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: 30/1/2018
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

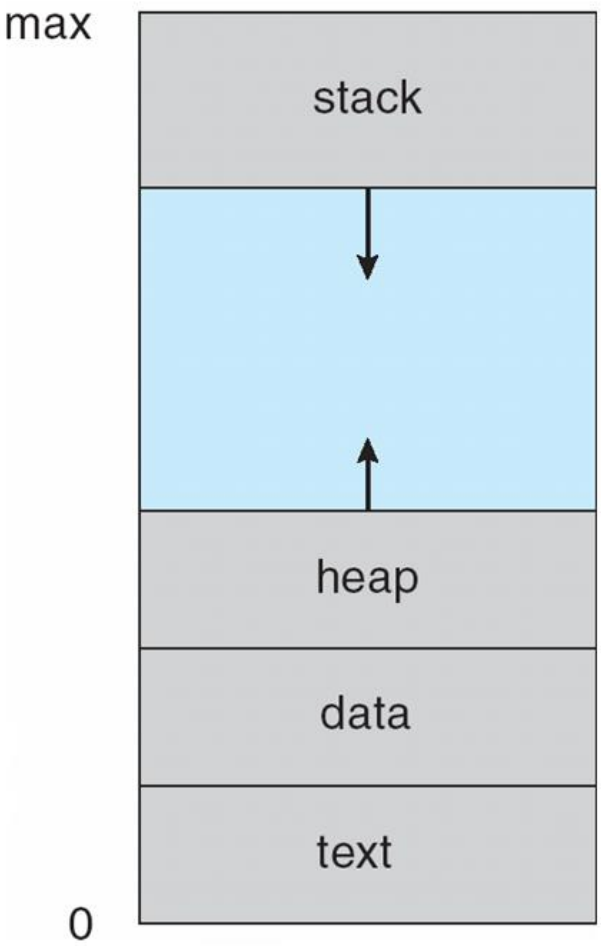
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN
Established in collaboration with MIT

Process Concept

- An operating system executes a variety of programs
- Process – a program in execution; execution of one process progresses in sequential fashion
  - Program is static (a file, e.g., "/bin/ls"); process is *dynamic* (a *running program*)
  - 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
  - dynamically allocated memory in *heap* – Java manages it automatically for you; for C, you use **malloc(3)** and **free(3)**
- Textbook uses the terms *job* and *process* almost interchangeably

Process in Memory

- Process also defines an *address space* (of memory)
  - 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?



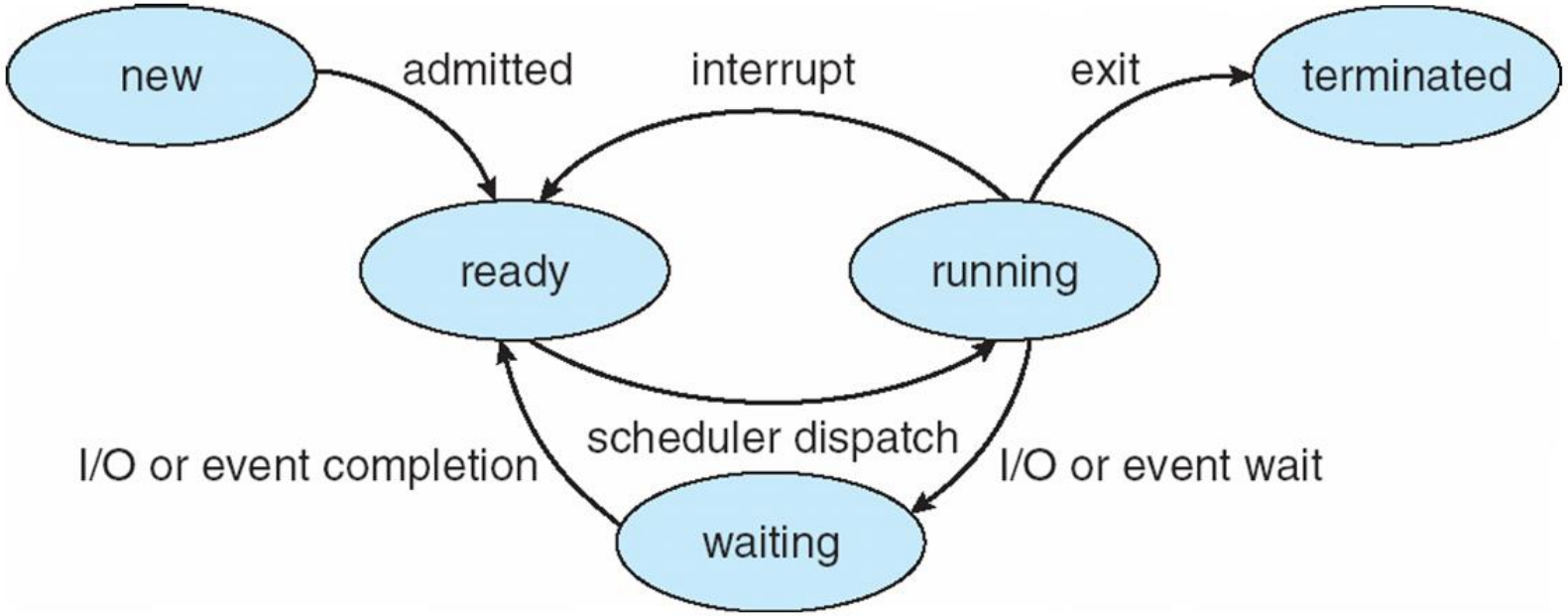
NB: *text* section is executuable code (i.e., program run by the process)

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN
Established in collaboration with MIT

Process Scheduling State

- As a process executes, it changes *scheduling state*
  - **new**: The process is being created
  - **running**: assigned the CPU; instructions are being executed
  - **waiting** (or **blocked**): The process is waiting for some event to occur (e.g., asked for input data from device, now waiting)
  - **ready (**including when unblocked when IO completes**)**: 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

Scheduling State Transitions



SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN
Established in collaboration with MIT

Process Control Block (PCB)

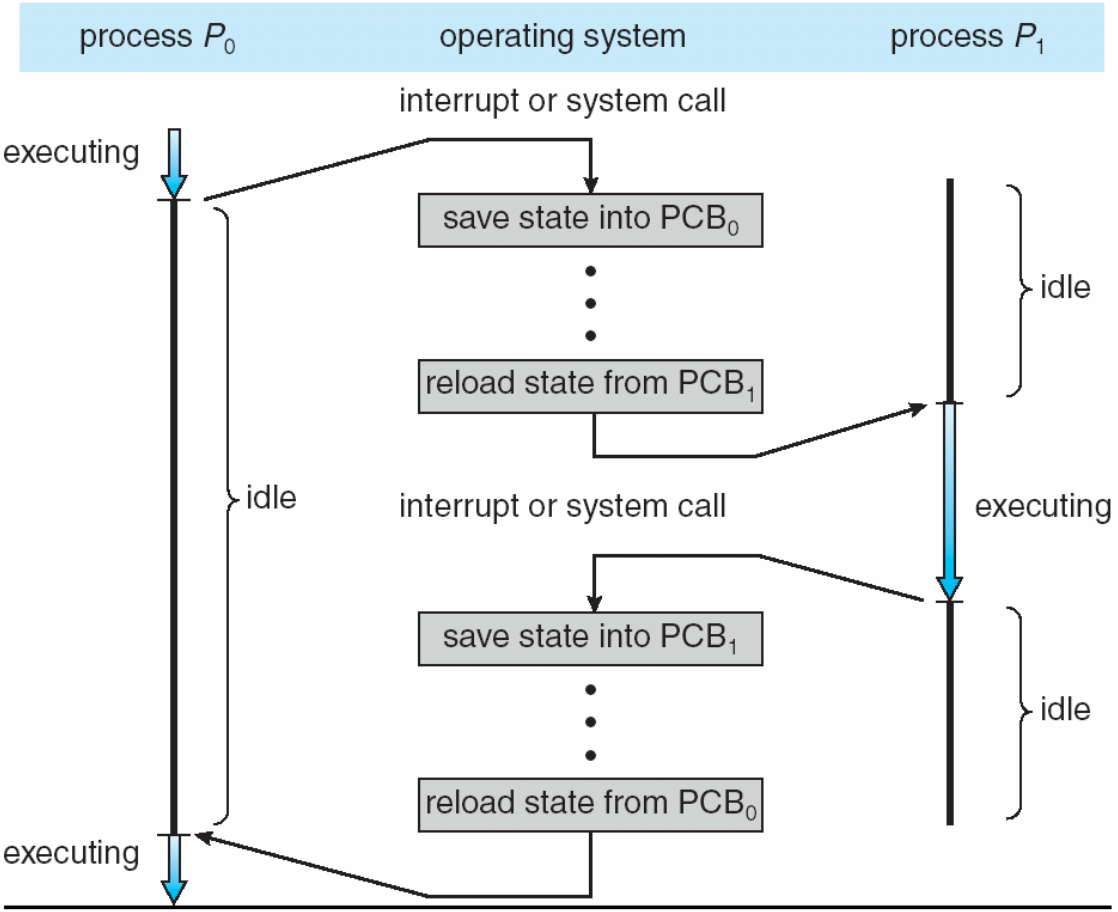
To manage a process, OS keeps information about each process in a data structure called PCB

- 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

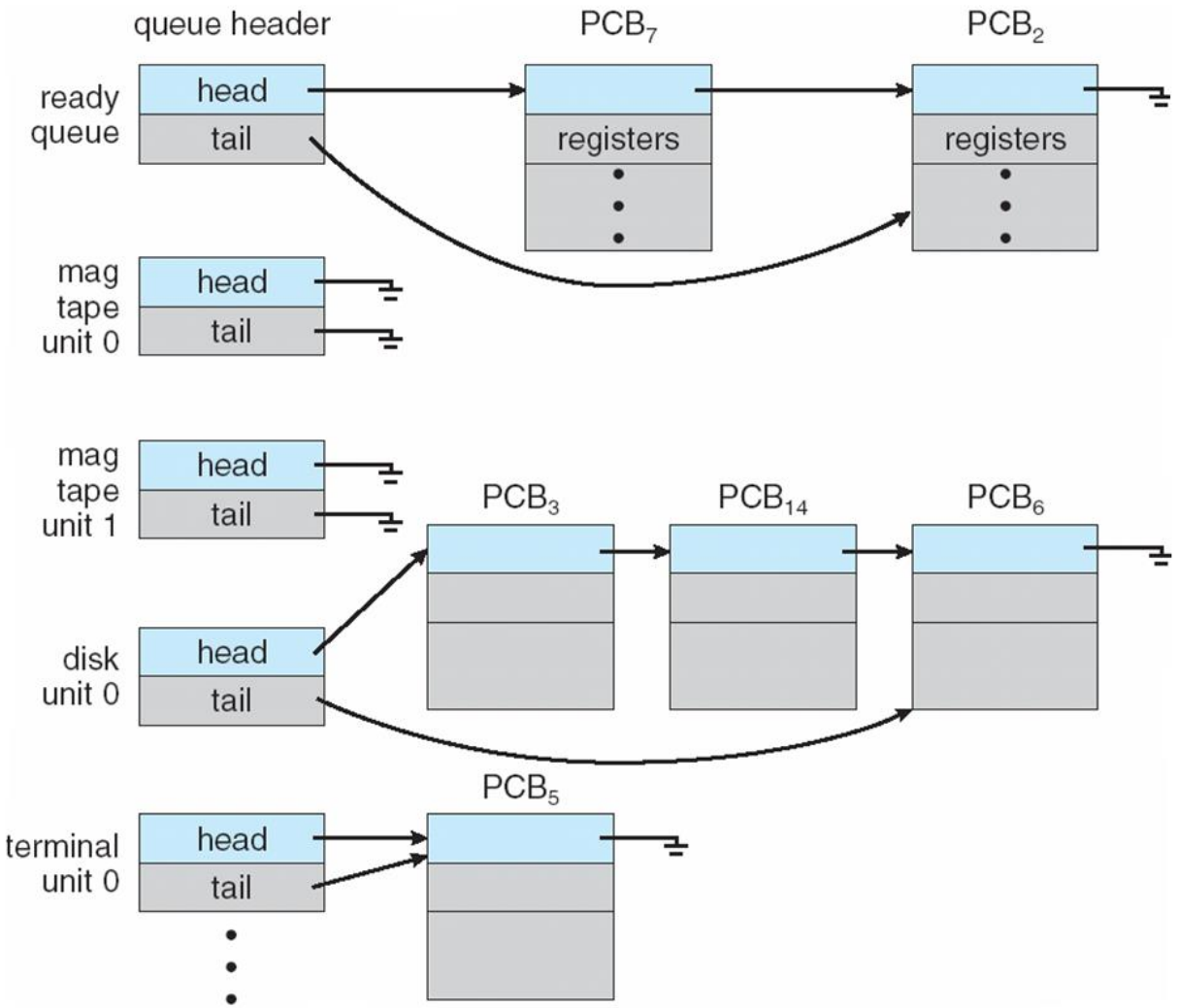
- Processes run independently (logically separate) and concurrently
- *Context switch*: change CPU from running one process to running another
- For a uniprocessor system, concurrency is an illusion
  - *As if* each process has its own CPU
  - Can view it as *virtual* CPU



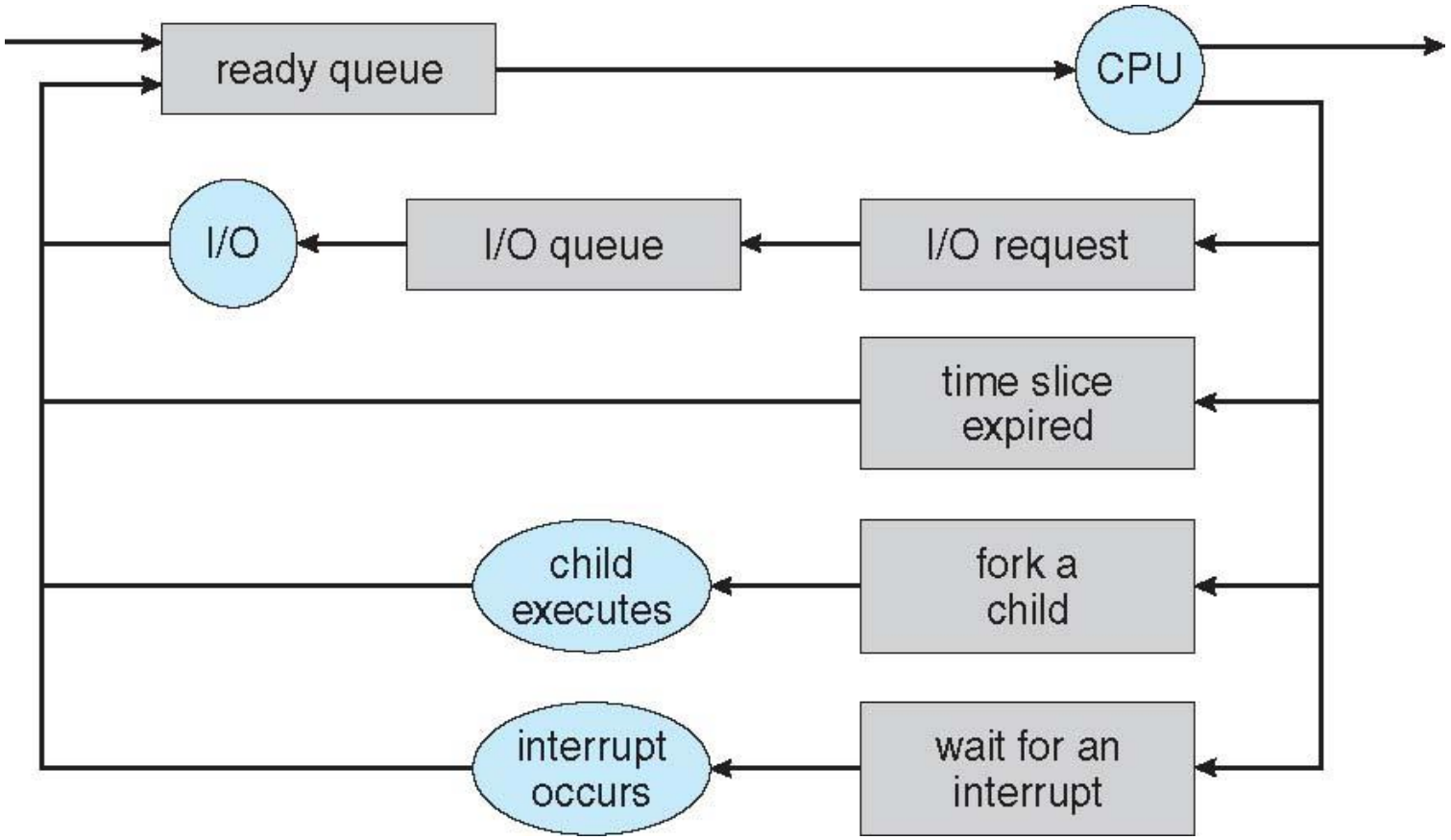
Process Scheduling Queues

- **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 (one queue for each device)
- Process migrates among the various queues during execution, depending on its scheduling state

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Note: create a process is also called *fork* (created process is *child* of creating 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 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*?
- Context switch time dependent on hardware support

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN
Established in collaboration with MIT

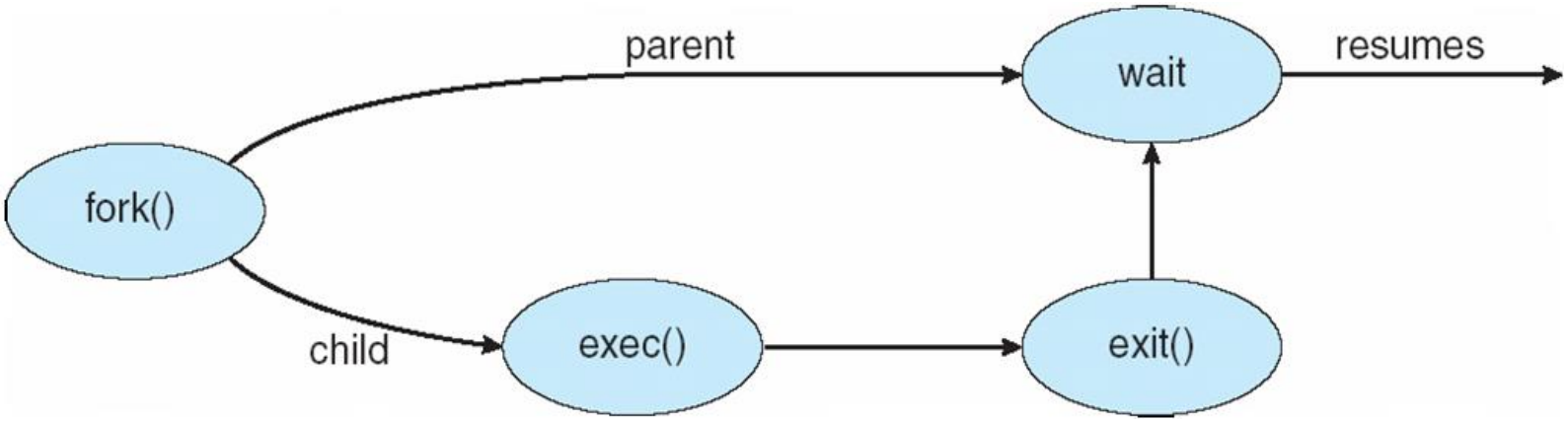
Process Creation

- Processes form a family tree!
- **Parent** process creates **child** processes, which in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **process identifier** (**pid:** positive integer)
- 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
  - How about the parent/child in your Ubuntu shell? Do they share stdout (standard output terminal of the process)? Working directory?
- 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 (including text section that stores the program being run)
  - Child then usually loads a new program into its address space
    - E.g., shell process creates child process, but child wants to run (say) "**ls**" program instead of being a duplicated shell
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call (used after a **fork)** loads new program (e.g., **ls**) 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

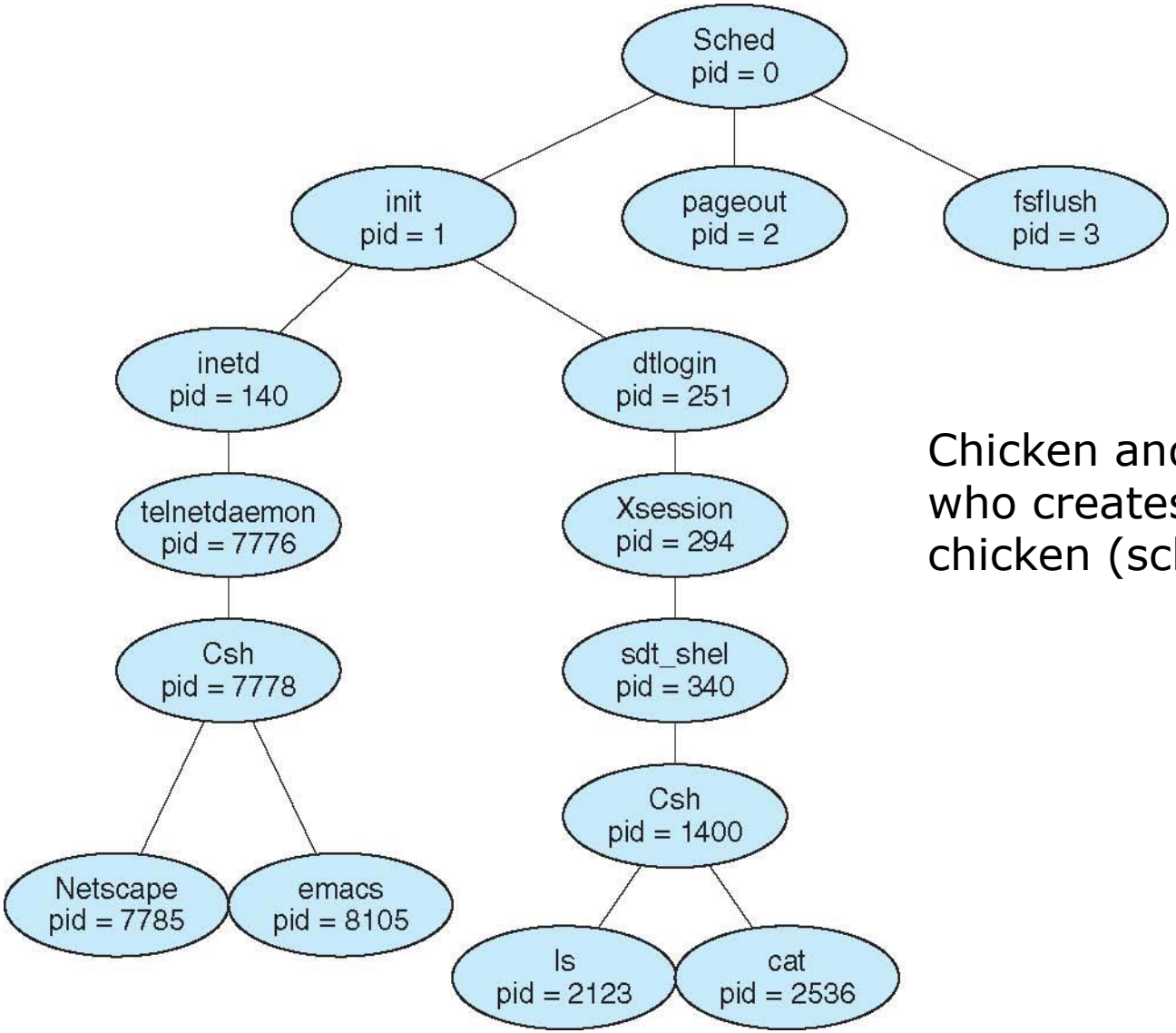
```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork(); /* both parent & child return from fork() */
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls",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

NB: (i) *Required (small) memorization*: how to distinguish parent's and child's returns based on return value of **fork**() system call! (ii) Will successful **execlp**() return to the code shown?

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 *C* to parent *P* (via *P*'s executing **wait()** system call)
  - Process's resources may be deallocated by OS
    - But sometimes, process needs to still exist as "zombie" (e.g., *C* terminates before *P*'s completes **wait** for *C*)
- 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 a synonym of process in this case

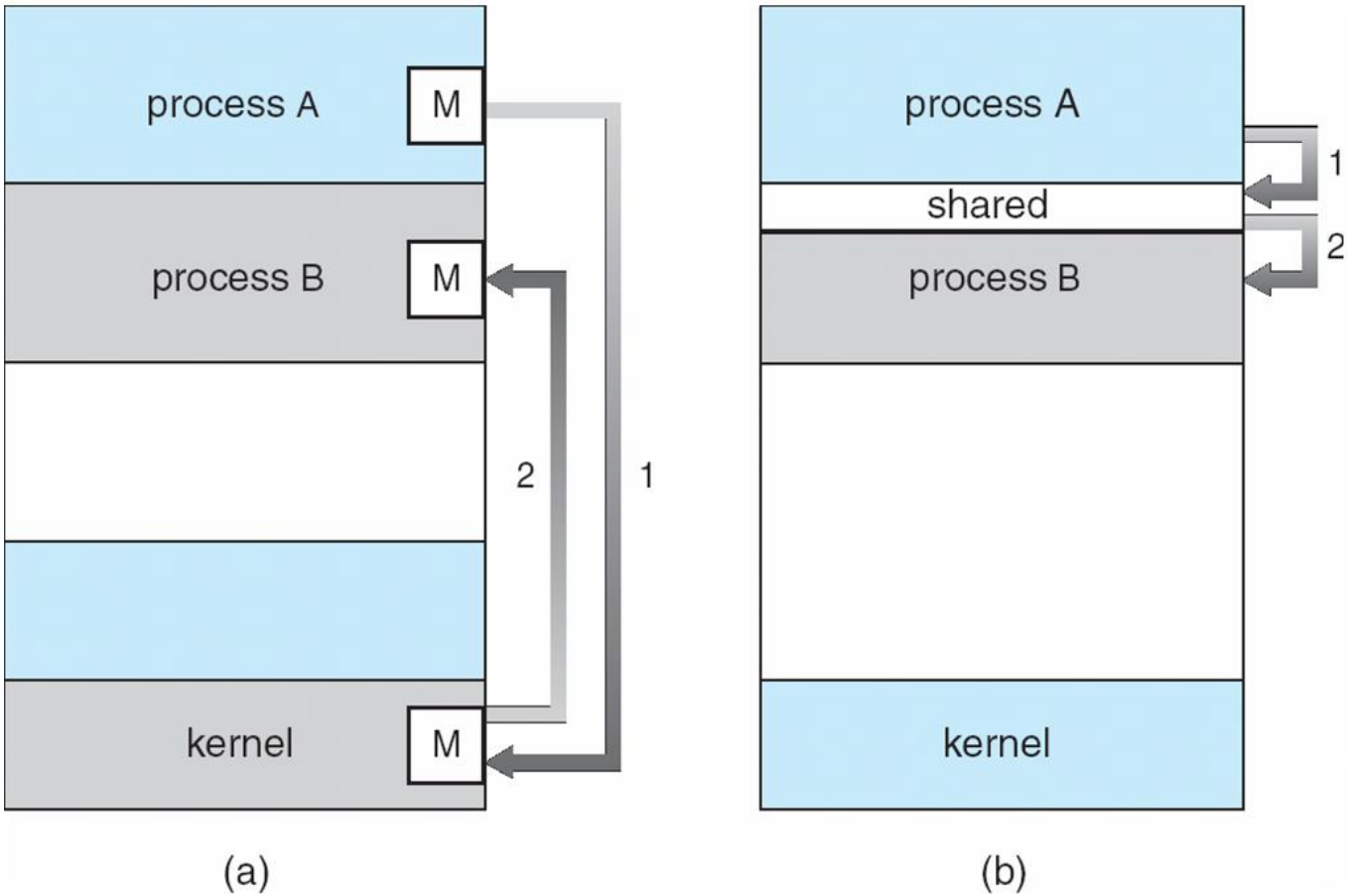
Interprocess Communication

- 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 – *try it on your Ubuntu shell*
- 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



1=system call to send message
2=system call to receive message

[*already used system calls (see next slide) to map shared memory in address spaces of both A and B*]
1=simple **store** instruction (no system call needed)
2=simple **load** instruction (no system call)

POSIX (C) 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 (if two processes both attach to it, they then share the memory)

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

  - Now the process could write to the shared memory

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

The data written by the `sprintf()` can now be read by another process that also attached to the segment id

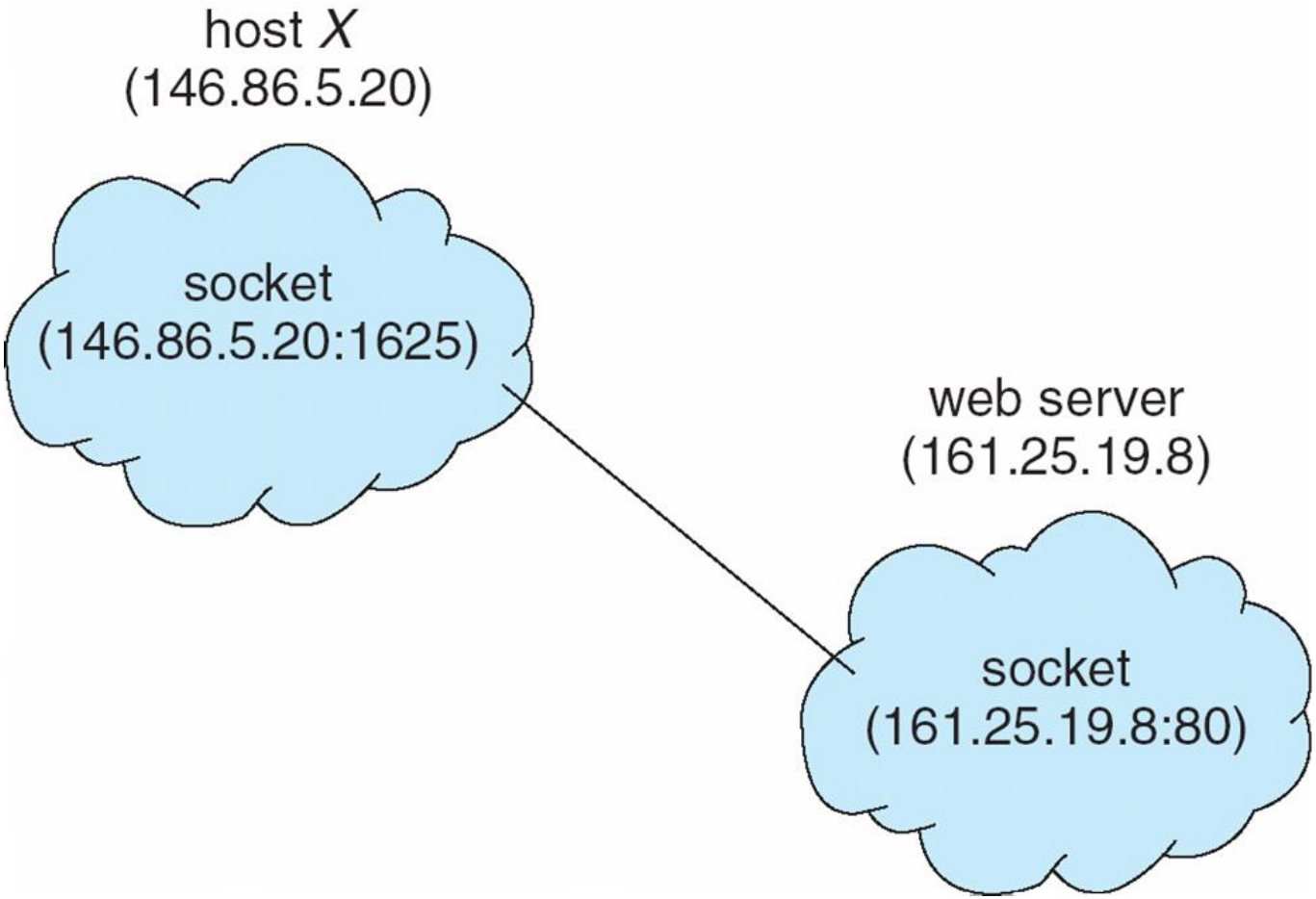
  - When done a process can detach the shared memory from its address space

```
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 (e.g., so called Unix domain sockets)
- 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

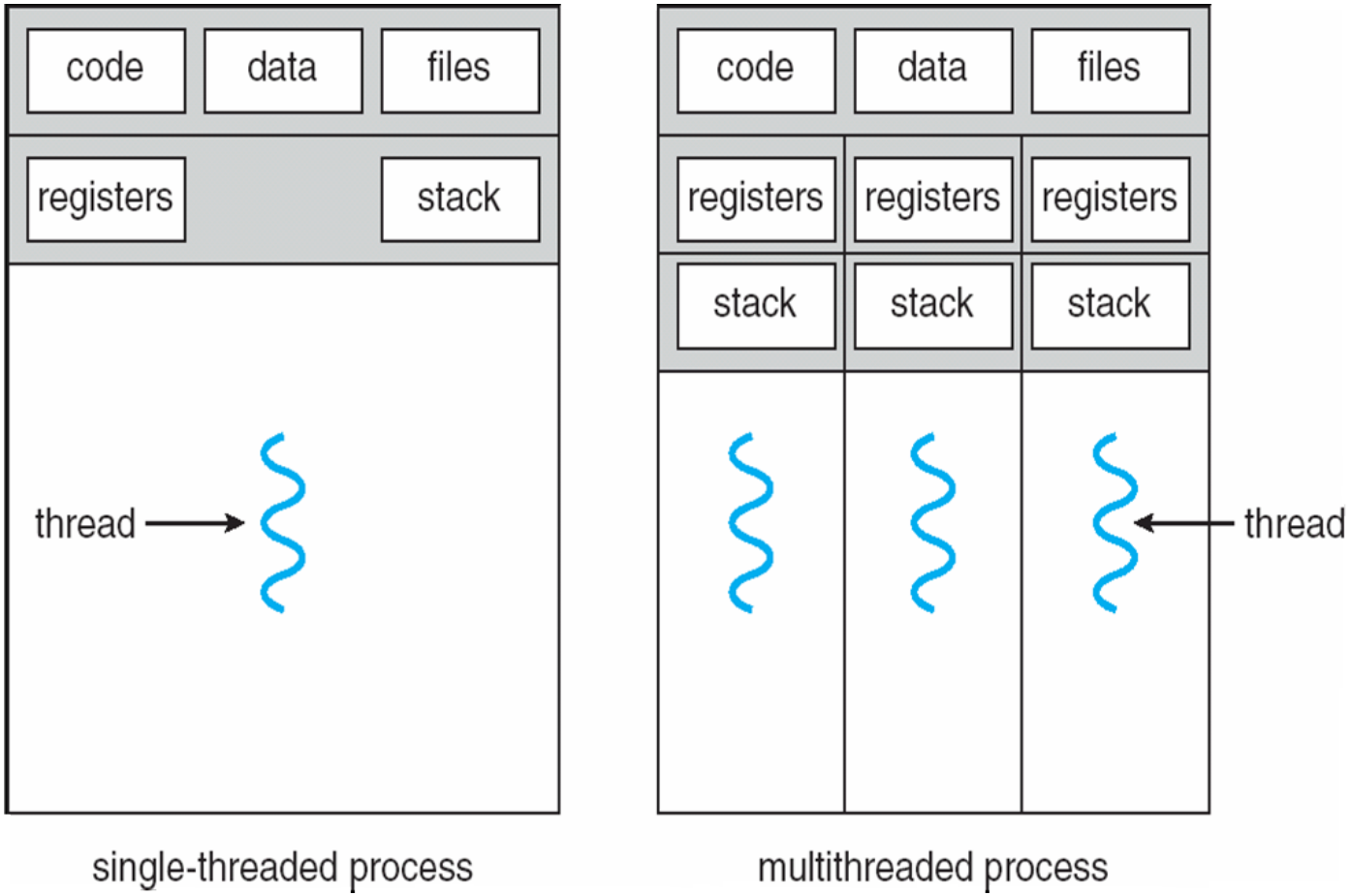
Socket Communication



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* (thread = line of execution, has registers + stack)
  - Many threads can run within a process, share the process's address space
    - No protection between them
    - But "IPC" (or inter-thread communication) is simple (e.g., no need for shmget, shmat, etc) and fast (much less to save/restore at context switch)

Single and Multithreaded Processes



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

NB.
i.Runnable is an abstract interface that must be implemented

ii.An object that implements **Runnable** can run as a separate thread

iii.The thread's execution starts at **run()** method

```
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

NB:
i.Summation, which implements **Runnable**, is passed as argument to **Thread** constructor as object to run in new thread **thrd**

ii.Calling thread uses **start**(), **join**() methods to respectively start **thrd** and wait for it to finish

```
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>");
   }
}
```

Pthreads (in C)

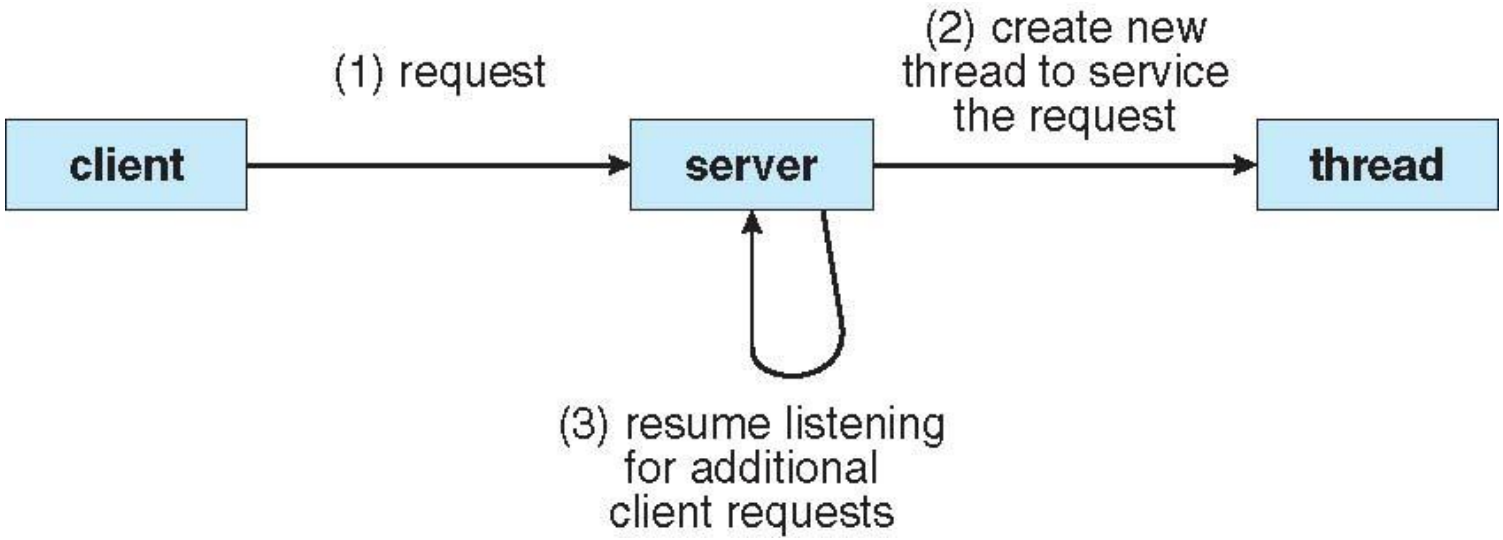
- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to developers of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS/X)
- Example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

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN
Established in collaboration with MIT

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



Server's main thread creates *new* thread to handle request by client. Main thread can quickly go back to accepting new requests (instead of waiting for previous request to finish, which may take long). This makes server *responsive* to (possibly many) clients.

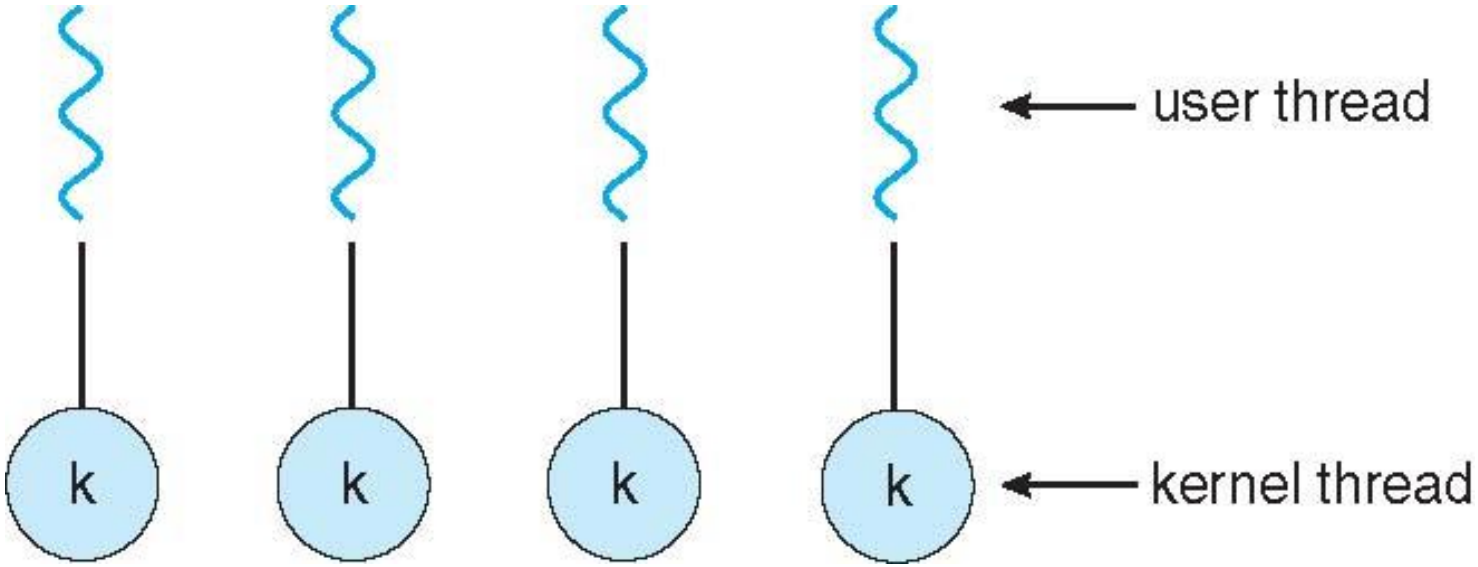
Two types of threads

- *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

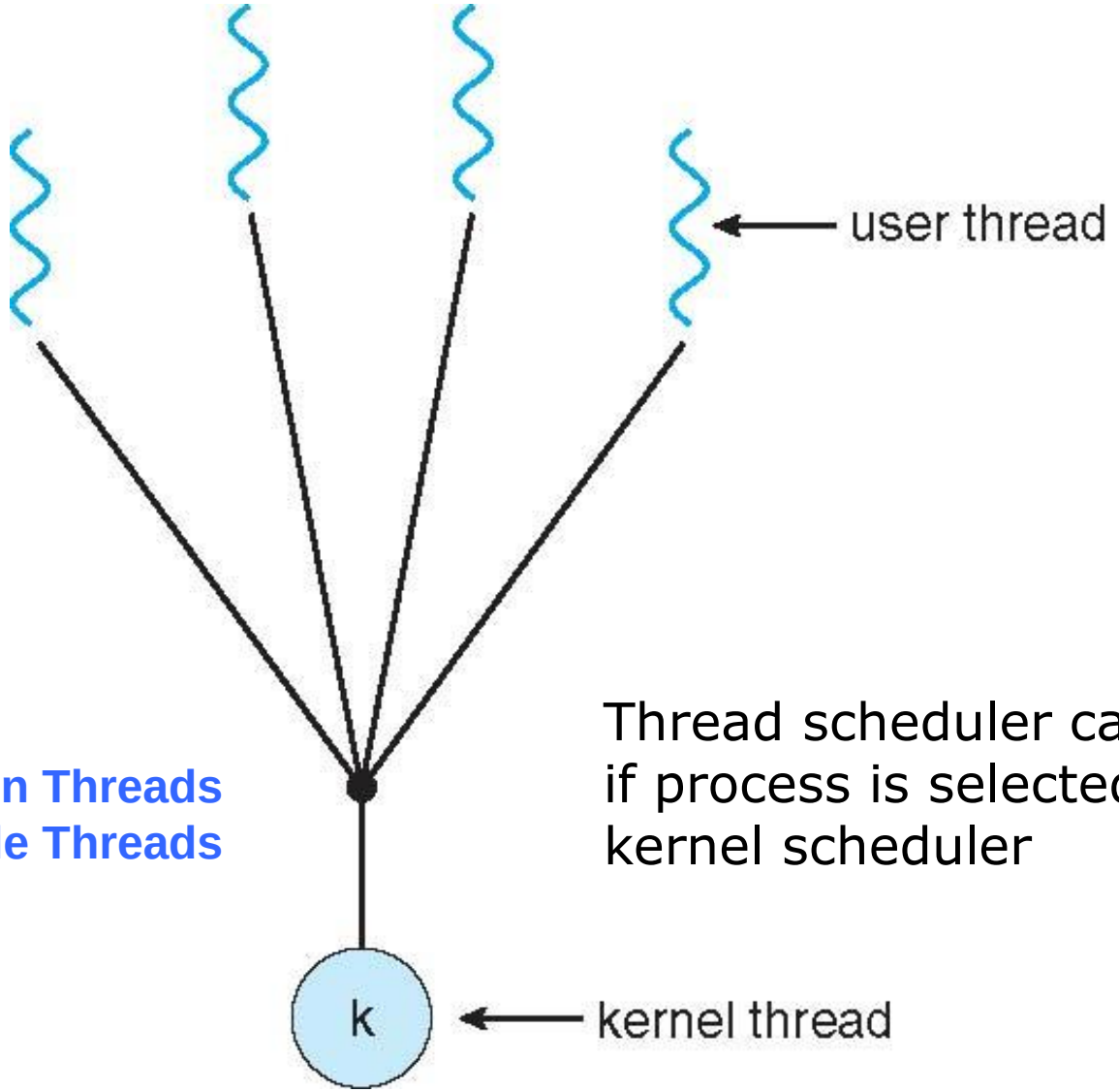
One-to-one Model



Examples

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

Many-to-One Model



Examples:

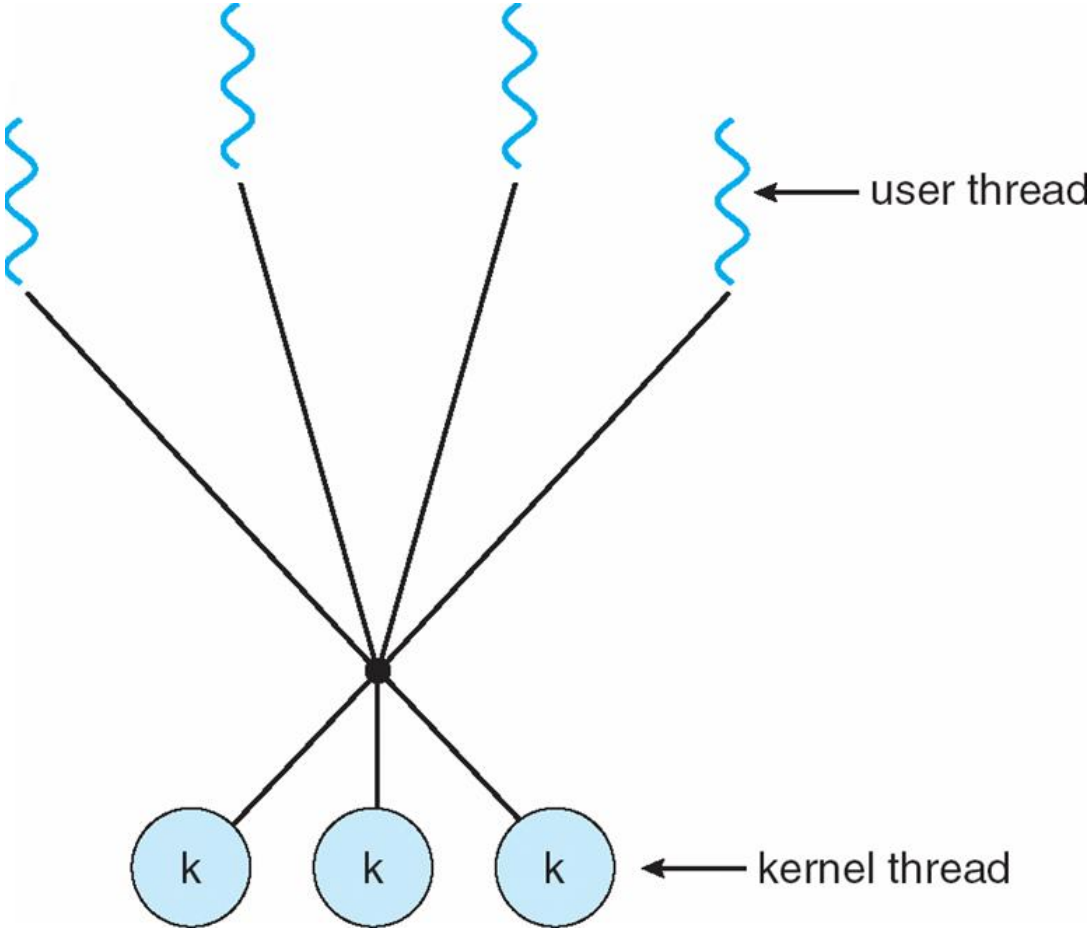
- **Solaris Green Threads**
- **GNU Portable Threads**

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

Many-to-Many Model

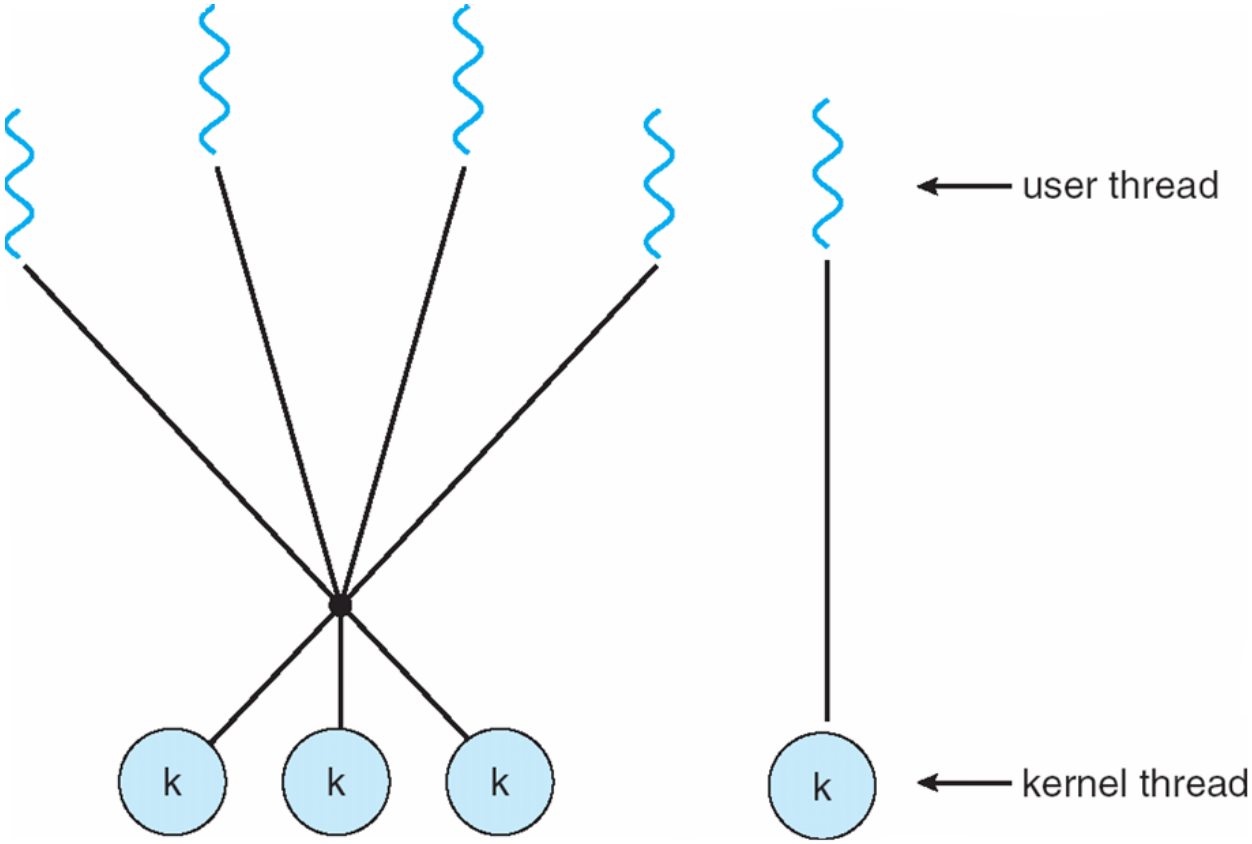
Examples

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



Hybrid 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 (assume many-to-one model)?
- Why is it generally faster to context switch user threads than kernel threads?