Chapter 3: Processes

Last update: 18/10/2023

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-process Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore inter-process communication using shared memory and message passing
- To describe communication in client-server systems

Process Concept

- Early computers allowed only one program to be executed at a time.
- This program had **complete control of the system** and
  - had access to **all the system's resources**.
- In contrast, contemporary computer systems allow **multiple programs** to be loaded into memory and executed **concurrently**.

- This evolution required firmer control and these needs resulted in the notion of a **process**, which **is a program in execution**.

- A **process** is the **unit of work** in a modern time-sharing system.

Process Concept

- Although main concern of OS is the execution of user programs, it also needs to take care of various system tasks
- A system therefore consists of a **collection of processes**:
  - **operating system processes** executing system code
  - **user processes** executing user code.
- Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them.
- By switching the CPU between processes, the operating system can make the computer more productive.

- In this chapter, you will learn about
  - what processes are
  - how they work.

Process Concept

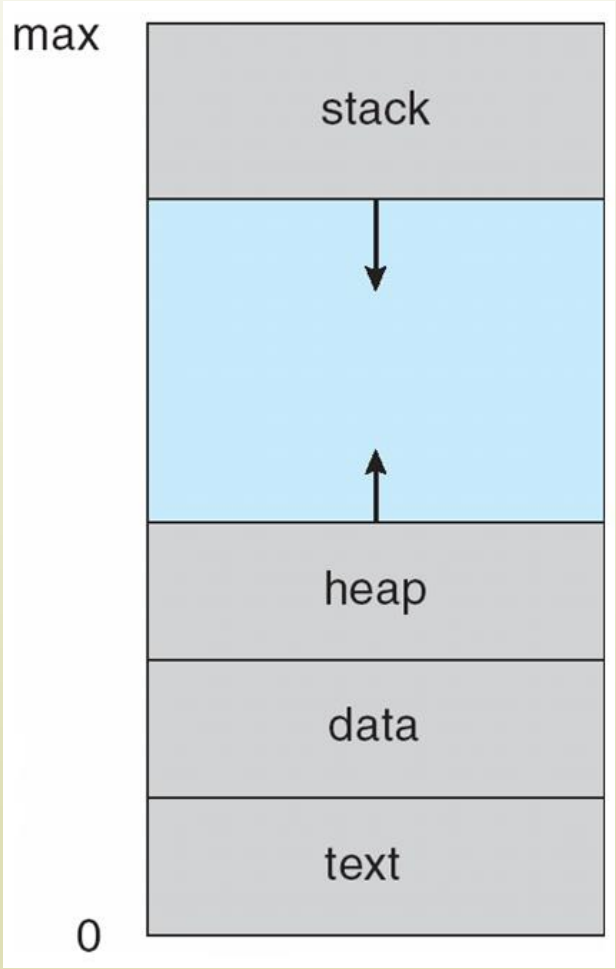
- The terms ***job*** and ***process*** are used almost interchangeably.
- We prefer the term ***process***
- It would be misleading to avoid the use of commonly accepted terms that include the word ***job*** (such as ***job scheduling***) simply because ***process*** has superseded (substituted) ***job.***

Process Concept

- Informally, a **process** is a program in **execution**.

- A process is more than the program code, which is sometimes known as the **text section**.
- It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's **registers**.
- A process generally also includes the process **stack**, which contains temporary data
  - (such as function parameters, return addresses, and local variables),
- and a **data section**, which contains global variables.
- A process may also include a **heap**, which is memory that is **dynamically** allocated during process run time.
- Now let's look at the figure.

Process in Memory



Process Concept – Summary

- **Process** – a program in execution;
- Multiple parts
  - The program code, also called **text section**
    - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

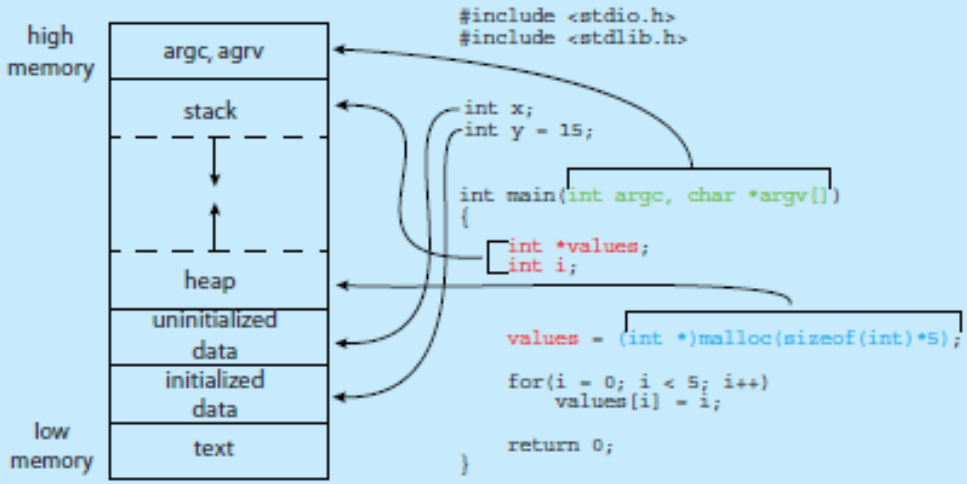
Process Concept - Summary (Cont.)

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via
  - GUI mouse clicks,
  - command line entry of its name,
- One program can be several processes
  - Consider multiple users executing the same program
  - For instance, the same user may invoke many copies of the web browser program.
  - Each of these is a separate process;
    - although the text sections are **equivalent**,
    - The data, heap, and stack sections vary.
- It is also common to have a process that creates many processes as it runs

MEMORY LAYOUT OF A C PROGRAM

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



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.

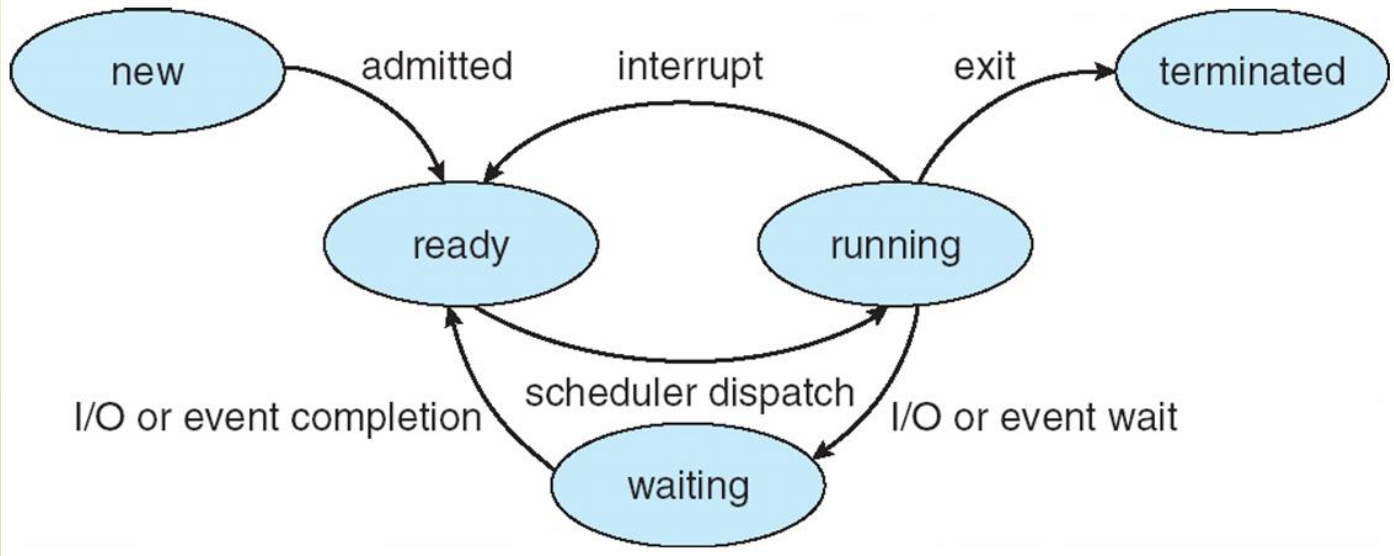
Process Concept

Process States:

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

Process Concept

- It is important to note that
  - **only one** process can be ***running*** on any processor at any instant.
- Many processes may be ***ready*** and ***waiting***, however.



Process Concept

Process Control Block:

- Each process is represented by a **process control block (PCB)**

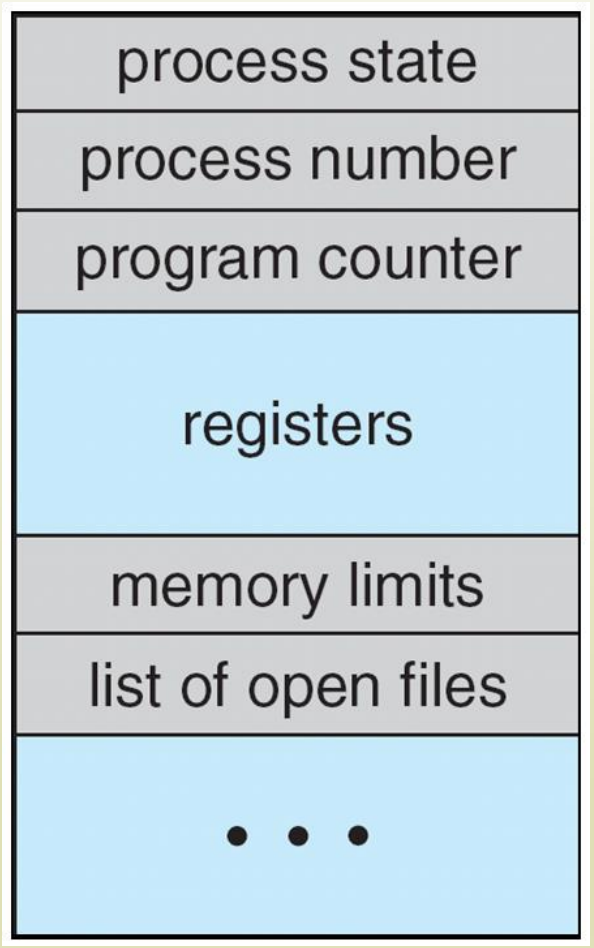
- It contains many pieces of information associated with a specific process, including these:
  - **Process state**. The state may be new, ready, running, waiting, halted, and so on.
  - **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
  - **CPU registers**. The registers vary in number and type, depending on the computer architecture. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
  - **CPU-scheduling information**. This information includes a **process priority**, pointers to scheduling queues, and any other scheduling parameters. (Chapter 6 describes process scheduling.)
  - **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 8).
  - **Accounting information**. This information includes **the amount of CPU** and real time used, **time limits**, account numbers, job or process numbers, and so on.
  - **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of **open files**, and so on.

Process Control Block (PCB)

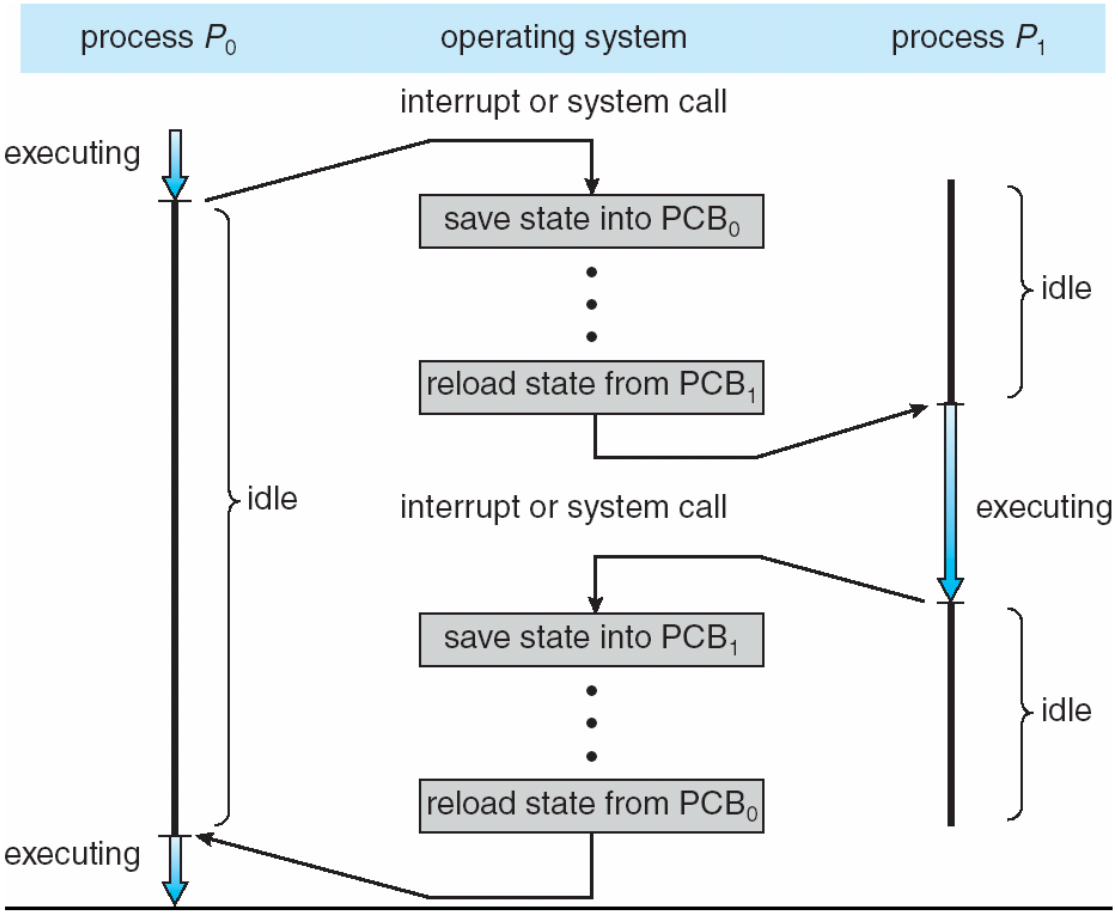
Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- 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



CPU Switch From Process to Process



Context Switch

- As discussed in 2 weeks ago, **interrupts** cause the operating system to change a CPU from its current task and to run a kernel routine.
- Such operations happen frequently on general-purpose systems.
- When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU so that it can restore that context when its processing is done,
  - essentially **suspending the process and then resuming it**.
- Generically, we perform a **state save** of the current state of the CPU and then a **state restore** to resume operations.
- Switching the CPU 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**.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is **pure overhead**, because the system does **no useful work** while switching.

Context Switch- Summary

- 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

Threads

Threads:

- **Single Thread**:
  - The process model discussed so far has implied that a process is a program that performs a single **thread** of execution.
    - For example, when a process is running a word-processor program, a single thread of instructions is being executed.
  - This single thread of control allows the process to perform only one task at a time.
  - The user **cannot** simultaneously type in characters and run the spell checker within the same process, for example.

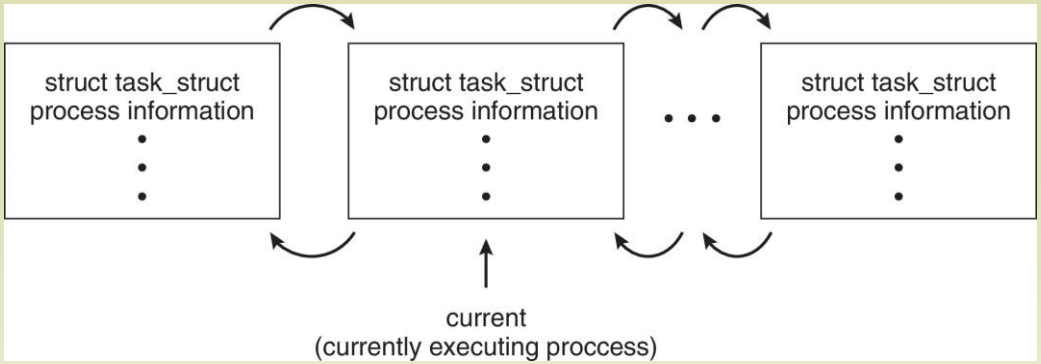
- **Multiple threads:**
  - Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and
    - thus to perform more than one task at a time.
  - This feature is especially beneficial on multicore systems, where multiple threads can run in **parallel.**
  - Must then have storage for thread details, multiple program counters in PCB
  - Other changes throughout the system are also needed to support threads.
- Next week we will explore threads in detail.

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 */
```

For example, the state of a process is represented by the field long state in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of task struct. The kernel maintains a pointer– current–to the process currently executing on the system, as shown below:



Process Scheduling

- The objective of multiprogramming is
  - to have some process running at all times, to maximize CPU utilization.
- The objective of time-sharing is to
  - switch the CPU among processes so **frequently** that
  - users can interact with each program while it is running.

- To meet these objectives,
  - the **process scheduler** selects an available process for program execution on the CPU.
- For a single-processor system, there will never be more than one running process.
  - If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.
  - Each CPU core can run one process at a time.

Process Scheduling

- Since devices run significantly slower than processors, the process will have to wait for the I/O to become available.
- Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a wait queue →

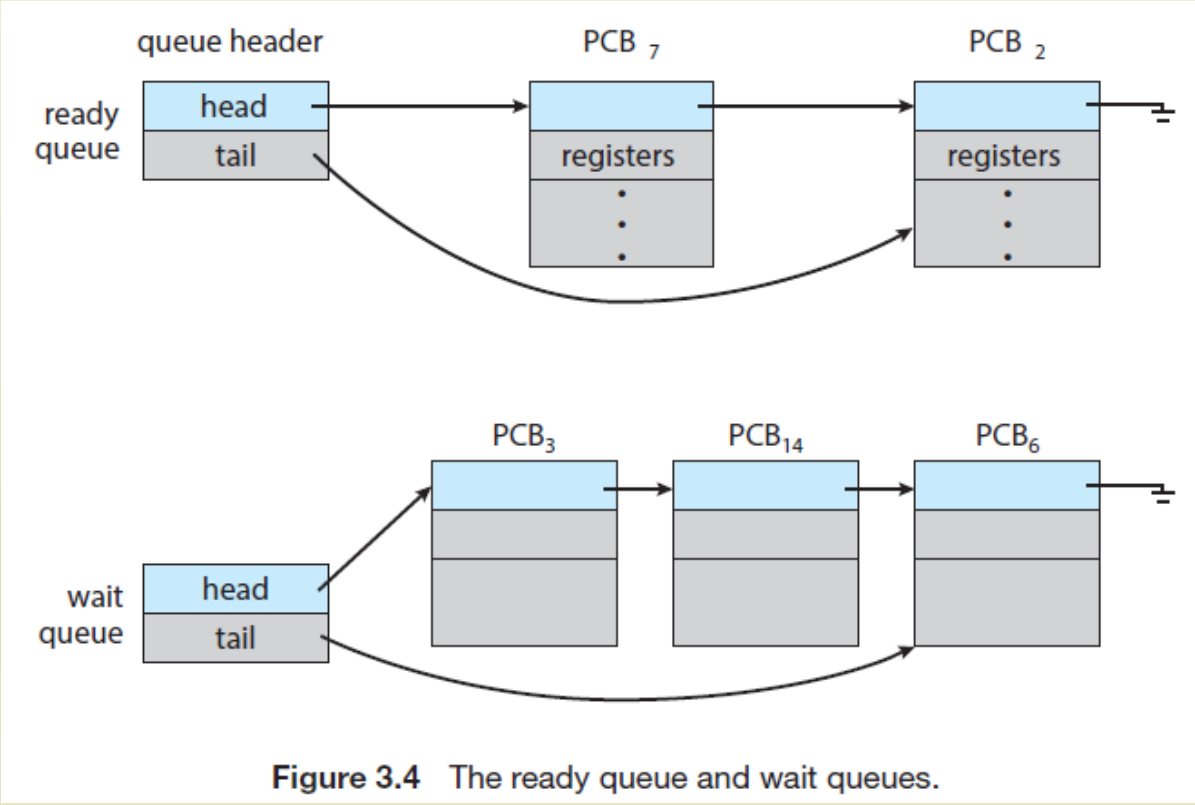
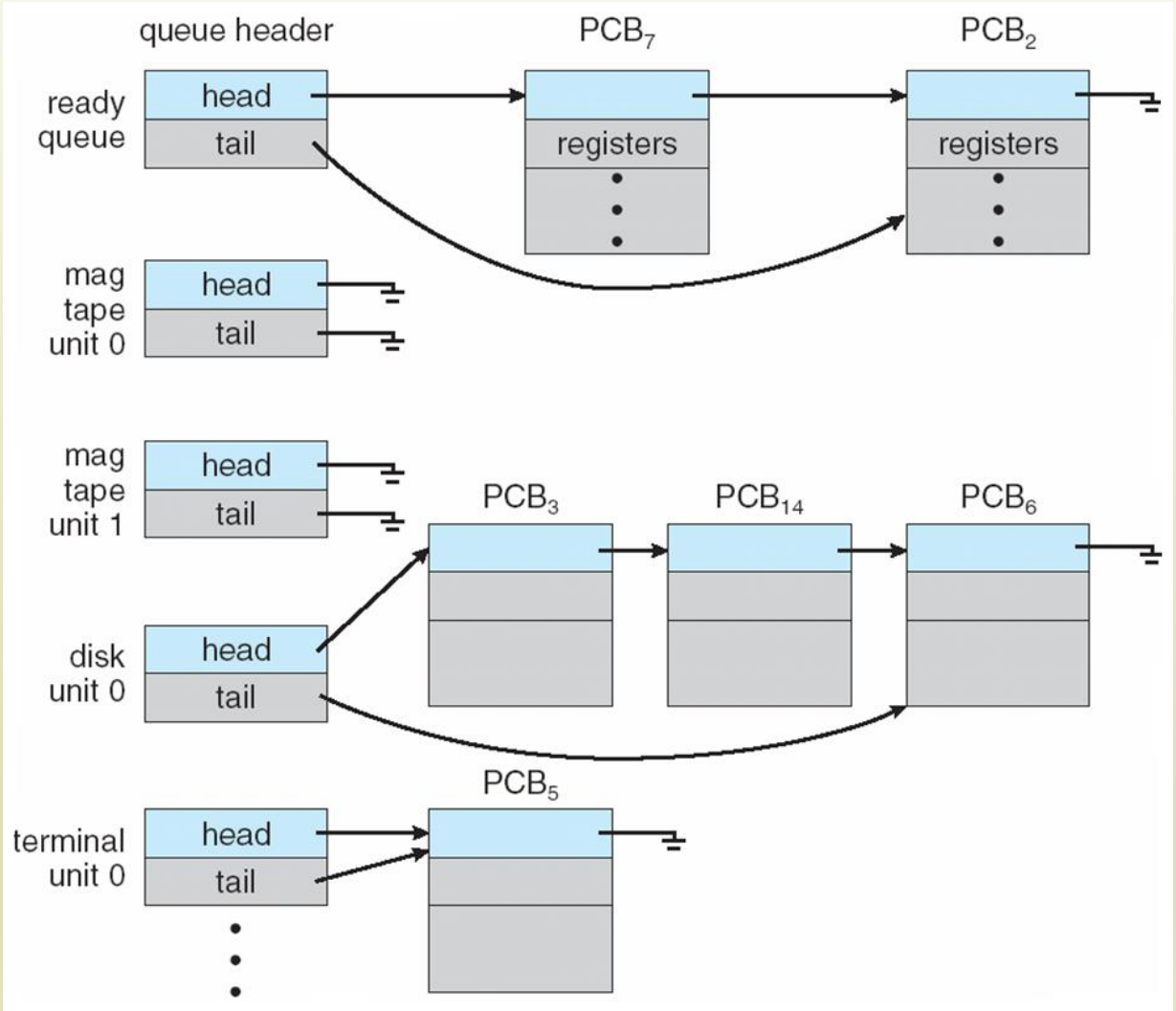


Figure 3.4 The ready queue and wait queues.

Device queues

- The system also includes queues.
- When a process is allocated the CPU,
  - it executes for a while and eventually
    - quits,
    - is interrupted, or
    - waits for the occurrence of a particular event, such as the **completion of an I/O request.**
- Suppose the process makes an I/O request to a shared device, such as a disk.
  - Since there are many processes in the system, the disk may be **busy with the I/O request** of some other process.
  - The process therefore may have to **wait** for the disk.
- The list of processes waiting for a particular I/O device is called a **device queue**.
  - Each device has its own device queue.

Ready Queue And Various I/O Device Queues

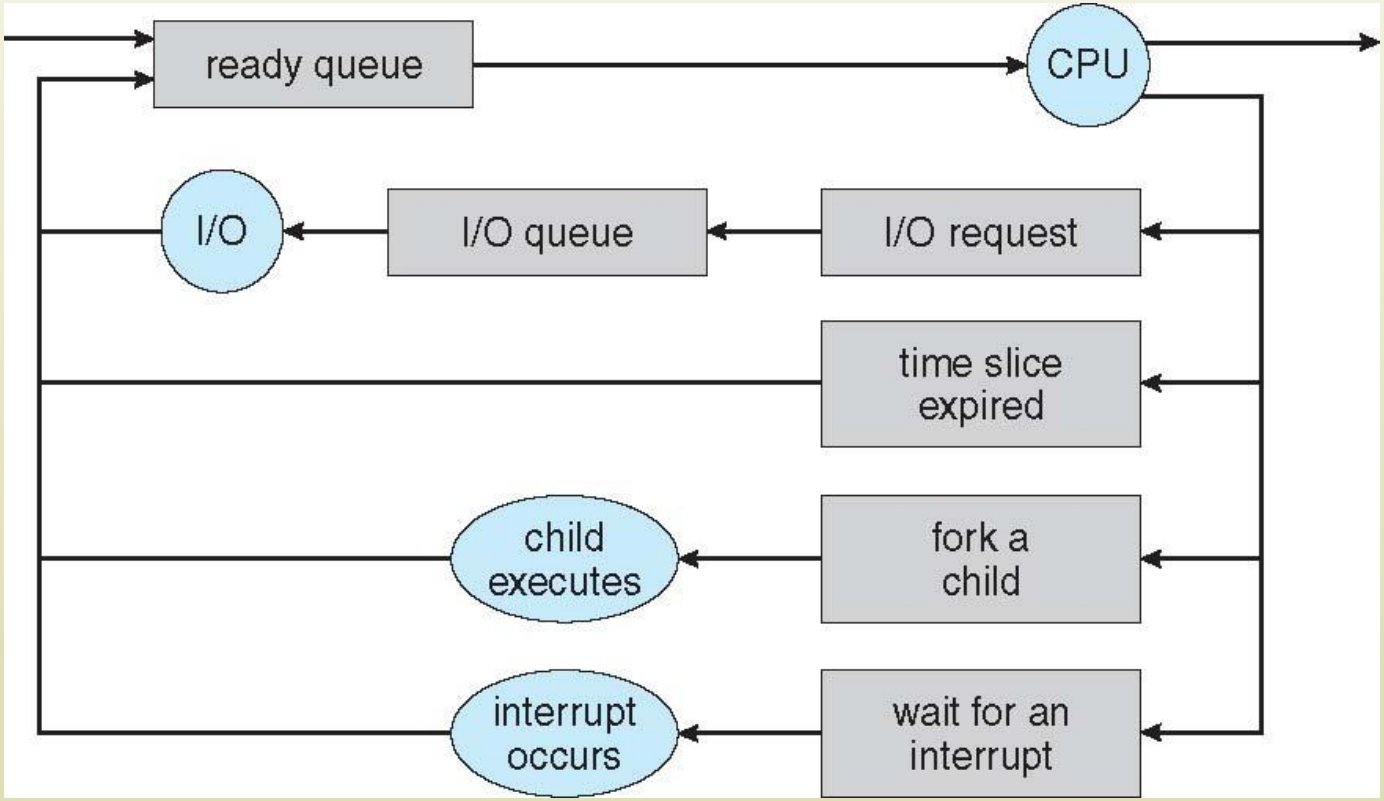


Representation of Process Scheduling

- A representation of process scheduling is a **queueing diagram**,
- Each rectangular box represents a queue. Two types of queues are present:
  - the ready queue and
  - a set of device queues.
- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**.
- Once the process is allocated the CPU and is executing, one of several events could occur:
  1. The process could issue an I/O request and then be placed in an **I/O queue**.
  2. The process could **create a new child process** and wait for the child's termination.
  3. The process could **be removed** forcibly from the CPU, as a result of an interrupt, and be put back in the **ready queue**.
- In the first two cases, the process eventually switches from
  - the waiting state to the ready state
  - and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows



Process Scheduling - Summary

- Maximize CPU use, quickly switch processes onto CPU for time sharing

- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **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
  - Processes migrate among the various queues

Schedulers

- A process migrates among the various scheduling queues throughout its lifetime.
- The OS must select processes from these queues in some fashion.
- The selection process is carried out by the scheduler.
- Usually, more processes are submitted than can be executed immediately.
- These processes reside in a mass-storage device (typically a disk), where they are kept for later execution.

- The **long-term scheduler**, or job scheduler, selects processes from this pool and loads them into memory for execution.

- The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

Schedulers

Short-term scheduler & long-term scheduler

- The distinction between these two schedulers lies in frequency of execution.
- The **short-term scheduler** must select a new process for the CPU frequently.
- A process may execute for only a **few milliseconds** before waiting for an I/O request.
- Often, the short-term scheduler executes at least once every 100 milliseconds.
  - Because of the short time between executions, the short-term scheduler must be **fast**.
  - If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100 + 10) = 9 percent of the CPU is being used (wasted) simply for scheduling the work.

Schedulers

- The **long-term scheduler** executes much less frequently;
  - minutes may separate the creation of one new process and the next.
- The long-term scheduler controls the **degree of multiprogramming**
  - the number of processes in memory
- If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- Thus, the long-term scheduler may need to be invoked only when a process leaves the system.
- Because of the longer interval between executions, the long-term scheduler can afford to **take more time to decide** which process should be selected for execution.

Schedulers

- It is important that the long-term scheduler make a careful selection.
- In general, most processes can be described as either I/O bound or CPU bound.
  - An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations.
  - A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- It is important that the long-term scheduler select a good ***process mix*** of I/O-bound and CPU-bound processes.
- If all processes are I/O bound, the ready queue will almost always be **empty**, and the short-term scheduler will have little to do.
- If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.
- The system with the **best performance** will thus have a combination of CPU-bound and I/O-bound processes.

Schedulers - summary

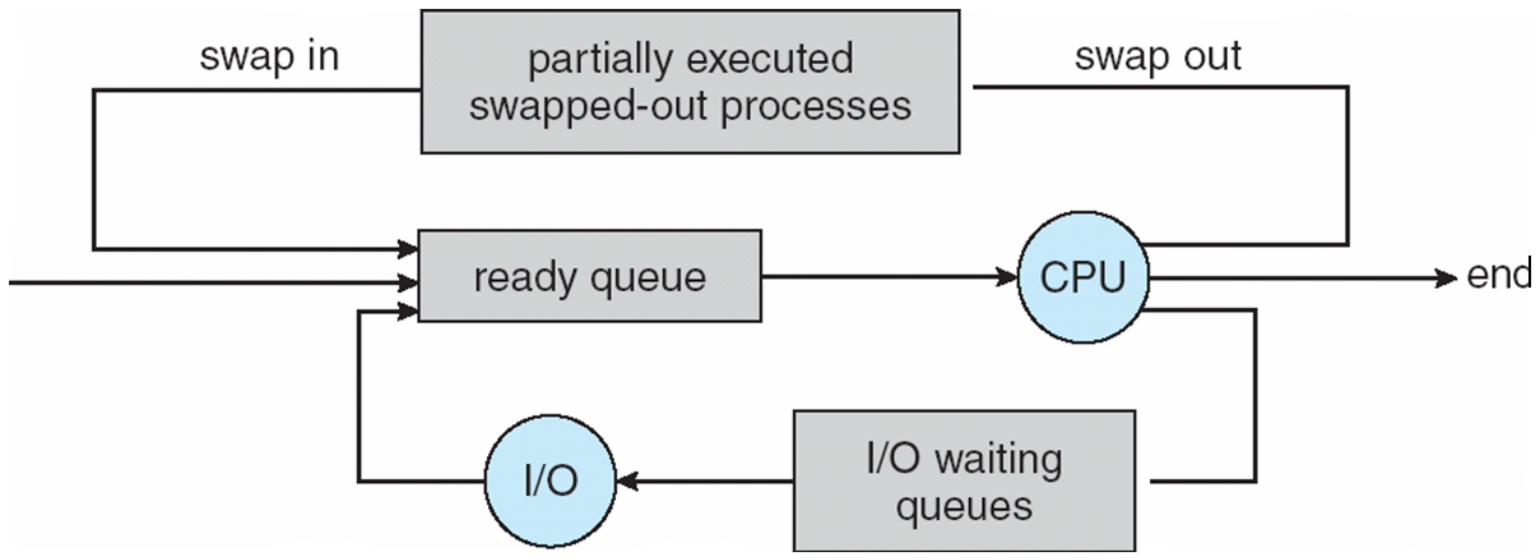
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **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
- Long-term scheduler make an effort for good ***process mix***

Schedulers

- Some operating systems, may introduce an additional, intermediate level of scheduling.
  - **Medium-term scheduler**
- The idea behind a medium-term scheduler is that 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**. The process is swapped out, and is later swapped in, by the medium-term scheduler.
- Swapping may be necessary to improve the process mix or because a change in memory requirements has consumed all available memory,
  - requiring memory to be freed up.
- Swapping will be discussed in detail 5 weeks later.

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Operations on Processes

- The processes in most systems can
  - execute concurrently,
  - and they may be created and deleted dynamically.
- Thus, system must provide mechanisms for:
  - process creation,
  - process termination

- In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

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 could be either:
  1. Parent and children share all resources
  2. Children share subset of parent's resources
  3. Parent and child share no resources
- Execution options could be either:
  1. Parent and children execute concurrently
  2. Parent waits until children terminate

Process Creation

- A process may create several new processes.
- The creating process is called **a parent process**,
  - and the new processes are called the **children** of that process.
- Each of these new processes may in turn create other processes, forming a **tree** of processes.
- Most operating systems identify processes according to a unique **process identifier** (or **pid**),
  - which is typically an integer number.
- The pid provides a unique value for each process and it can be used as an index to access various attributes of a process within the kernel.

A Tree of Processes in Linux

- Typical process tree for the Linux operating system, showing the name of each process and its pid.
- The recent Linux distributions have replaced **init** with **systemd**.

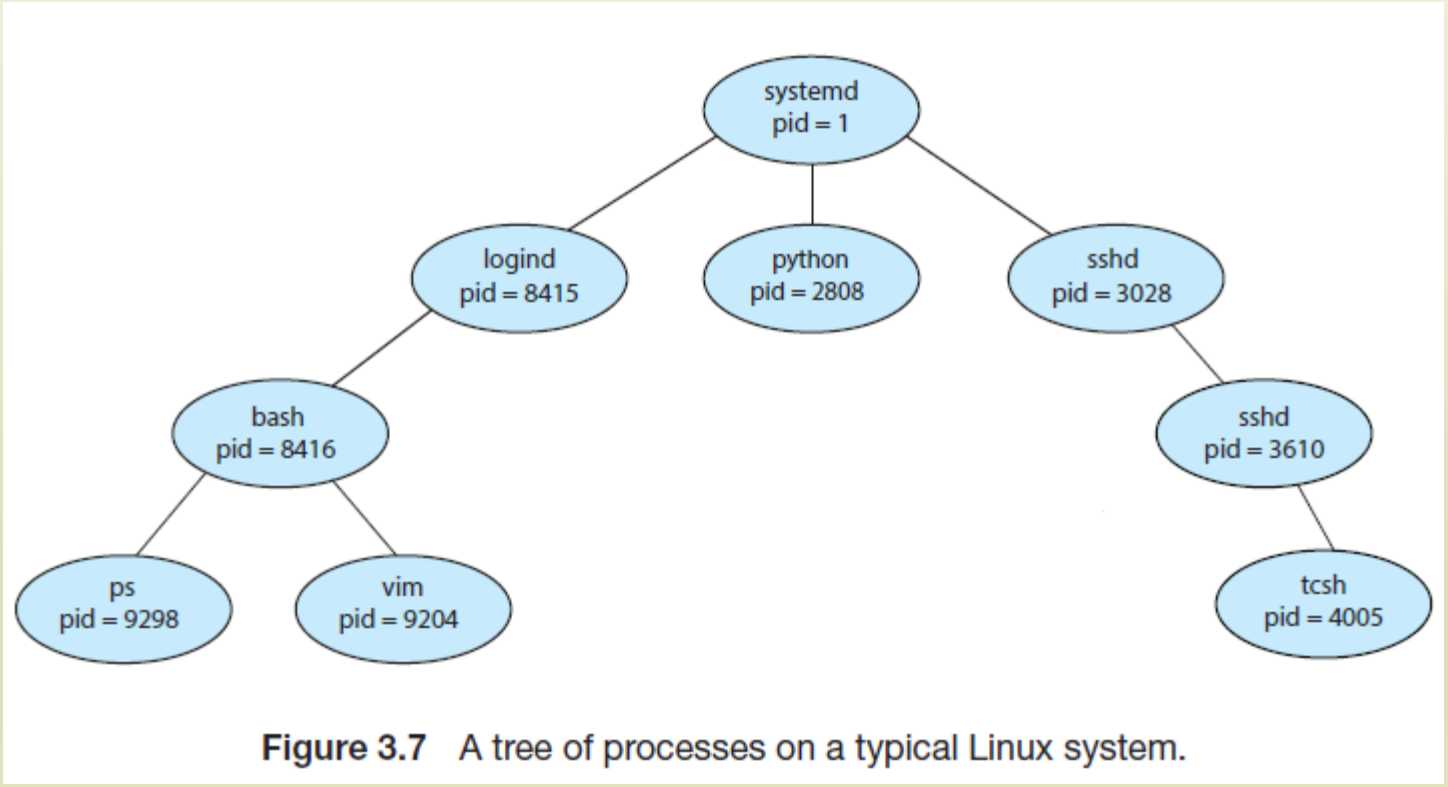


Figure 3.7 A tree of processes on a typical Linux system.

Process Creation

- The **init (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 init process can also create various user processes, such as a web or print server, a ssh server.

- In the figure, we see two children of **systemd**—**logind** and **sshd**.
- The **logind** process is responsible for managing clients that directly log onto the system.
  - In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416.
  - Using the bash command-line interface, this user has created the process ps as well as the vim editor.

- The **sshd** process is responsible for managing clients that connect to the system by using ssh (which is short for secure shell).

Process Creation

- When a process creates a new process, two possibilities for execution exist:
  - **1.** The parent continues to execute concurrently with its children.
  - **2.** The parent waits until some or all of its children have terminated.

- 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

- To illustrate these differences, let's first consider the UNIX operating system.
- A new process is created by the **fork()** system call.
- The new process consists of a **copy of the address space** of the original process.
- This mechanism allows the parent process to communicate easily with its child process.
- Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference:
  - the return code for the fork() is **zero for the new (child) process**, whereas **the (nonzero) process identifier of the child** is **returned** to the parent.

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;
}
```

C Program Forking Separate Process

- We now have two different processes running copies of the same program.

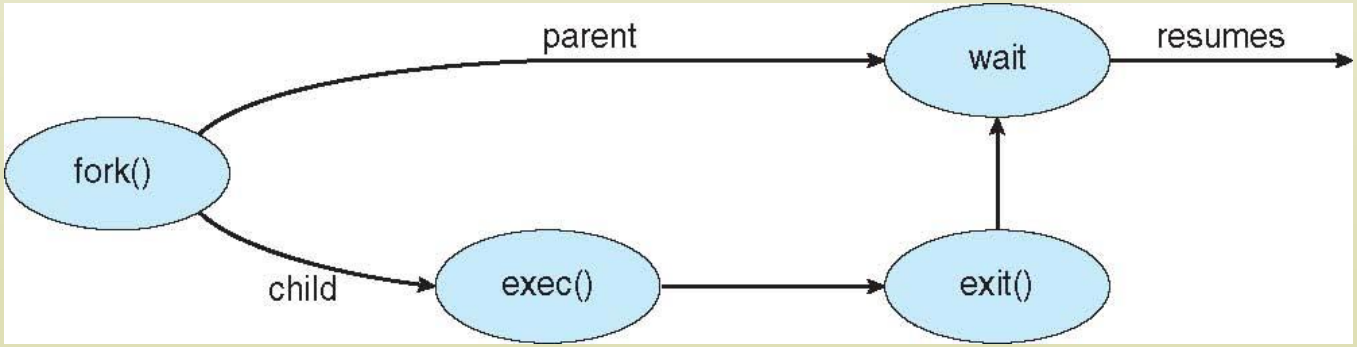
- The only difference is that the returning value of fork() for the child process is 0,
  - while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process).

- The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files.
- In the example, the child process then overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call).

- The parent waits for the child process to complete with the wait() system call.
- When the child process completes the parent process resumes from the call to wait(),
  - where it completes using the **exit**() system call.

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- 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



Creating a Separate Process via Windows API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

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.
  - Returns status data from child to parent (via **`wait()`**)
    - At that point, the process may return a status value (typically an integer) to its parent process (via the **wait**() system call).
  - Process' resources are deallocated by operating system
    - All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the 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 OS does not allow a child to continue if its parent terminates

Process Termination

- Some OS 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_t pid;
int status;
pid = wait(&status);
```

Process Termination

- 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.
- All processes transition to this state when they terminate, but generally they exist as zombies only briefly.
- Once the parent calls **wait**(), the process identifier of the zombie process and its entry in the process table are released.

Process Termination

- Now consider what would happen if a parent did not invoke **wait**() and instead terminated,
  - thereby leaving its child processes as **orphans**.
- Traditional UNIX systems addressed this scenario by assigning the **init** process as the new parent to orphan processes.

- The **init** process periodically invokes **wait**(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

- 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 single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multi-process 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
  - **Plug-in** process for each type of plug-in

Interprocess Communication

- A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system.
  - Any process that does not share data with any other process is independent.
- 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.

- There are several reasons for providing an environment that allows process cooperation:
  - **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
  - **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.
    - Notice that such a speedup can be achieved only if the computer has **multiple processing cores**.
  - **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads

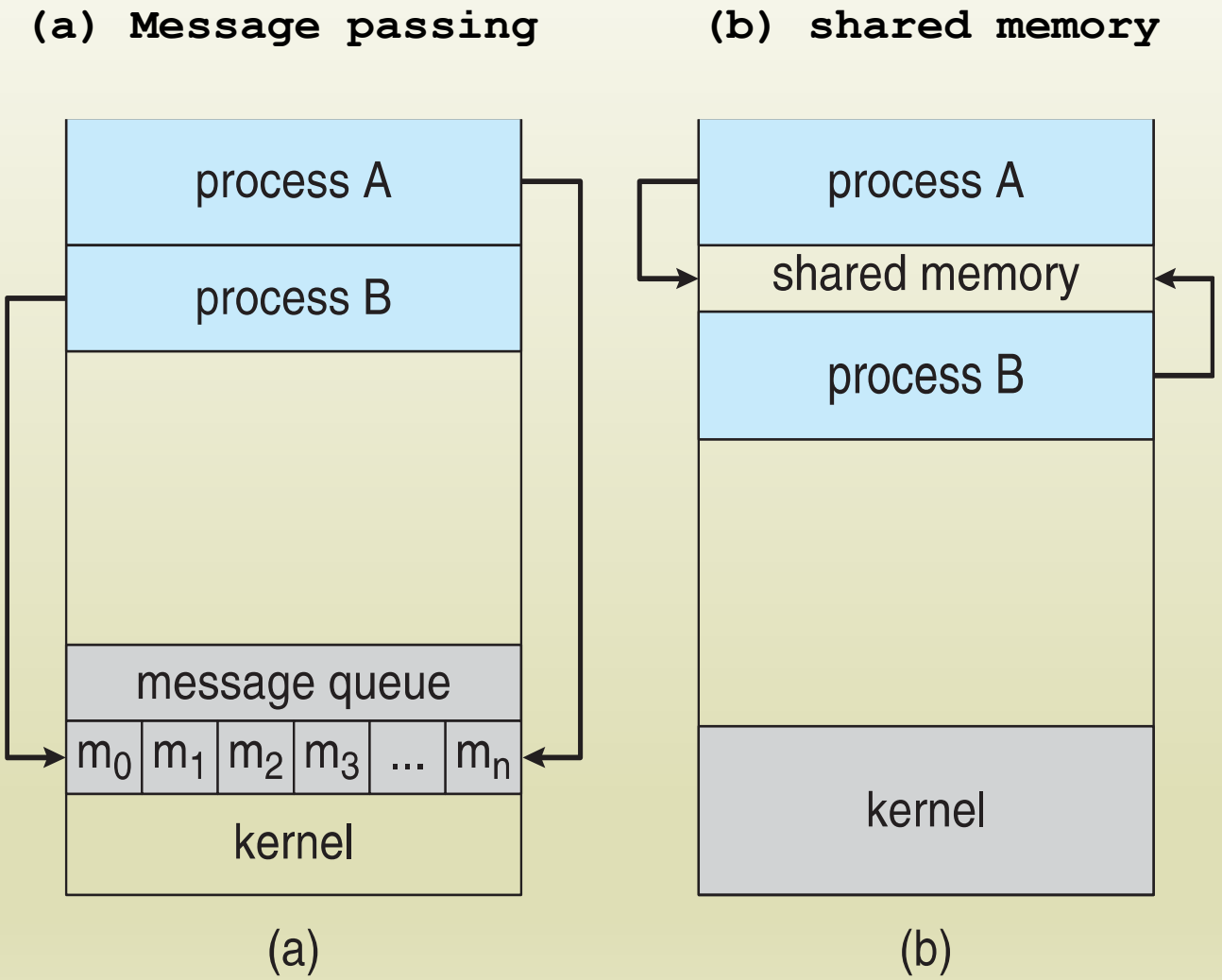
Interprocess Communication

- Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information.
- There are two fundamental models of interprocess communication:
  - **shared memory** and
  - **message passing**.

- In the **shared-memory model**, a region of memory that is shared by cooperating processes is established.
  - Processes can then exchange information by reading and writing data to the shared region.

- In the **message-passing model**, communication takes place by means of messages exchanged between the cooperating processes.

Communications Models



Shared-Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Processes can exchange information by reading and writing data in the shared areas in the memory.
- The processes are responsible for ensuring that they are not writing to the same location simultaneously.

Producer-Consumer Problem

- To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes.
- A **producer** process produces information that is consumed by a **consumer** process.
  - One solution to the producer–consumer problem uses shared memory.

- To allow producer and consumer processes to run concurrently, we must have available **a buffer of items** that can be **filled** by the producer and **emptied** by the consumer.
  - **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
    - the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full

Bounded-Buffer – Shared-Memory Solution

- 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 array** with two logical pointers: **in** and **out**.
- The buffer is empty when **in == out**;
- The buffer is full when ((in + 1) % BUFFER_SIZE) == out.

Bounded-Buffer – Producer

- The code for the producer process is shown below, and the code for the consumer process is shown in next slide
- The producer has a local variable **next_produced** in which the new item to be produced is stored.

Code for the Producer:

```
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;

}
```

Bounded Buffer – Consumer

- The consumer process has a local variable **next_consumed** in which the item to be consumed is stored.

Code for the Consumer :

```
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 – Shared Memory

- An area of memory is shared among the processes that wish to communicate
- The communication is **under the control** of the users processes not **the operating system**
- Requiring some mechanisms that will allow the user processes to **synchronize** their actions when they access shared memory.
- **Synchronization** is discussed in great details later in the semester.

3.4.2 Message-Passing Systems

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

- It is 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.
- A message-passing facility provides at least two operations:
  - send(message)
  - receive(message)
- Messages sent by a process can be either fixed or variable in size.

Examples of IPC Systems

- We first cover the **POSIX API** for shared memory.
- Next, we present **Windows IPC.**
- We conclude with **pipes**, one of the earliest IPC mechanisms on UNIX systems.

Examples of IPC Systems

- Several IPC mechanisms are available for POSIX systems, including
  - shared memory and
  - message passing.
- Here, we explore the POSIX API for **shared memory.**

- POSIX shared memory is organized using **memory-mapped files**, which associate the region of shared memory with a file.
- A process must first create a shared-memory object using the shm_open() system call, as follows:
  - `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`

Examples of IPC Systems

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- The first parameter specifies the name of the shared-memory object.
- The subsequent parameters specify that
  - the shared-memory object is to be created if it does not yet exist (O_CREAT)
  - that the object is open for reading and writing (O_RDWR).
- The last parameter establishes the directory permissions of the shared-memory object.
- A successful call to shm_open() returns an integer file descriptor for the shared-memory object.

POSIX API - IPC

- Once the object is established, the **ftruncate**() function is used to configure the size of the object in bytes.
- The call

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

  - sets the size of the object to 4,096 bytes.

- Finally, the **mmap**() function establishes a memory-mapped file containing the shared-memory object.
- It also returns a pointer to the memory-mapped file that is used for accessing the shared-memory object.

IPC POSIX Producer

- The producer, shown in the next slide
  - creates a shared-memory object named OS and
  - writes the string "Hello World!" to shared memory.
- The program memory-maps a shared-memory object of the specified size and allows writing to the object.
- The flag MAP_SHARED specifies that changes to the shared-memory object will be visible to all processes sharing the object.

IPC POSIX Producer

```
int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

Real code for producer

To compile: gcc producer.c -o producer -lrt

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main(){
   int size=4096;
   char *name = "CENG305";
   char *msg1="Hello ";
   char *msg2="Students";
   int shm_fd;
   void *ptr;
   shm_fd = shm_open(name,O_CREAT | O_RDWR, 0666);
   ftruncate(shm_fd,size);
   ptr = mmap(0,size,PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
   sprintf(ptr," %s ",msg1);
   ptr += strlen(msg1);
   sprintf(ptr," %s ",msg2);
   ptr += strlen(msg2);

   return 0;
}
```

IPC POSIX Consumer

- The consumer process, shown in the following slide, reads and outputs the contents of the shared memory.
- The consumer also invokes the **shm_unlink**() function, which removes the shared-memory segment after the consumer has accessed it.
- Please study further for exercises using the POSIX shared-memory API in the programming exercises at the end of this chapter.

IPC POSIX Consumer

```
int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Real code for consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main(){
   int size=4096;
   char *name = "CENG305";
   int shm_fd;
   void *ptr;
   shm_fd = shm_open(name, O_RDONLY, 0666);
   ptr = mmap(0,size,PROT_READ , MAP_SHARED, shm_fd, 0);

   printf("The message of producer:  [  %s  ]\n",(char *)ptr);
   shm_unlink(name);
   return 0;
}
```

Windows IPC

- Windows provides support for multiple operating environments, or
  - ***subsystems.***
- Application programs communicate with these subsystems via a message-passing mechanism.
- Thus, application programs can be considered clients of a subsystem server.
- The message-passing facility in Windows is called the **advanced local procedure call (ALPC)** facility.
- It is used for communication between two processes on the same machine.
  - It is similar to the standard remote procedure call (RPC)

Pipes

- A **pipe** acts as a canal allowing two processes to communicate.
- Pipes were one of **the first IPC mechanisms** in early UNIX systems.
- In implementing a pipe, four issues must be considered:

  1. Does the pipe allow **bidirectional (two-way)** communication, or is communication **unidirectional**?

  2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions)?

  3. Must a relationship (such as ***parent–child***) exist between the communicating processes?

  4. Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

- Two common types of pipes used on both UNIX and Windows systems:
  - **ordinary pipes** and **named pipes.**

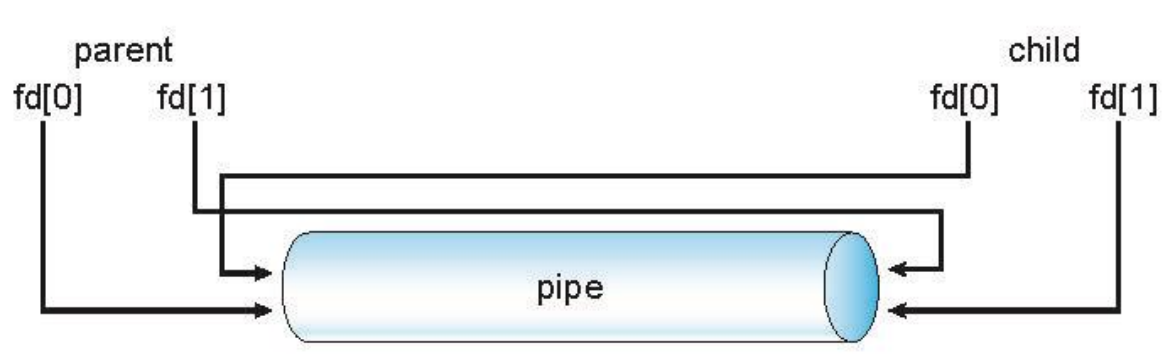
Pipes

Ordinary pipes:

- Allow two processes to communicate in standard producer–consumer fashion:
  - the producer writes to one end of the pipe (the **write-end**)
  - and the consumer reads from the other end (the **read-end**).
- As a result, ordinary pipes are **unidirectional**, allowing only one-way communication.
- If **two-way** communication is required, **two** pipes must be used, with each pipe sending data in a different direction.
- We next illustrate constructing ordinary pipes on both UNIX and Windows systems.
- In both program examples, one process writes the message “Greetings” to the pipe, while the other process reads this message from the pipe.

Ordinary Pipes

- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Require parent-child relationship between communicating processes
  - An ordinary pipe **cannot be accessed from outside** the process that created it.
  - A **parent** process creates a pipe and uses it to communicate with a **child** process that it creates via **fork**().



- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook

Real code for Ordinary pipe

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

#define BUFFER_SIZE 25
#define READ_END        0
#define WRITE_END       1

int main(void)
{
            char write_msg[BUFFER_SIZE] = "Greetings";
            char read_msg[BUFFER_SIZE];
            pid_t pid;
            int fd[2];    // an array of 2 integers fd[0] and fd[1]

            if (pipe(fd) == -1) { fprintf(stderr,"Pipe failed"); return 1;}

            pid = fork();
            if (pid < 0) { fprintf(stderr, "Fork failed"); return 1; }

            if (pid > 0) /* parent */ {
                        close(fd[READ_END]); //close read end (index 0) - it is unused here
                        write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
                        close(fd[WRITE_END]);
            }
            else { /* child process */
                        close(fd[WRITE_END]); //close write end (index 1) - it is unused here
                        read(fd[READ_END], read_msg, BUFFER_SIZE);
                        printf("child read %s\n",read_msg);
                        close(fd[READ_END]);
            }
            return 0;
}
```

Named Pipes

Named Pipes (also known as a **FIFO):**

- Ordinary pipes provide a simple mechanism for allowing a pair of processes to communicate.
  - However, ordinary pipes **exist only** while the processes are communicating with one another.
- Once the processes have finished communicating and have terminated,
  - the ordinary pipe closes.
- But, **named pipes** can remain **open** and provide a 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.
- A named pipe can have several writers.
- Additionally, named pipes continue to exist after communicating processes have finished.
- The **mkfifo** command lets you create such named pipes in Linux

Named pipe - writer

```
// C program to implement one side of FIFO
// This side writes first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
                int fd;
                // FIFO file path
                char * myfifo = "/tmp/myfifo";

                // Creating the named file(FIFO)
                // mkfifo(<pathname>, <permission>)
                mkfifo(myfifo, 0666);
                char arr1[80], arr2[80];
                while (1)         {
                                // Open FIFO for write only
                                fd = open(myfifo, O_WRONLY);

                                // Take an input arr2ing from user.
                                // 80 is maximum length
                                fgets(arr2, 80, stdin);

                                // Write the input arr2ing on FIFO
                                // and close it
                                write(fd, arr2, strlen(arr2)+1);
                                close(fd);

                                // Open FIFO for Read only
                                fd = open(myfifo, O_RDONLY);

                                // Read from FIFO
                                read(fd, arr1, sizeof(arr1));

                                // Print the read message
                                printf("User2: %s\n", arr1);
                                close(fd);
                }
return 0; }
```

Named Pipe - reader

```
// C program to implement one side of FIFO
// This side reads first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
                int fd1;
                // FIFO file path
                char * myfifo = "/tmp/myfifo";

                // Creating the named file(FIFO)
                // mkfifo(<pathname>,<permission>)
                mkfifo(myfifo, 0666);
                char str1[80], str2[80];
                while (1)
                {
                                // First open in read only and read
                                fd1 = open(myfifo,O_RDONLY);
                                read(fd1, str1, 80);

                                // Print the read string and close
                                printf("User1: %s\n", str1);
                                close(fd1);

                                // Now open in write mode and write
                                // string taken from user.
                                fd1 = open(myfifo,O_WRONLY);
                                fgets(str2, 80, stdin);
                                write(fd1, str2, strlen(str2)+1);
                                close(fd1);
                }
return 0; }
```

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, the UNIX **ls** command produces a directory listing.
- For especially long directory listings, the output may scroll through several screens.
- The command **less** manages output by displaying only one screen of output at a time where the user may use certain keys to move forward or backward in the file.
- Setting up a pipe between the **ls** and **less** commands 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. The complete command is

  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

- Since the beginning of the lecture, we described how processes can communicate using shared memory and message passing.
- These techniques can be used for communication in client–server systems as well.
- Three other strategies for communication in client–server systems:
  - sockets,
  - remote procedure calls (RPCs), and
  - pipes.

Sockets

- A **socket** is defined as an endpoint for communication.
- A pair of processes communicating over a network employs a pair of sockets—one for each process.
- A socket is identified by an IP address concatenated with a **port number**.

- In general, 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 telnet, FTP, and HTTP) listen to well-known ports;
  - a telnet server listens to port 23;
  - 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 ports;*** we can use them to implement standard services.

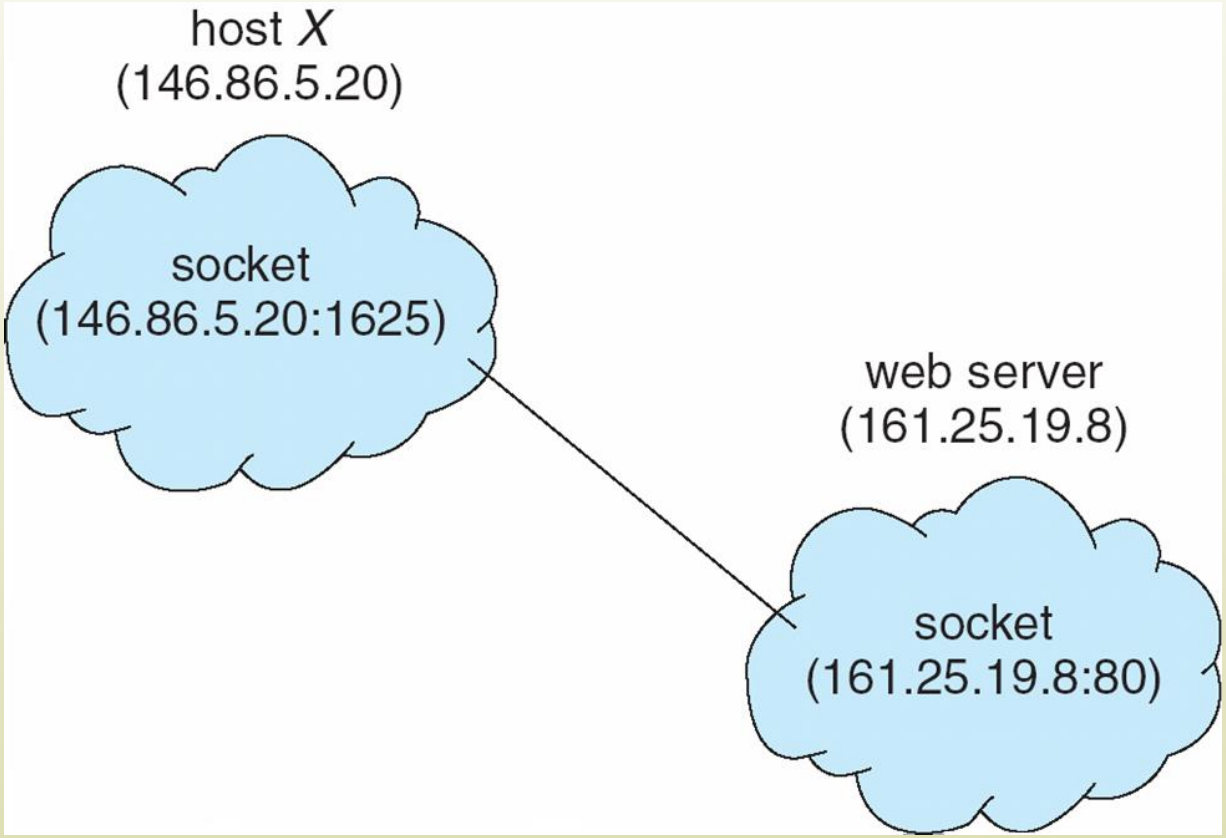
Sockets

- When a client process initiates a request for a connection, it is assigned a port by its host computer.
- This port has some arbitrary number greater than 1024.

- For example,
  - if a client on host X with IP address **146.86.5.20** wishes to establish a connection with a web server (which is listening on port 80) at address **161.25.19.8**, host X may be assigned port 1625.

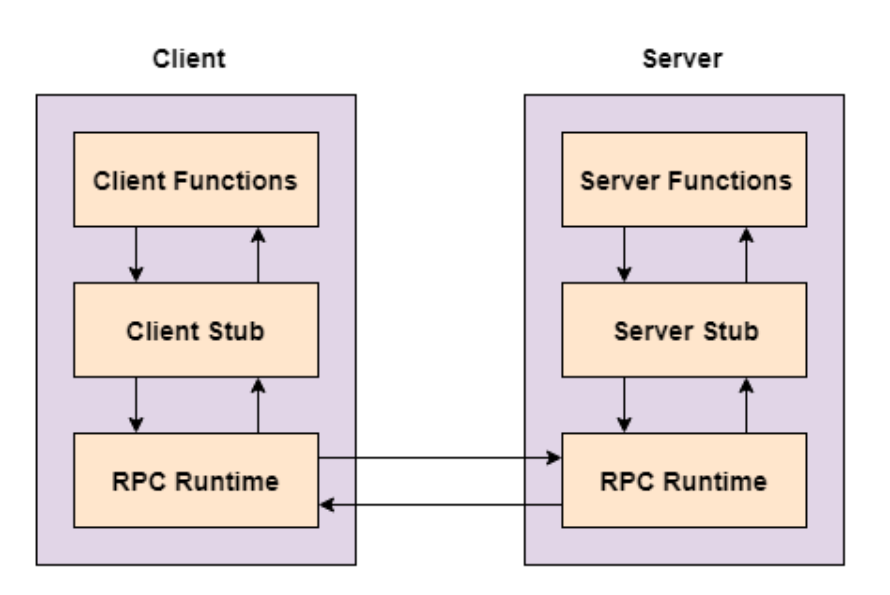
- The connection will consist of a pair of sockets: (**146.86.5.20:1625**) on host X and (**161.25.19.8:80**) on the web server.

Socket Communication



Remote Procedure Calls - RPC

- A remote procedure call is an inter-process communication technique that is used for client-server-based applications.
- A client has a request message that the RPC translates and sends to the server.
- This request may be a procedure or a function call to a remote server.
- When the server receives the request, it sends the required response back to the client.
- The client is blocked while the server is processing the call and only resumed execution after the server is finished.



End of Chapter 3