Chapter 3: Processes

Chapter 3: Processes

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 interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

Notion of a **process**

Contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently.

A system therefore consists of a collection of processes, some executing user code, others executing operating system code.

All these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them.

1. Process Concept

Program vs Process

A program by itself is not a process.

Process – a program in execution; process execution must progress in sequential fashion

- **A program** is a ***passive*** entity, such as a file containing a list of instructions stored on disk (often called an **executable file**).
- In contrast, **a process** is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources

- The status of the current activity is represented by the value of the program counter and the contents of the processor's registers.

A program becomes a process when an executable file is loaded into memory. Common techniques for loading executable files:

  - double-clicking an icon representing the executable file
  - entering the name of the executable file on the command line (as in prog.exe or a.out).

Separate execution sequences

Although two processes may be associated with the same program, they
are nevertheless considered two separate execution sequences.

For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program.

Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

JVM example

- A process itself can be an execution environment for other code.
  - The Java programming environment provides a good example. - executable Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code and takes actions (via native machine instructions) on behalf of that code.
  - For example, to run the compiled Java program Program.class, we would enter

    ```
    java Program
    ```

    - The command java runs the JVM as an ordinary process, which in turns executes the Java program in the virtual machine.
- The concept is the same as simulation, except that the code, instead of being written for a different instruction set, is written in the Java language.

Layout of Process in Memory

Memory layout of a process divided into multiple parts:

1. **Text section -** The executable code
2. **Data section** **-** Global variables
3. **Heap** **-** Memory that is dynamically allocated during program run time
4. **Stack section** **-** Temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

Process in Memory

Stack and Heap sections can shrink and grow dynamically during program execution.

Stack:

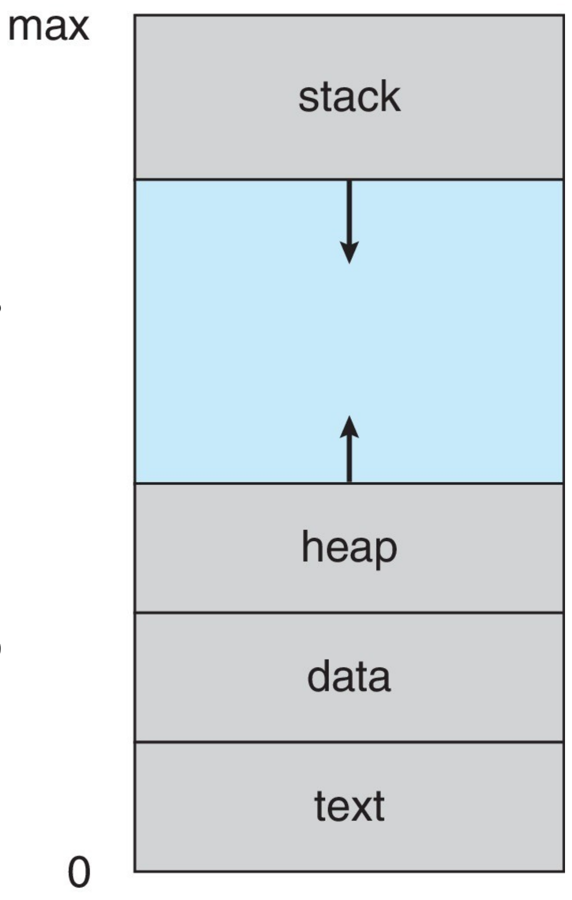
- Each time a function is called, an activation record containing function parameters, local variables, and return address is pushed on a Stack; when control is returned from the function, the activation record is popped from the Stack.

Heap:

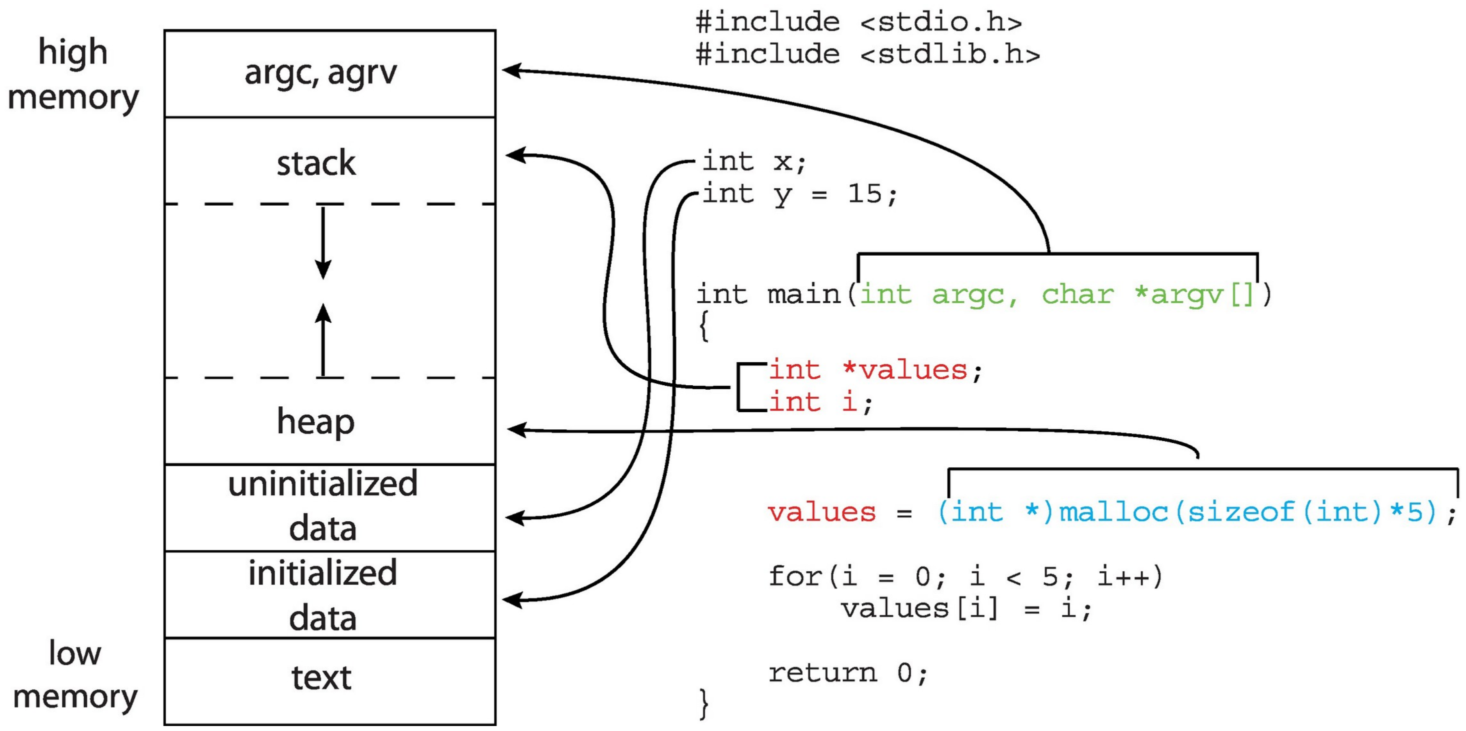
- Similarly, Heap will grow a memory is dynamically allocated and will shrink when memory is returned to the system.

Text, Data:

Sizes of Text and Data sections are fixed, as their size do not change during program run time.



Memory Layout of a C Program



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
   int *values;
   int i;

   values = (int *)malloc(sizeof(int)*5);

   for(i = 0; i < 5; i++)
       values[i] = i;

   return 0;
}
```

Figure shows how different sections of a process relate to actual C program.

- A separate section is provided for the argc and argv parameters passed to the main() function
- Global data section is divided into initialized and uninitialized

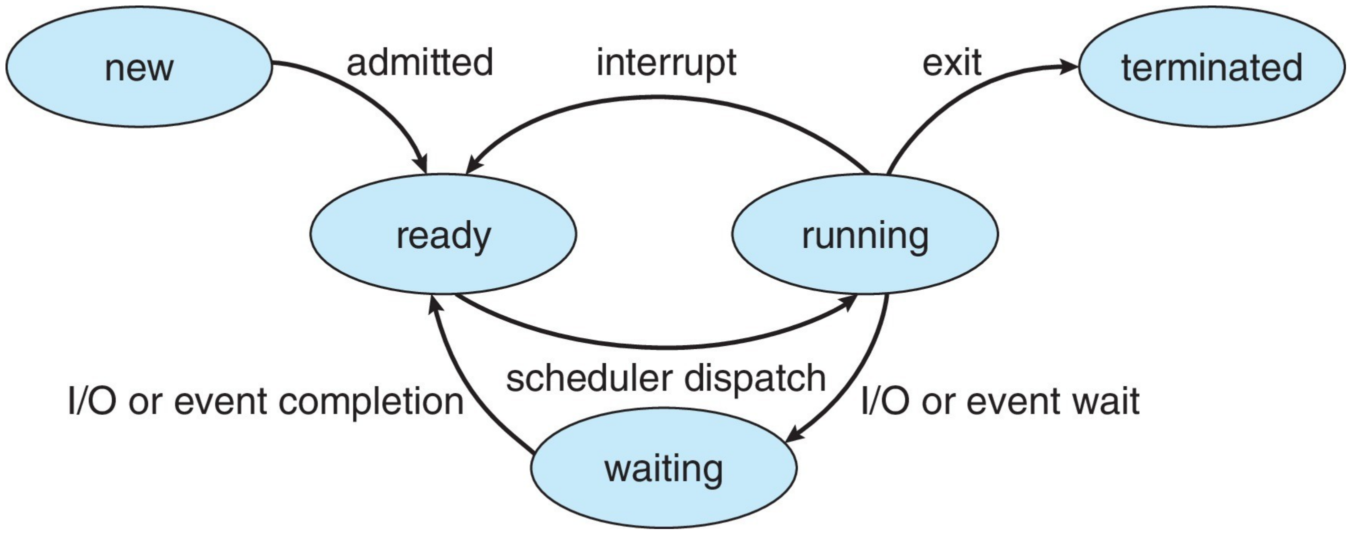
GNU Size command

The GNU size command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is memory, the following is the output generated by entering the command size memory:

```
text    data    bss     dec     hex     filename
1158    284     8       1450    5aa     memory
```

The data field refers to uninitialized data, and bss refers to initialized data.(bss is a historical term referring to block started by symbol.) The dec and hex values are the sum of the three sections represented in decimal and hexadecimal, respectively.

Process State Diagram



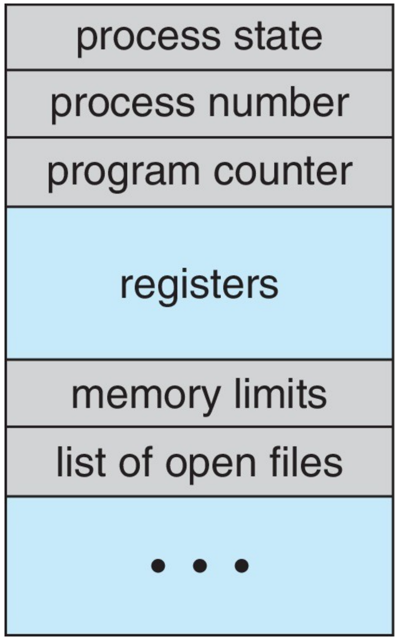
- As a process executes, it changes **state.** The state of a process is defined in part by the current activity of that process.
  - **New**: The process is being created
  - **Ready**: The process is waiting to be assigned to a processor
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
  - **Terminated**: The process has finished execution

Process Control Block (PCB)

Also called **task control block**

Each process is represented in the OS with its **own PCB** containing many pieces of information associated with it such as:

- Process state – ready, running, waiting, etc.
- Program counter – address of the next instruction to be executed
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process, base and limit registers, page tables or segment tables
- 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



2. Process Scheduling

Process Scheduling

Process scheduler selects among available processes for next execution on CPU core

1. **Objective of multiprogramming** – have some process running at all times to

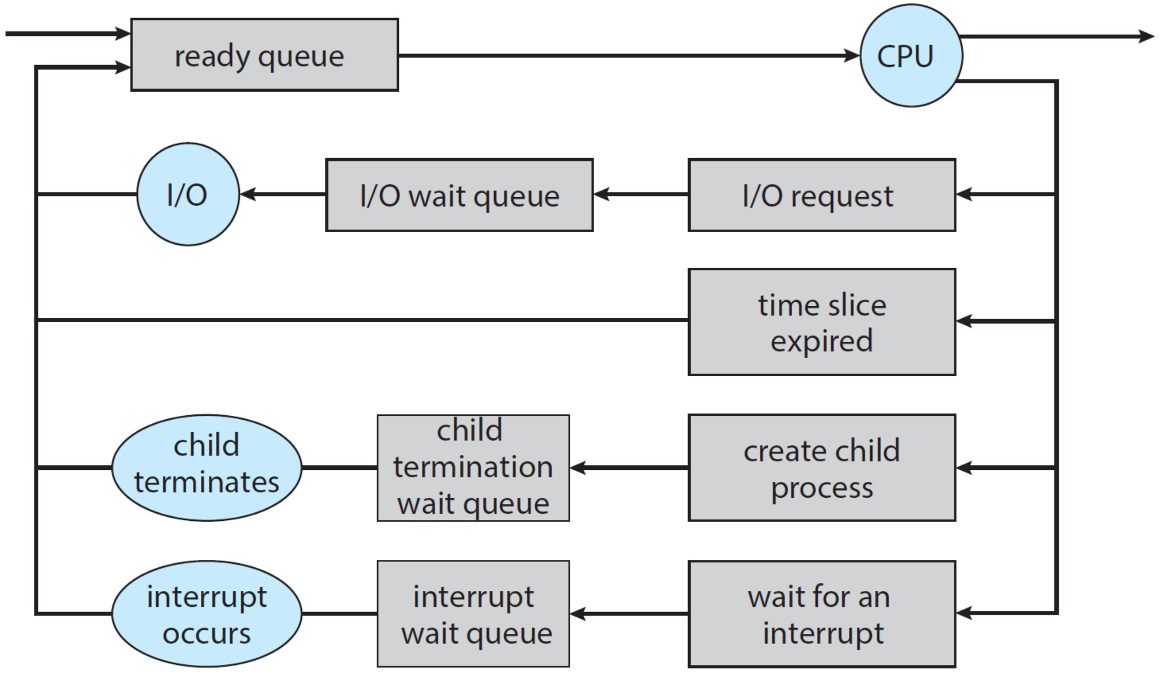
   Maximize CPU utilization

2. **Objective of time sharing** is to **switch a CPU core among processes** so frequently that users can interact with each program while it is running

- For a system with a single CPU core, there will never be more than one process running at a time
- A multicore system can run multiple processes at one time. If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled.
- The number of processes currently in memory is known as the **degree of multiprogramming**.

The Ready Queue and a set of Wait Queues

As processes enter the system, they are put into a **ready queue**, where they are ready and waiting to execute on a CPU's core.

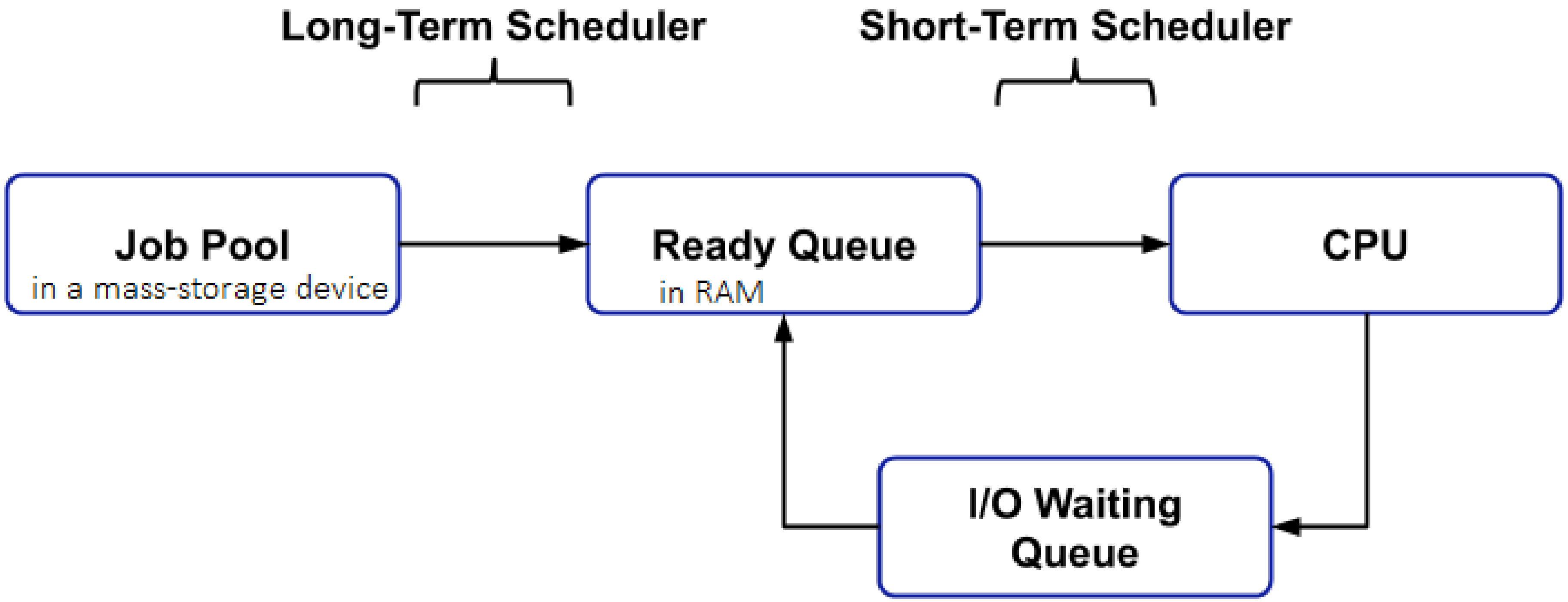


- The process could issue an I/O request and then be placed in an I/O **wait queue**
- The process could create a new child process and then be placed in a wait
queue while it awaits the child's termination.
- The process could be removed forcibly from the core, as a result of

Long-term, Short-term Scheduling

Executable files reside permanently in the file system in mass-storage. Long-term scheduler selects processes from the JOB pool to the Ready state for its execution

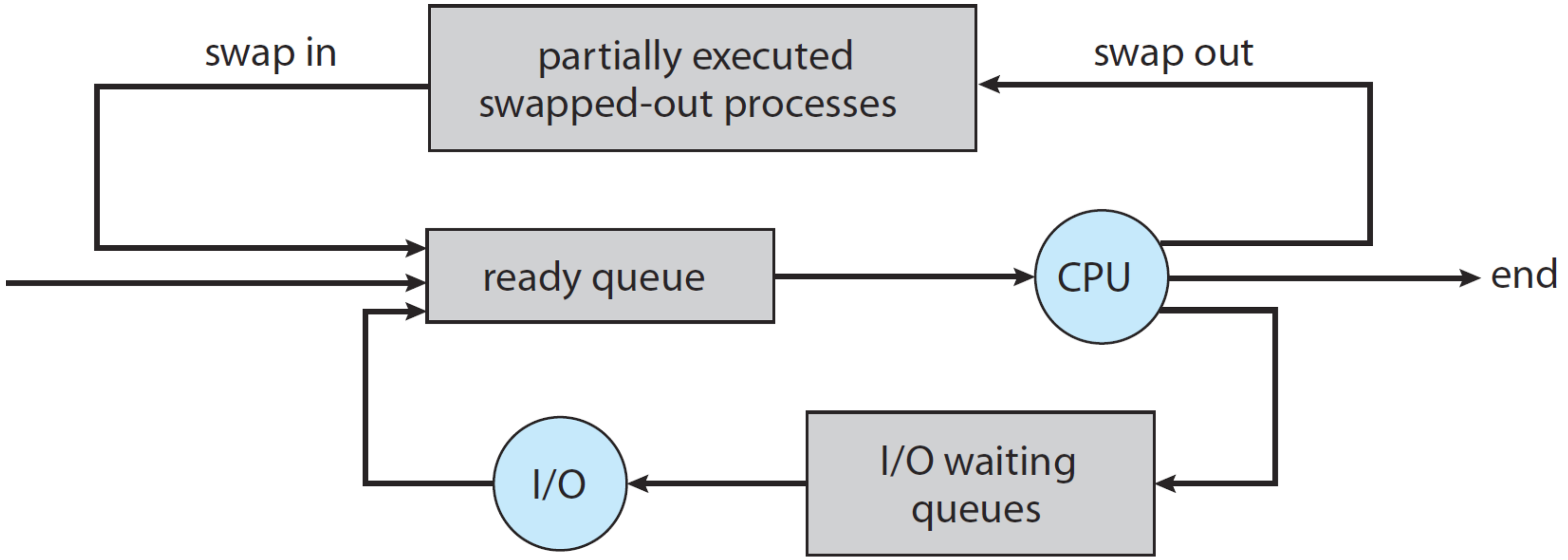
Short-Term Scheduler is also known as **CPU scheduler** and is responsible for selecting one process from the ready state for scheduling it on the running state using a scheduling algorithms such as FCFS, Round-Robin, or SJF



Medium-term Scheduling (Swapping)

Sometimes, OS needs to send a running process to the ready state or to the wait/block state. For example, in the round-robin process, after a fixed time quantum, the process is again sent to the ready state from the running state.

Sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.

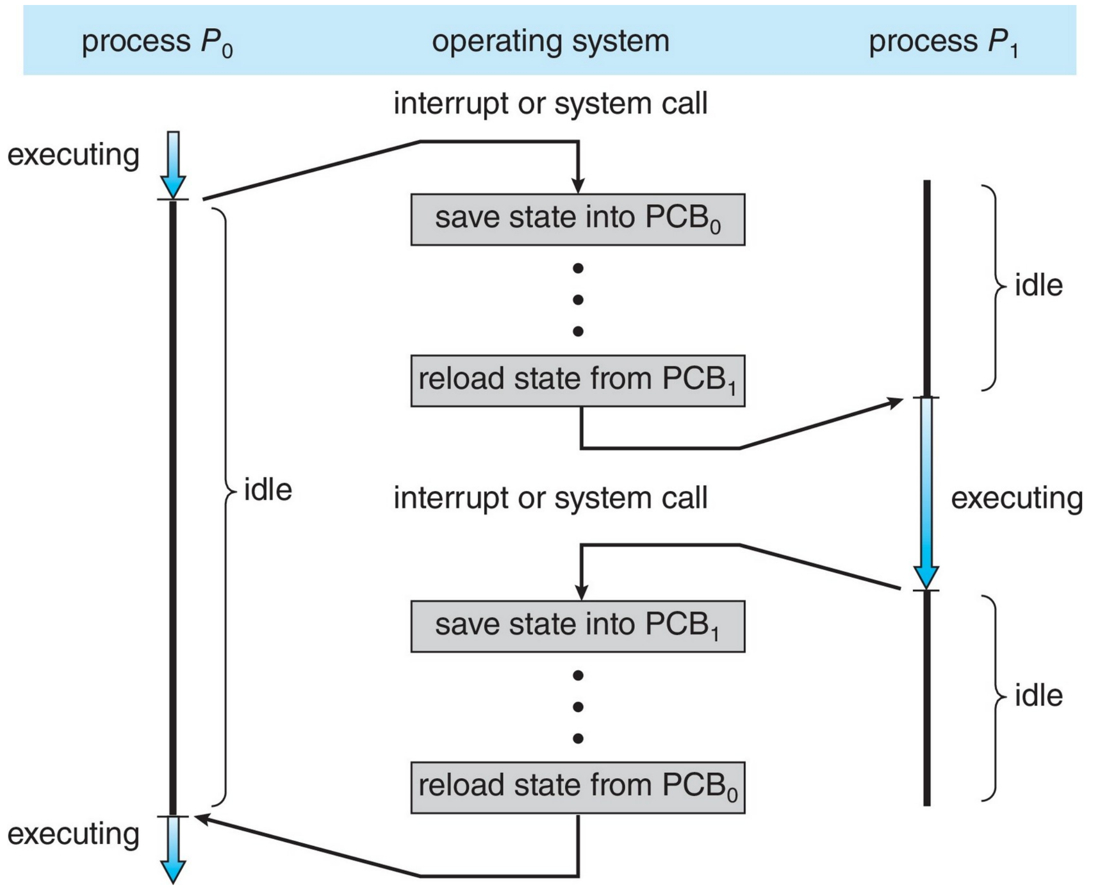


Context Switch

- Interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine.
- When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state, and memory-management information
- Generically, we perform a **state save** of the current state of the CPU core, be it in kernel or user mode, and then a **state restore** to resume operations.
- Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process.
- This task is known as a **context switch**

CPU Switch From Process to Process

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



Context Switch - an overhead

- 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

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits **iOS** provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and **with limits**
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- **Android** runs **foreground and background**, with **fewer limits**
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

3. Operations on Processes

Process Creation

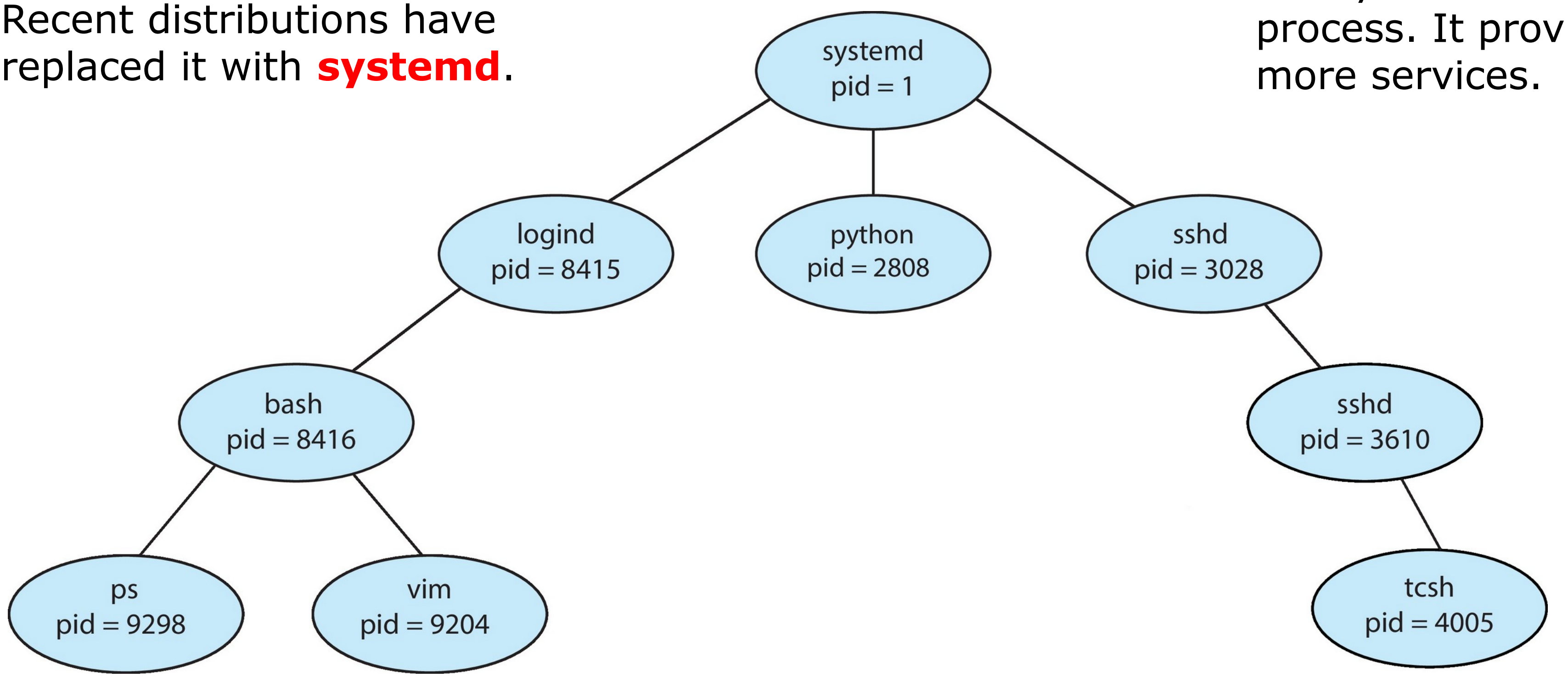
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options:
  1. Parent and children share all resources
  2. Children share subset of parent's resources
  3. Parent and child share no resources
- Execution options
  a. Parent and children execute concurrently
  b. Parent waits until children terminate

A Tree of Processes in Linux

In Unix-based computer operating systems, **init** (short for *initialization*) is the first process started during booting of the computer system. **init** is a **daemon process** that continues running until the system is shut down. It is the direct or indirect ancestor of all other processes and automatically adopts all orphaned processes. Init is started by the kernel during the booting process; Init is typically assigned **p**rocess **id**entifier **1**.

Linux initially adapted **init** Recent distributions have replaced it with **systemd**.

systemd serves as the system's initial process. It provides more services.



Process creation

The **systemd** process (which always has a pid of 1) serves as the root parent process for all user processes.

Once the system has booted, the systemd process creates processes which provide additional services such as a web server, ssh server, and the like.

On UNIX and Linux systems, the command

```
ps -elf
```

will list every running process (**-e**) and a full listing (**-f**) of all processes currently active in the system

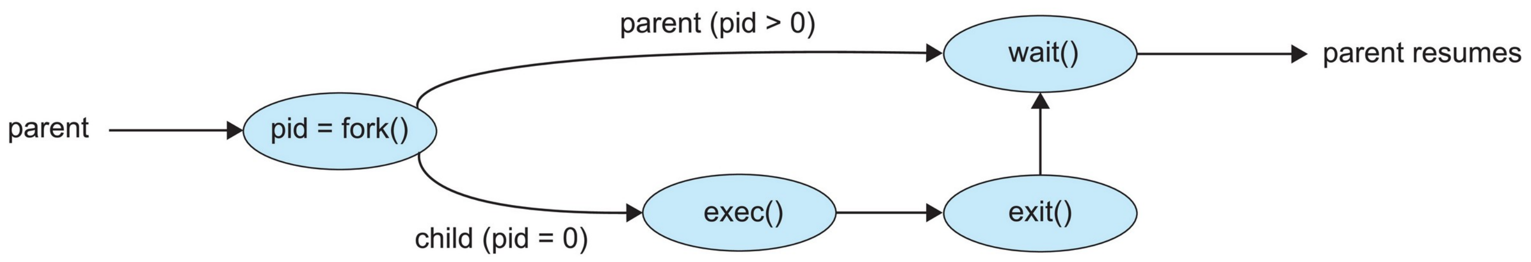
Address Space:
There are also **two address-space possibilities** for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).

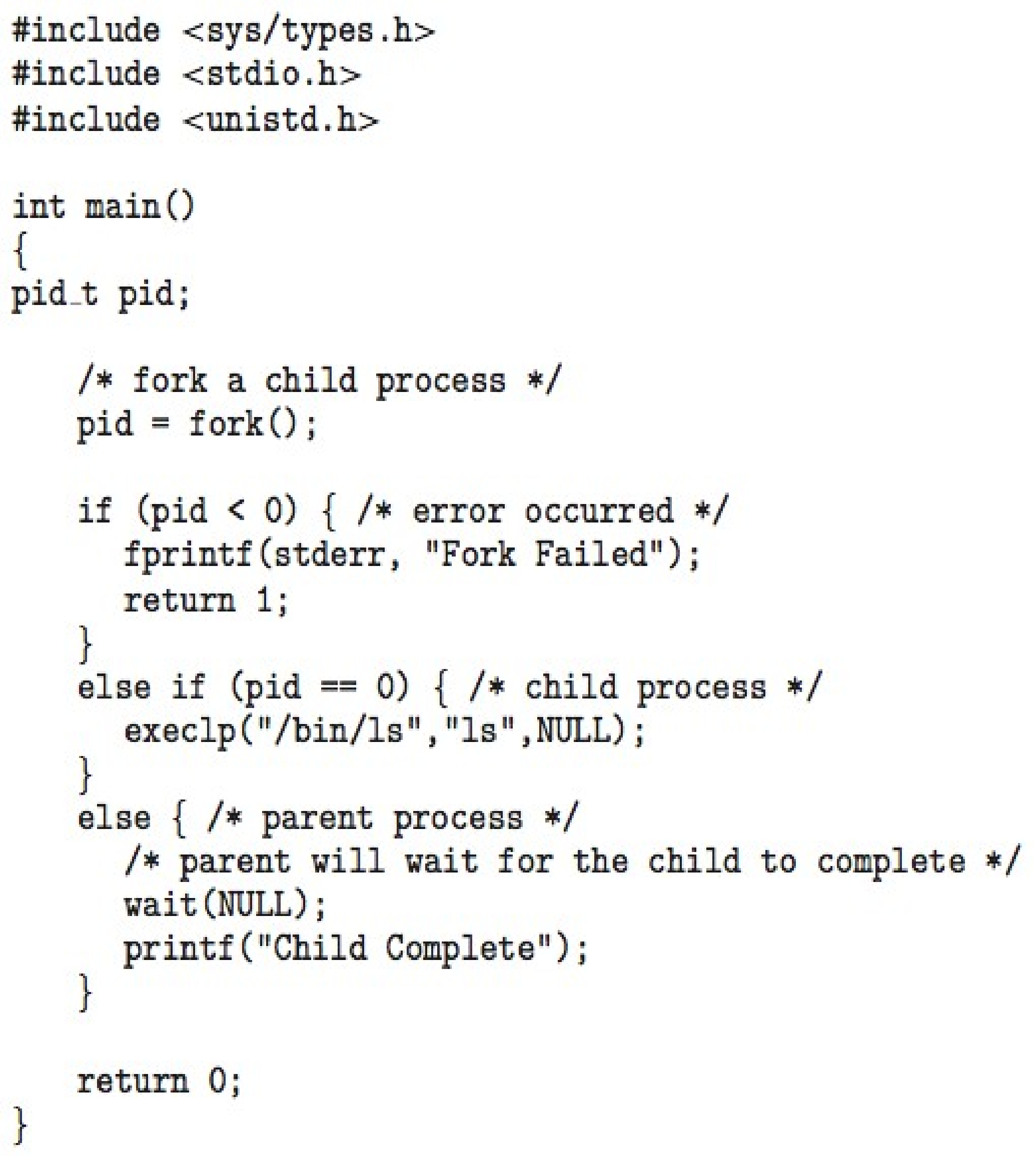
2. The child process has a new program loaded into it.

Process Creation (Cont.)

- **A new process is created by the fork() system call**
- **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
- The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways.
- The parent can then create more children
- If nothing else to do, parent process calls **`wait()`** `system call` to move itself off the ready queue until the child terminates
- A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution



C Program forking separate process



```
#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;
}
```

Process termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.

```
/* exit with status 1 */
exit(1);
```

- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated and reclaimed by the operating system

Zombie & Orphan Processes

- 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 a parent did not invoke **`wait()`** system call, the child process becomes a **zombie process**
  - When a process terminates, its resources are deallocated by the OS. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie** process.
- If a parent process did not invoke **`wait`** () and instead terminated, its child processes remain as **orphan processes**

Android Process Importance Hierarchy

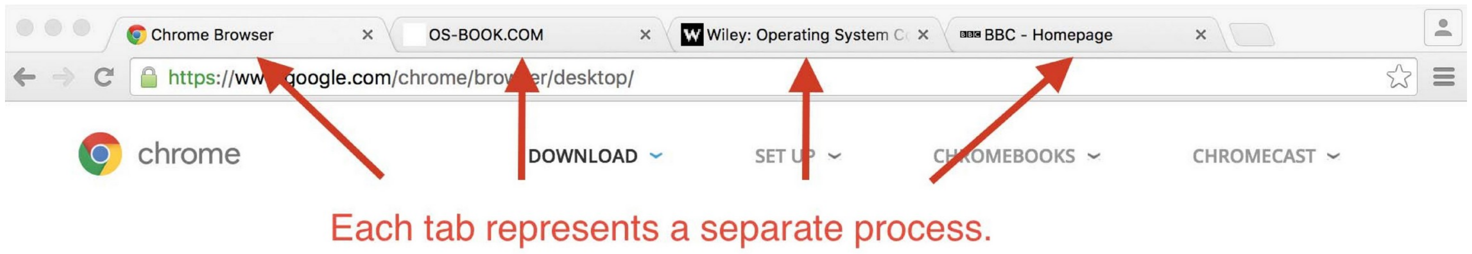
- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- Android will begin terminating processes that are least important.

Multiprocess Architecture – Chrome Browser

- Most contemporary web browsers provide tabbed browsing. A problem with this approach is that if a web application in any tab crashes, the entire process —including all other tabs displaying additional websites— crashes.
- Google's Chrome web browser was designed to address this issue by using a multiprocess architecture. The advantage of the multiprocess approach is that websites run in isolation from one another.

Chrome identifies three different types of processes:

- The **browser** process is responsible for managing the user interface as well as disk and network I/O. Only one browser process is created.
- **Renderer** processes contain the logic for handling HTML, Javascript, images, and so forth. If one website crashes, only its renderer process is affected. Renderer processes run in a **sandbox**, which means that access to disk and network I/O is restricted, minimizing the effects of any security exploits.
- A **plug-in** process is created for each type of plug-in (such as Flash) in use.



Chrome Browser
OS-BOOK.COM
Wiley: Operating System C
BBC - Homepage
https://www.google.com/chrome/browser/desktop/
chrome
DOWNLOAD
SET UP
CHROMEBOOKS
CHROMECAST
Each tab represents a separate process.

4. Interprocess Communication

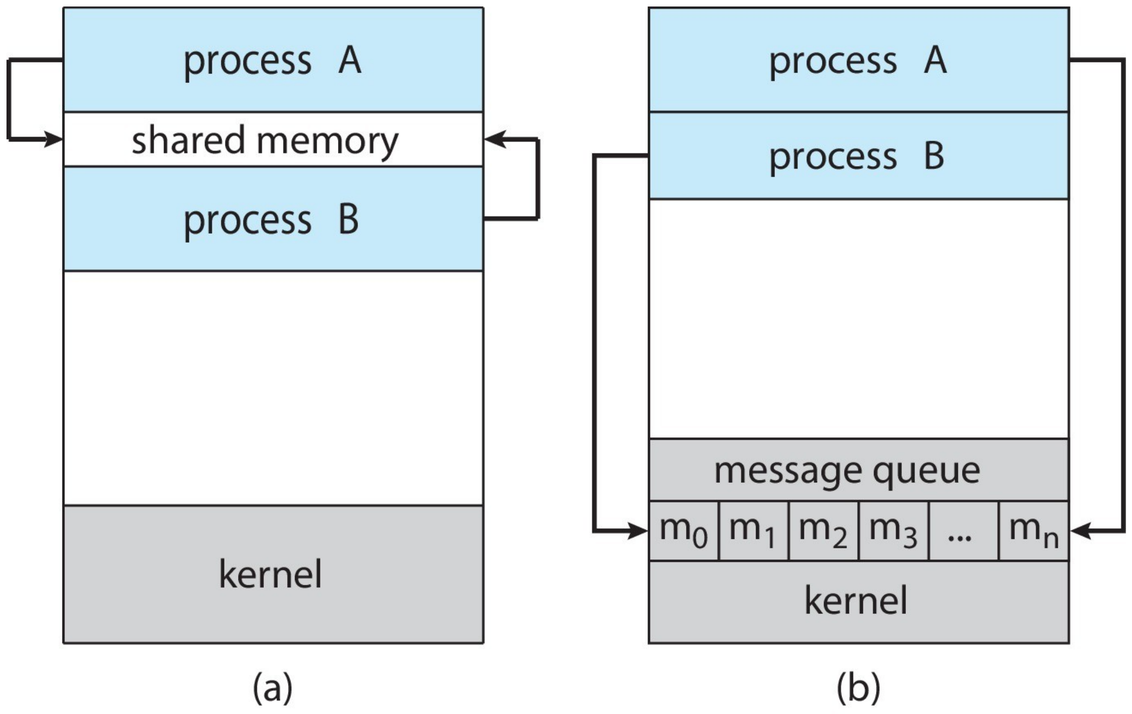
Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

- A process is ***independent*** if it does not share data with any other processes executing in the system.
- A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is a cooperating process.
  - Reasons for cooperating processes:
    - Information sharing - concurrent access to same information
    - Computation speedup – if computer has multiple processing cores, break task into subtasks, each executing in parallel
    - Modularity - dividing the system functions into separate processes
- Cooperating processes need **interprocess communication** (**IPC**) to exchange data
- Two fundamental models of IPC:
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Shared memory (b) Message passing



- A shared-memory region resides in the address space of the process creating the shared-memory segment.
- Data and the location are determined by processes and are not under OS control

- Message passing is easier to implement in a distributed system.
- Implemented using system calls.

Producer-Consumer Problem

- Paradigm for cooperating processes, ***producer*** process produces information that is consumed by a ***consumer*** process
  - Example: A compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.
  - In the client–server paradigm, server is the producer and a client is the consumer.

- Types of buffers:
  - **Unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
  - **Bounded buffer** assumes that there is a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Bounded-Buffer – Shared-Memory IPC

- The following variables reside in a region of memory shared by the producer and consumer processes:

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

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

- The shared buffer is implemented as a circular arraywith two logical pointers: **in** and **out**.
- The variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer.
- Buffer is empty when in == out;
- Buffer is full when ((in + 1) % BUFFER SIZE) == out.
- Solution is correct, but can only use **`BUFFER_SIZE-1`** elements

Producer Consumer Processes – Shared Memory

Producer Process:

```
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:

```
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 */
}
```

Interprocess Communication – Message Passing Systems

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

- Particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
  - For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.
- If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other: a ***communication link*** must exist between them.
  - Direct or indirect communication
  - Synchronous or asynchronous communication
- IPC facility provides at least two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Direct Communication

Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication.

- The processes need to know only each other's identity to communicate
- 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 or ports
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox

    send(A, message) —Send a message to mailbox A.

    receive(A, message) —Receive a message from mailbox A.
- 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

- Mailbox sharing
  - 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.

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 Consumer processes - message passing

Producer Process:

```
message next_produced;

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

      send(next_produced);
}
```

Consumer Process:

```
message next_consumed;

while (true) {
      receive(next_consumed)

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

Pipes

Acts as a conduit allowing two processes to communicate

- **Ordinary pipes** – cannot be accessed from outside the process that created it.
  - Require a parent–child relationship between the communicating processes on both UNIX and Windows systems.
  - Can be used only for communication between processes on the same machine
  - Once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.
- **Named pipes** – much more powerful communication tool.
  - Communication can be bidirectional, and no parent–child relationship is required.
  - Once a named pipe is established, several processes can use it for communication. In a typical scenario, a named pipe has several writers.
  - Continue to exist after communicating processes have finished.

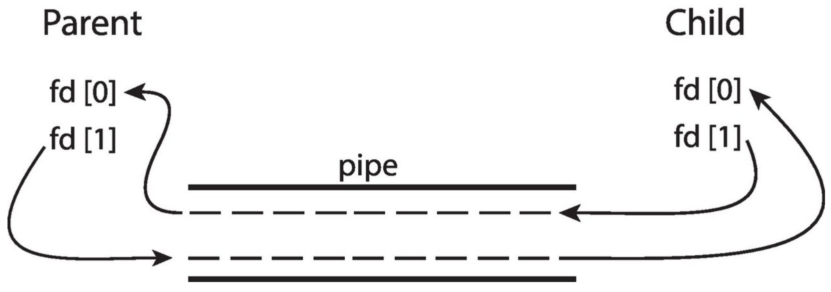
Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are unidirectional. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.

On UNIX systems, ordinary pipes are constructed using the function: pipe(int fd[])

fd[0] is the read end of the pipe, and fd[1] is the write end.

Both the parent process and the child process initially close their unused ends.



- Windows calls these **anonymous pipes**

Named Pipes

- Provided on both UNIX and Windows systems
- Referred to as FIFOs in UNIX systems.
- Once created, they appear as typical files in the file system. A FIFO is created with the mkfifo() system call and manipulated with the ordinary open(), read(), write(),and close() system calls.
- Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted. If data must travel in both directions, two FIFOs are typically used.
- Additionally, the communicating processes must reside on the same machine. If intermachine communication is required, sockets must be used.

PIPES IN PRACTICE

Pipes are used quite often in the UNIX command-line environment for situations in which the output of one command serves as input to another.

- For example, setting up a pipe between the ls and less commands (which are running as individual processes) allows the output of ls to be delivered as the input to less, enabling the user to display a large directory listing a screen at a time.
- A pipe can be constructed on the command line using the | character.

```
ls | less
```

- In this scenario, the ls command serves as the producer, and its output is consumed by the less command.

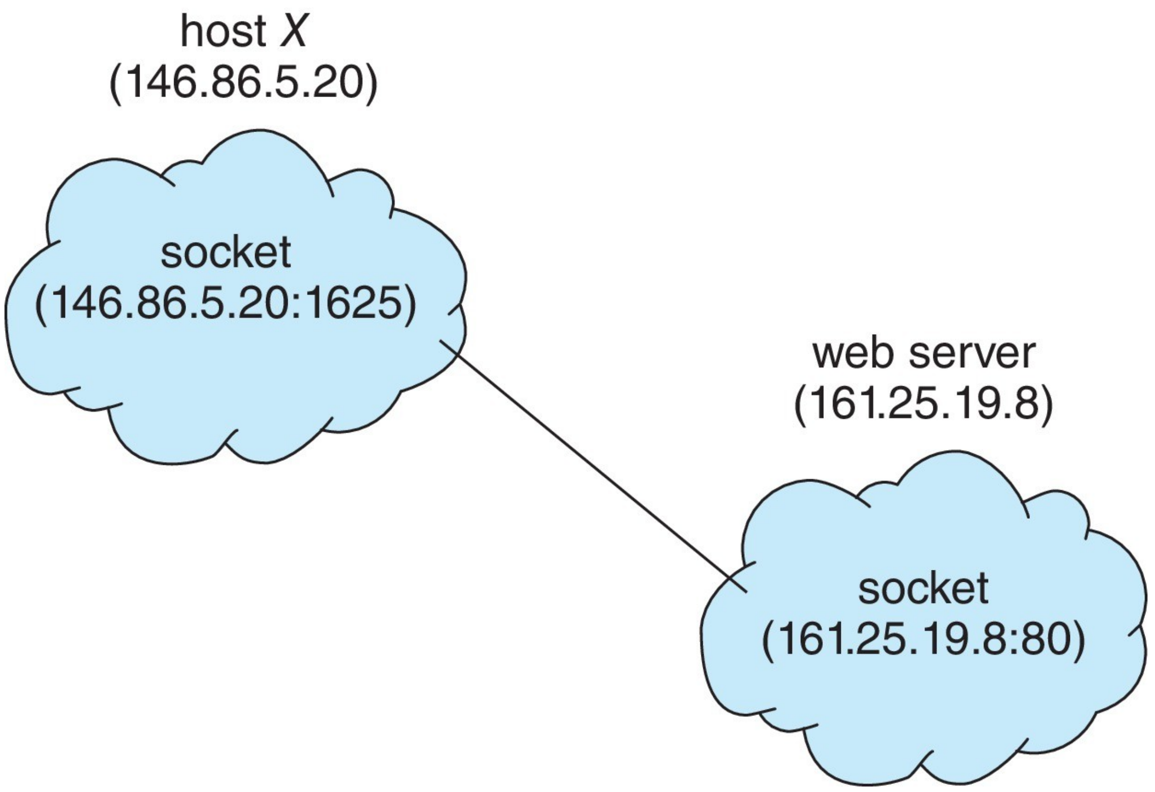
Communications in Client-Server Systems

- Sockets
  - A pair of processes communicating over a network employs a pair of sockets—one for each process.
  - Sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection
  - Servers implementing specific services (such as SSH, FTP, and HTTP) listen to well-known ports (an SSH server listens to port 22; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80).
  - All ports below 1024 are considered well known and are used to implement standard services.

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication



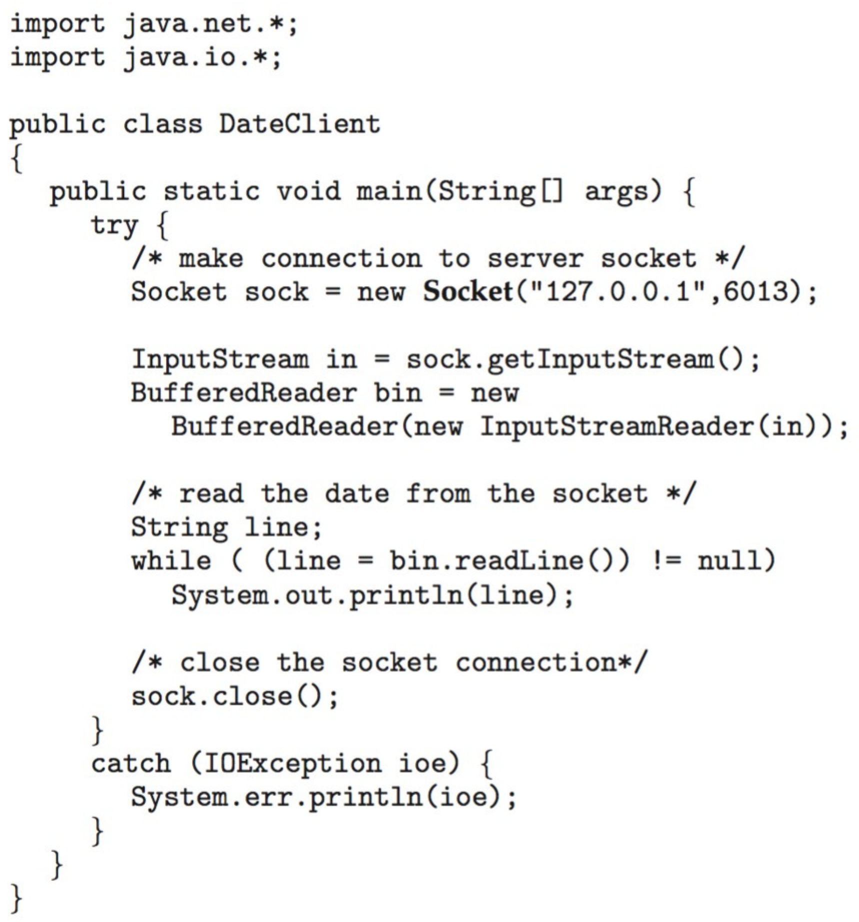
Sockets in Java

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

```
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);
      }
   }
}
```

Sockets in Java

The equivalent Date client



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

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);
      }
   }
}
```

End of Chapter 3