

# Chapter 3: Processes

---





# Chapter 3: Processes

---

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess 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 interprocess communication using shared memory and message passing
- To describe communication in client-server systems





# Process Concept

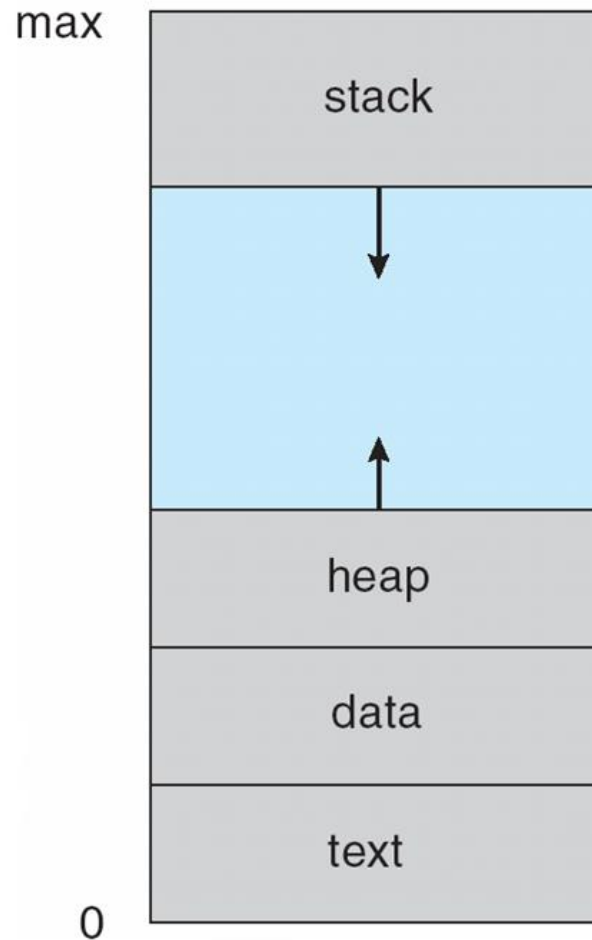
---

- ❑ An operating system executes a variety of programs:
  - ❑ Batch system – **jobs**
  - ❑ Time-shared systems – **user programs** or **tasks**
- ❑ **Process** – a program in execution; process execution must progress in sequential fashion
- ❑ The memory layout of a process is typically divided into multiple sections. These sections include:
  - ❑ **Text section** - the executable code
  - ❑ Current activity including **program counter**, processor registers
  - ❑ **Stack** - temporary data storage
  - ❑ **Data section** - global variables
  - ❑ **Heap** - memory that is dynamically allocated during program run time





# Process in Memory



***Layout of Process in memory***





# Process Concept (Cont.)

- ❑ Notice that the sizes of the text and data sections are fixed, as their sizes do not change during program run time.
- ❑ However, the stack and heap sections can shrink and grow dynamically during program execution.
- ❑ Each time a function is called, an activation record containing function parameters, local variables, and the return address is pushed onto the stack; when control is returned from the function, the activation record is popped from the stack.
- ❑ Similarly, the heap will grow as memory is dynamically allocated and will shrink when memory is returned to the system.
- ❑ Although the stack and heap sections grow toward one another, the operating system must ensure they do not overlap one another.





# Process Concept (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, etc
- One program can be several processes
  - 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.





# Process State

---

□ Go to Book Page 108

## MEMORY LAYOUT OF A C PROGRAM







# Process State

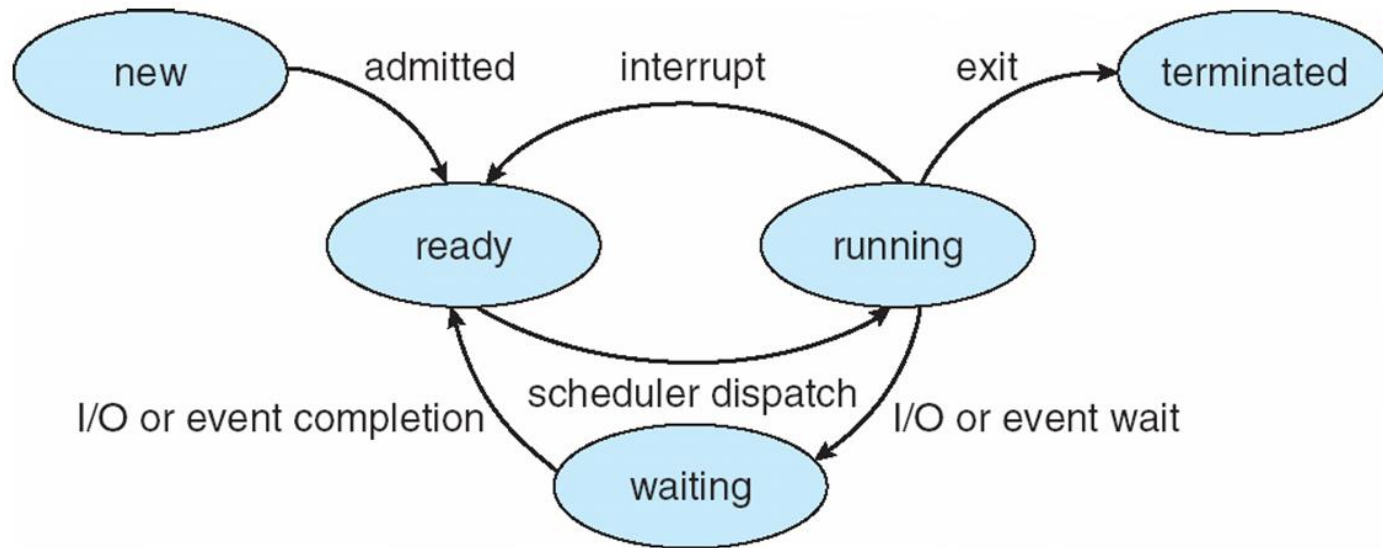
---

- 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 (such as an I/O completion or reception of a signal).
  - **Ready.** The process is waiting to be assigned to a processor





# Diagram of Process State

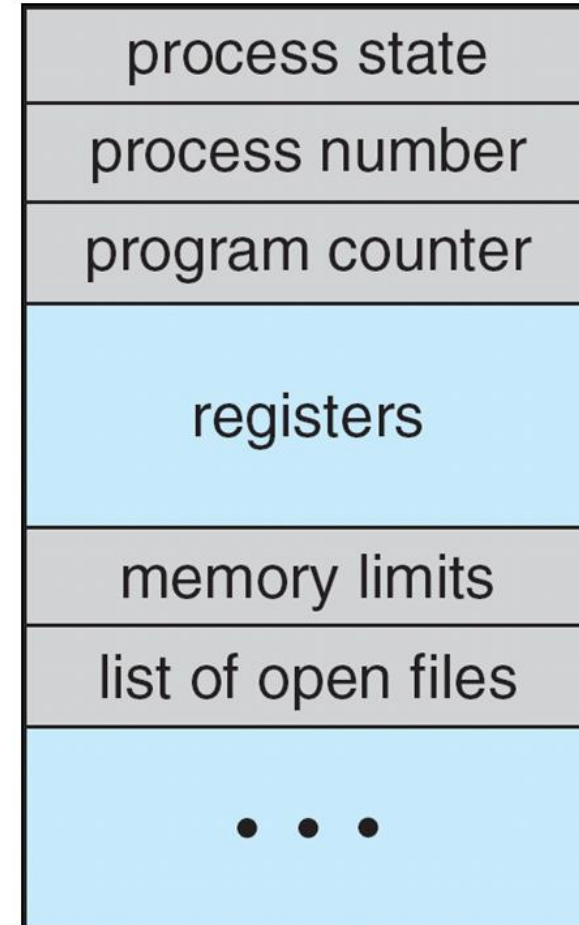




# Process Control Block (PCB)

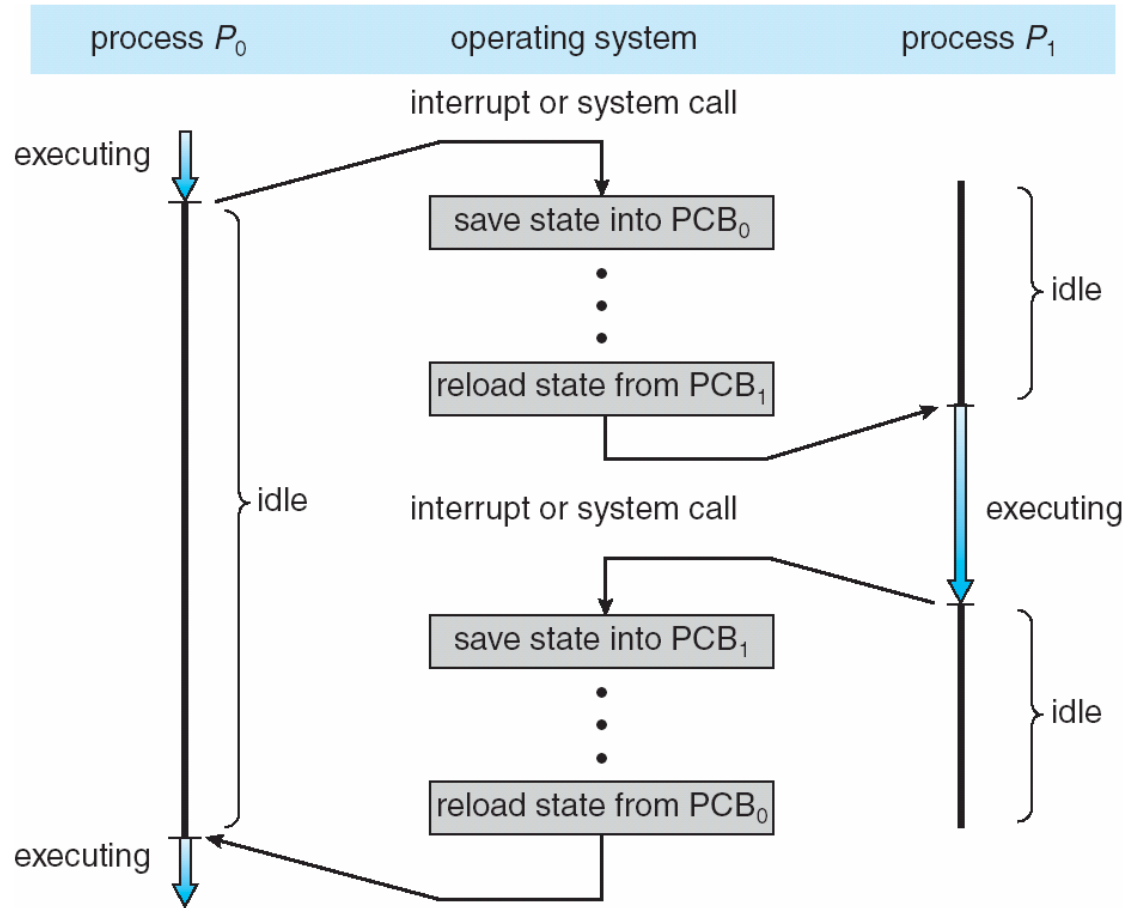
Each process is represented in the operating system by a process control block (PCB)—(also called **task control block**). Contain information associated with specific process, Including:

- ❑ **Process state** – new, ready, running, waiting, etc
- ❑ **Program counter** – location of instruction to next execute
- ❑ **CPU registers** – General purpose, accumulator etc
- ❑ **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





# Process Scheduling

---

- The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization.
- The 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.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on a core.
- Each CPU core can run one process at a time.





# Process Scheduling

---

- **Process scheduler** selects among available processes for next execution on CPU
- I/O bound processes
- CPU bound processes

## Scheduling Queues

- 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





# Process Scheduling

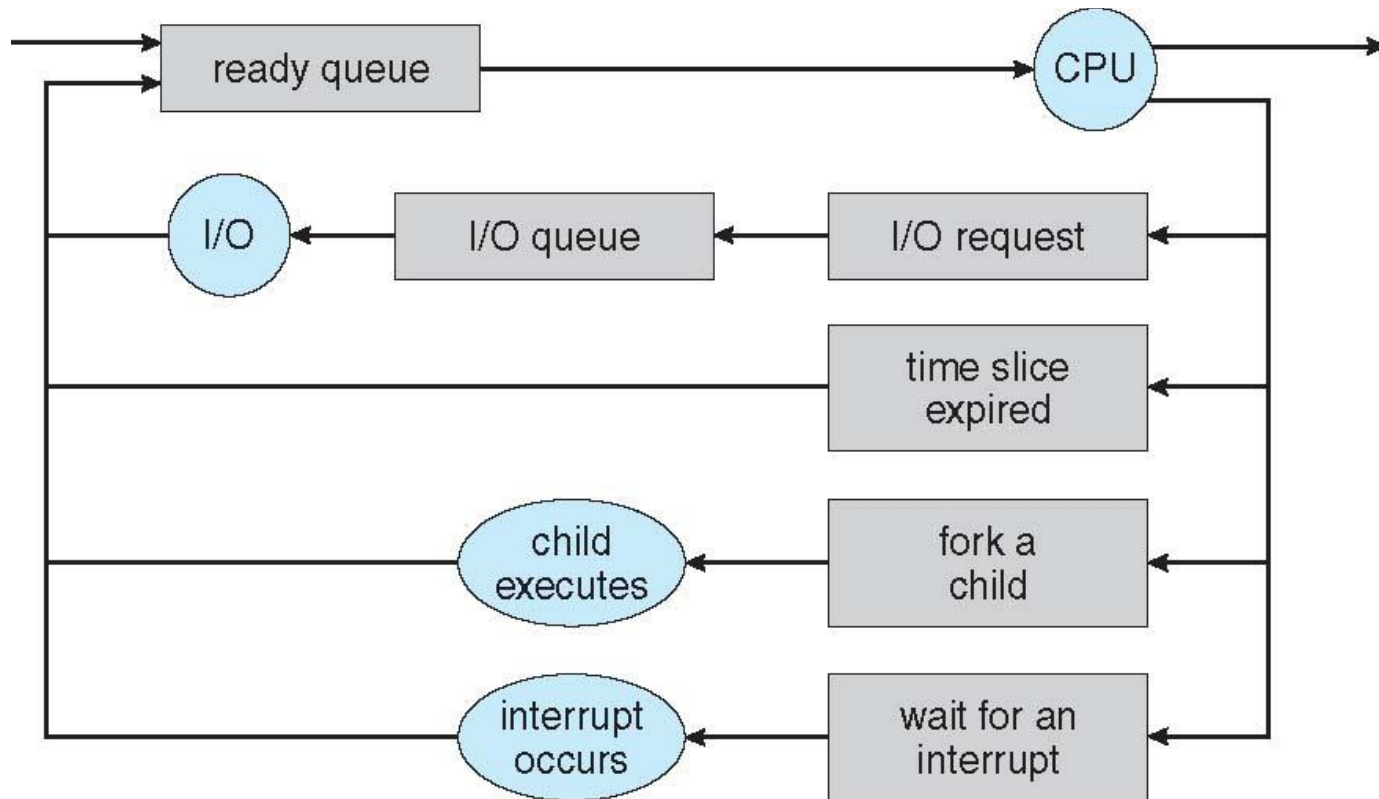
- A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. Once the process is allocated a CPU core and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O wait queue.
  - 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 an interrupt or having its time slice expire, and be put back in the ready queue.





# Representation of Process Scheduling

- Queueing diagram represents queues, resources, flows







# Schedulers

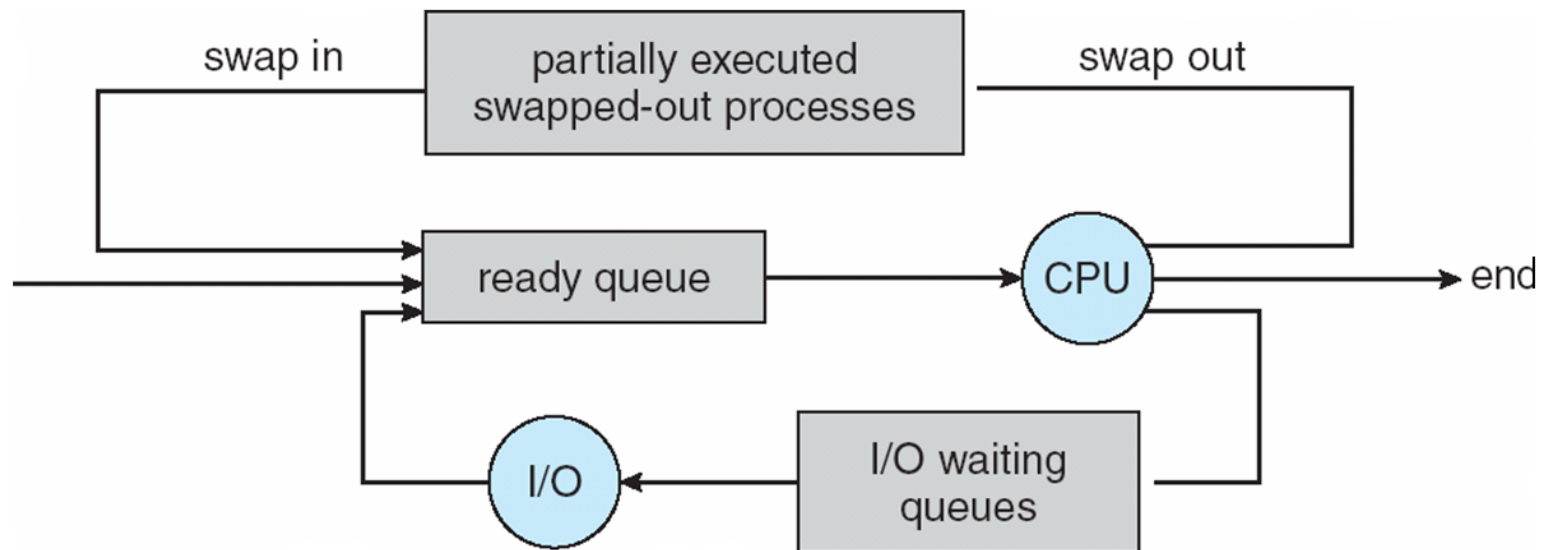
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Invoked frequently  $\Rightarrow$  (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Invoked infrequently  $\Rightarrow$  (may be slow)
  - 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 strives for good ***process mix***





# 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**





# 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,
  - and so on as detailed next





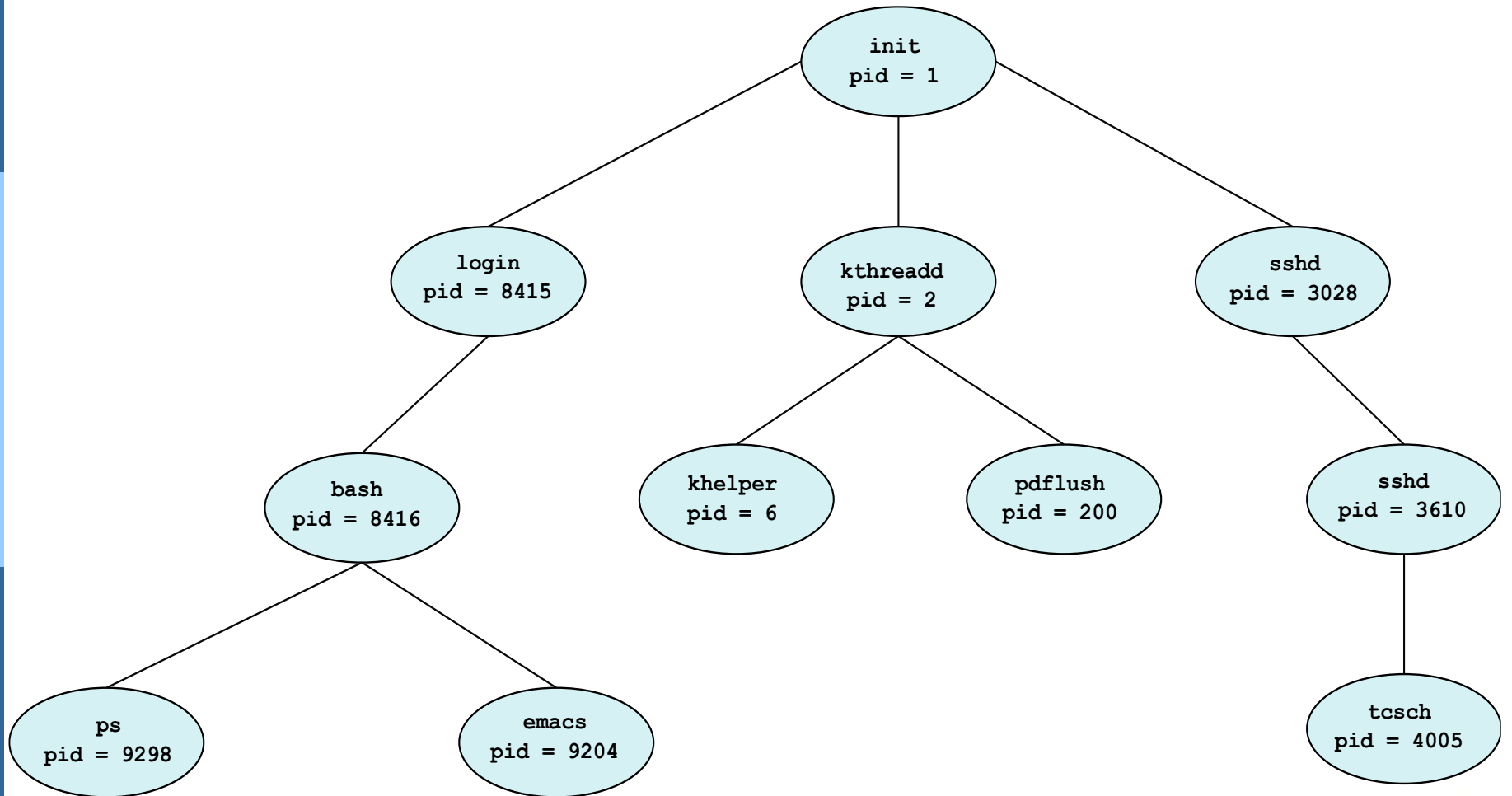
# Process Creation

- ❑ **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
  - ❑ Parent and children share all resources
  - ❑ Children share subset of parent's resources
  - ❑ Parent and child share no resources
- ❑ Execution options
  - ❑ Parent and children execute concurrently
  - ❑ Parent waits until children terminate





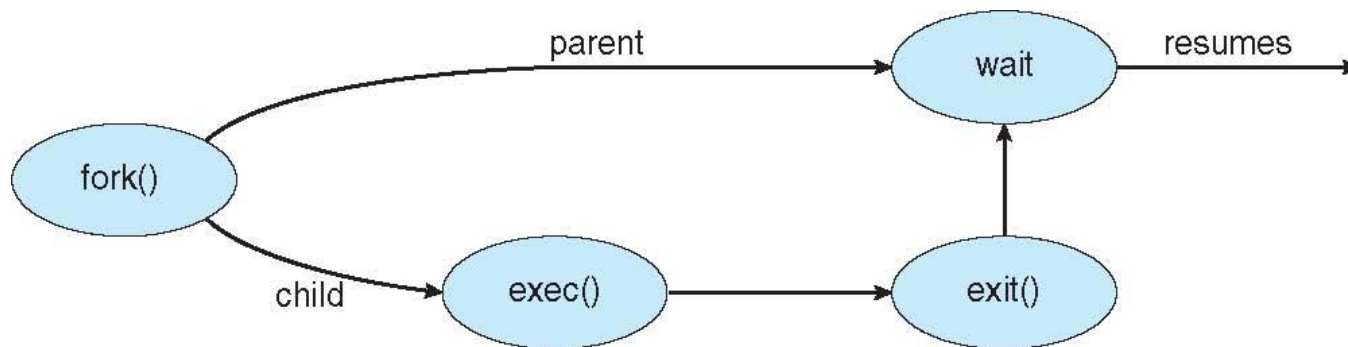
# A Tree of Processes in Linux





# 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





# 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;
}
```





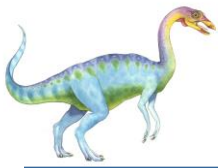


# Process Termination

---

- 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

- Some operating systems do not allow child to exist 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**





# Interprocess Communication

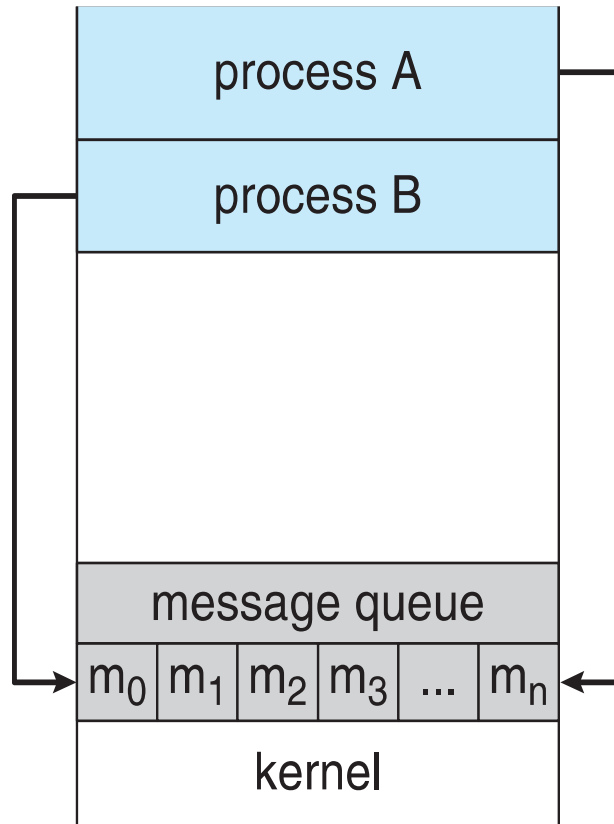
- ❑ Processes within a system may be *independent* or *cooperating*
- ❑ Cooperating process can affect or be affected by other processes, including sharing data
- ❑ Reasons for cooperating processes:
  - ❑ Information sharing
  - ❑ Computation speedup
  - ❑ Modularity
  - ❑ Convenience
- ❑ Cooperating processes need **interprocess communication (IPC)**
- ❑ Two models of IPC
  - ❑ **Shared memory**
  - ❑ **Message passing**





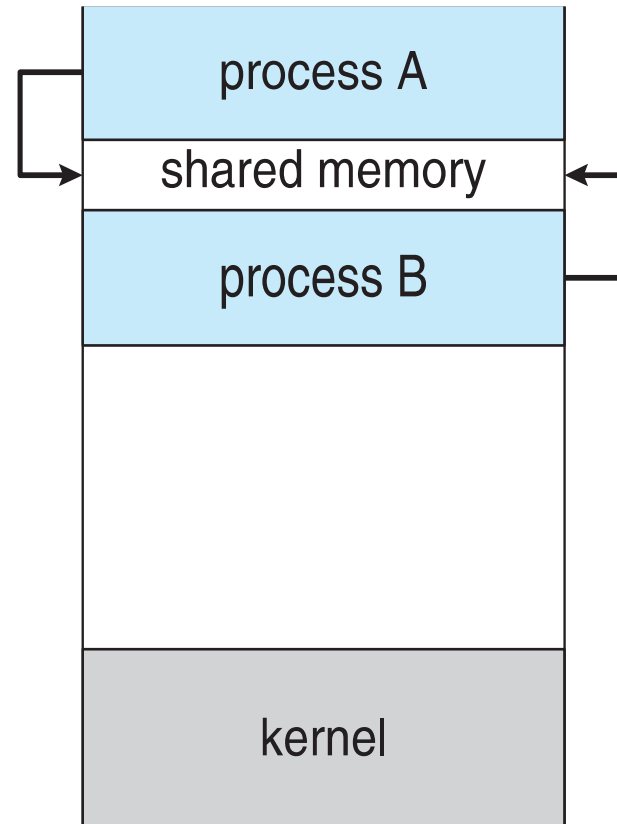
# Communications Models

(a) Message passing.



(a)

(b) shared memory.



(b)





# Cooperating Processes

---

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience





# Producer-Consumer Problem

---

- 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





# Bounded-Buffer – 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

---

```
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 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 Chapter 5.





# Interprocess 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?





# Message Passing (Cont.)

---

- Implementation of communication link
  - Physical:
    - ▶ Shared memory
    - ▶ Hardware bus
    - ▶ Network
  - Logical:
    - ▶ Direct or indirect
    - ▶ Synchronous or asynchronous
    - ▶ Automatic or explicit buffering





# 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

---

- Operations
  - create a new mailbox (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

- 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**





# Synchronization (Cont.)

---

## □ Producer-consumer becomes trivial

```
message next_produced;  
while (true) {  
    /* produce an item in next produced */  
    send(next_produced);  
}
```

```
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
  1. Zero capacity – no messages are queued on a link.  
Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of  $n$  messages  
Sender must wait if link full
  3. Unbounded capacity – infinite length  
Sender never waits





# Communications in Client-Server Systems

---

- Sockets
- Remote Procedure Calls
- Pipes





# Sockets

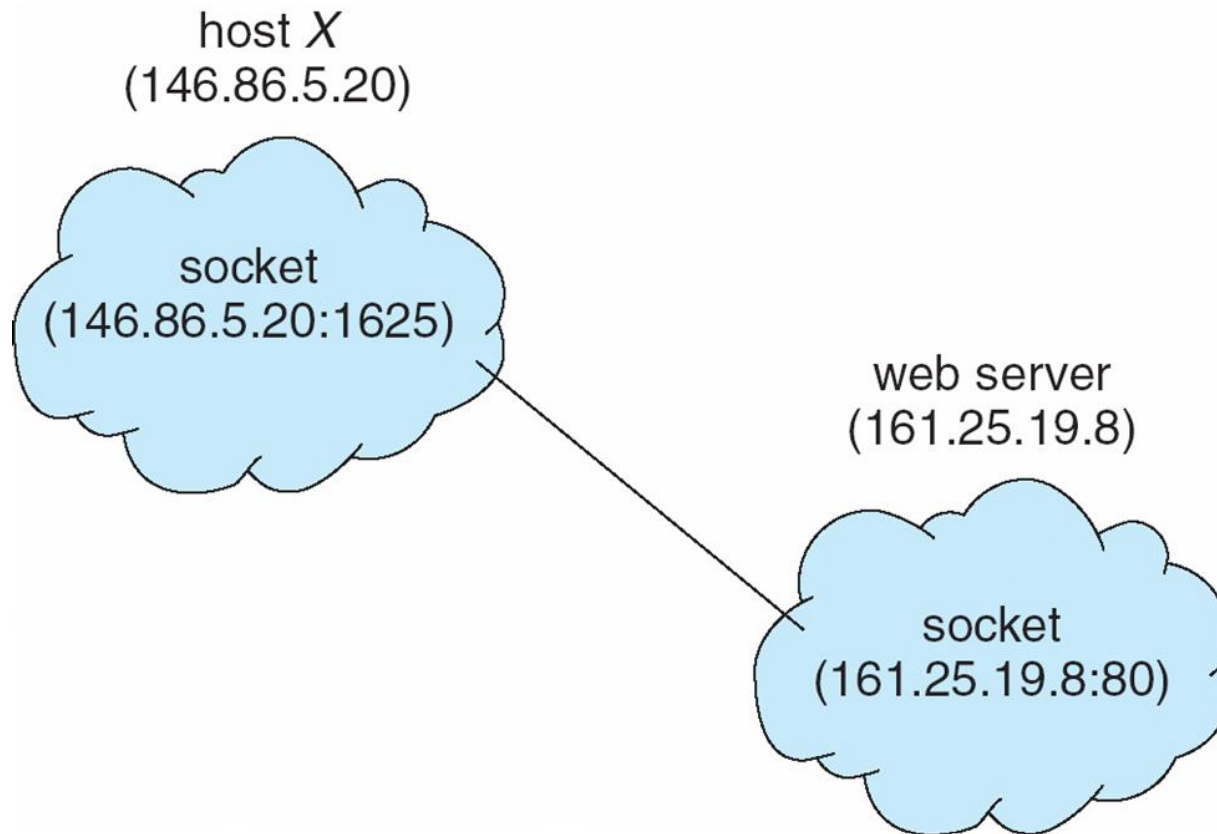
---

- 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**
- Communication consists between a pair of sockets
- All ports below 1024 are **well known**, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running





# Socket Communication





# Pipes

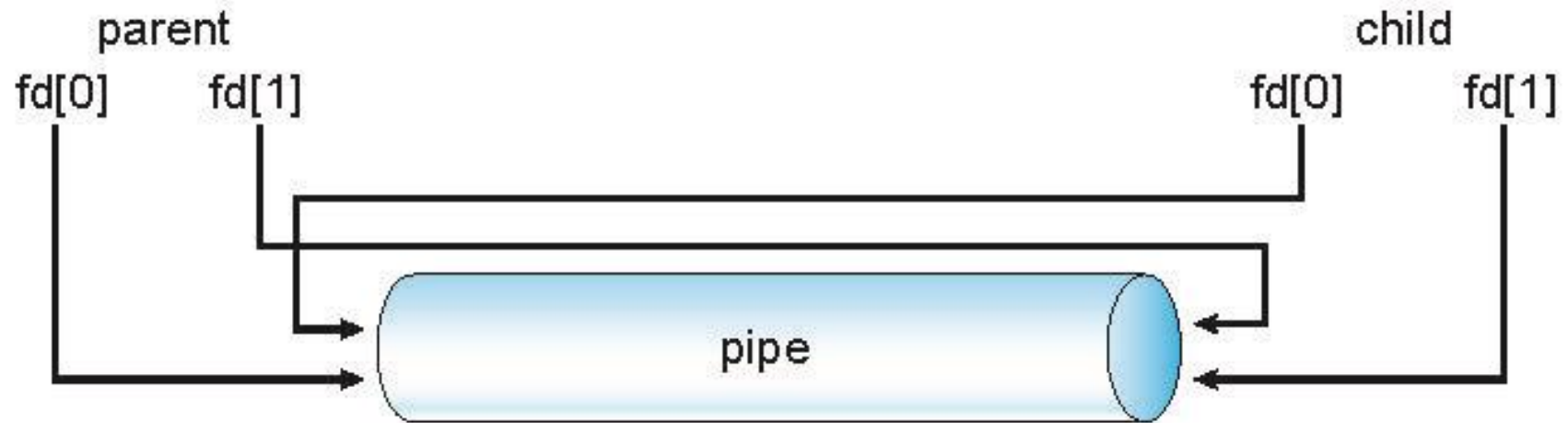
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.







# Ordinary Pipes





# Named Pipes

---

- ❑ Named Pipes are more powerful than ordinary pipes
- ❑ Communication is bidirectional
- ❑ No parent-child relationship is necessary between the communicating processes
- ❑ Several processes can use the named pipe for communication
- ❑ Provided on both UNIX and Windows systems



# End of Chapter 3

---

