Chapter 3: Processes

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-Process Communication (IPC)
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast *inter-process communication* using shared memory and message passing.
- Design *programs that uses pipes and POSIX shared memory* to perform inter-process communication.
- Describe *client-server communication* using sockets and remote procedure calls.
- Design *kernel modules* that interact with the Linux operating system.

Process Concept

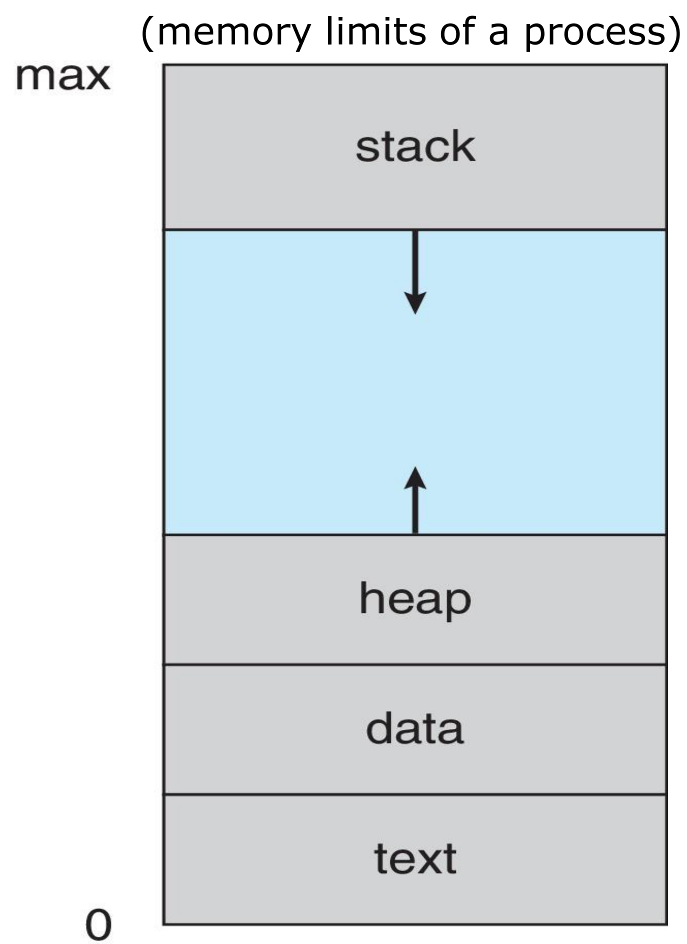
- An operating system executes a variety of programs that run as processes
- ***Process*** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The *program code*, also called ***text*** *section*
  - Current activity including *program counter*, and *processor registers*
  - ***Stack*** *section* containing temporary data
    - Function parameters, return addresses, local variables
  - ***Data*** *section* containing global variables
  - ***Heap*** *section* containing memory dynamically allocated during run time

Process Concept (Cont.)

- *Program* is *passive* entity stored on disk (e.g., *executable file*)
- *Process* is *active* entity
  - Program becomes process when executable file loaded into memory
- *Execution of program* can be started via GUI mouse clicks, command line (CLI) entry of its name, etc.
- One program can be several processes
  - E.g., Consider multiple users executing the same program

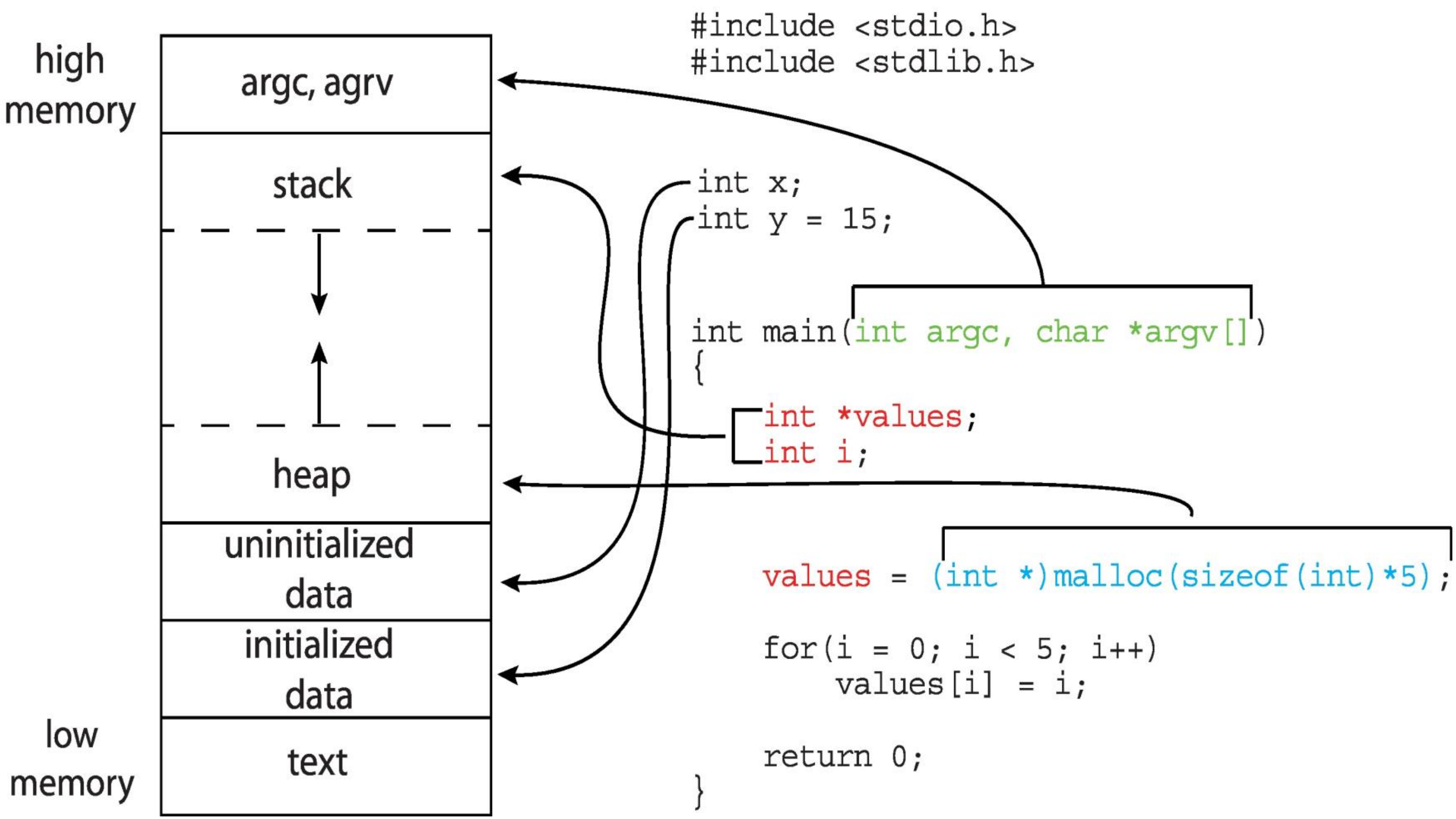
```
#ps -aux
```

Process in Memory



#size <pid>

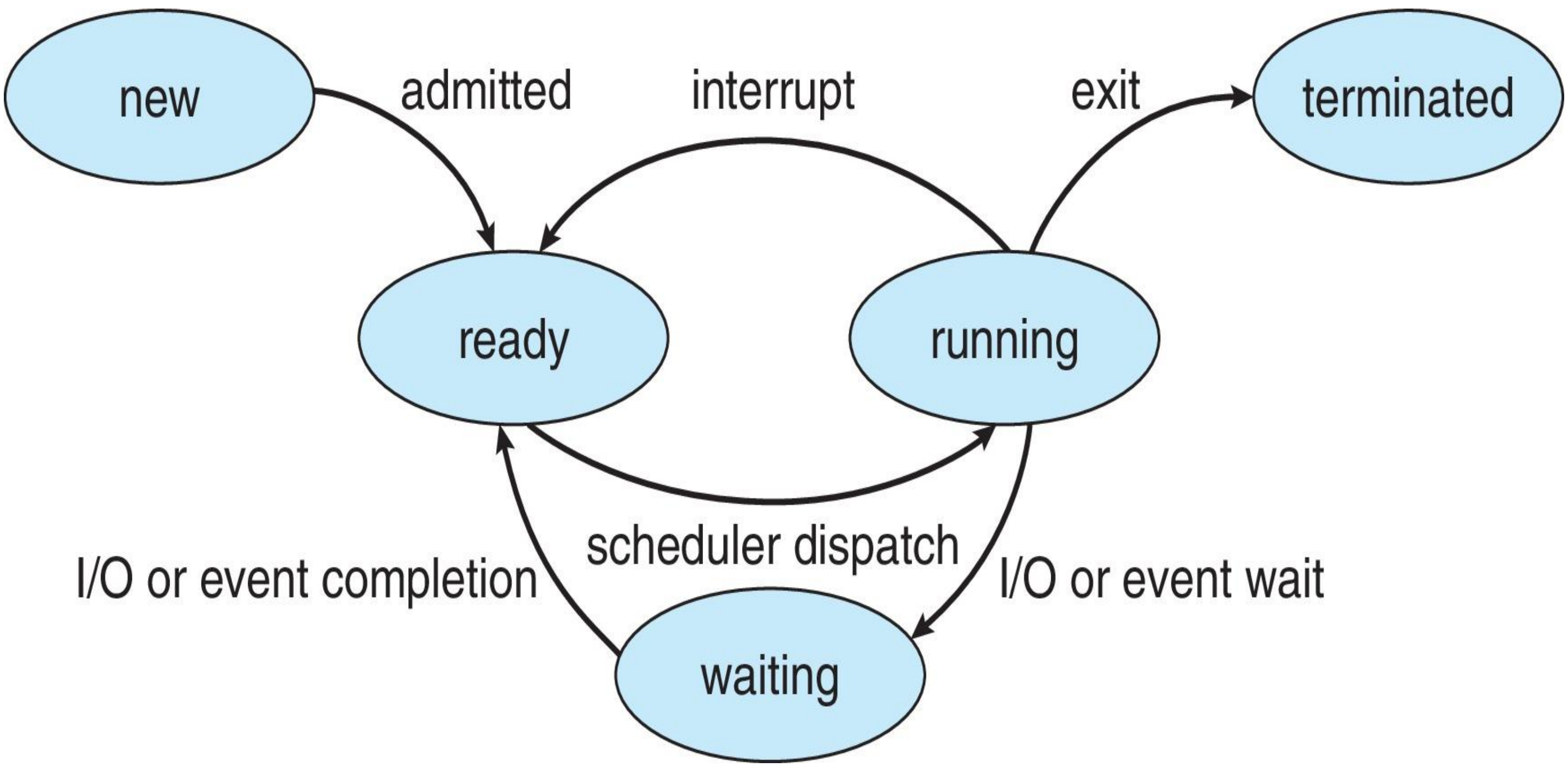
Memory Layout of a C Program



Process State

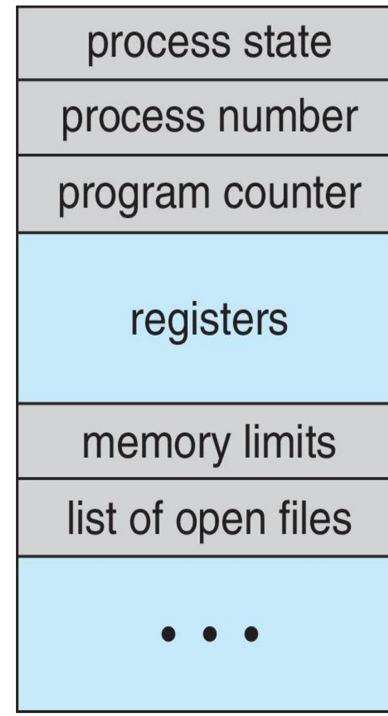
- As a process executes, it changes *state*
  - *New* – The process is being created
  - *Running* – Instructions are being executed
  - *Waiting* – The process is waiting for some event to occur
  - *Ready* – The process is waiting to be assigned to a processor
  - *Terminated* – The process has finished execution

Diagram of Process State



Process Control Block (PCB)

- **Process Control Block** (**PCB**) – Information associated with each process, also called **Task Control Block** (**TCB**), includes:
  - *Process state* – running, waiting, etc.
  - *Process number* – identity of the process
  - *Program counter* – location of instruction to next execute
  - *CPU registers* – contents of all process-centric registers
  - *CPU scheduling info* – priorities, scheduling queue pointers
  - *Memory-management information* – memory allocated to the process
  - *Accounting information* – CPU used, clock time elapsed since start, time limits
  - *I/O status information* – I/O devices allocated to process, list of open files



Threads

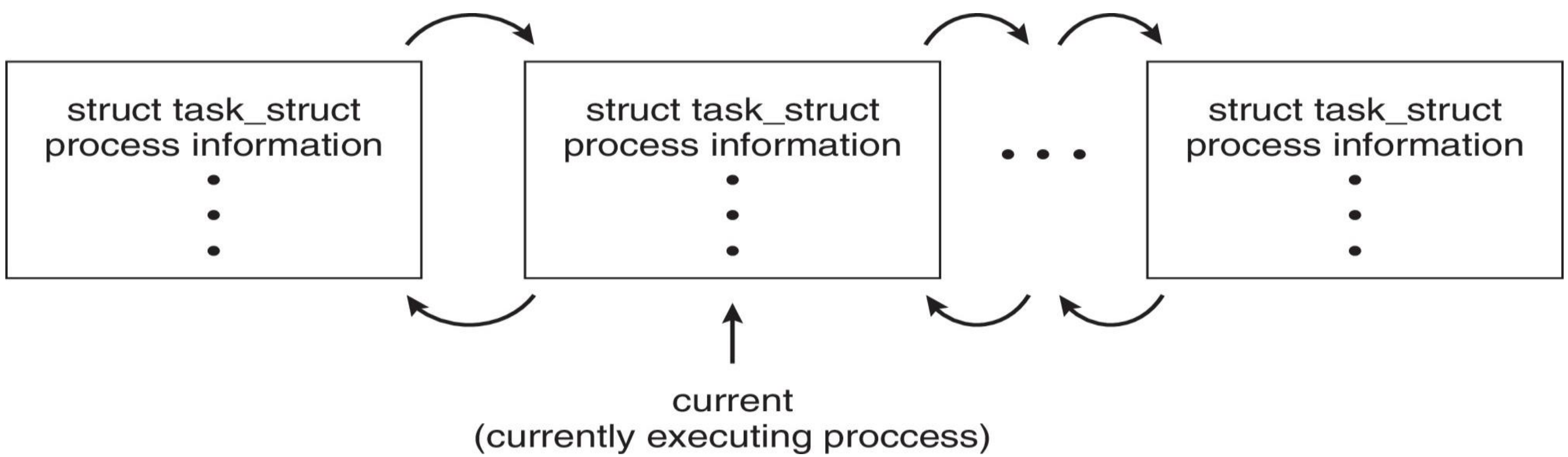
- So far, process has a *single thread* of execution
- Consider having *multiple program counters per process*
  - Multiple locations can execute at once
    - Multiple threads of control –> ***threads***
- Must then have *storage for thread* details
- Multiple program counters in PCB

(*Explore in detail in Chapter 4*)

Process Representation in Linux

- Represented by the C structure **`task_struct`**

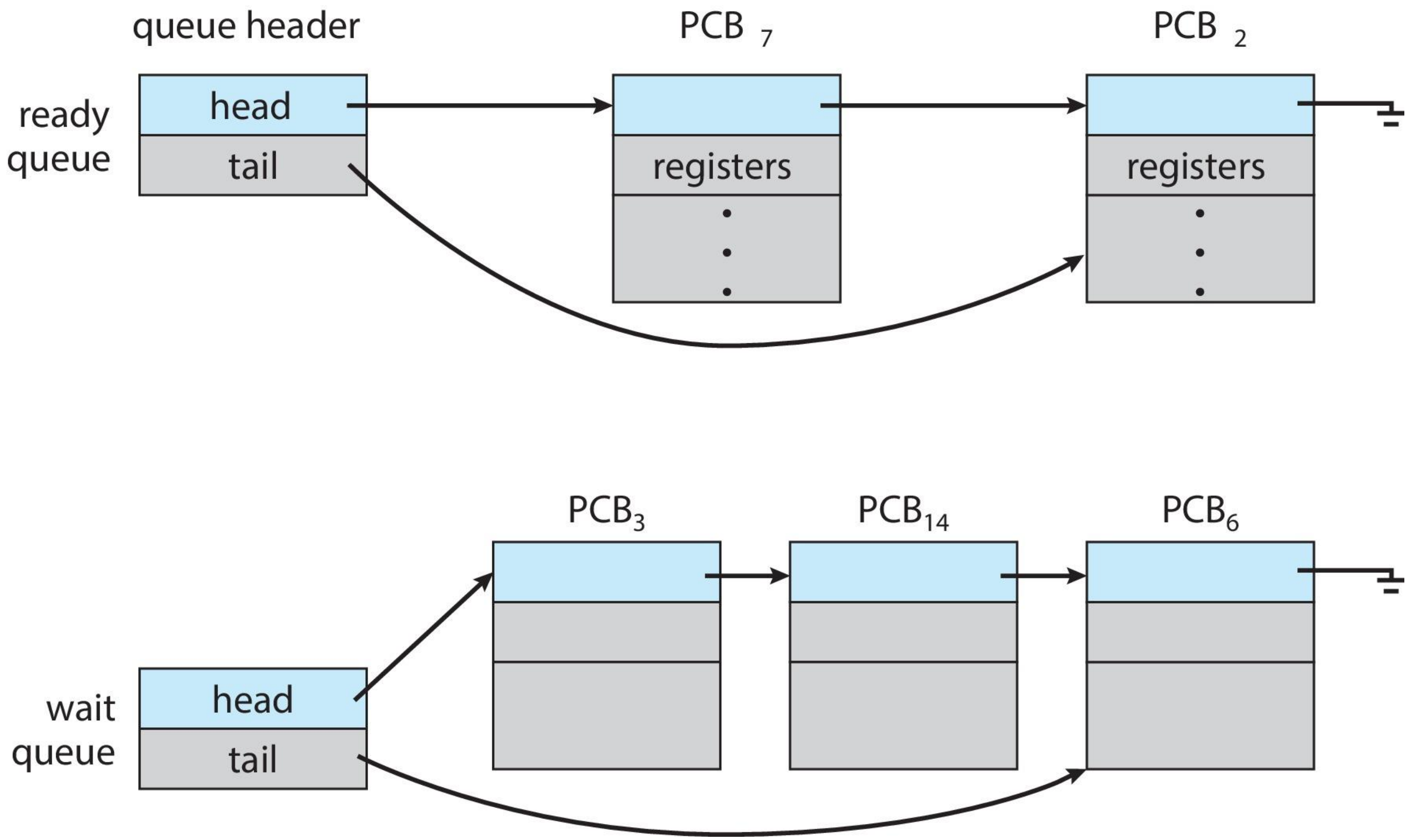
```
pid t_pid;                        /* process identifier */
long state;                       /* state of the process */
unsigned int time_slice           /* scheduling information */
struct task_struct *parent;       /* this process's parent */
struct list_head children;        /* this process's children */
struct files_struct *files;       /* list of open files */
struct mm_struct *mm;       /* address space of this process */
```



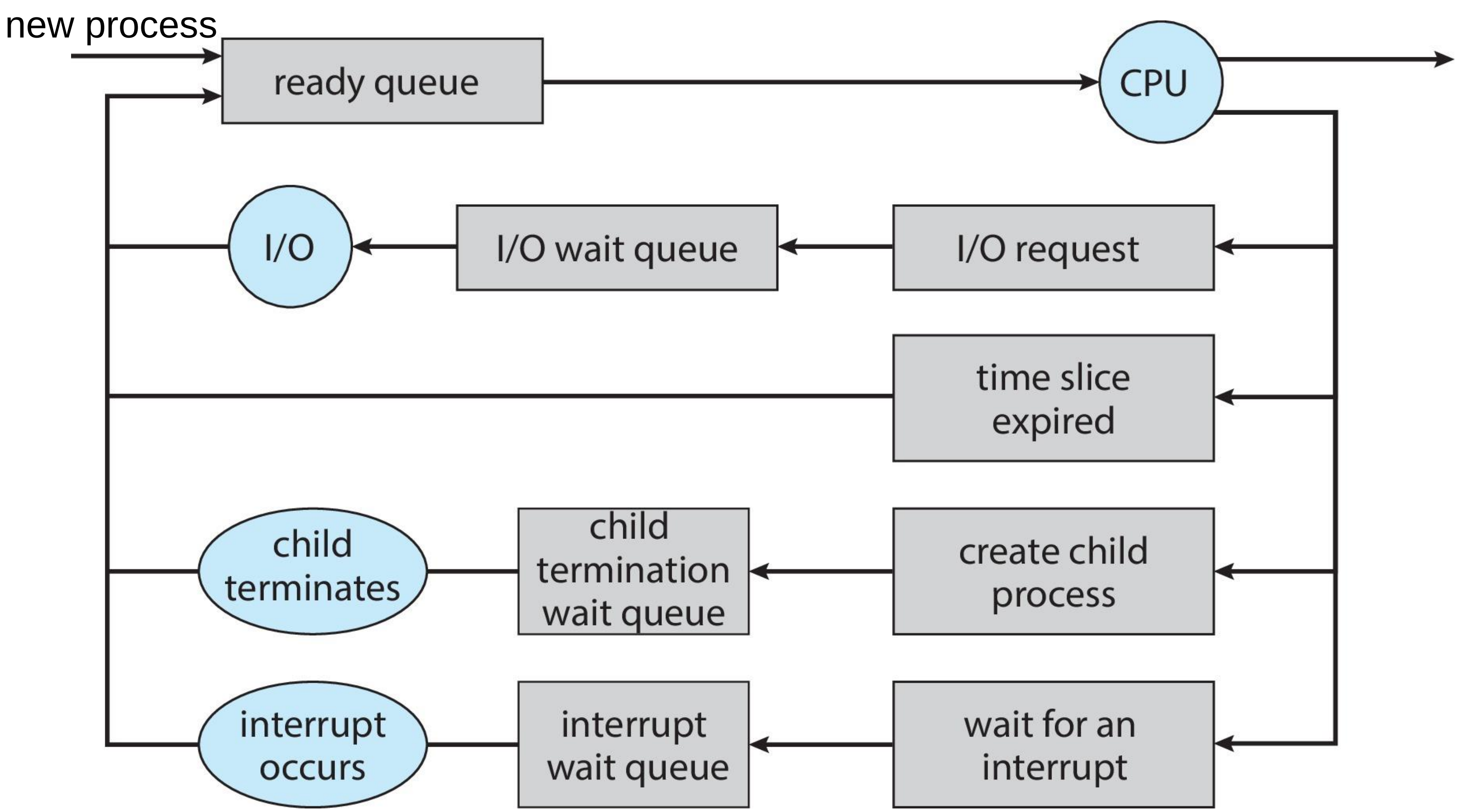
Process Scheduling

- Maximize CPU use ⇢ quickly switch processes onto CPU core
- *Process scheduler* selects one process among available (ready) processes for next execution on CPU core
- Maintains *scheduling queues* of processes
  - *Ready queue* – set of all processes residing in main memory, ready and waiting to execute
  - *Wait queues* – set of processes waiting for an event (e.g., I/O)
- Processes migrate among the various queues

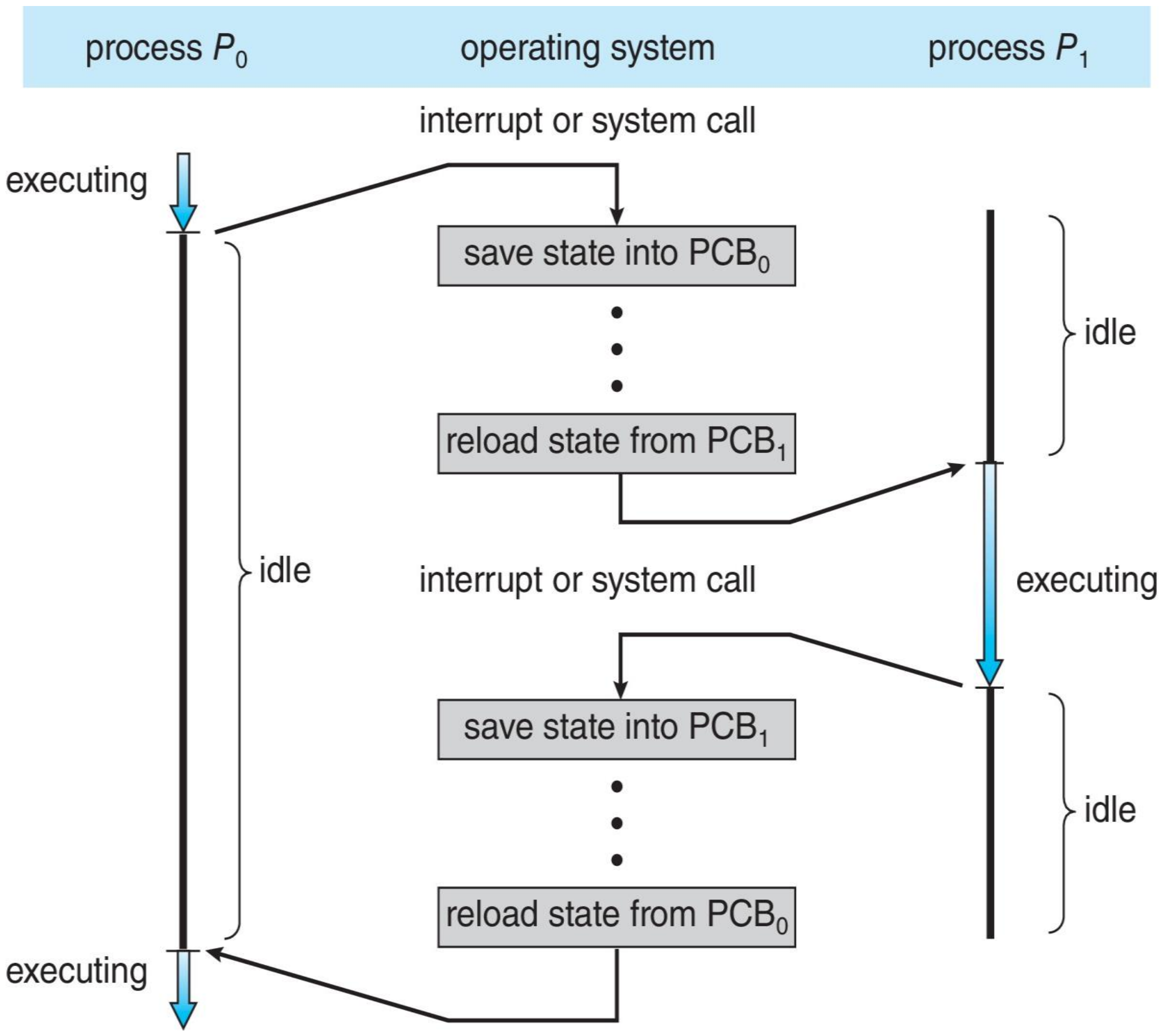
Ready and Wait Queues



Representation of Process Scheduling



CPU Switch from Process to Process



- A *context switch* occurs when the CPU switches from one process to another.

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* of a process represented in the **PCB**
- Context-switch time is *overhead,* the system does no useful work while switching
  - The more complex the OS and the PCB, the longer the context switch
- *Time* dependent on hardware support
  - Some hardware provides *multiple sets of registers per CPU*, multiple contexts loaded at once

Operations on Processes

- System must provide mechanisms for:
  - process creation
  - process termination

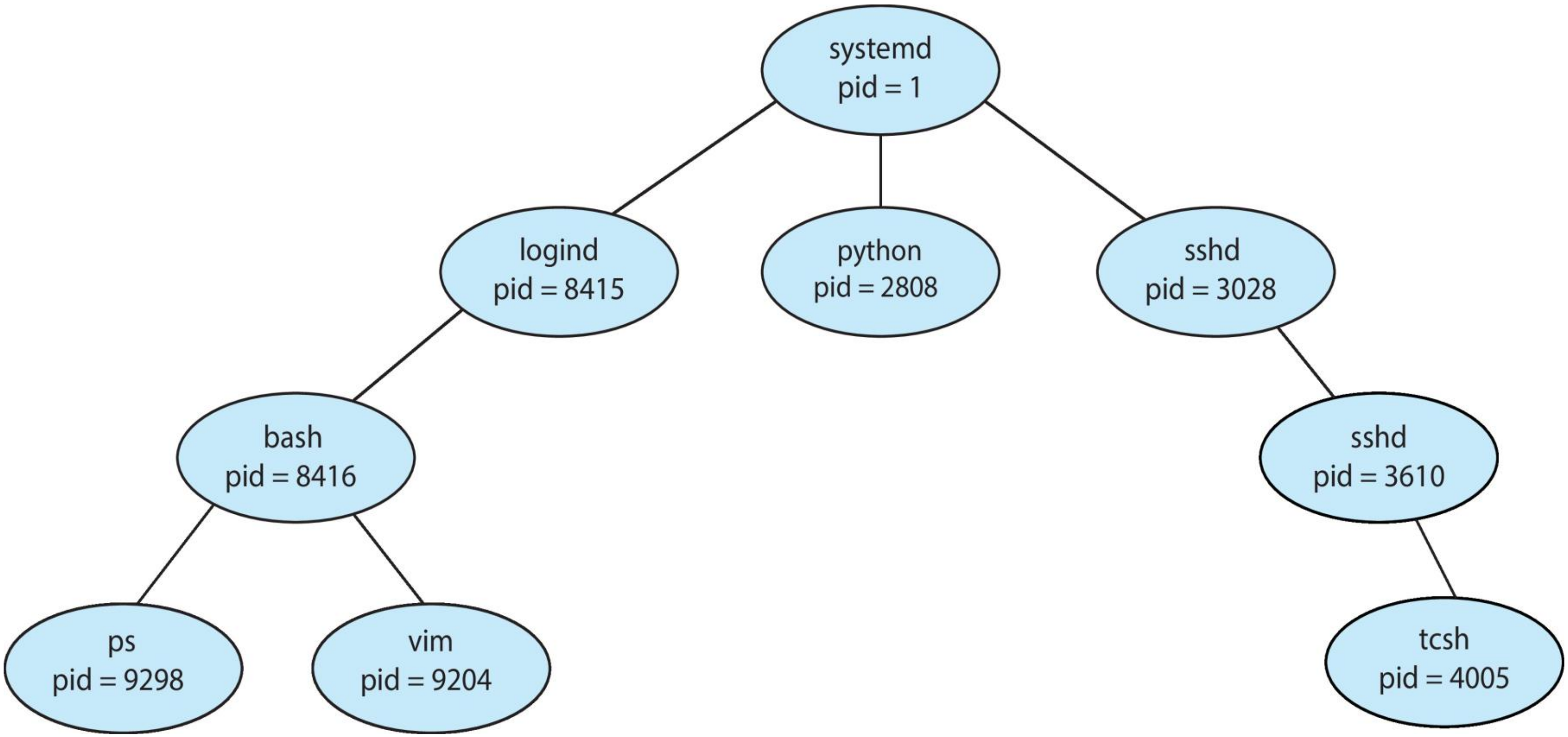
Process Creation

- *Parent processes* create *children processes*, which, in turn create other processes, forming a *tree of processes*
- Process identified and managed via a **Process Identifier** (**PID**)
- **Resource sharing options**
  - Parent and children share *all* resources
  - Children share *subset* of parent's resources
  - Parent and child share *no* resources

Process Creation (Cont.)

- **Execution options**
  - Parent and children execute concurrently
  - Parent waits until children terminate
- **Address space**
  - Child duplicate of parent
  - Child has a program loaded into it

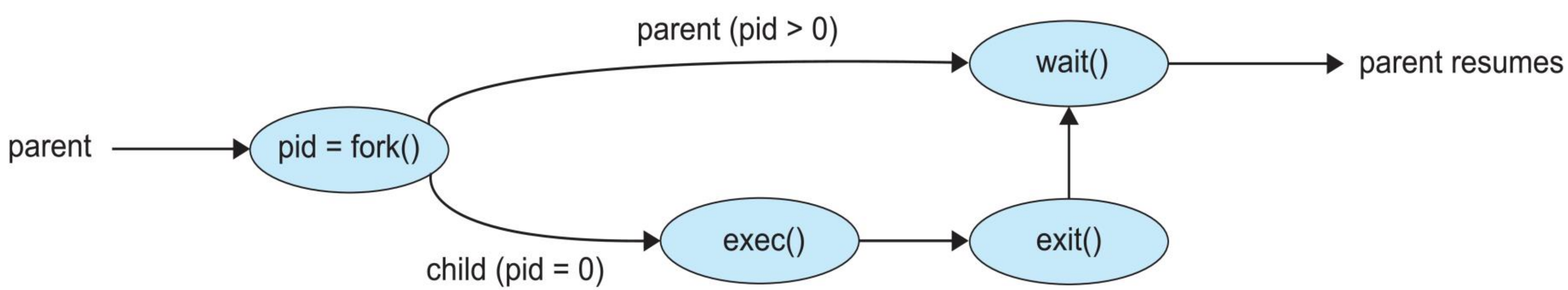
A Tree of Processes in Linux



#pstree

Process Creation (Cont.)

- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - Parent process calls **`wait()`** waiting for the child to terminate



C Program Forking A Separate Process

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 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");
   }

   return 0;
}
```

Fork() example

```
int main(){

        (1): printf("hi \n");
        (2): fork();
        (3): fork();
        (4): printf("hello \n");
        (5): return 0;
}
```

Process Termination

- Process executes *last statement* and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the *operating systems does not allow a child to continue if its parent terminates*

Process Termination (Cont.)

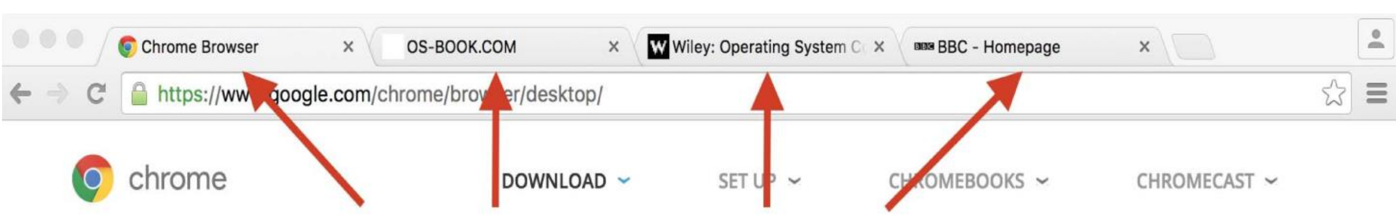
- Some operating systems do not allow child to exists if its parent has terminated. *If a process terminates, then all its children must also be terminated.*
  - **Cascading termination:** All children, grandchildren, etc. are terminated
  - The termination is initiated by the operating system
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the **pid** of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**), process is a *zombie*
- If parent terminated without invoking **`wait()`**, process is an *orphan*

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as a single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - *Browser process* manages user interface, disk and network I/O
  - *Renderer process* renders web pages, deals with HTML, JavaScript. A new renderer created for each website opened
    - Runs in *sandbox* restricting disk and network I/O, minimizing effect of security exploits
  - *Plug-in process* for each type of plug-in



Each tab represents a separate process.

Inter-Process Communication (IPC)

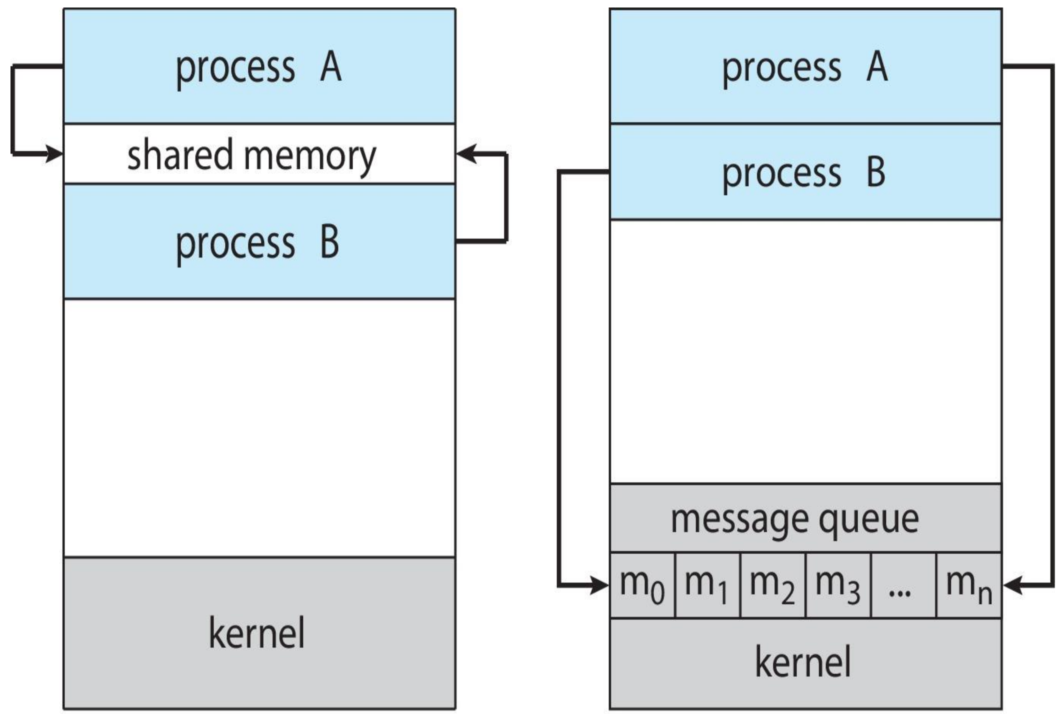
- Processes within a system may be *independent* or *cooperating*
  - *Independent process* does not share data with any other processes executing in the system
  - *Cooperating process* can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Cooperating processes need **Inter-Process communication** (**IPC**)

Communication Models

- Two models of IPC
  - *Shared memory*
  - *Message passing*

Shared memory

Message passing



(a) (b)

Inter-Process Communication – Shared Memory

- An *area of memory shared among the processes* that wish to communicate
- The communication is *under the control of the users processes*, not the operating system.
- Major issues is to provide mechanism that will allow the user processes to *synchronize their actions* when they access shared memory.

(*Synchronization is discussed in great details in Chapters 6 & 7*)

Producer-Consumer Problem

- *Producer-Consumer relationship*
- Paradigm for cooperating processes, *producer process* produces information that is consumed by a *consumer process*
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
 . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Producer Process – Shared Memory

```
item next_produced;

while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
       while (in == out)
              ; /* do nothing */
       next_consumed = buffer[out];
       out = (out + 1) % BUFFER_SIZE;

       /* consume the item in next consumed */
}
```

Inter-Process Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message)
  - receive(message)
- The message size is either fixed or variable

Message Passing (Cont.)

- If processes P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via ***send/receive***
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - 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 unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from *mailboxes* (also referred to as *ports*)
  - Each mailbox has a *unique ID*
  - Processes can communicate *only if they share a mailbox*
- Properties of communication link
  - 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 unidirectional or bi-directional

Indirect Communication (Cont.)

- Operations
  - create a new mailbox (or port)
  - send and receive messages through mailbox
  - destroy a mailbox
- **Primitives** are defined as:
  - **`send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A

Indirect Communication (Cont.)

- **Mailbox sharing**
  - **Example**
    - P_1, P_2, and P_3 share mailbox A,
    - P_1 sends; P_2 and P_3 receive.
    - Who gets the message?
  - **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 arbitrarily the receiver. Sender is notified who the receiver was

Message Passing – Synchronization

- Message passing may be either *blocking* or *non-blocking*
- *Blocking* is considered *synchronous*
  - *Blocking send* – the sender is blocked until the message is received
  - *Blocking receive* – the receiver is blocked until a message is available
- *Non-blocking* is considered *asynchronous*
  - *Non-blocking send* – the sender sends the message and continue
  - *Non-blocking receive* – the receiver receives:
    - A valid message, or Null message
- Different combinations possible
  - If both send and receive are blocking, we have a *rendezvous*

Producer – Message Passing

```
message next_produced;

while (true) {
      /* produce an item in next_produced */

       send(next_produced);
}
```

Consumer – Message Passing

```
message next_consumed;

while (true) {
      receive(next_consumed)

      /* consume the item in next_consumed */
   }
```

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  - *Zero capacity* – no messages are queued on a link
    - Sender must wait for receiver (rendezvous)
  - *Bounded capacity* – finite length of *n* messages
    - Sender must wait if link full
  - *Unbounded capacity* – infinite length
    - Sender never waits

Examples of IPC Systems - POSIX

- **POSIX Shared Memory**
  - Process first creates shared memory segment

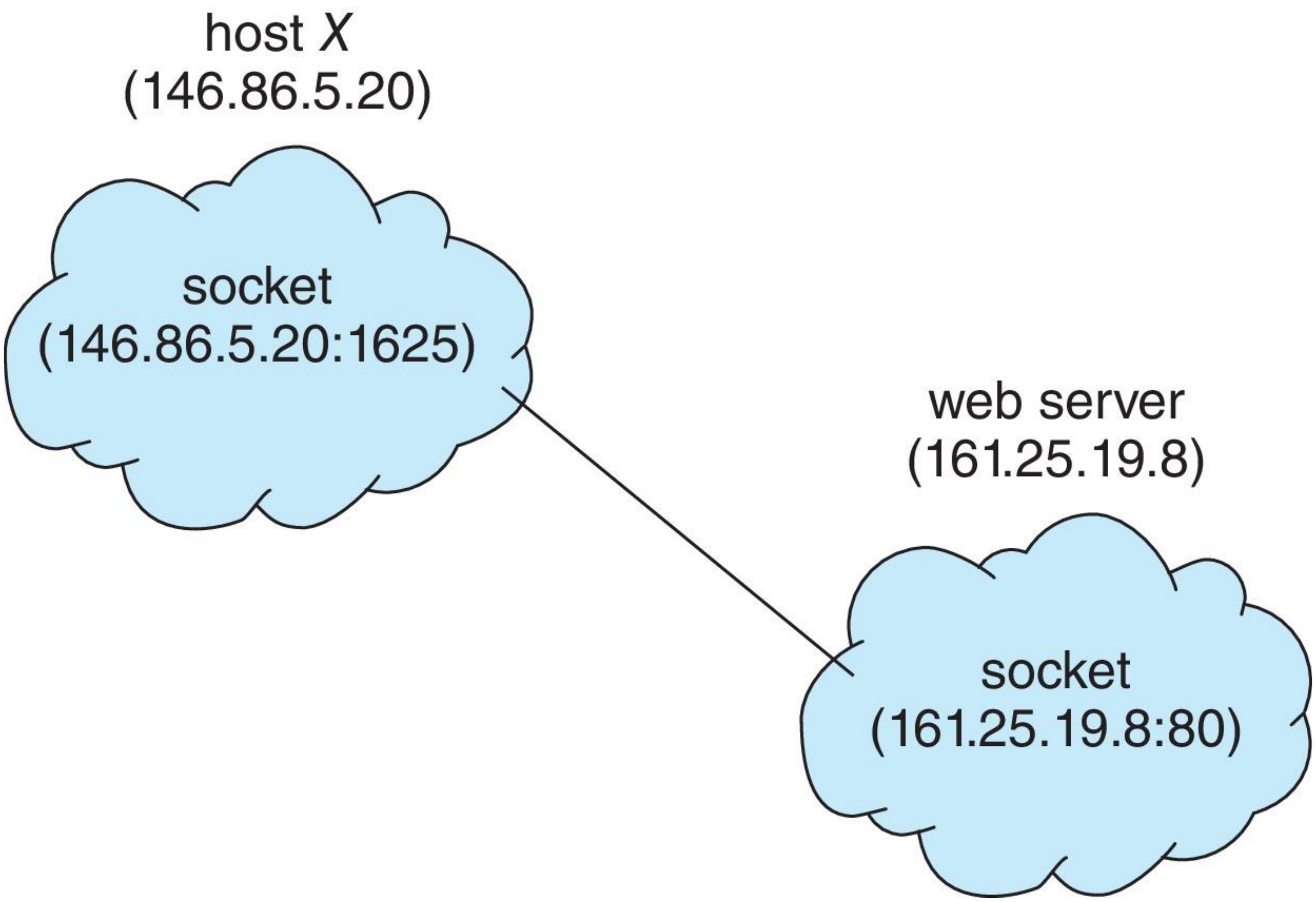
    ```
    shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
    ```

  - Also used to open an existing segment
  - Set the size of the object

    ```
    ftruncate(shm_fd, 4096);
    ```

  - Use **`mmap()`** to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by **`mmap()`**.

Socket Communication



Sockets in Java – Server

```
import java.net.*;
import java.io.*;

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);
      }
   }
}
```

- Three types of sockets
  - *Connection-oriented (TCP)*
  - *Connectionless (UDP)*
  - **`MulticastSocket`** class– data can be sent to multiple recipients
- Consider this “Date” *server* in Java:

Remote Procedure Calls

- **Remote Procedure Call** (**RPC**) abstracts procedure calls between processes on *networked systems*
  - Again uses *ports* for service differentiation
- *Stubs* – proxies for the actual procedure on the server and client sides
  - The *client-side stub* locates the server and *marshals* the parameters
  - The *server-side stub* receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On **Windows**, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Summary

- A ***process*** is a program in execution, and the status of the current activity of a process is represented by the program counter, as well as other registers.
- The ***layout of a process in memory*** is represented by four different sections: (1) text, (2) data, (3) heap, and (4) stack.
- As a process executes, it changes state. There are ***four general states of a process***: (1) ready, (2) running, (3) waiting, and (4) terminated.
- A ***process control block (PCB)*** is the kernel data structure that represents a process in an operating system.
- The role of the ***process scheduler*** is to select an available process to run on a CPU.

Summary (Cont.)

- An operating system performs a ***context switch*** when it switches from running one process to running another.
- The ***`fork()`*** and ***`CreateProcess()`*** system calls are used to create processes on UNIX and Windows systems, respectively.
- When ***shared memory*** is used for communication between processes, two (or more) processes share the same region of memory. POSIX provides an API for shared memory.
- Two processes may communicate by exchanging messages with one another using ***message passing***. The Mach operating system uses message passing as its primary form of inter-process communication. Windows provides a form of message passing as well.

Summary (Cont.)

- A ***pipe*** provides a conduit for two processes to communicate. There are two forms of pipes, ordinary and named. Ordinary pipes are designed for communication between processes that have a parent-child relationship. Named pipes are more general and allow several processes to communicate.
- **UNIX** systems provide ordinary pipes through the `pipe()` system call. Ordinary pipes have a read end and a write end. A parent process can, for example, send data to the pipe using its write end, and the child process can read it from its read end. Named pipes in UNIX are termed FIFOs.

Summary (Cont.)

- **Windows** systems also provide *two forms of pipes*—anonymous and named pipes. Anonymous pipes are similar to UNIX ordinary pipes. They are unidirectional and employ parent-child relationships between the communicating processes. Named pipes offer a richer form of inter-process communication than the UNIX counterpart, FIFOs.
- Two common forms of ***client-server communication*** are *sockets* and *remote procedure calls* (**RPCs**). Sockets allow two processes on different machines to communicate over a network. RPCs abstract the concept of function (procedure) calls in such a way that a function can be invoked on another process that may reside on a separate computer.
- The **Android** operating system uses RPCs as a form of inter-process communication using its *binder framework*.

End of Chapter 3