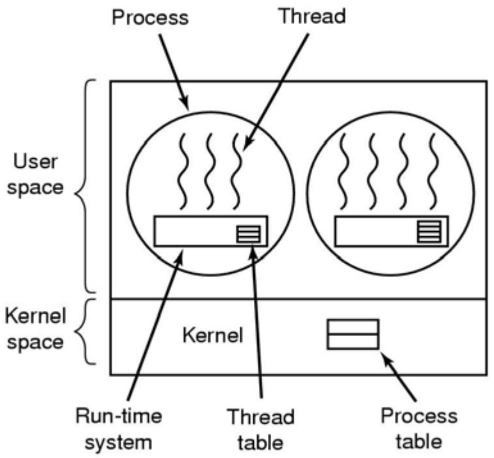
User-Level Threads

- Thread management done by user-level threads library

- Examples

  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*
  - Java threads

Implementing Threads in User Space

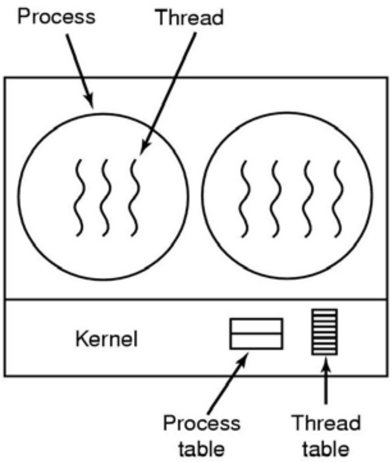


A user-level threads package

Kernel Threads

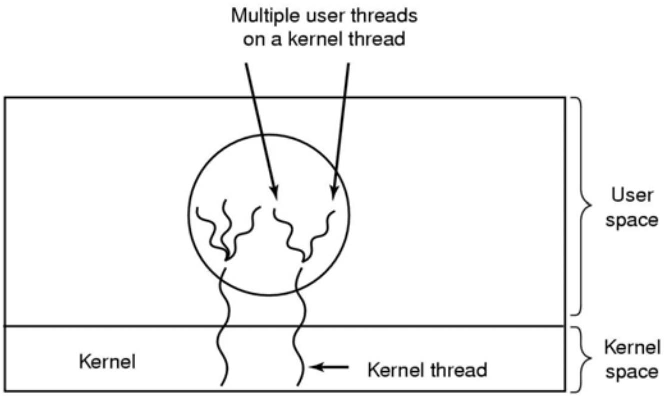
- Supported by the Kernel
- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux

Implementing Threads in the Kernel



A threads package managed by the kernel

Hybrid Implementations



Multiplexing user-level threads onto kernel-level threads

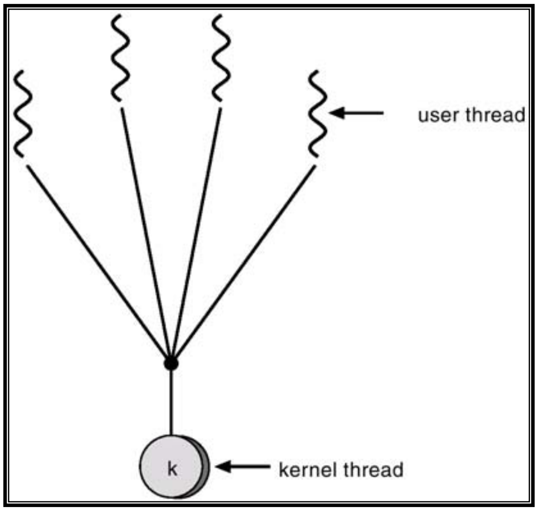
Multithreading Models

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

Many-to-One

- Many user-level threads mapped to single kernel thread.
  - If one user-level thread makes a blocking system call, the entire process is blocked even though other user-level threads may be “ready”
- Used on systems that do not support kernel threads.

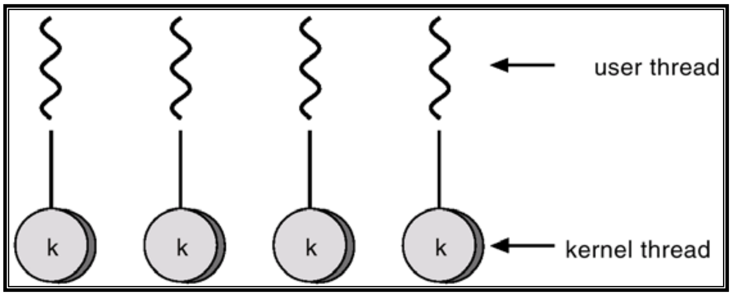
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread.
- Examples
  - Windows 95/98/NT/2000
  - OS/2

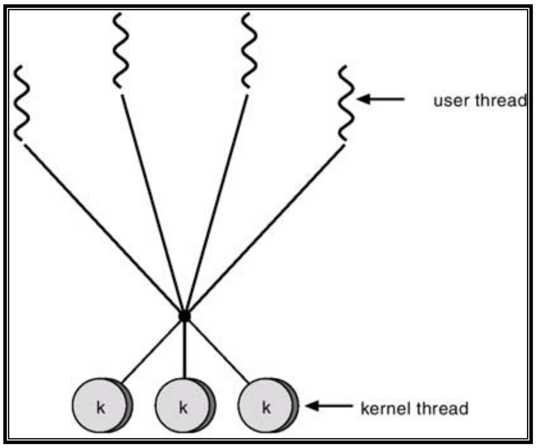
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- Solaris 2
- Windows NT/2000 with the *ThreadFiber* package

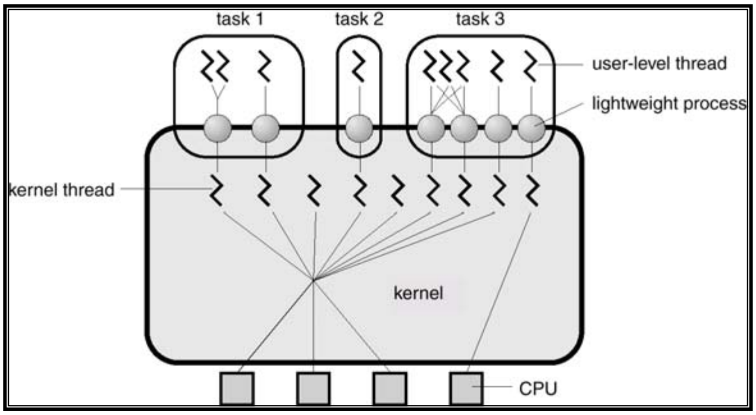
Many-to-Many Model



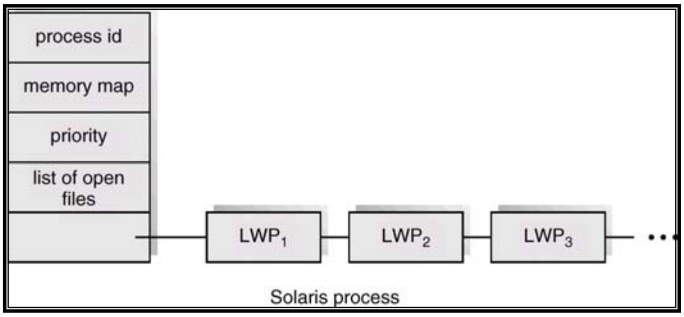
Pthreads

- 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.

Solaris 2 Threads



Solaris Process



Windows 2000 Threads

- Implements the one-to-one mapping.
- Each thread contains
  - a thread id
  - register set
  - separate user and kernel stacks
  - private data storage area

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)

Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java threads are managed by the JVM.

Creating and Using threads

- Pthreads Multi-threading Library
  - Supported on Solaris, Linux, Windows (maybe)
  - pthread_create, pthread_join, pthread_self, pthread_exit, pthread_detach
- Java
  - provides a Runnable interface and a Thread class as part of standard Java libraries
    - users program threads by implementing the Runnable interface or extending the Thread class

Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
Set and return the thread's priority.

run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()
Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(int millisecs)
Cause the thread to enter the *SUSPENDED* state for the specified time.

yield()
Enter the *READY* state and invoke the scheduler.

destroy()
Destroy the thread.

Creating threads

```
class Simple implements Runnable {
   public void run() {
     System.out.println(“this is a thread”);
   }
}

Runnable s = new Simple();
Thread t = new Thread(s);
t.start();
```

Alternative strategy: Extend Thread class (not recommended unless you are creating a new type of Thread)

Race Conditions

Consider two threads T1 and T2 repeatedly executing the code below

```
int count = 100;   // global

increment ( ) {
     int  temp;

     temp = count;
     temp = temp + 1;
     count = temp;
}
```

Time ↓

Thread T1	Thread T2
temp = 100	
count = 101	
	temp = 101
	count = 102
temp = 102	
	temp = 102
count = 103	
	count = 103

We have a *race condition* if two processes or threads want to access the same item in shared memory at the same

Assignment 1

- Three experiments
  - All you have to do is compile and run programs
  - Linux/Solaris
- First two experiments illustrate differences between processes and threads
- Third experiment shows a race condition between two threads