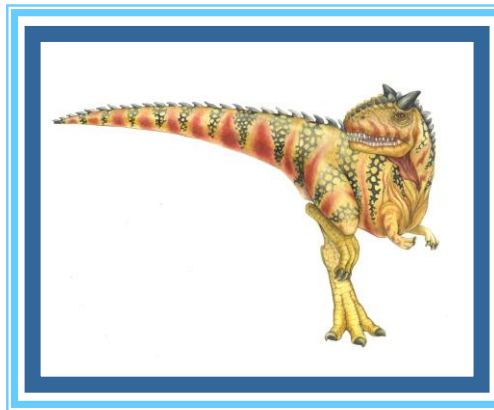
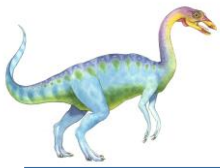


Chapter 3: Processes





Chapter 3: Processes

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





Objectives

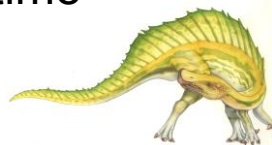
- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing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

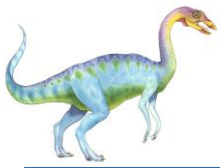




Process Concept

- ❑ An operating system executes a variety of programs:
 - ❑ Batch system – **jobs**
 - ❑ Time-shared systems – **user programs** or **tasks**
- ❑ Textbook uses the terms **job** and **process** almost interchangeably
- ❑ **Process** – a program in execution; process execution must progress in sequential fashion
- ❑ Multiple parts
 - ❑ The program code, also called **text section**
 - ❑ Current activity including **program counter**, 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





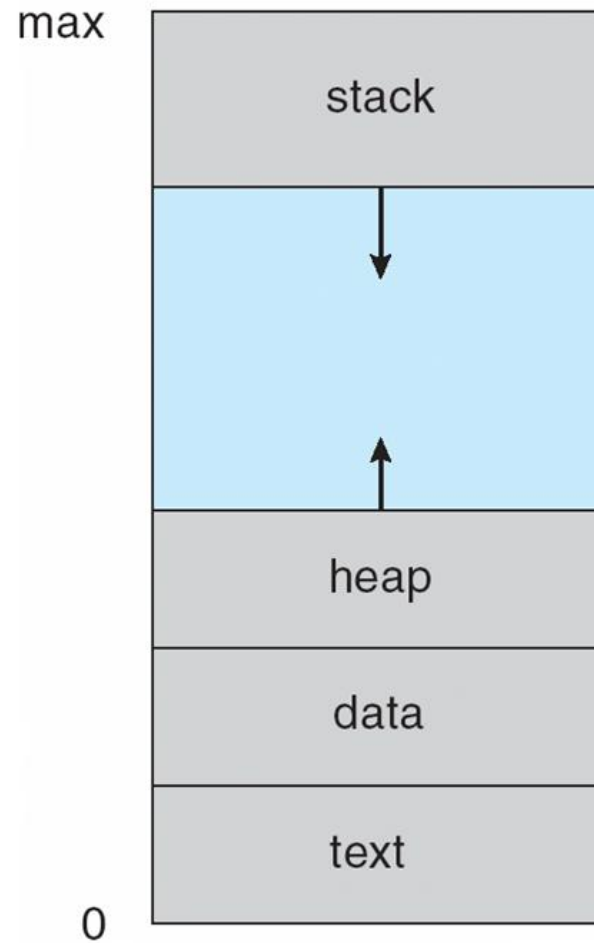
Process Concept (Cont.)

- 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
 - Consider multiple users executing the same program





Process in Memory





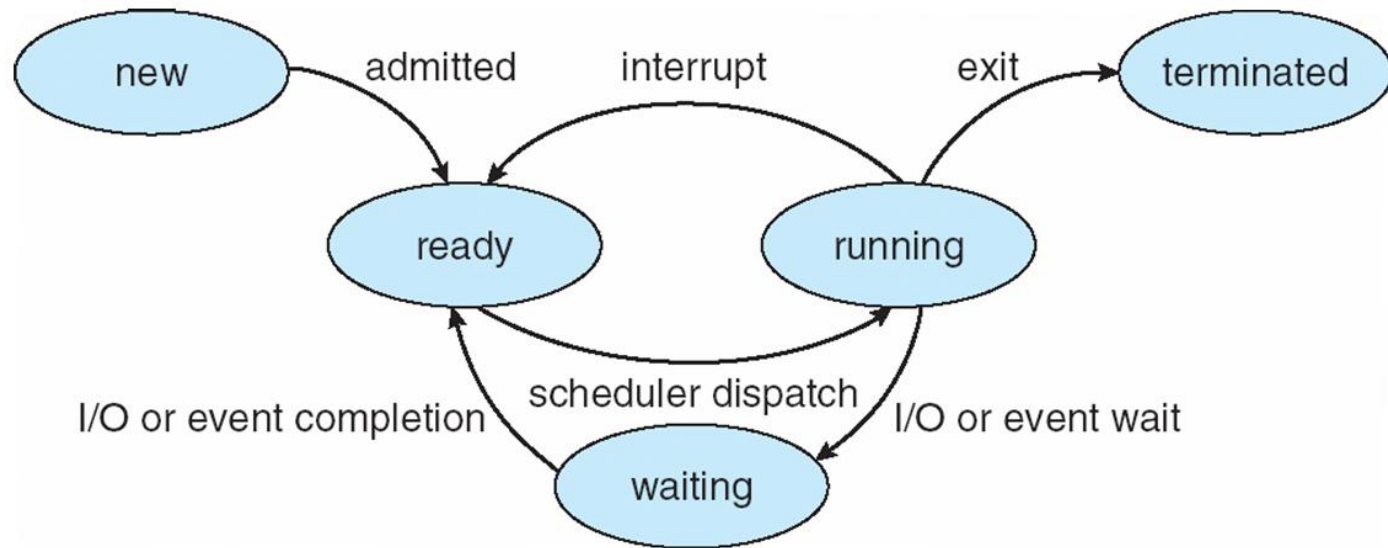
Process State

- As a process executes, it changes **state**
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution





Diagram of Process State

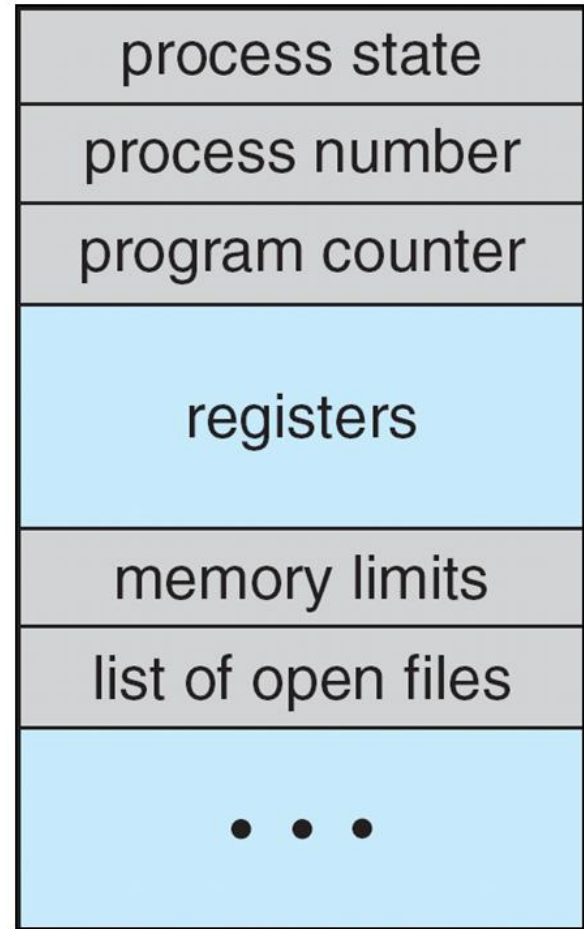




Process Control Block (PCB)

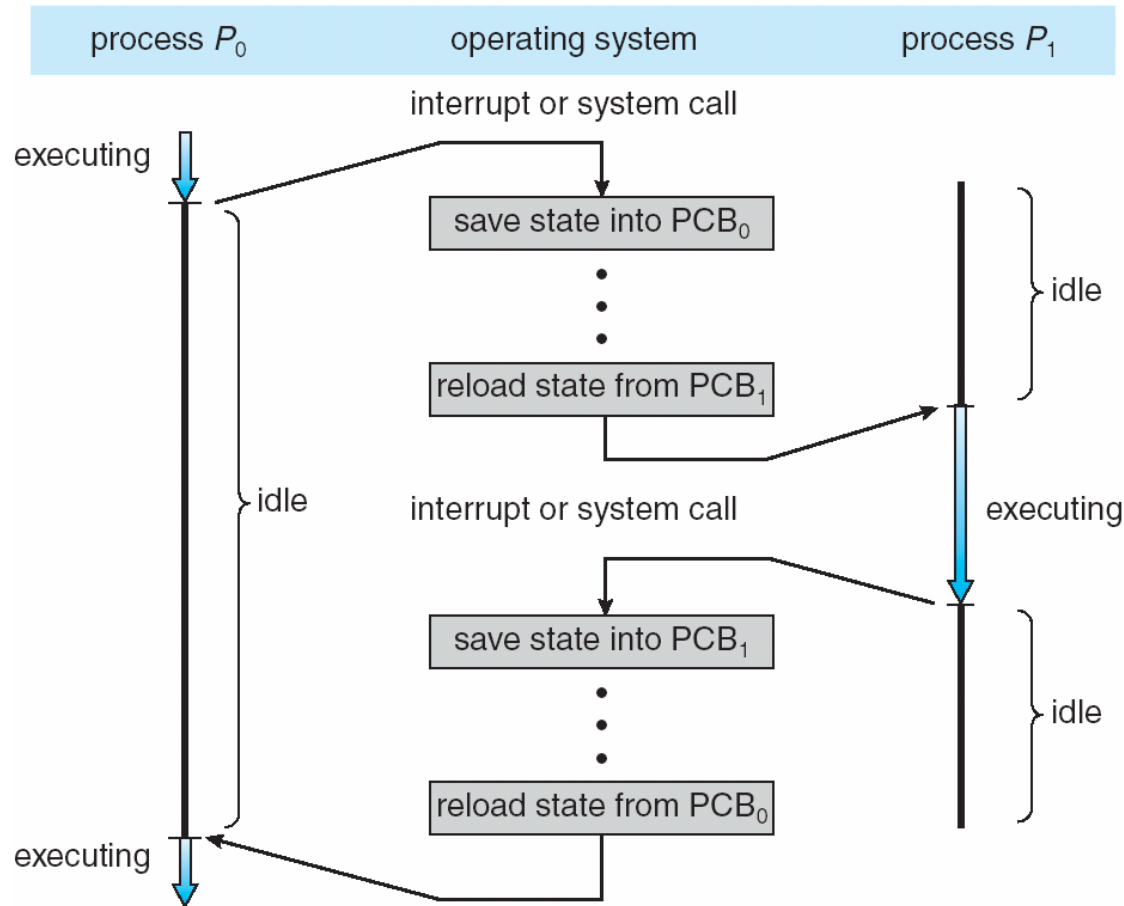
Information associated with each process
(also called **task control block**)

- ❑ 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





Threads

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

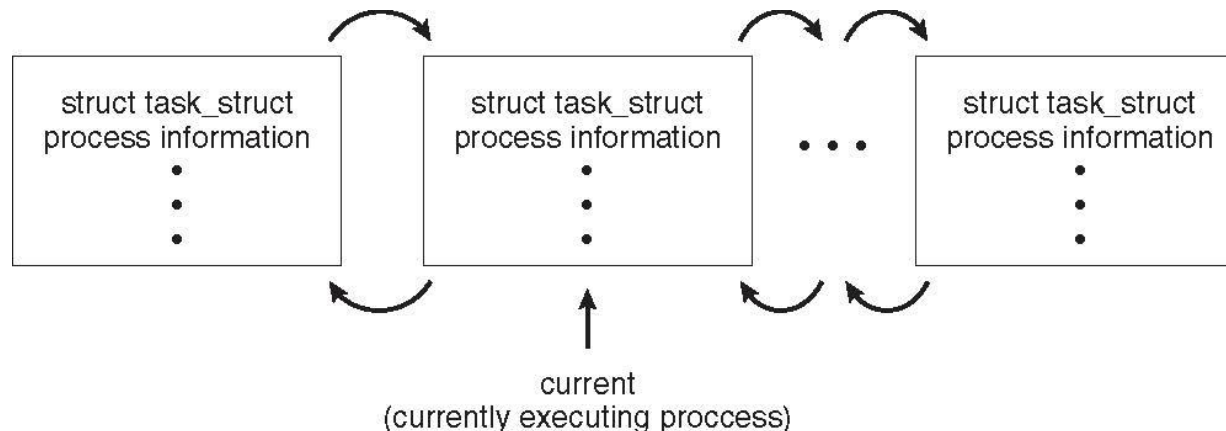




Process Representation in Linux

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 */
```





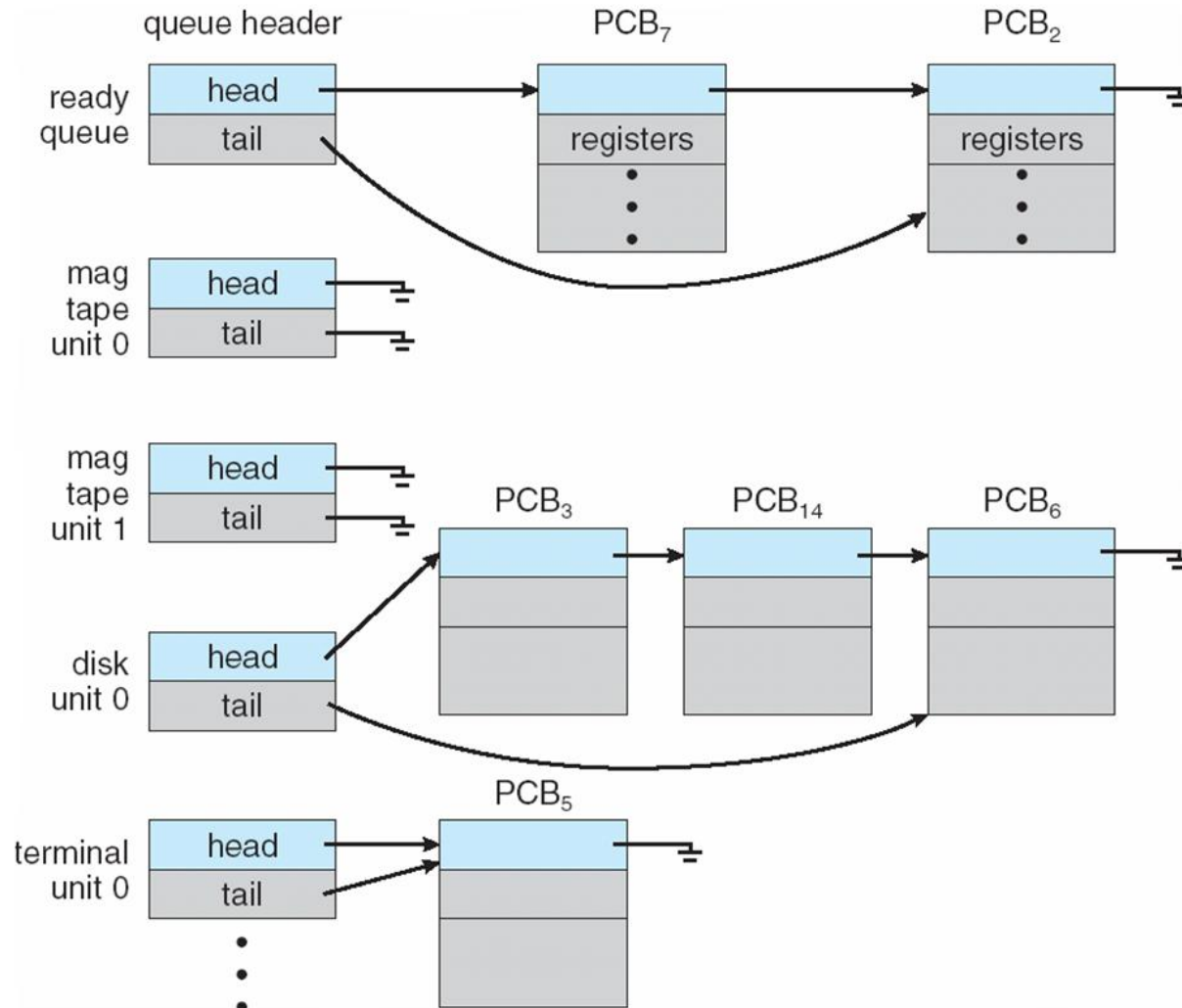
Process Scheduling

- 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





Ready Queue And Various I/O Device Queues





-
- ```
graph LR; In(()) --> RQ[ready queue]; RQ --> CPU((CPU)); CPU --> IOR[I/O request]; IOR --> IOQ[I/O queue]; IOQ --> IO((I/O)); IO --> RQ; CPU --> TSE[time slice expired]; TSE --> RQ; CPU --> FC[fork a child]; FC --> CE((child executes)); CE --> RQ; CPU --> WFI[wait for an interrupt]; WFI --> IOO((interrupt occurs)); IOO --> RQ;
```





# 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)  $\Rightarrow$  (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)  $\Rightarrow$  (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 strives for good ***process mix***

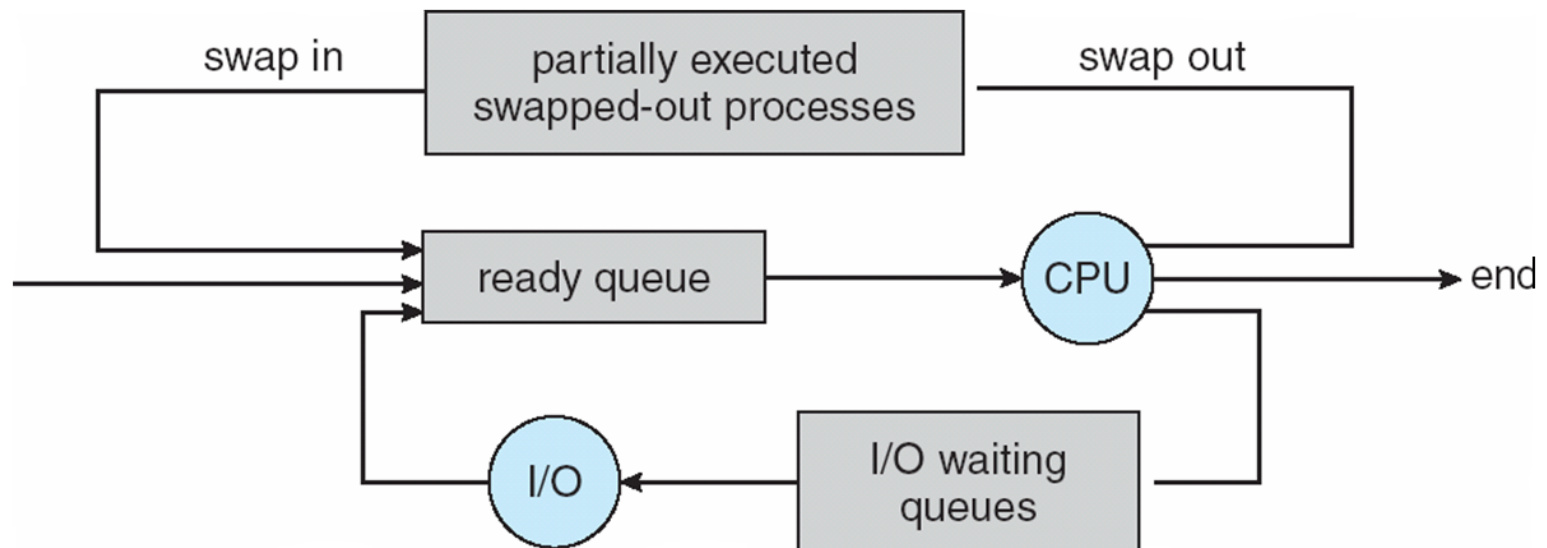






# Addition of Medium Term Scheduling

- **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

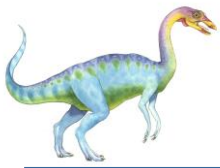




# Context Switch

- 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





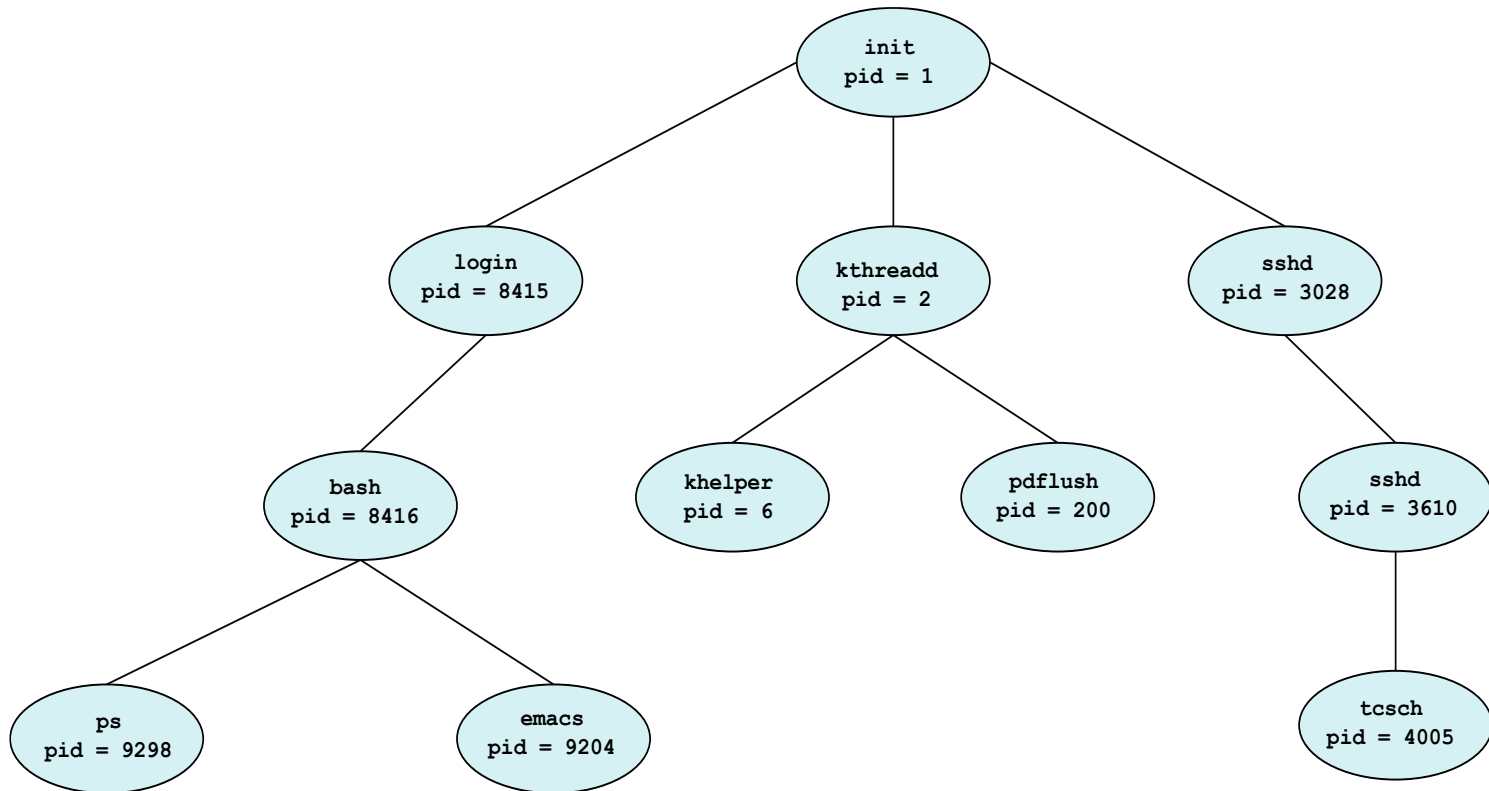
# Process Creation

- ❑ **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





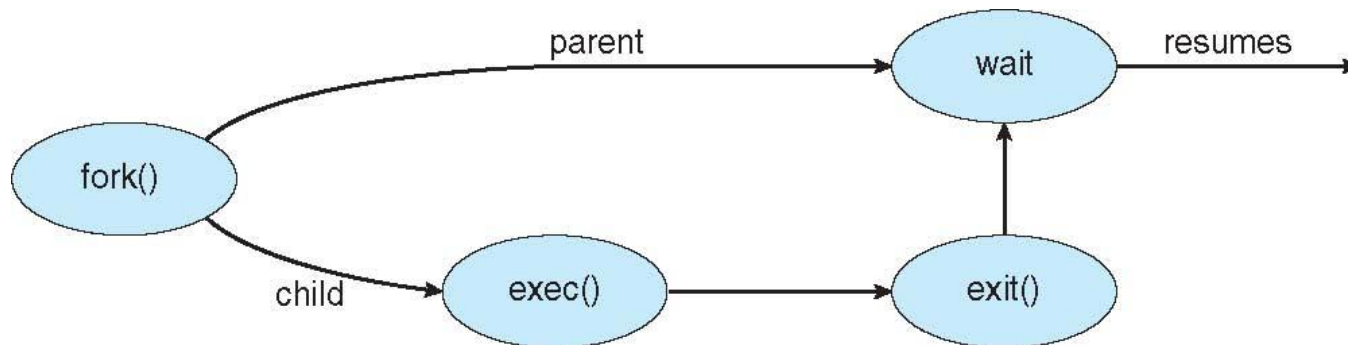
# A Tree of Processes in Linux





# 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;

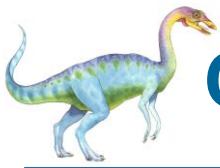
 /* fork a child process */
 pid = fork();

 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 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**





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





Interprocess Communication

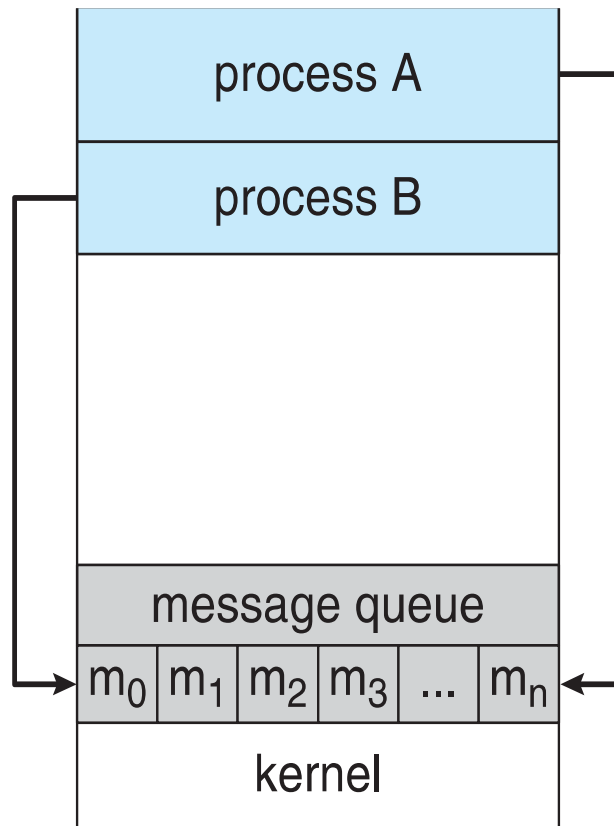
- ❑ Processes within a system may be *independent* or *cooperating*
- ❑ Cooperating process can affect or be affected by other processes, including sharing data
- ❑ Reasons for cooperating processes:
 - ❑ Information sharing
 - ❑ Computation speedup
 - ❑ Modularity
 - ❑ Convenience
- ❑ Cooperating processes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



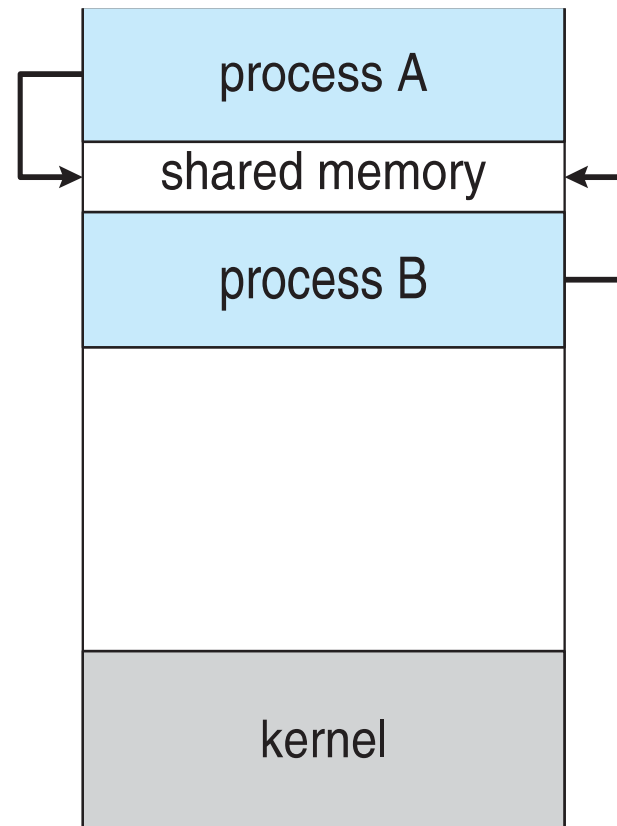


Communications Models

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

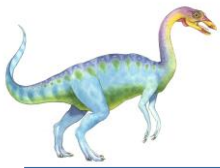


(a)



(b)





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

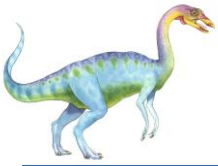
□ Shared data

```
#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

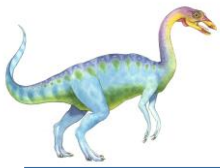




Indirect Communication

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





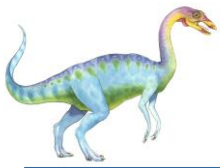
Synchronization (Cont.)

□ 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 - POSIX

? POSIX Shared Memory

- ? Process first creates shared memory segment

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

- ? Also used to open an existing segment to share it

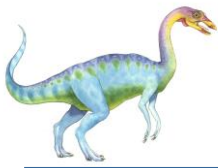
- ? Set the size of the object

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

- ? Now the process could write to the shared memory

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





IPC POSIX Producer

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

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 object */
    void *ptr;

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

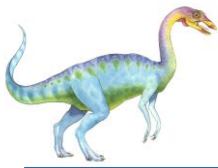
    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* 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;
}
```





IPC POSIX Consumer

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

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 object */
    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;
}
```





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





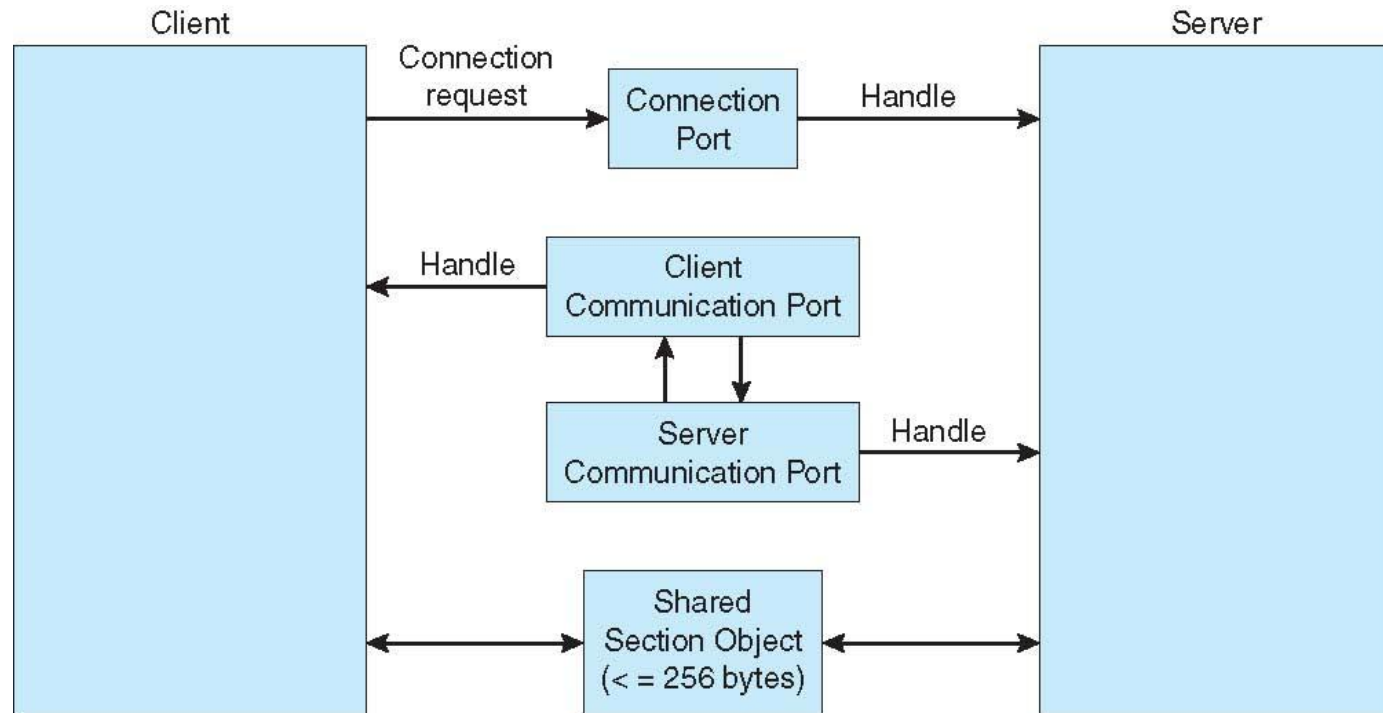
Examples of IPC Systems – Windows

- 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

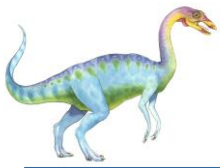




Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)





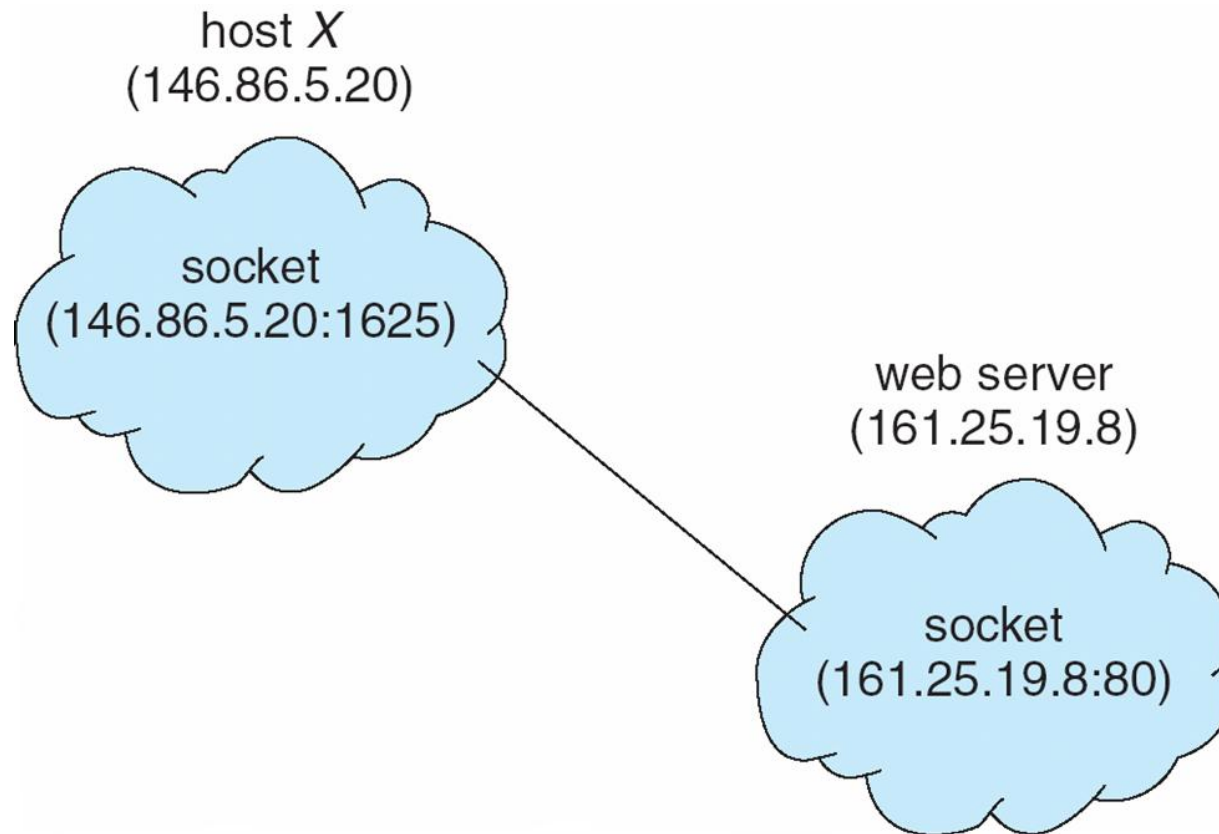
Sockets

- 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





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:

```
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);
        }
    }
}
```

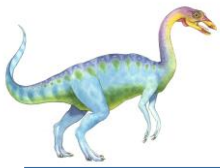




Remote Procedure Calls

- 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)**





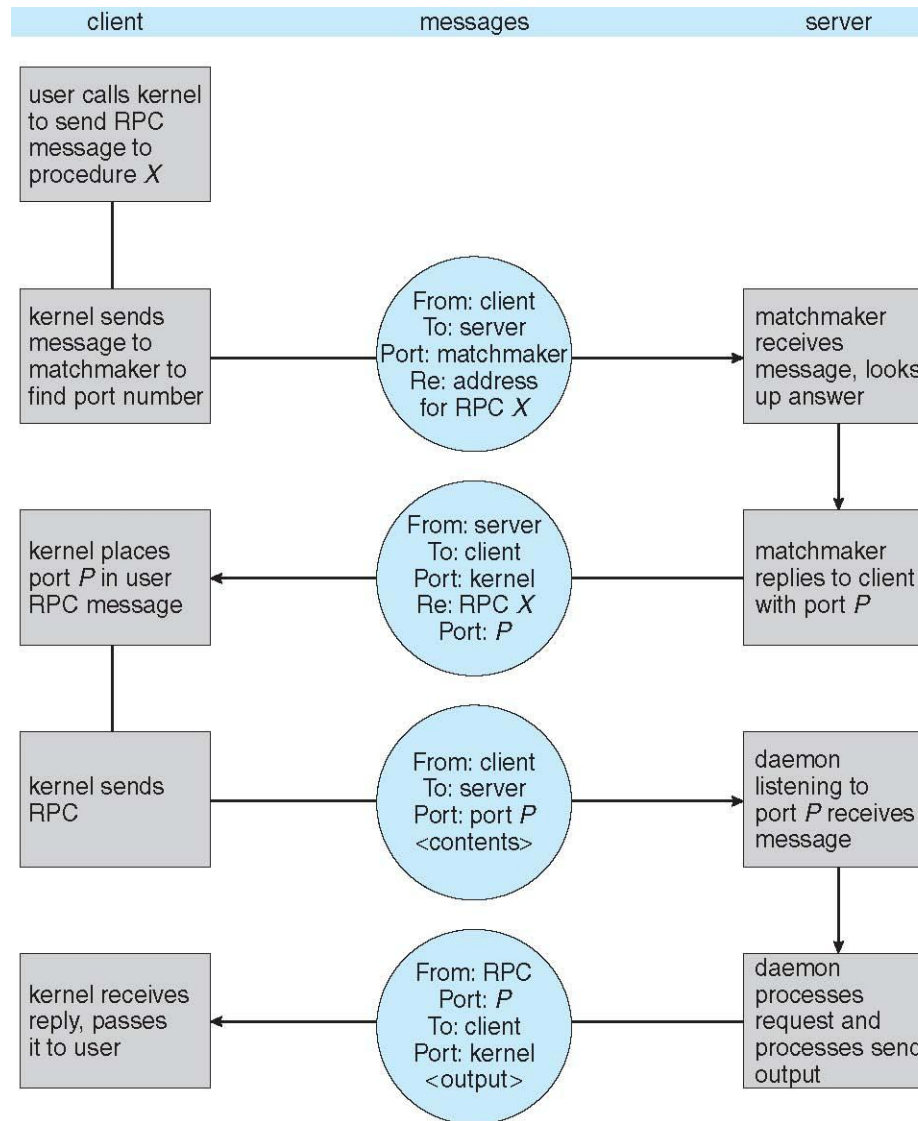
Remote Procedure Calls (Cont.)

- 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 (or **matchmaker**) service to connect client and server





Execution of RPC





Pipes

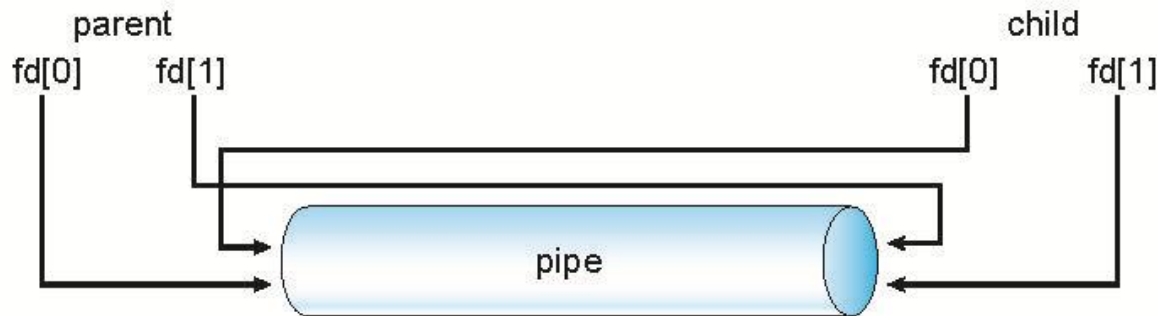
- Acts as a conduit allowing two processes to communicate
- Issues:
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
 - Can the pipes 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



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





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

