

Processes

Operating Systems (CS-220)
Fall 2020, FAST NUCES

COURSE SUPERVISOR: ANAUM HAMID

anaum.hamid@nu.edu.pk

COURSE SUPERVISOR: ANAUM HAMID



ROAD MAP

- 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 and 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

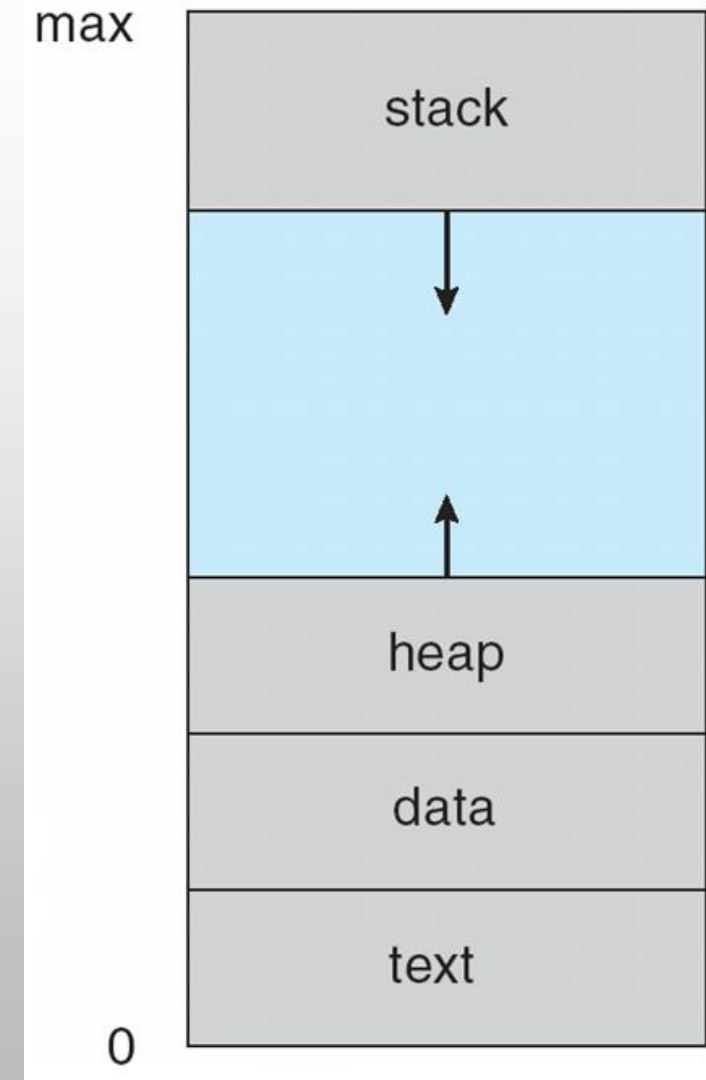
PROCESS CONCEPT

- An operating system executes a variety of programs:
 - Batch system – **jobs**
 - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** –A process is an instance of a program in execution. Process execution must progress in sequential fashion.

PROCESS CONCEPT

- 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
 - Consider multiple users executing the same program

PROCESS IN MEMORY



COURSE SUPERVISOR: ANAUM HAMID

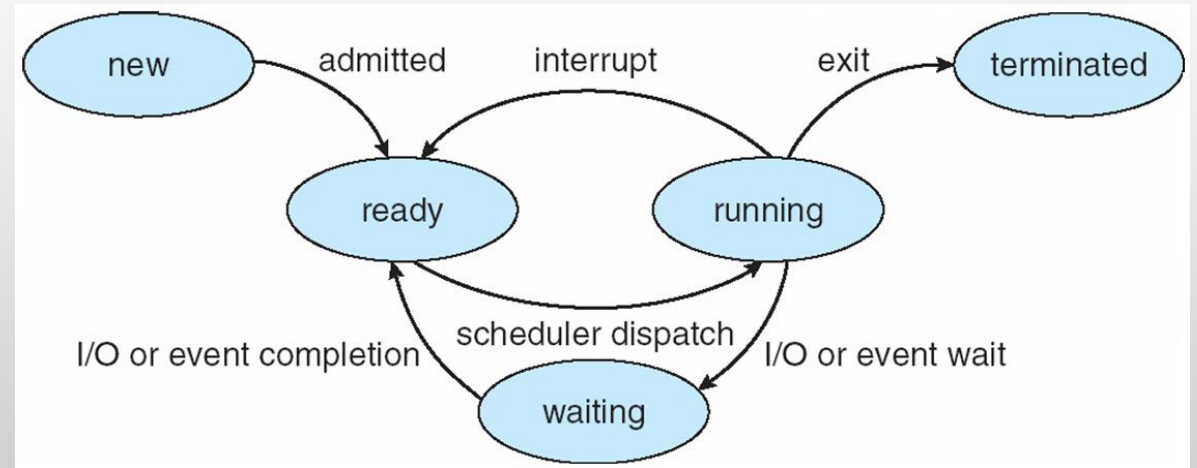
- **Text section** the text section comprises the compiled program code, read from storage when the program is launched. Current activity including **program counter**, processor registers
- **Stack** containing temporary data
 - Function parameters, return addresses, local variables.
 - Note that the stack is also used for function return values, and the exact mechanisms of stack management may be language specific.
- **Data section** containing global variables
 - The data section stores global and static variables, allocated and initialized prior to executing main.
- **Heap** containing memory dynamically allocated during run time.
 - The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.

PROCESS IN MEMORY

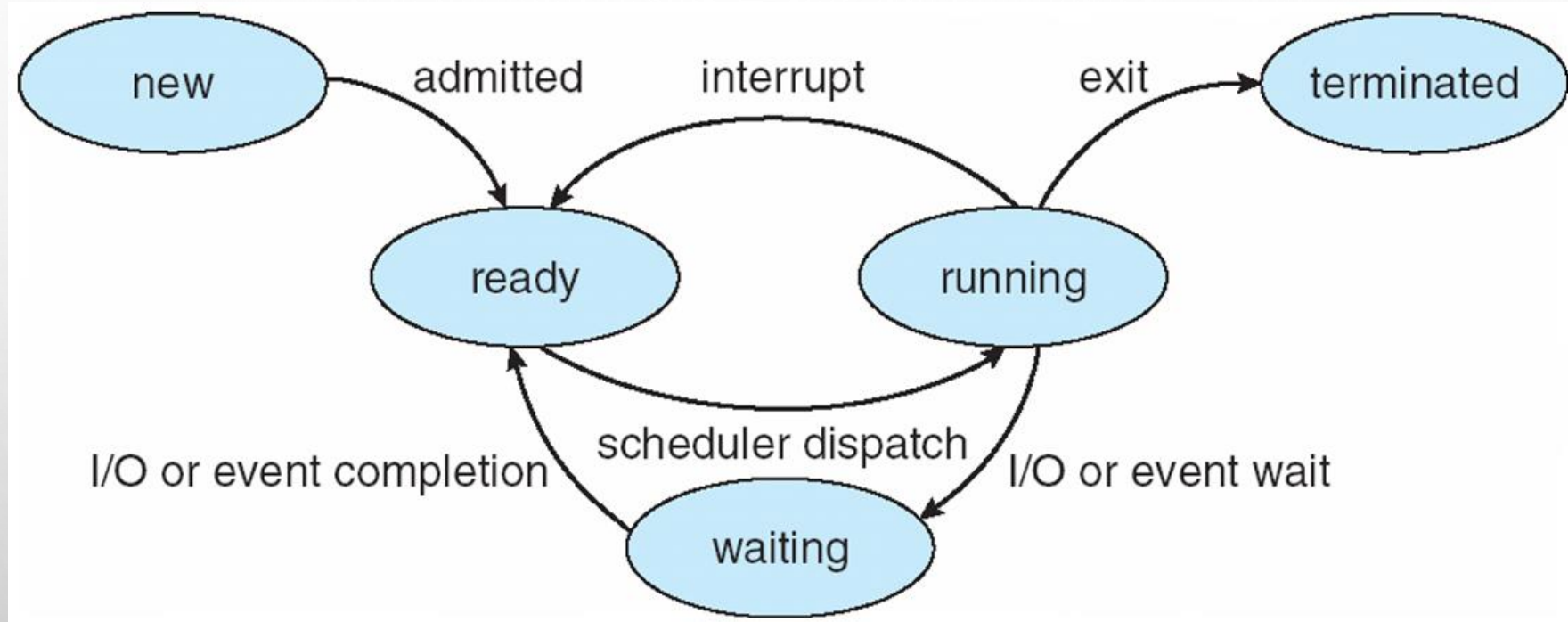
- Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.
- When processes are swapped out of memory and later restored, additional information must also be stored and restored. Key among them are the program counter and the value of all program registers.

PROCESS STATE

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



STATE TRANSITION DIAGRAM OF A PROCESS

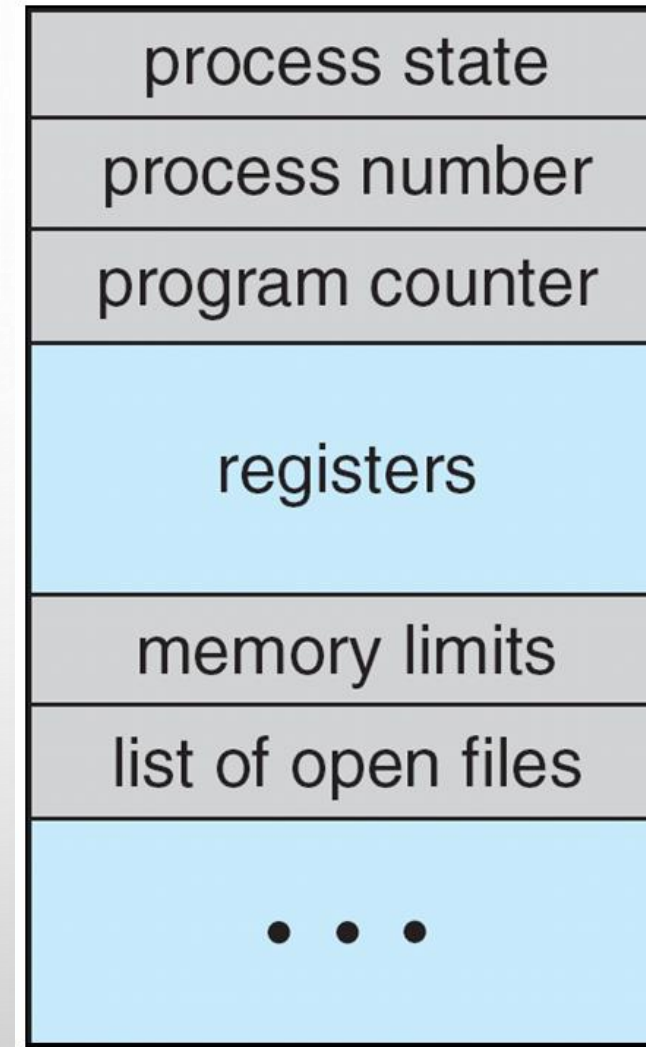


PROCESS CONTROL BLOCK (PCB)

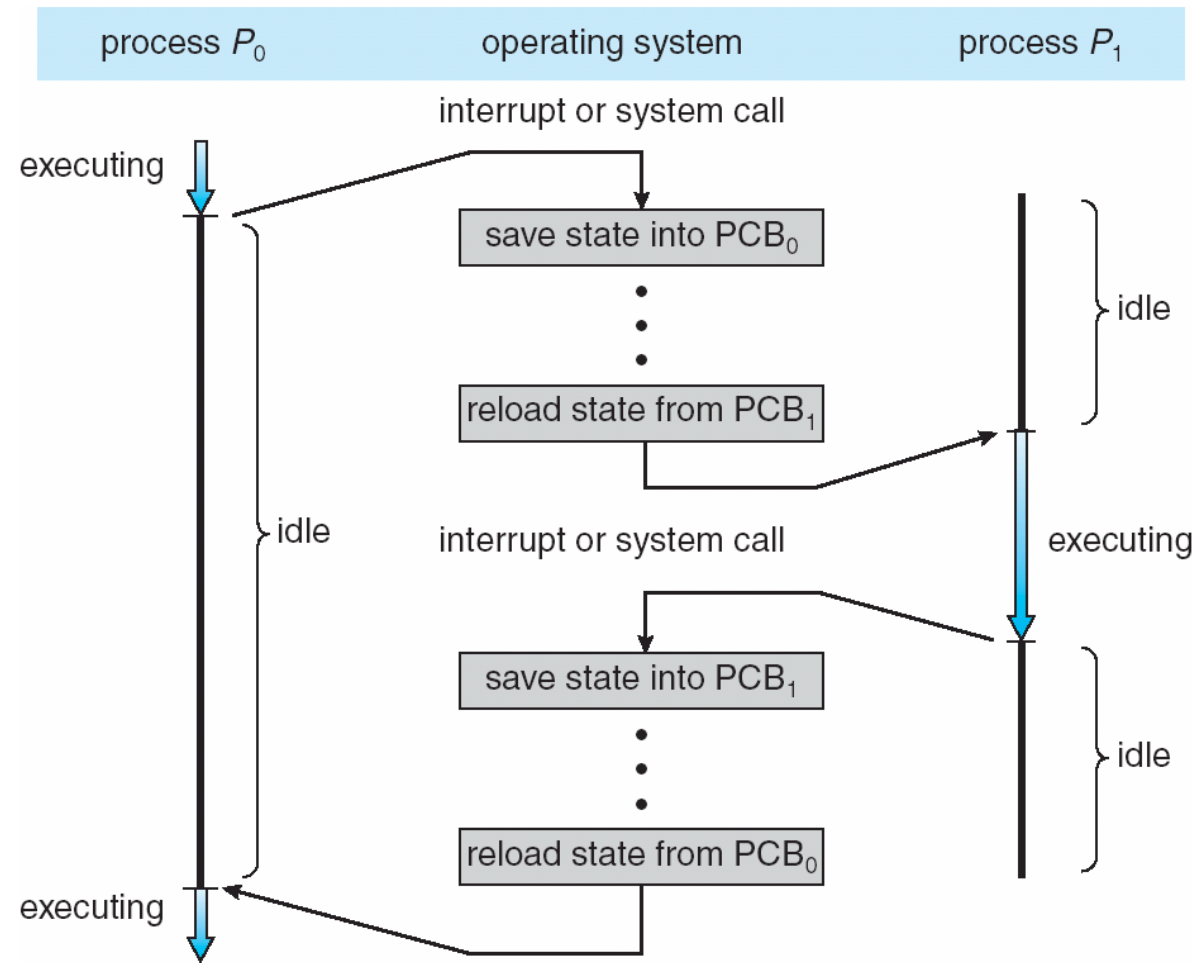
Information associated with each process

(also called **task control block**)

- **Process state** – running, waiting, etc.
- **Process ID**, and parent process ID
- **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



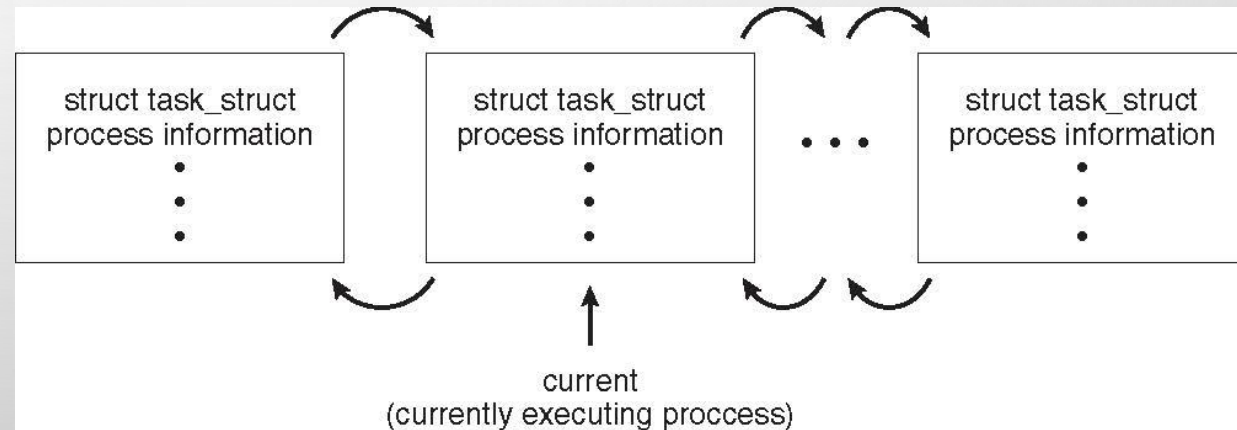
THREADS

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

PROCESS REPRESENTATION IN LINUX

REPRESENTED BY THE C STRUCTURE TASK_STRUCT

```
PID T_PID; /* PROCESS IDENTIFIER */
LONG STATE; /* STATE OF THE PROCESS */
UNSIGNED INT TIME_SLICE /* SCHEDULING INFORMATION */
STRUCT TASK_STRUCT *PARENT; /* THIS PROCESS'S PARENT */
STRUCT LIST_HEAD CHILDREN; /* THIS PROCESS'S CHILDREN */
STRUCT FILES_STRUCT *FILES; /* LIST OF OPEN FILES */
STRUCT MM_STRUCT *MM; /* ADDRESS SPACE OF THIS PROCESS */
```



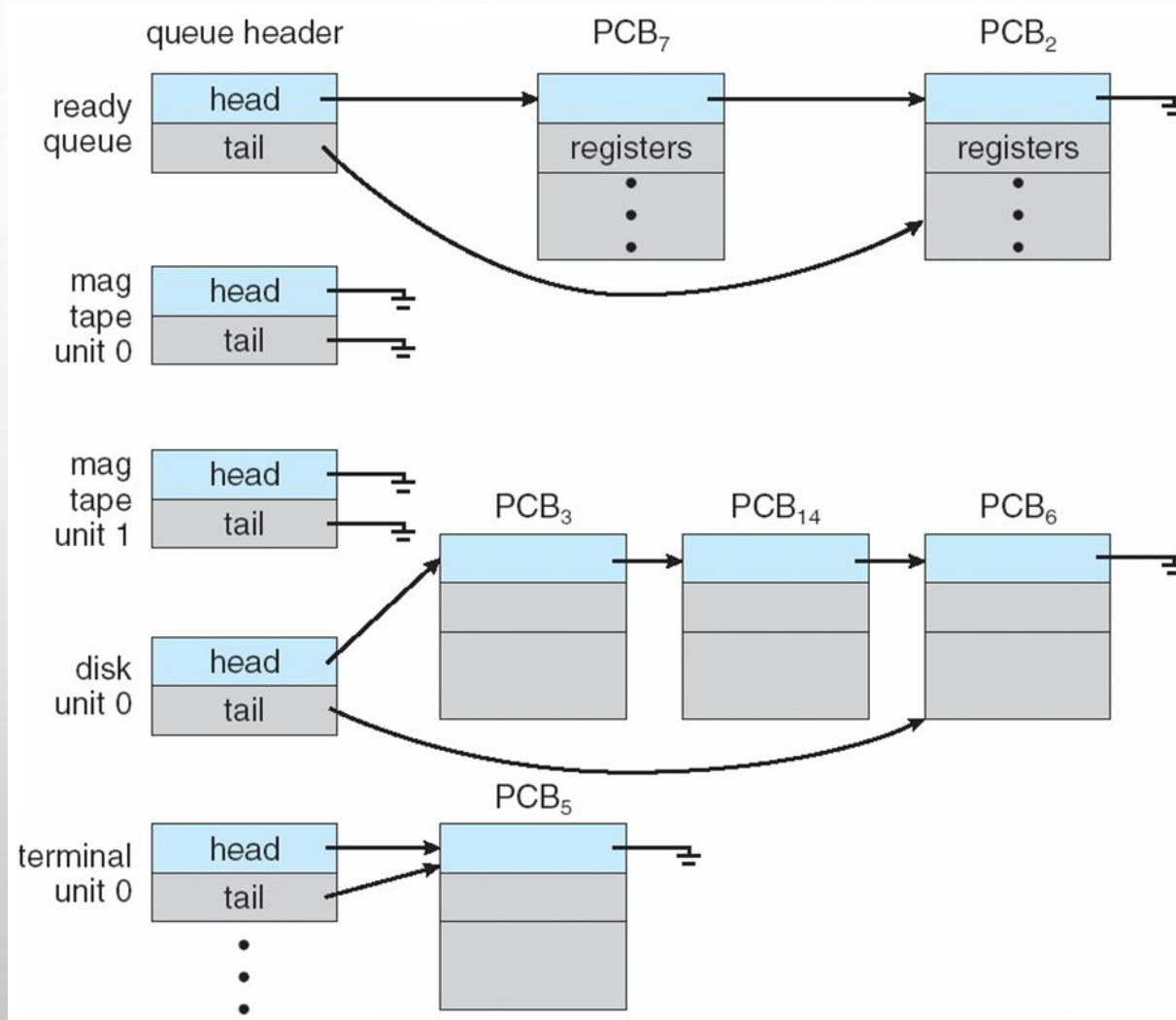
PROCESS SCHEDULING

- The two main objectives of the process scheduling system are always to keep the CPU busy and to deliver "acceptable" response times for all programs, particularly for interactive ones.
- The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU.
- 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

PROCESS SCHEDULING QUEUES

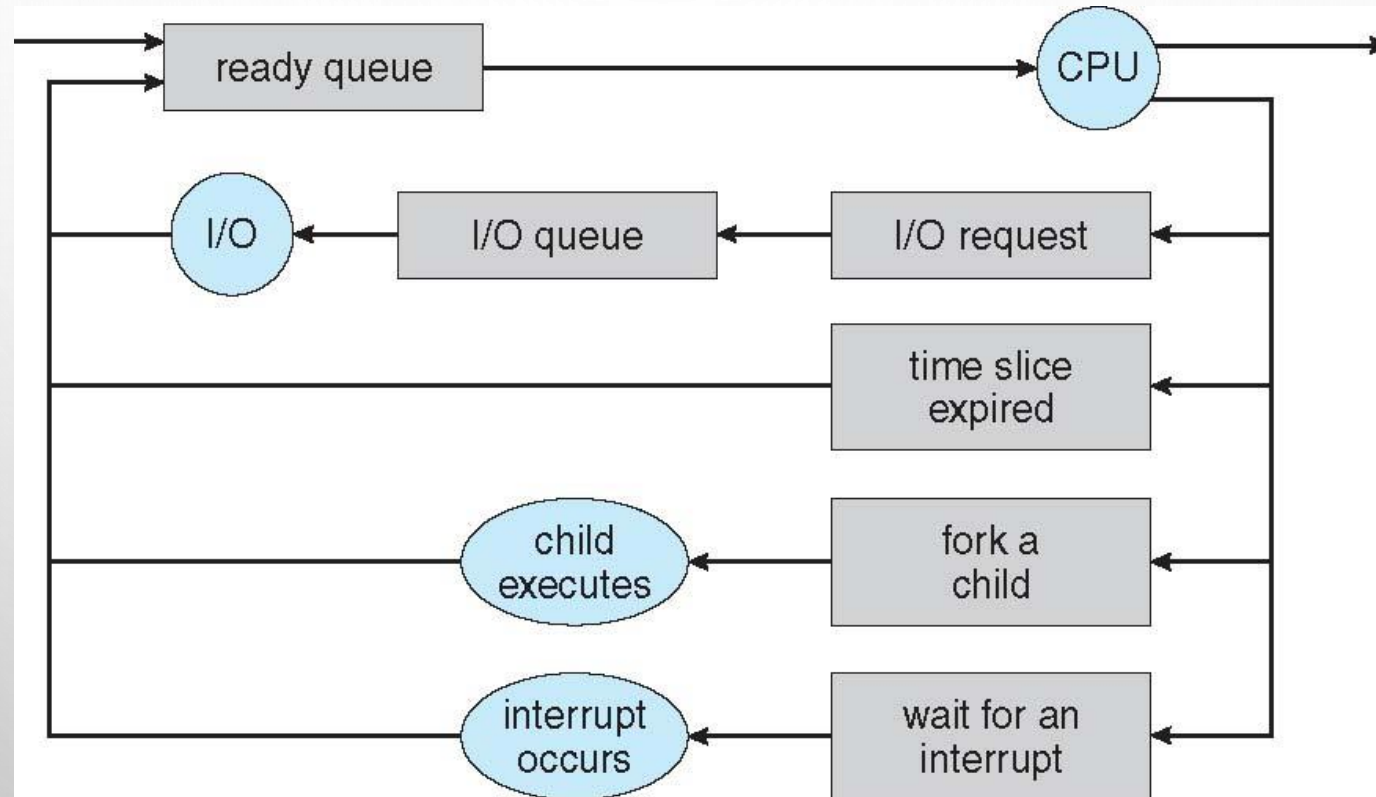
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

READY QUEUE AND VARIOUS I/O DEVICE QUEUES



REPRESENTATION OF PROCESS SCHEDULING

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



SCHEDULERS

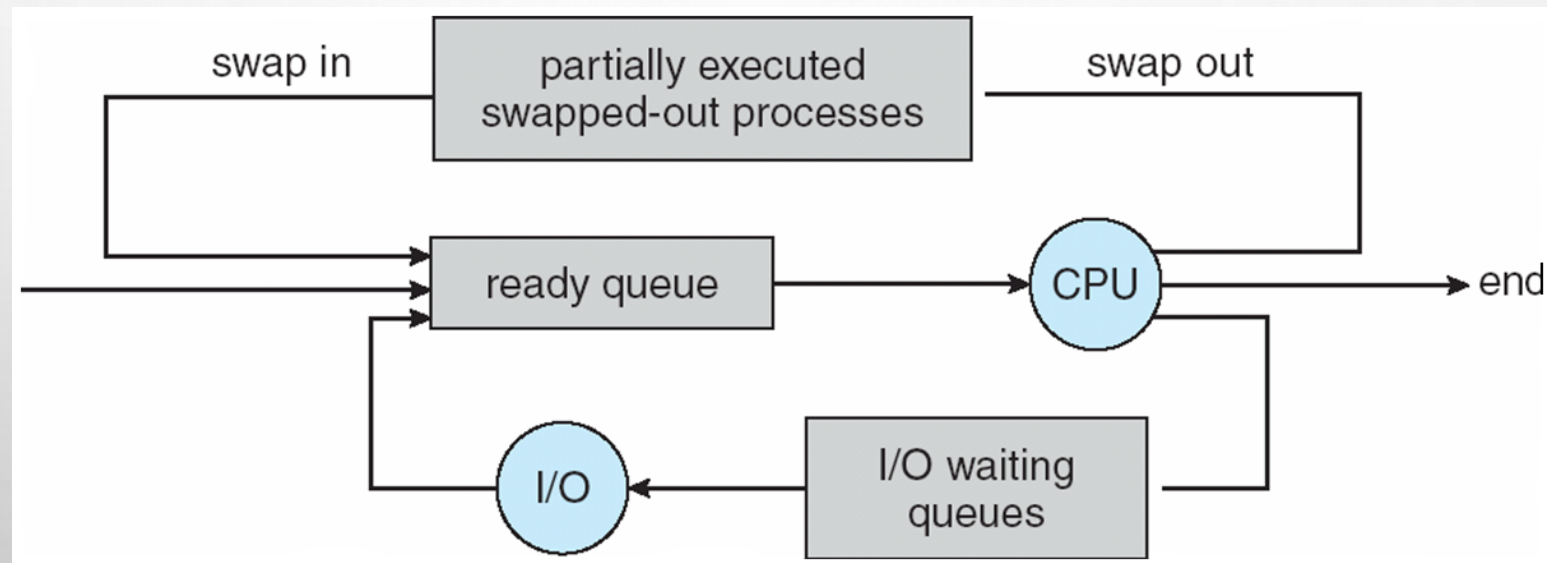
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
 - Sometimes the only scheduler in a system
 - Short-term scheduler is invoked frequently (milliseconds)
⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which 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**

SCHEDULERS

- 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**



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

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

COURSE SUPERVISOR: ANAUM HAMID

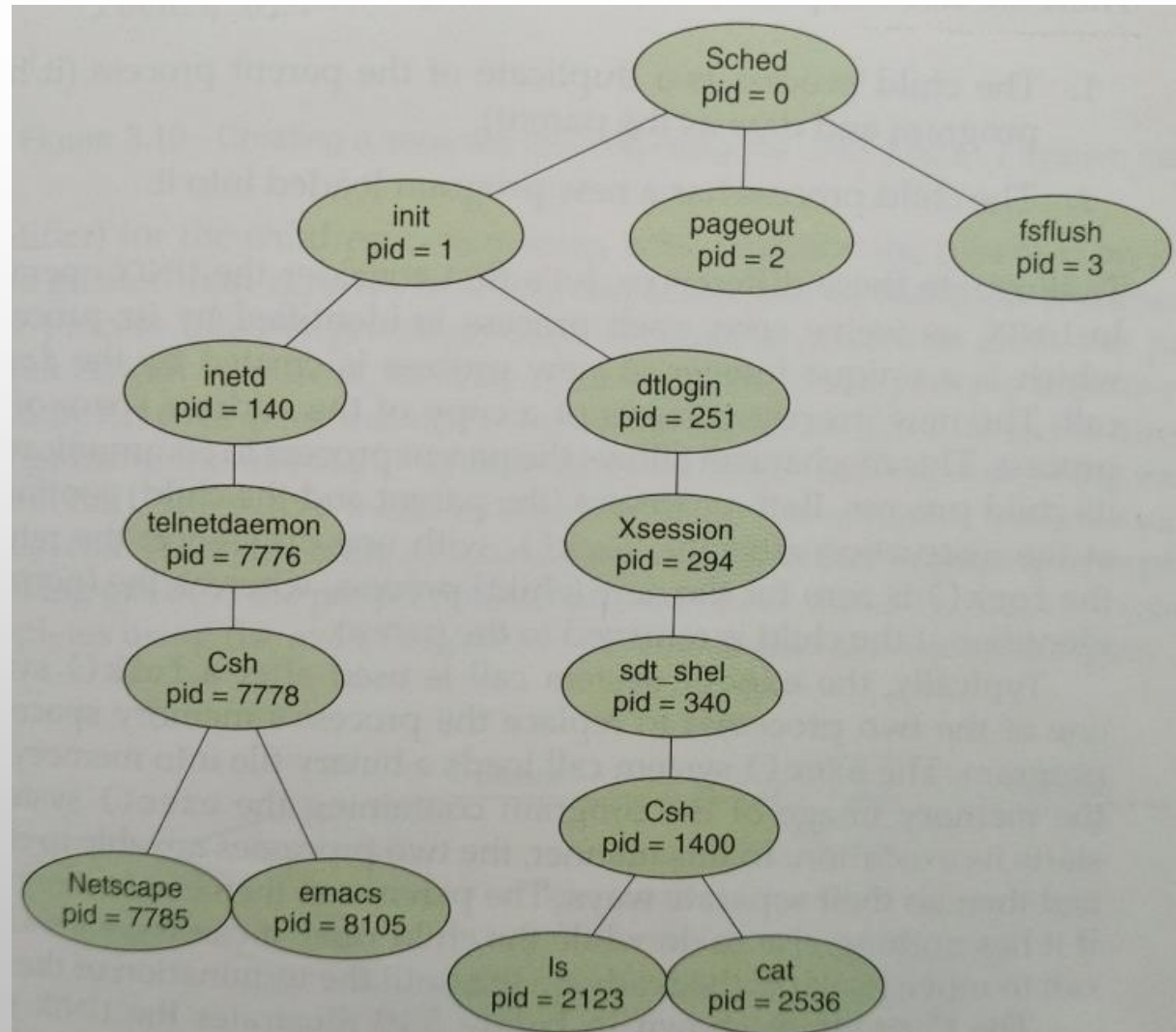
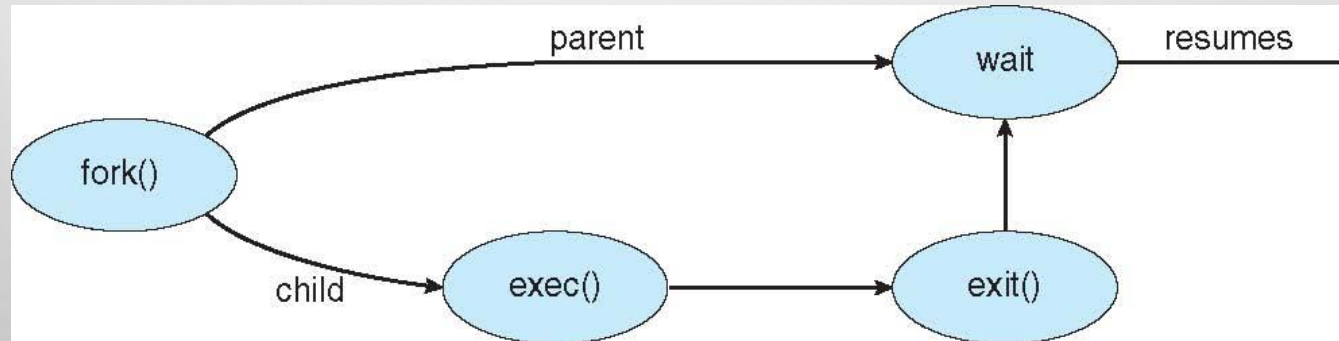


Figure 3.9 A tree of processes on a typical Solaris system.

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;
}
```

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

- 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 DE allocated 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) ;
```

PROCESS TERMINATION

- When a child process terminates, its resources are de-allocated by the operating system. 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 did not invoke `wait()` and instead terminated, thereby leaving its child processes as **ORPHANS**.

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 multiprocessing with 3 different types of processes:
 - **Browser** process manages user interface, disk and network I/O
 - **Renderer** process renders web pages, deals with HTML, JavaScript. A new renderer created for each website opened
 - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
 - **Plug-in** process for each type of plug-in



COOPERATING PROCESSES

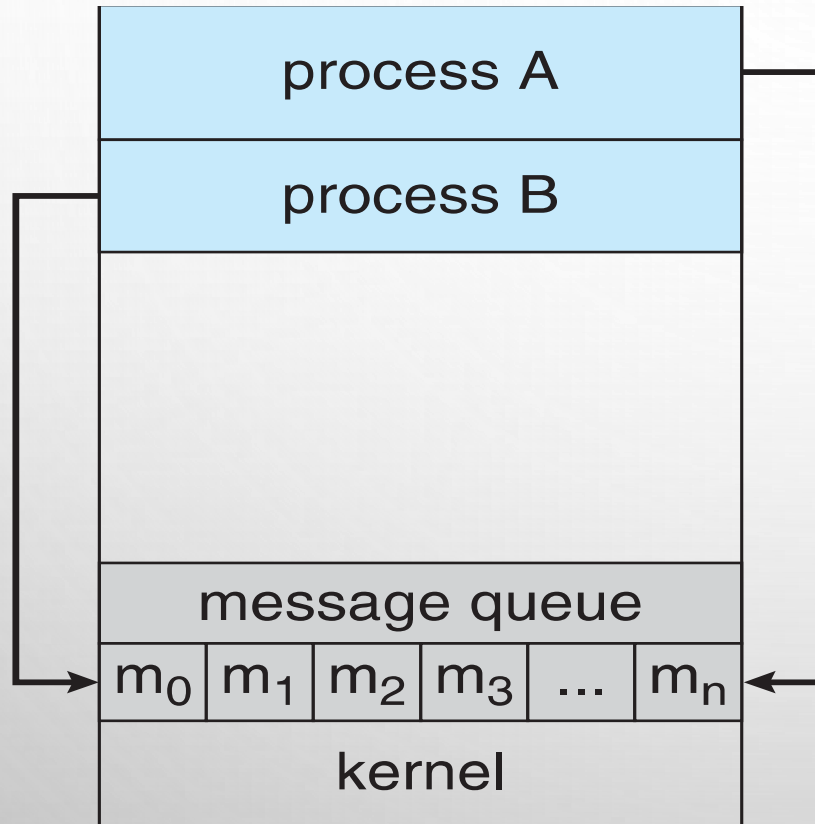
- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience

INTERPROCESS COMMUNICATION

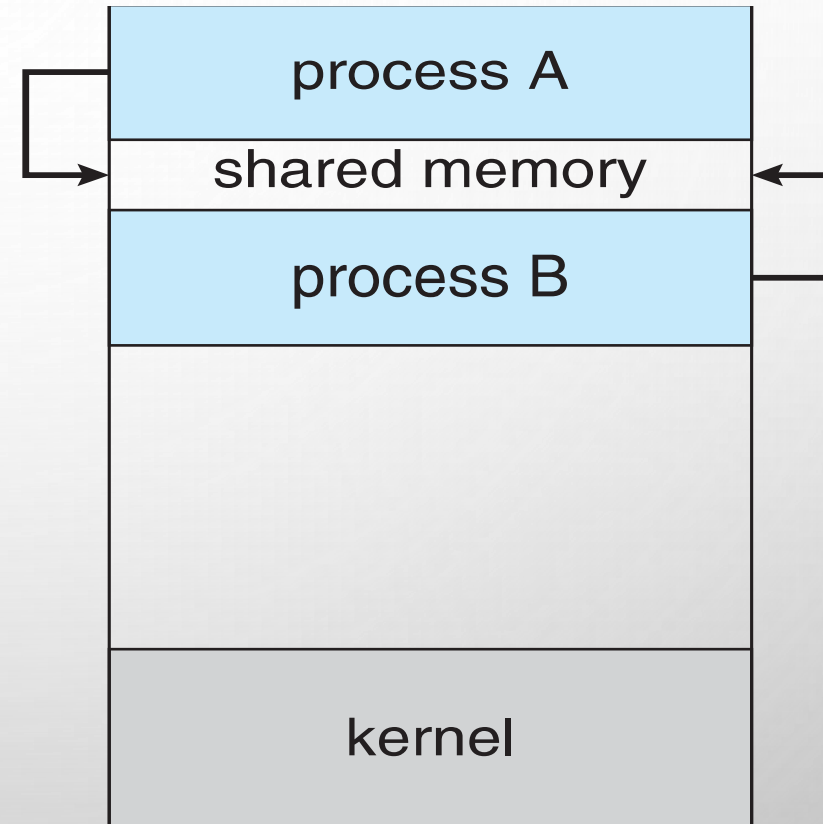
- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes need **interprocess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

COMMUNICATIONS MODELS

(a) Message passing. (b) shared memory.



(a)



(b)

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 */

}

EXAMPLES OF IPC SYSTEMS - POSIX

□ POSIX Shared Memory

- Process first creates shared memory segment

```
segment id = shmget(IPC PRIVATE, size, S  
IRUSR | S IWUSR);
```

- Process wanting access to that shared memory must attach to it
- ```
shared memory = (char *) shmat(id, NULL, 0);
```

- Now the process could write to the shared memory

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

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

```
shmdt(shared memory);
```



# INTER PROCESS COMMUNICATION – MESSAGE PASSING

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

# MESSAGE PASSING (CONT.)

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

# 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** -- the receiver receives:
    - ☐ A valid message, or
    - ☐ null message
- Different combinations possible
  - ☐ If both send and receive are blocking, we have a **rendezvous**

# 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

# EXAMPLES OF IPC SYSTEMS - MACH

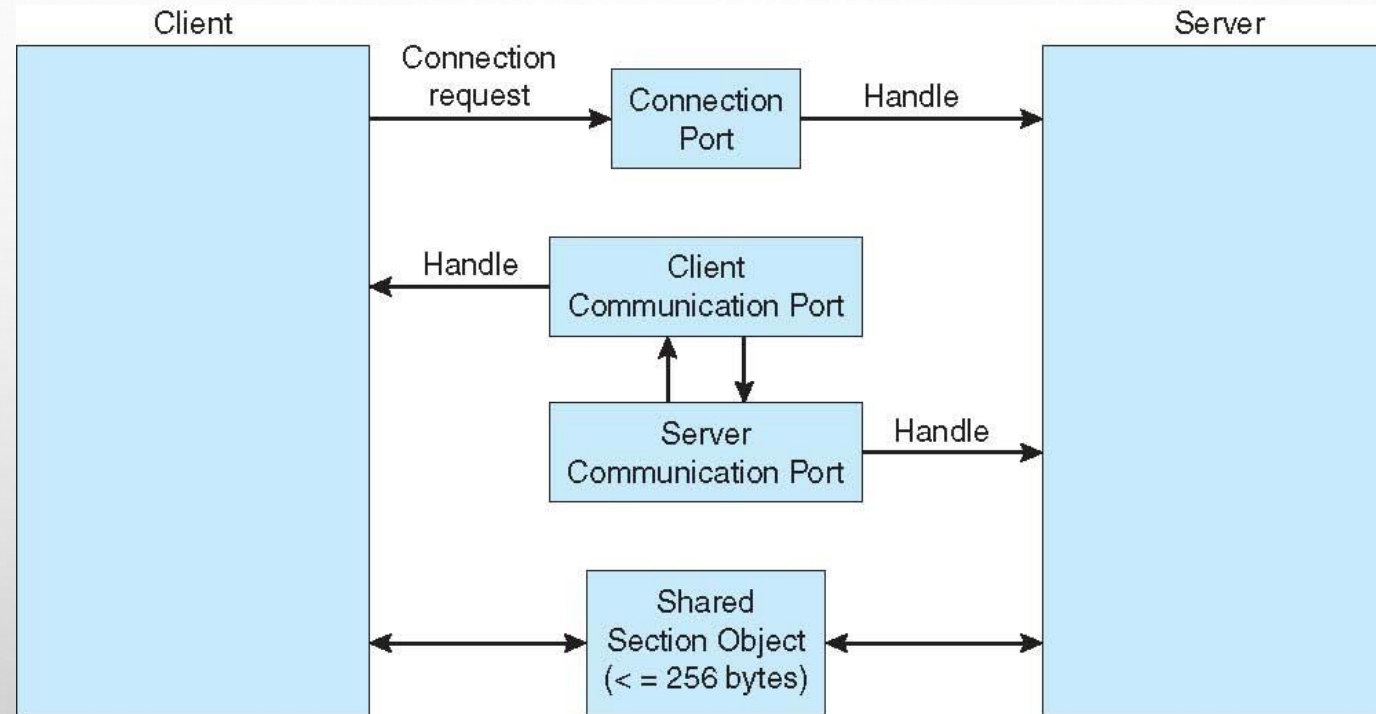
- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- kernel and notify
  - Only three system calls needed for message transfer  
**Msg\_send(), msg\_receive(), msg\_rpc()**
  - Mailboxes needed for communication , created via  
**Port\_allocate()**
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

# EXAMPLES OF IPC SYSTEMS – WINDOWS

- Message-passing centric via **advanced local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.



# LOCAL PROCEDURE CALLS IN WINDOWS



# COMMUNICATIONS IN CLIENT-SERVER SYSTEMS

1. Sockets
2. Remote procedure calls
3. 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

host X  
(146.86.5.20)

socket  
(146.86.5.20:1625)

web server  
(161.25.19.8)

socket  
(161.25.19.8:80)

# SOCKET COMMUNICATION

```

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

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

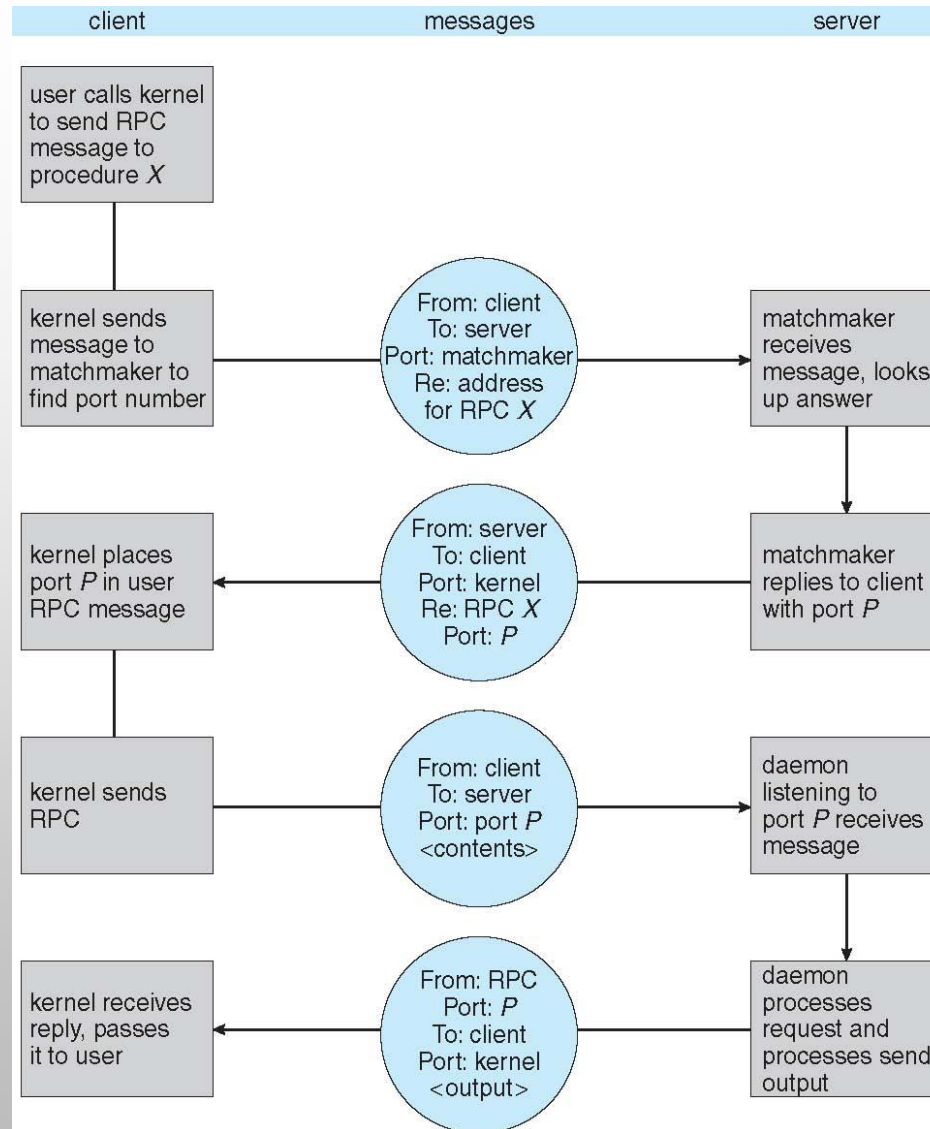
# REMOTE PROCEDURE CALLS

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **Marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On windows, stub code compile from specification written in **microsoft interface definition language (MIDL)**



# REMOTE PROCEDURE CALLS (CONT.)

- Data representation handled via **external data representation (XDL)** format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous('rändəˌvoo or **matchmaker**) service to connect client and server



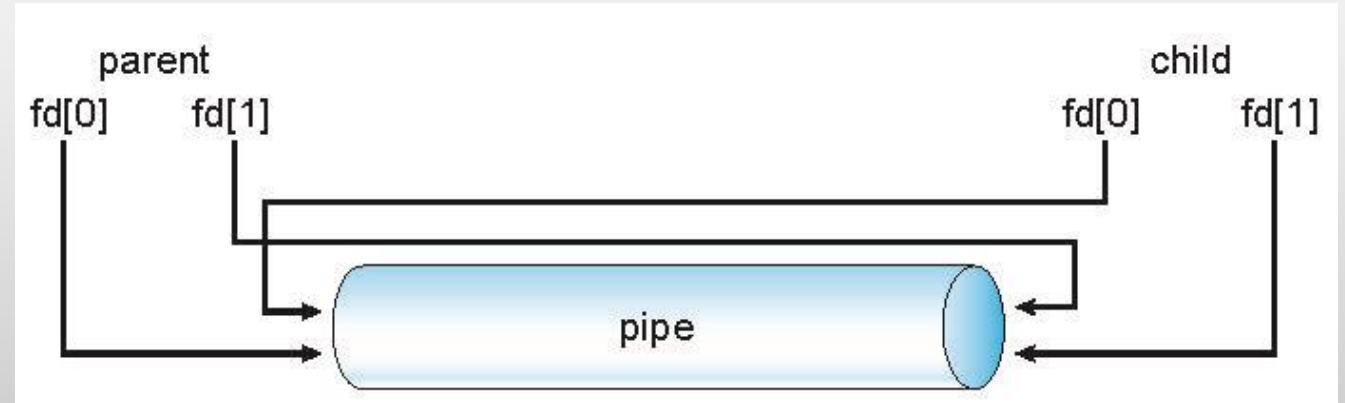
# EXECUTION OF RPC

# 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.

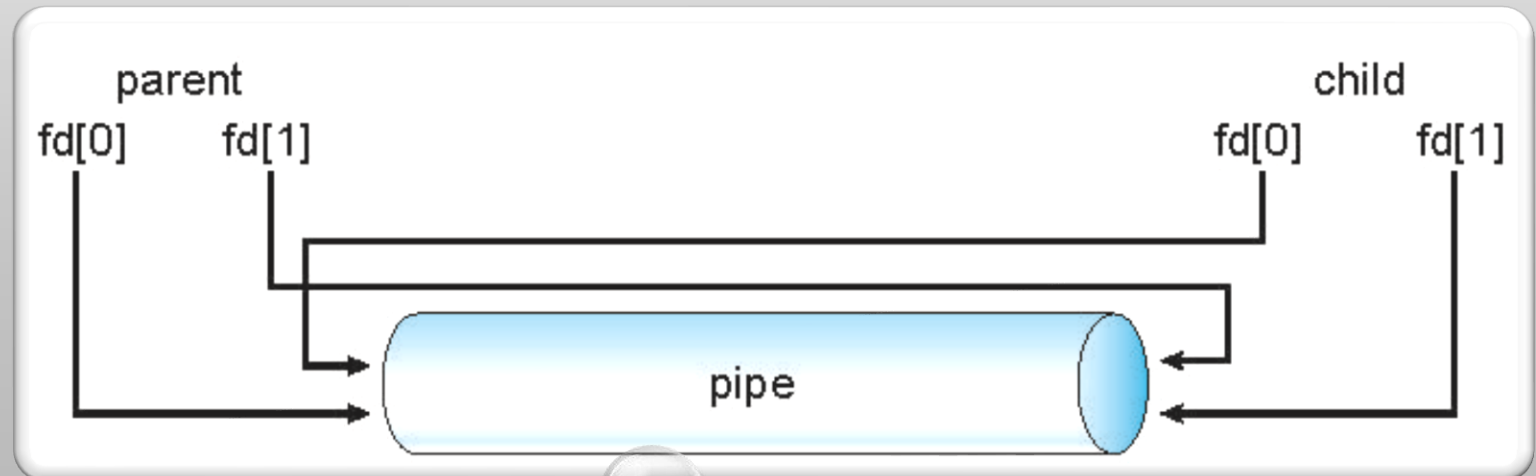
# ORDINARY PIPES

- ❑ 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 therefore unidirectional.
- ❑ Require parent-child relationship between communicating processes.



# ORDINARY PIPES

- Ordinary pipes are uni-directional, with a reading end and a writing end. ( If bidirectional communications are needed, then a second pipe is required. )
- In UNIX ordinary pipes are created with the system call "int pipe( int fd [ ])".
  - The return value is 0 on success, -1 if an error occurs.
  - The int array must be allocated before the call, and the values are filled in by the pipe system call:
    - Fd[0] is filled in with a file descriptor for the reading end of the pipe
    - Fd[1] is filled in with a file descriptor for the writing end of the pipe.



# ORDINARY PIPES

- UNIX pipes are accessible as files, using standard `read( )` and `write( )` system calls.
- Ordinary pipes are only accessible within the process that created them.
  - Typically a parent creates the pipe before forking off a child.
  - When the child inherits open files from its parent, including the pipe file(s), a channel of communication is established.
  - Each process ( parent and child ) should first close the ends of the pipe that they are not using. For example, if the parent is writing to the pipe and the child is reading, then the parent should close the reading end of its pipe after the fork and the child should close the writing end.



# ORDINARY PIPES IN WINDOWS

- Ordinary pipes in windows are termed as ***anonymous*** pipes
- They are still limited to parent-child relationships.
- They are read from and written to as files.
- They are created with `createpipe()` function, which takes additional arguments.
- In windows it is necessary to specify what resources a child inherits, such as pipes.

# NAMED PIPES

- Named pipes support bidirectional communication, communication between non parent-child related processes, and persistence after the process which created them exits. Multiple processes can also share a named pipe, typically one reader and multiple writers.

# NAMED PIPES

- In unix, named pipes are termed fifos, and appear as ordinary files in the file system.
  1. Created with `mkfifo( )` and manipulated with `read( )`, `write( )`, `open( )`, `close( )`, etc.
  2. Unix named pipes are bidirectional, but half-duplex, so two pipes are still typically used for bidirectional communications.
  3. Unix named pipes still require that all processes be running on the same machine. Otherwise sockets are used.
- In Windows, Named pipes provide richer communications.
  1. Full-duplex is supported.
  2. Processes may reside on the same or different machines
  3. Created and manipulated using `createnamedpipe( )`, `connectnamedpipe( )`, `readfile( )`, and `writefile( )`.

# THANK YOU!