Process and Thread

Process concept. Process management. PCB and context switch. Short-term vs. long-term CPU scheduling. Process creation and termination. Inter-process communication: shared memory and message passing. Thread vs. process. Kernel vs. user thread.

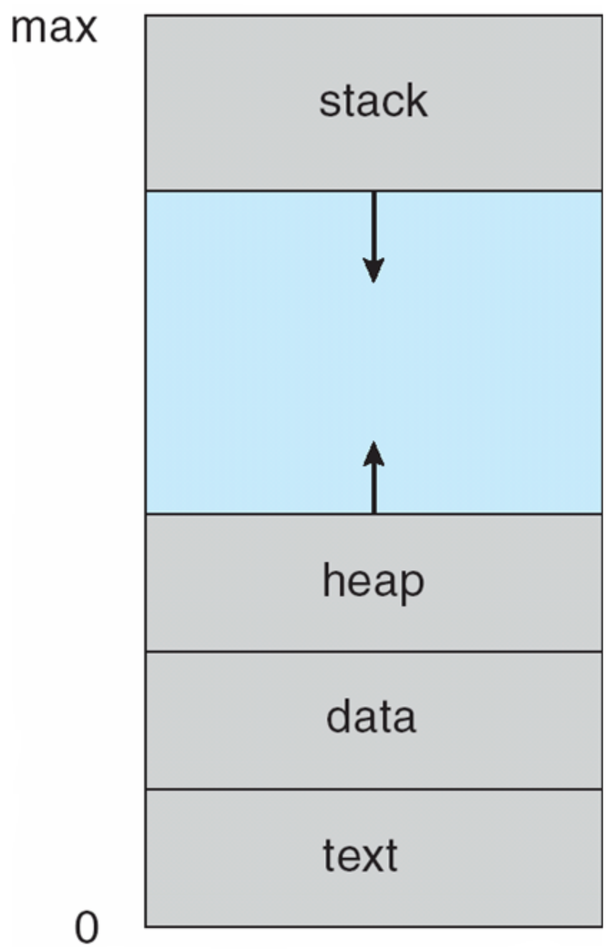
OS3: 1/2/2016
Textbook (SGG): Ch. 3.1-3.3,3.4.1,3.6.1,4.1-4.2,4.3.1,4.4,4.5.1-4.5.2

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
  - Program is static (a file); process is dynamic
  - You can run the same program *n* times (e.g., edit *n* files): one program, *n* processes
- A process defines a line of *concurrency*, includes:
  - program counter
  - stack
  - data section

Process in Memory

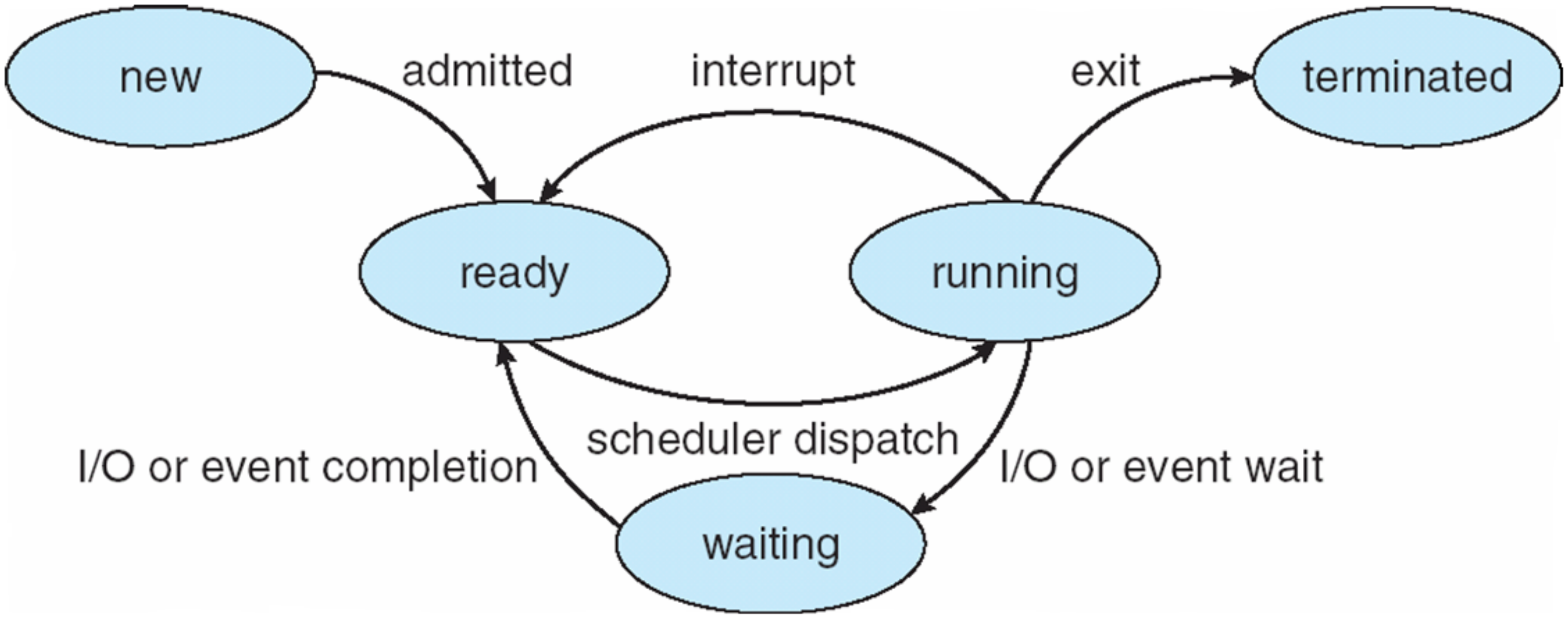
- Process also defines an address space
  - Address space is *private*
  - Not accessible (by default) from another process
- Hence, process couples **two** abstractions
  - Concurrency
  - Protection
- Can OS determine direction of stack growth?
- Advantage of stack and heap growing in opposite directions?



Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting** (or **blocked**): The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor (runnable)
    - What's needed to change it from runnable to running?
  - **terminated**: The process has finished execution

Diagram of Process State



Process Control Block (PCB)

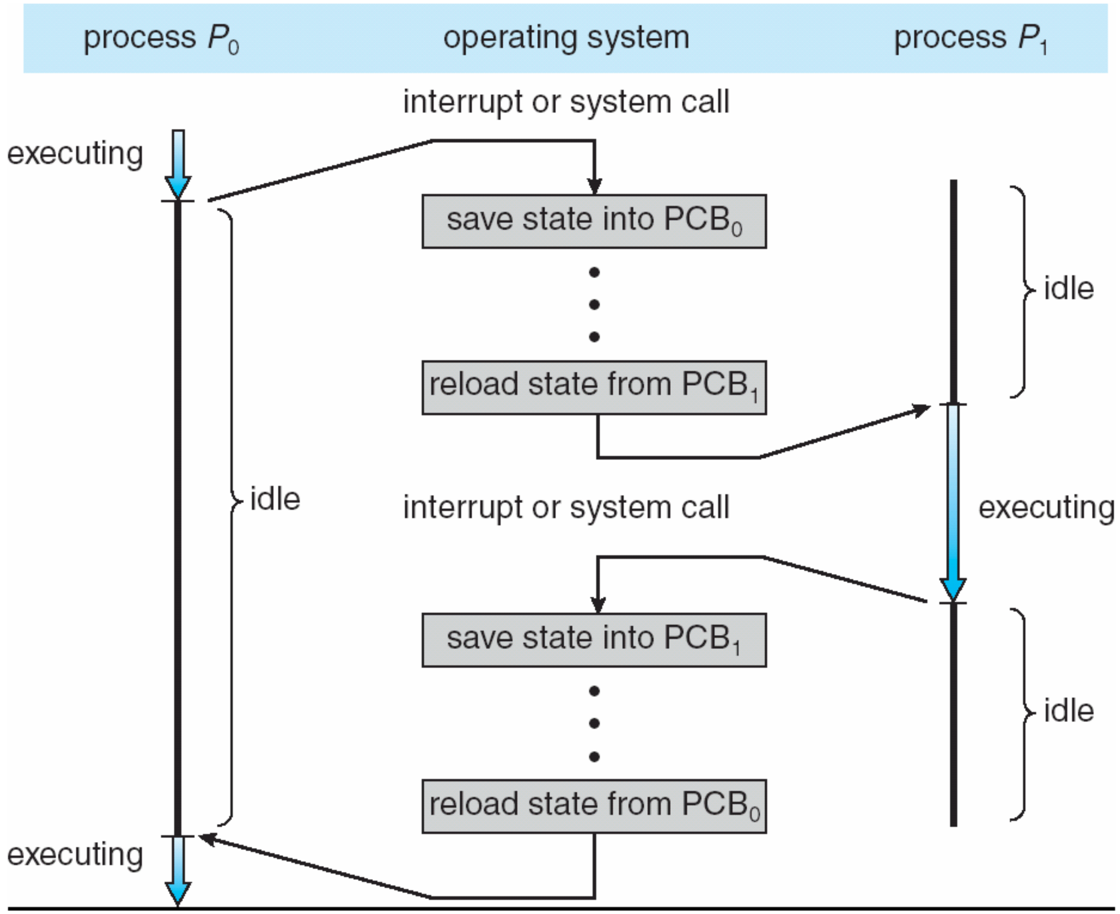
To manage a process, OS keeps information about each process in a data structure called PCB

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)

CPU Switch From Process to Process

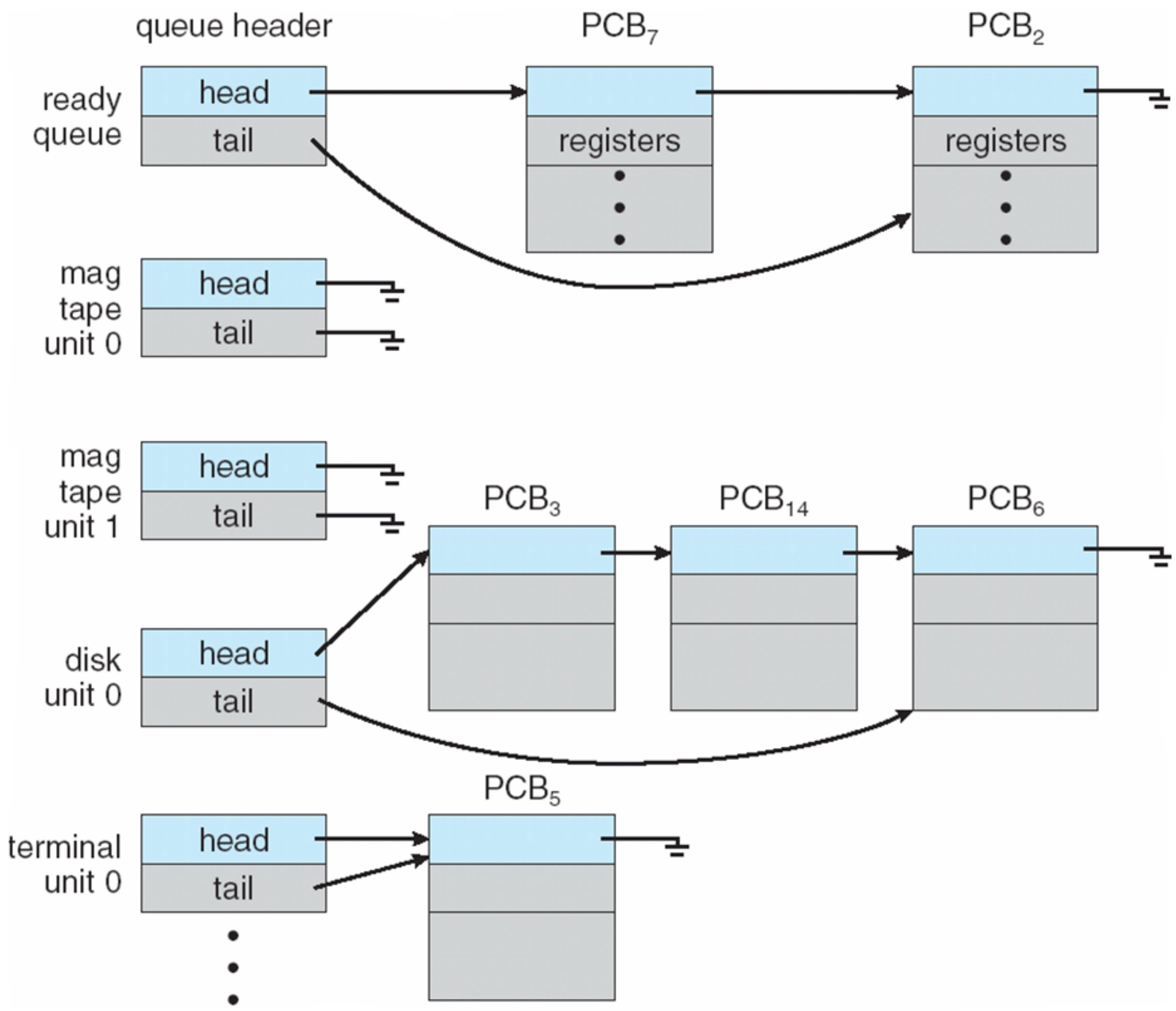
- Processes run independently (logically separate) and concurrently
- For a uniprocessor system, this is an illusion
  - *As if* each process has its own CPU
  - Can view it as *virtual* CPU



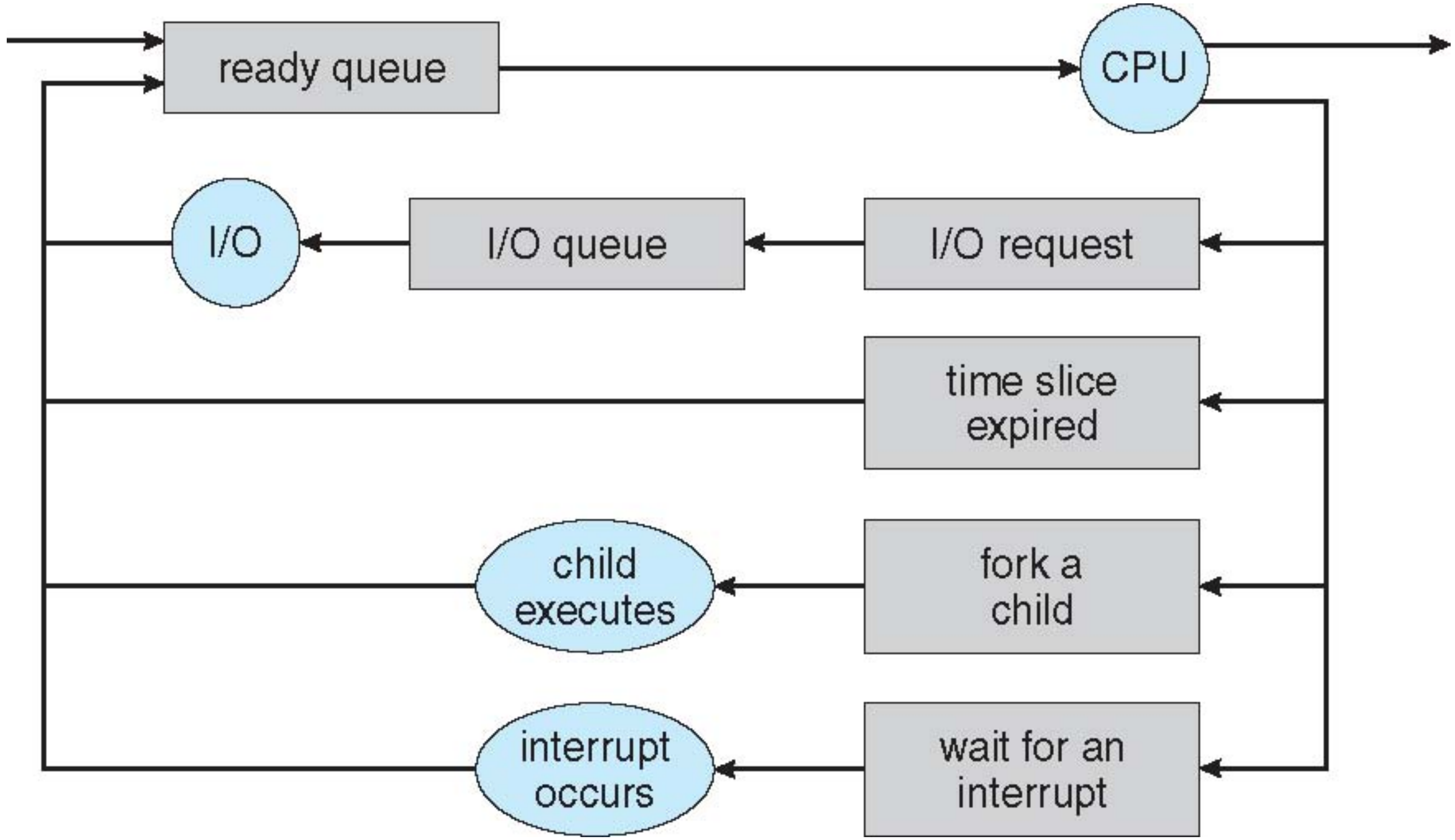
Process Scheduling Queues

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

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Two Kinds of CPU Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - i.e., loaded into main memory (swapped in)
  - If memory is limited, some processes can be *swapped out* (where are they then?)
- **Short-term scheduler** (or CPU scheduler) – selects which (in-memory) process should be executed next and allocates CPU to that process

Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (e.g., milliseconds) ⇒ must be fast
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ can be slow
- The long-term scheduler controls the *degree of multiprogramming*
  - i.e., How many processes are allowed to run at the same time
- Processes can behave quite differently, e.g.,
  - **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

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 via a **context switch**
- **Context** (where the process is in its execution) of a process stored in the PCB
- Context-switch time is *overhead*
  - System does no useful work while switching
  - So, don't want to do it too much, but what is *just enough*?
- Time dependent on hardware support

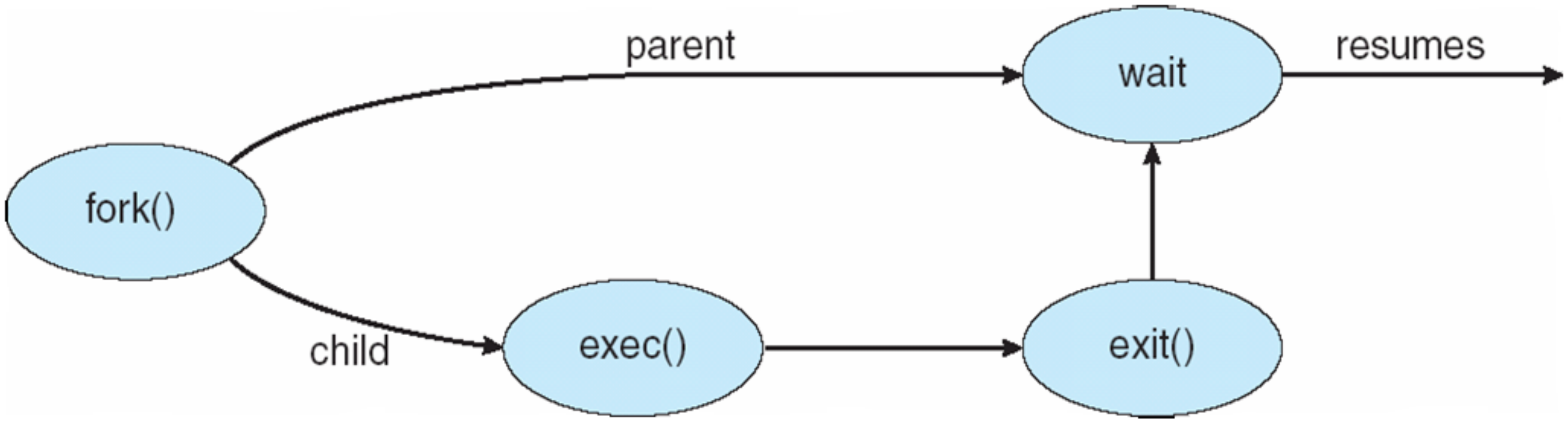
Process Creation

- Processes form a family tree!
- **Parent** process create **child** processes, which in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **process identifier** (**pid**)
- Parent/children can share resources (e.g., opened files) in different ways
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent can wait for children to terminate (**wait**() system call)

Process Creation (Cont.)

- Address space
  - Child gets its own address space, whose content is initially a duplicate of parent's
  - Child then usually loads a new program into its address space
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call (used after a **fork)** loads new program into the process's memory space

Process Creation



- After parent creates child, execution for both resumes as return from fork()
- How do you tell parent's return from child's return then?

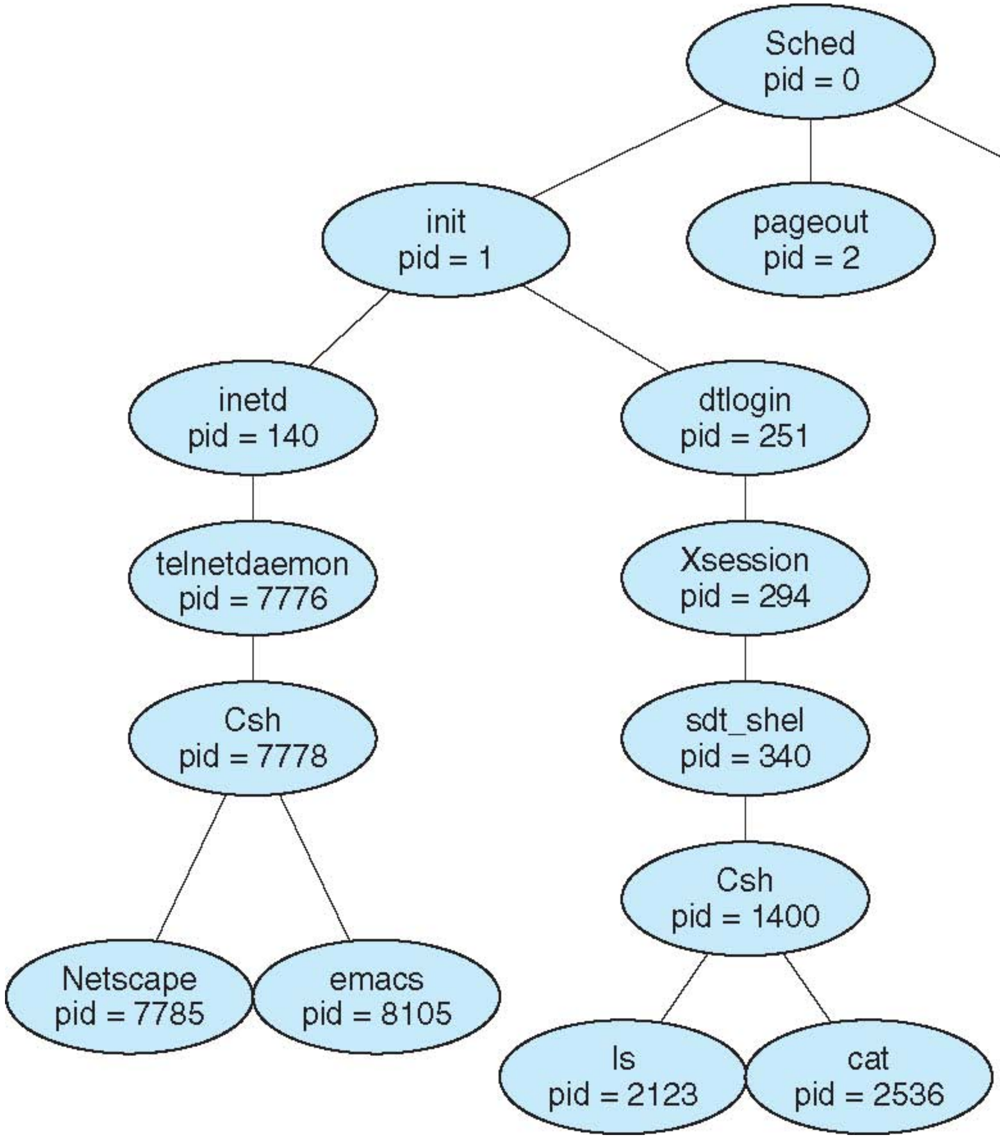
C Program Forking Separate Process

```
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);
	}
}
```

Process Creation in Java

```
import java.io.*;

public class OSProcess
{
 public static void main(String[] args) throws IOException {
  if (args.length != 1) {
   System.err.println("Usage: java OSProcess <command>");
   System.exit(0);
  }

  // args[0] is the command
  ProcessBuilder pb = new ProcessBuilder(args[0]);
  Process proc = pb.start();

  // obtain the input stream
  InputStream is = proc.getInputStream();
  InputStreamReader isr = new InputStreamReader(is);
  BufferedReader br = new BufferedReader(isr);

  // read what is returned by the command
  String line;
  while ( (line = br.readLine()) != null)
    System.out.println(line);

  br.close();
 }
}
```

A tree of processes (Solaris OS)



Chicken and egg: who creates first chicken (sched)?

Process Termination

- Process executes last statement and informs OS (via **exit()** system call)
  - Output data from child to parent (via **wait**)
  - Process's resources may be deallocated by OS
    - But sometimes, process can also still exist as "zombie"
- Parent may terminate execution of child processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent exits
    - Some operating systems don't allow its children to continue
      - All children terminated - **cascading termination**
    - UNIX has *job control* feature that defines different behaviors
      - So a UNIX job means a related group of processes, not synonymous with process

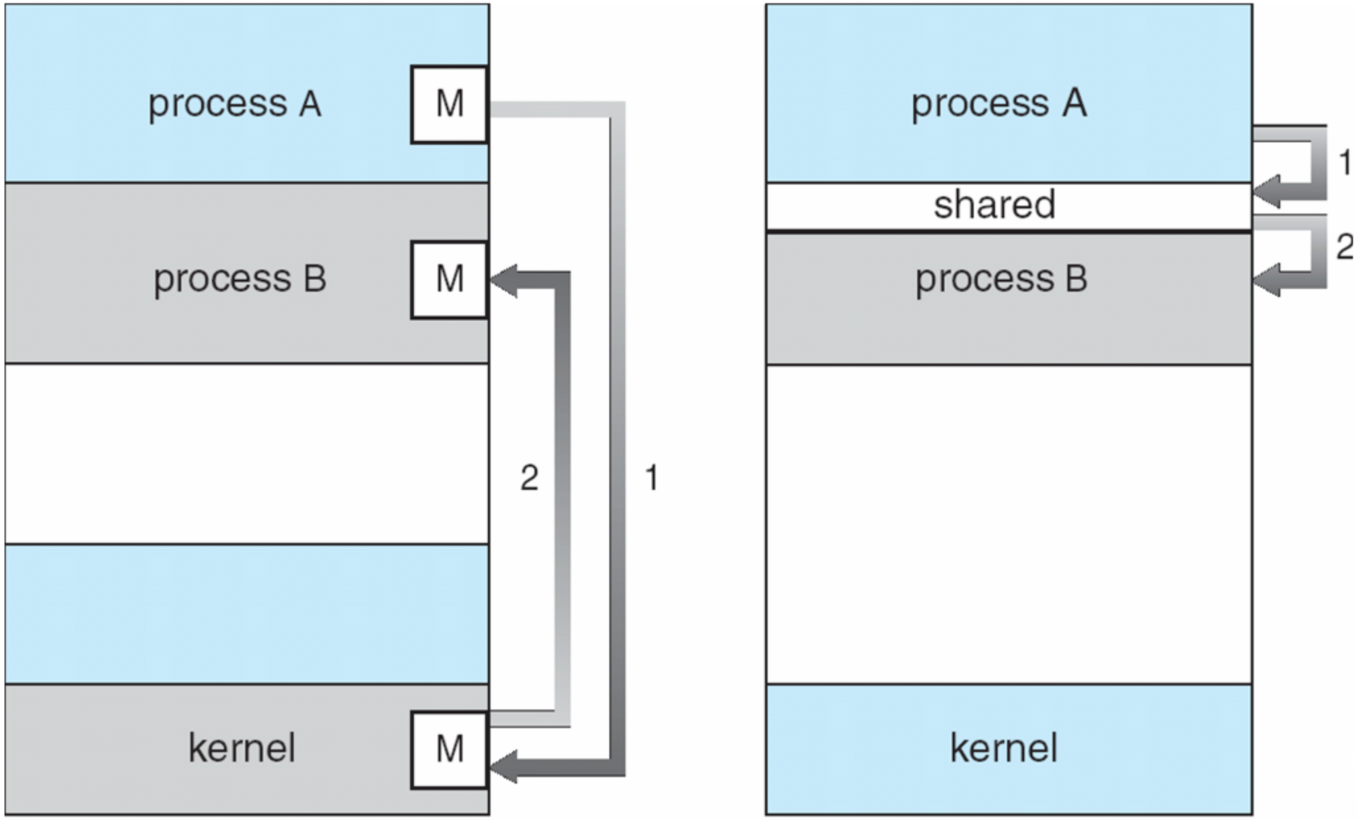
Interprocess Communication

- Processes are by default **independent,** but they can also agree to be **cooperating**
- Cooperating processes may affect each other, mainly through sharing data
- Reasons for cooperating processes:
  - Share information
  - Speed up computation
  - Achieve modularity (hence protection) in spite of cooperation
  - Convenience (e.g., "ls -l | wc -l" roughly counts how many files you have)
- Cooperating processes need **interprocess communication** (**IPC**)
- Two basic models of IPC
  - Shared memory
  - Message passing

Activity 3.1: Multiprocess Program

- If you run a task as N processes, what is the maximum speedup you can expect over a single-process implementation?
- If x% of the task is sequential (i.e., can't be parallelized), what then is the maximum speedup?
- On a practical system, what other important factors will limit your actual speedup below the maximum possible?

Communications Models



(a) (b)

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - Items produced and waiting to be consumed are put in *shared* buffer
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

POSIX Example of Shared Memory IPC

- POSIX Shared Memory
  - Process first creates shared memory segment (w/ read, write access for owner)

```
segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

  - Process wanting access to that shared memory must attach to it

```
shared_memory = (char *) shmat(id, NULL, 0);
```

  - Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

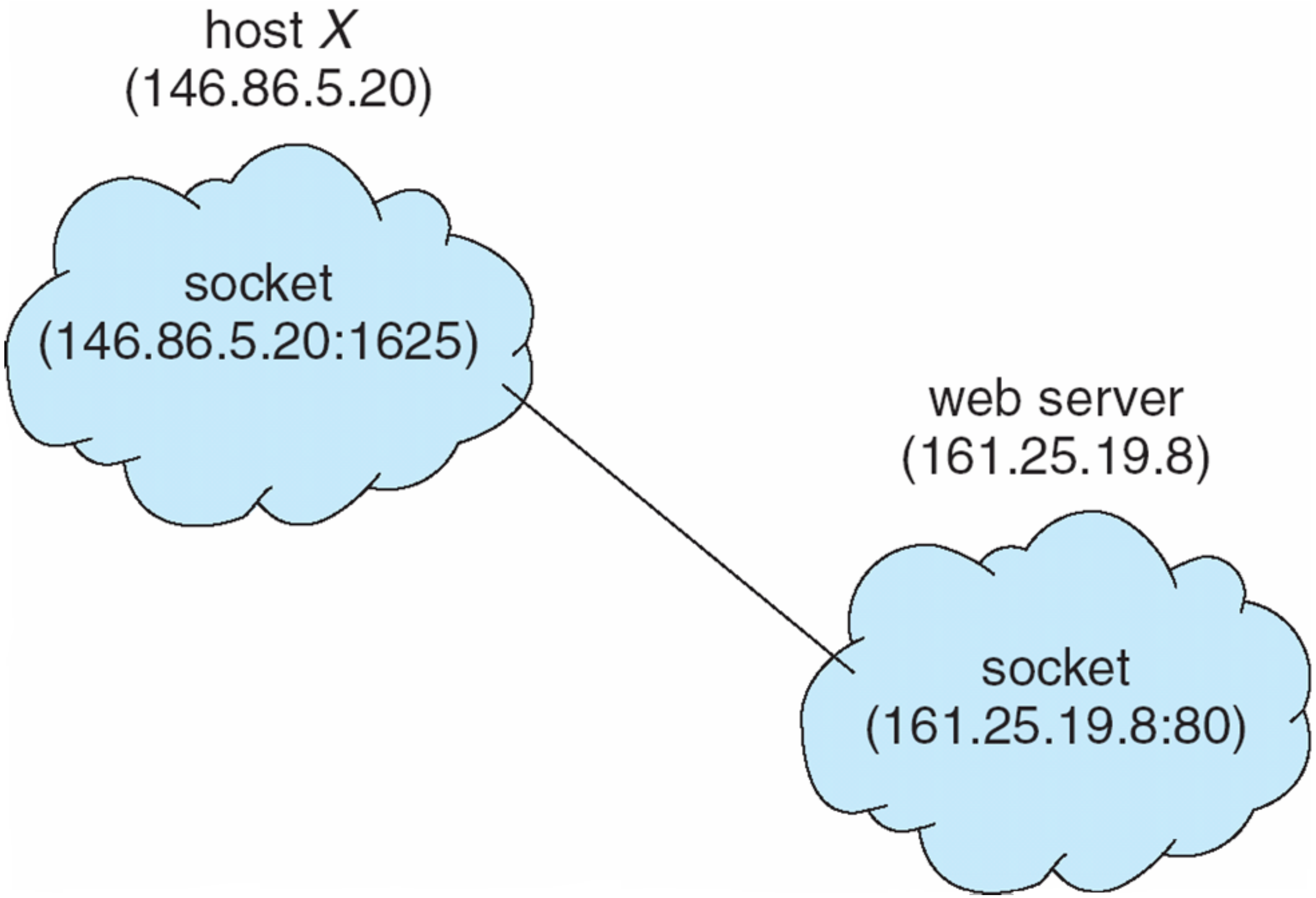
  - When done a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

Message Passing Example: Sockets

- A socket is defined as an *endpoint for communication*
  - Usually for network communication (e.g., TCP/IP), but also IPC within single machine
- Endpoint specified as concatenation of IP address and (TCP or UDP) port
  - E.g., **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication occurs between a pair of sockets – two flavors:
  - Connection oriented (e.g., TCP)
  - Connectionless (e.g., UDP)
- More details when we study networking

Socket Communication



Socket Communication in Java

```
public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         // now listen for connections
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            // write the Date to the socket
            pout.println(new java.util.Date().toString());

            // close the socket and resume
            // listening for connections
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

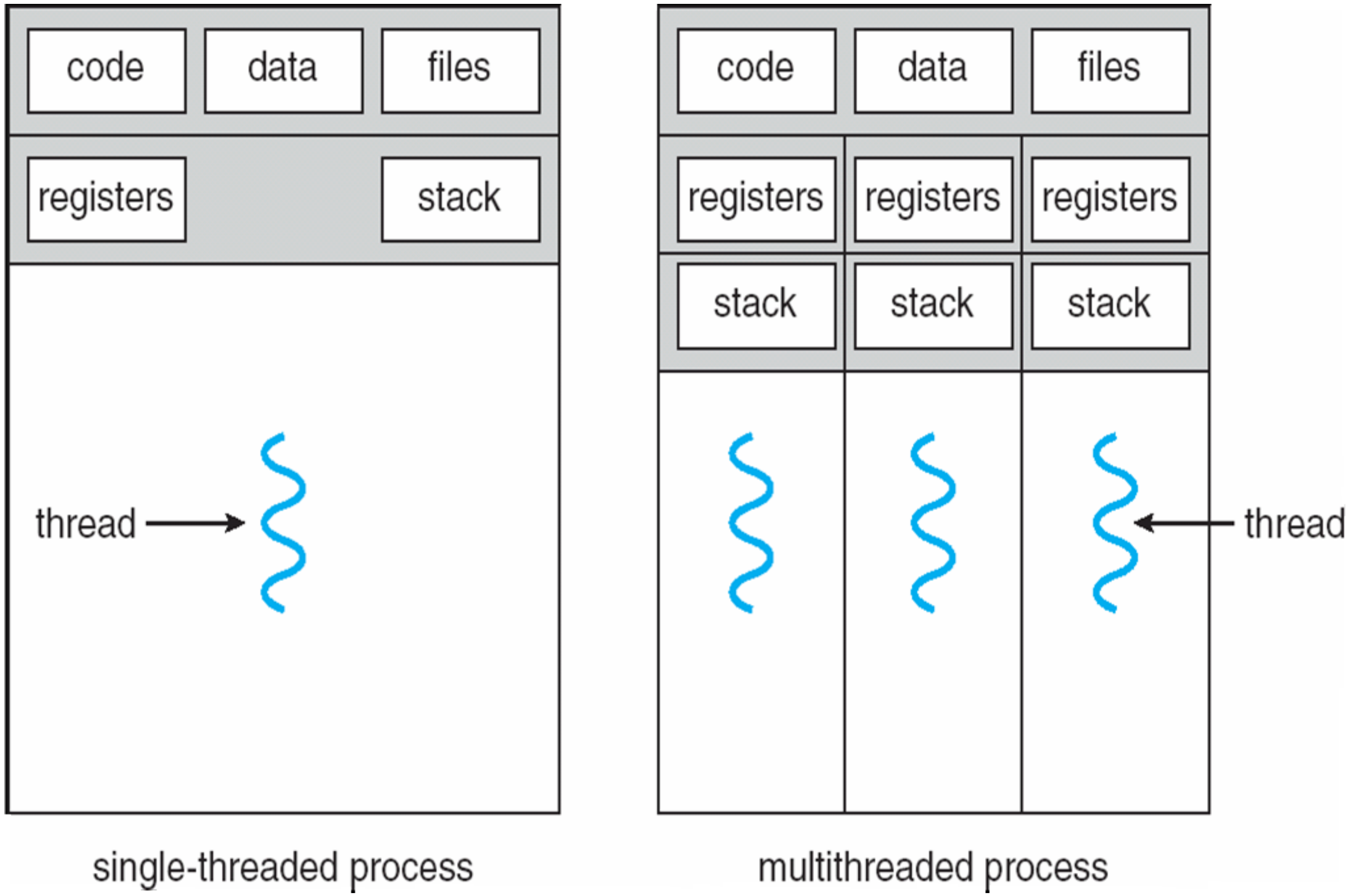
Socket Communication in Java

```
public class DateClient
{
   public static void main(String[] args) {
      try {
         //make connection to server socket
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         // read the date from the socket
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         // close the socket connection
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Thread vs. Process

- Recall: Process couples two abstractions: concurrency and protection
- Can I decouple the two, e.g., have concurrency *without* protection?
  - Yes, use *threads*
  - Many threads can run within a process, share the process's address space
    - No protection between them
    - But IPC is simple (e.g., no need for shmget, shmat, etc) and fast (much less to save/restore at context switch)

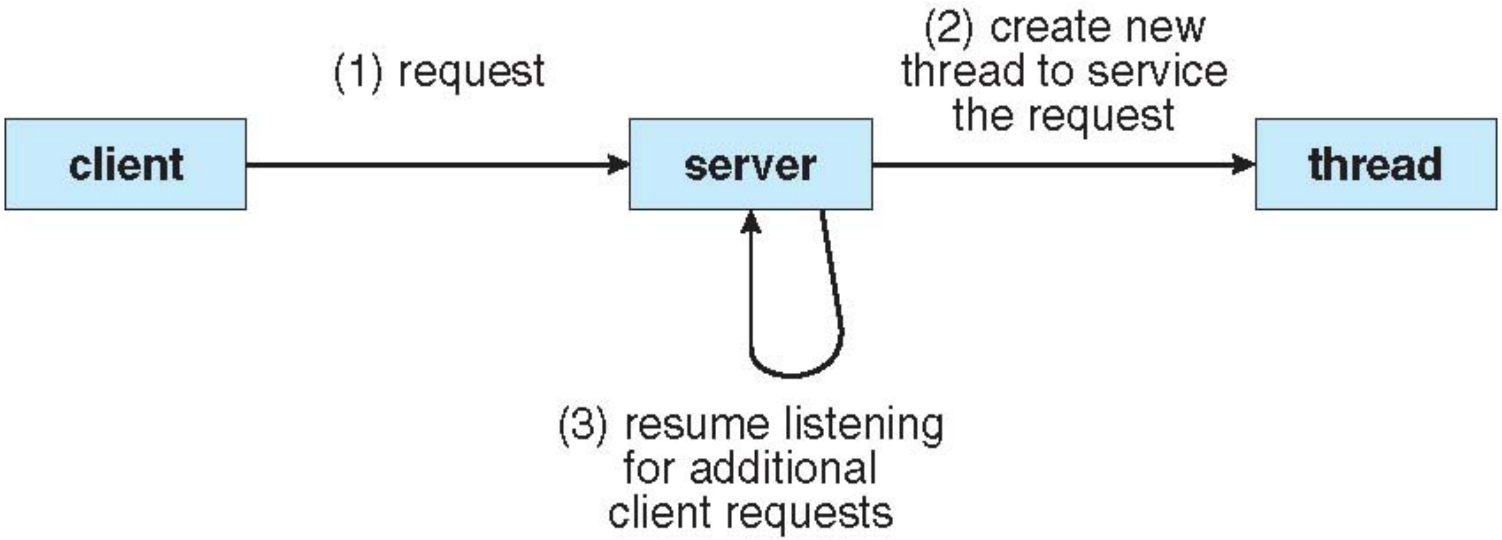
Single and Multithreaded Processes



Why (or why not) threads?

- Speedup by parallel execution (e.g., your Lab 2)
  - On multiprocessor or multicore systems
- Responsiveness
  - While one thread is blocked for IO, another thread can be executing and doing useful computation
- Logical modularity
  - Though without fault isolation
- Disadvantages
  - Context switch + synchronization overheads
  - Can be much harder to program and get right!

Multithreaded Server Architecture



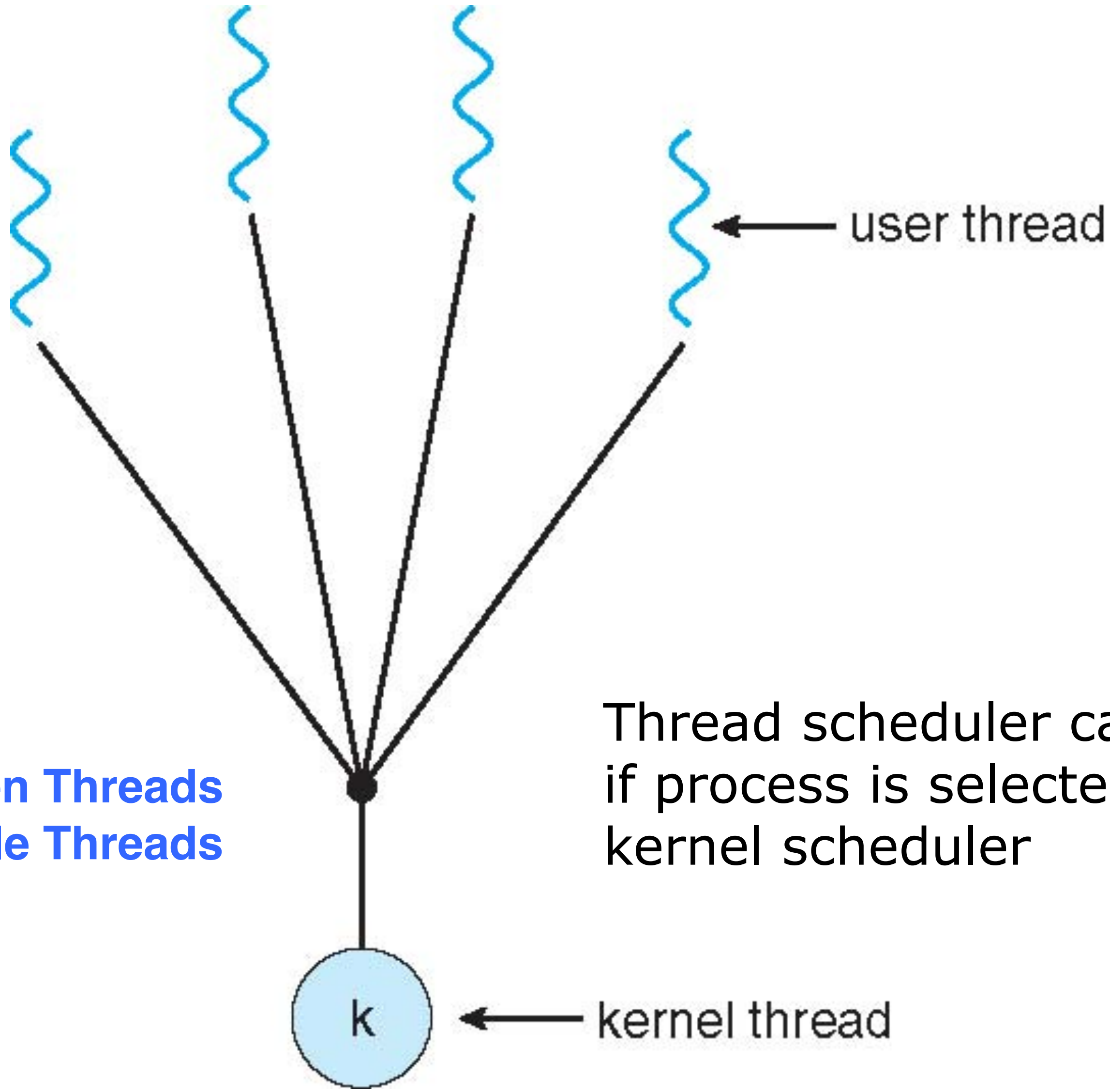
Two types of threads

- *Kernel* threads
  - Known to OS kernel
  - Scheduled by kernel CPU
  - Take up kernel data structure (e.g., Thread Control Block like PCB)
  - More expensive
- *User* threads
  - Not known to OS kernel
  - Scheduled by thread scheduler (running in user mode) in thread library (e.g., POSIX pthread or Java threads) linked with process
  - Less expensive

Mapping from user to kernel threads

- Many-to-One
- One-to-One
- Many-to-Many
  - Two-level model

Many-to-One Model

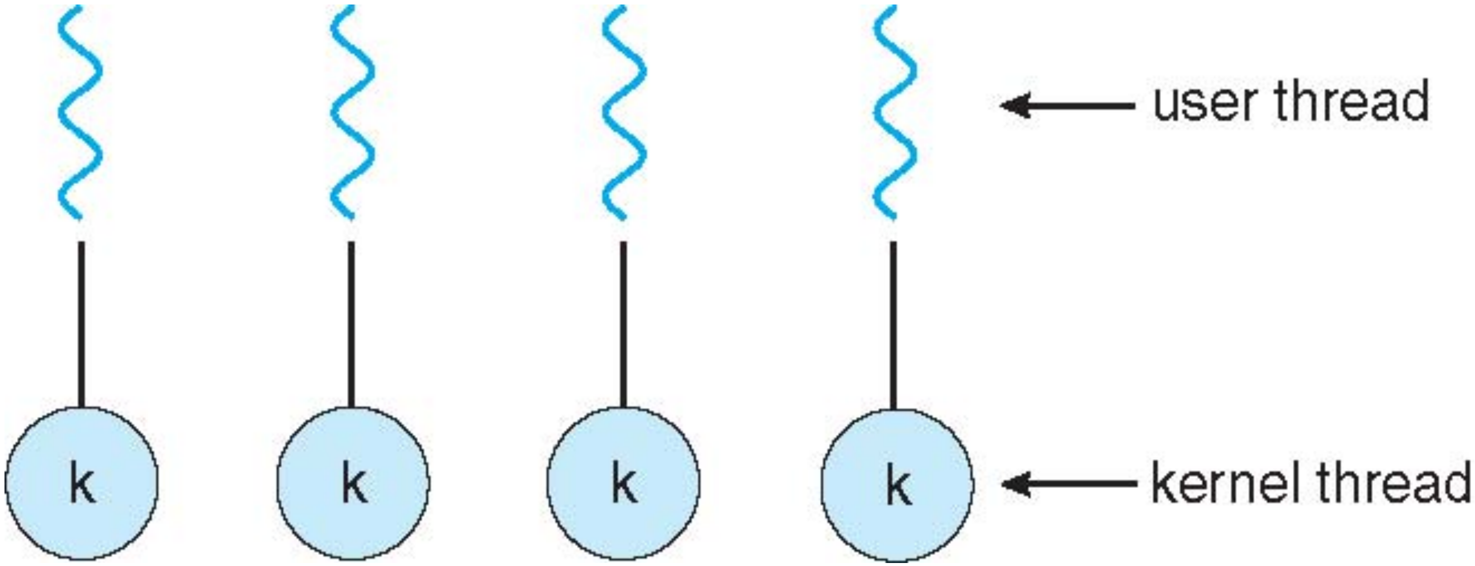


Examples:

- **Solaris Green Threads**
- **GNU Portable Threads**

Thread scheduler can run only if process is selected to run by kernel scheduler

One-to-one Model



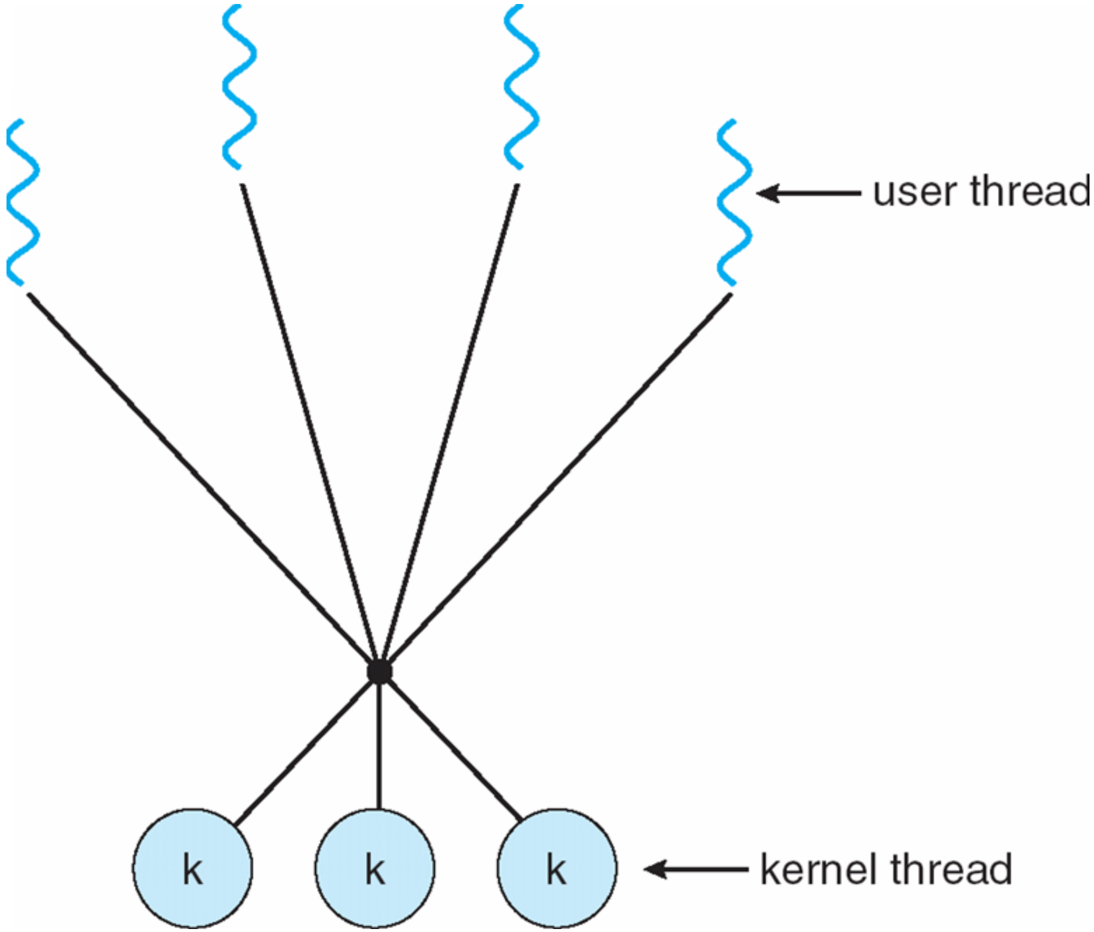
Examples

- Windows NT/XP/2000
- Linux
- Solaris 9 and later

Many-to-Many Model

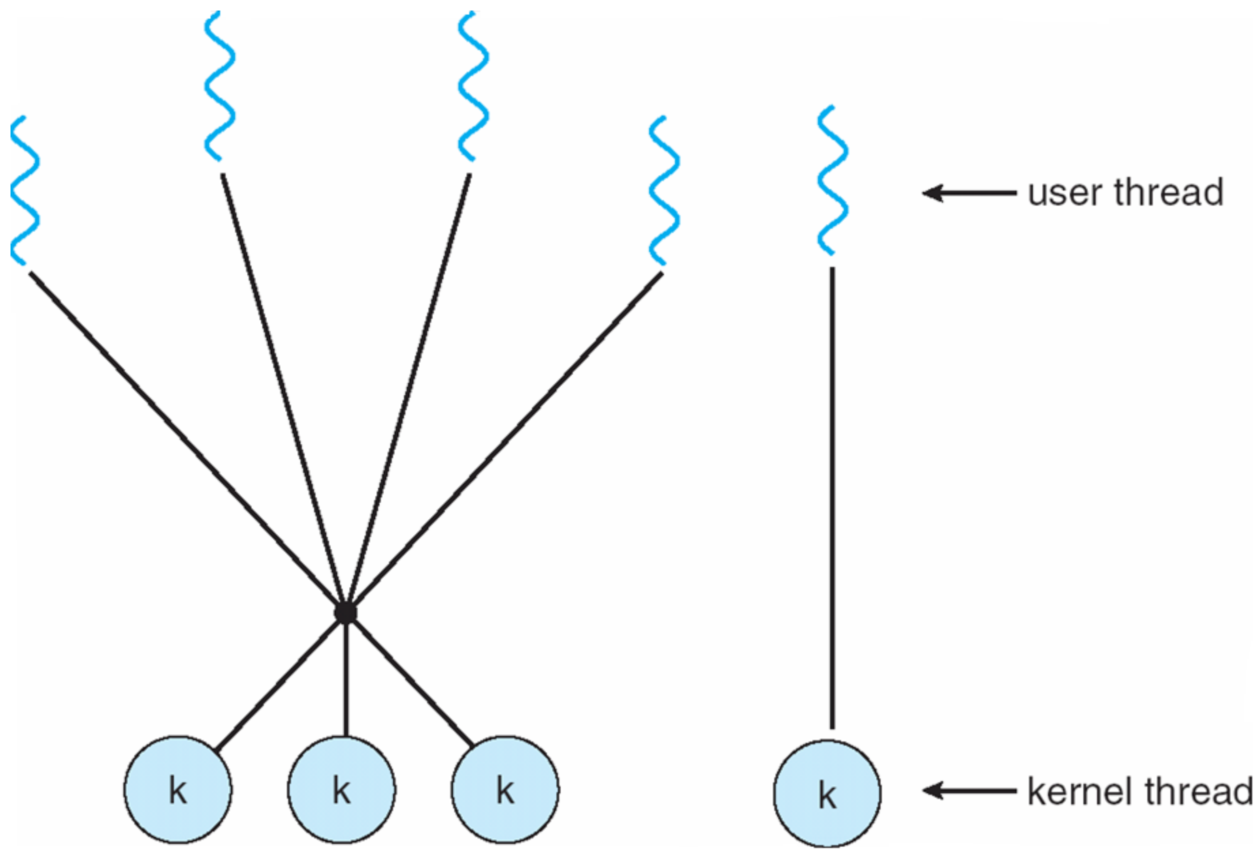
Examples

- Solaris before v9 (Solaris LWP is user thread's door to kernel thread)
- Windows NT/2000 w/ ThreadFiber package



Two-level Model

Like many-to-many, except possible to bind user thread to kernel thread



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

Activity 3.2: Kernel vs. User Threads

- You have a multithreaded process in execution on a uniprocessor. One of the threads is in the middle of processing a previously accepted user command and it is runnable. Another thread, which is running, attempts to read a user command from the network and blocks in the kernel.
- What will happen next if these threads are kernel threads?
- What will happen next if these threads are user threads?
- Why is it generally faster to context switch user threads than kernel threads?

Pthreads

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

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
  - Implementing the Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

Java Threads - Example Program

```
class MutableInteger
{
  private int value;
  public int getValue() {
   return value;
  }
  public void setValue(int value) {
   this.value = value;
  }
}

class Summation implements Runnable
{
  private int upper;
  private MutableInteger sumValue;
  public Summation(int upper, MutableInteger sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }
  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setValue(sum);
  }
}
```

Java Threads - Example Program

```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0)
       System.err.println(args[0] + " must be >= 0.");
      else {
       // create the object to be shared
       MutableInteger sum = new MutableInteger();
       int upper = Integer.parseInt(args[0]);
       Thread thrd = new Thread(new Summation(upper, sum));
       thrd.start();
       try {
          thrd.join();
          System.out.println
                 ("The sum of "+upper+" is "+sum.getValue());
       } catch (InterruptedException ie) { }
      }
     }
     else
      System.err.println("Usage: Summation <integer value>");
     }
}
```

Java Thread States