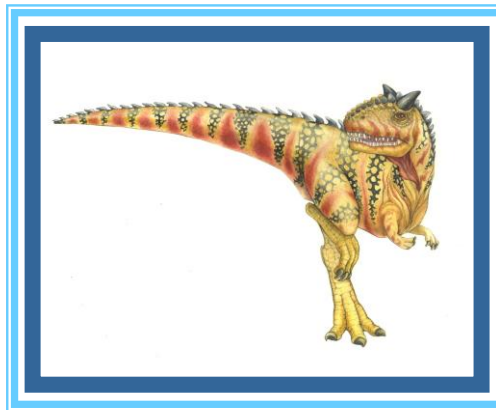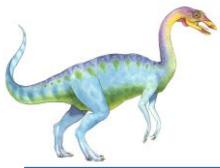# Chapter 3: Processes

Outline

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

Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.

- Describe how processes are created and terminated in an operating system using system calls.

- Describe and contrast interprocess communication using shared memory and message passing.

- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.

- Describe client-server communication using sockets and remote procedure calls.

- Design kernel modules that interact with the Linux operating system.

Process Concept

- An operating system executes a variety of programs that run as a process.

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

- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process

- Multiple parts

  - The executable 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
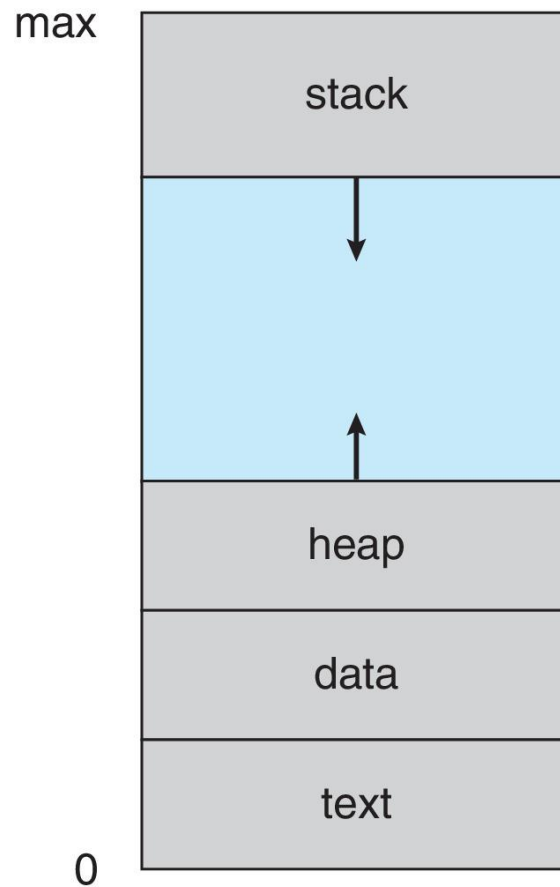
Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**

    - Program becomes process when an executable file is loaded into memory

- Execution of program started via GUI mouse clicks, command line entry of its name, etc.

- One program can be associated with several processes

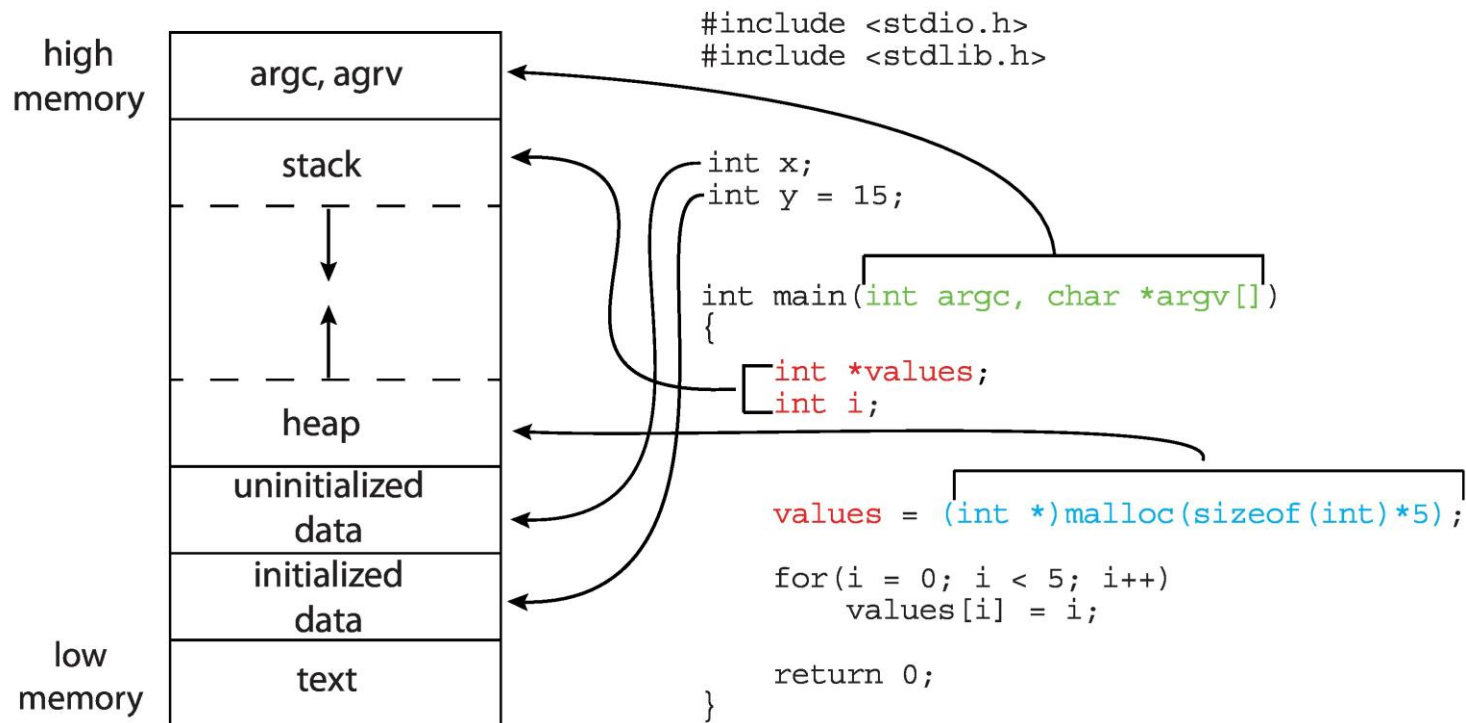   - Consider multiple users executing the same program
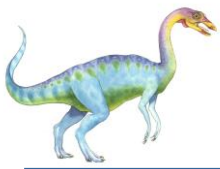
Process in Memory

- Sizes of the text and data sections are fixed (Why??)
- Operating system must ensure they do not *overlap* one another

Memory Layout of a C Program


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

int x;
int y = 15;

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

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

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

    return 0;
}
```

Process State

- As a process executes, it changes **state**

    - **New**: The process is being created

    - **Running**: Instructions are being executed

        ‣ only one process can be running on any processor core at any instant

    - **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
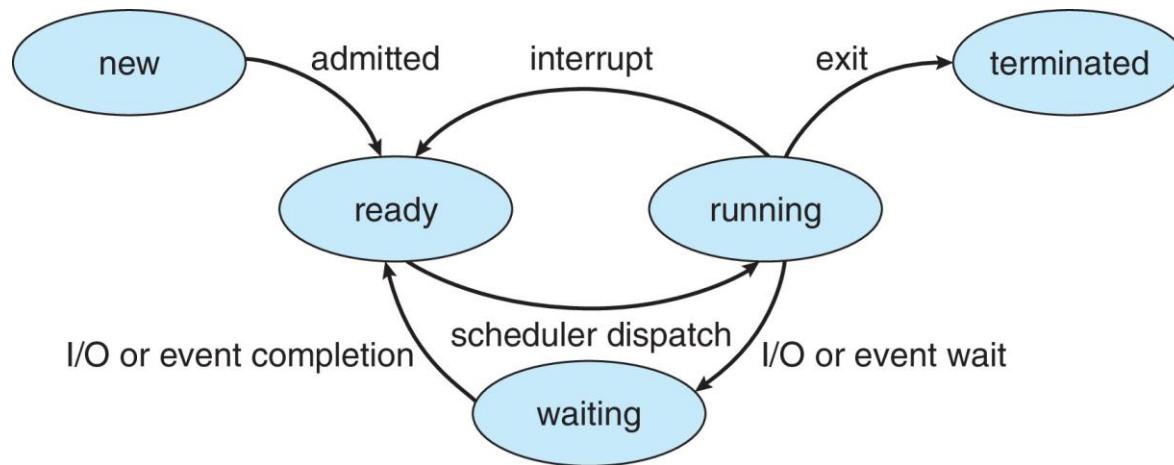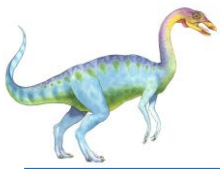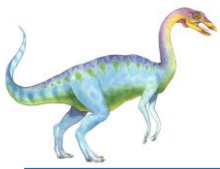
Diagram of Process State

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 e.g., accumulators, stack pointers, 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

process state
process number
program counter
registers
memory limits
list of open files
• • •

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**

- PCB must have storage for thread details, multiple program counters

- Example:

  - A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker.

  - Video players could use multiple threads for decoding videos, synchronizing sound, and reading a network stream.

# Process Representation in Linux

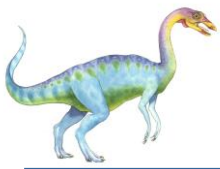Represented by the C structure `task_struct` in `<include/linux/sched.h>`

```
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 */
```
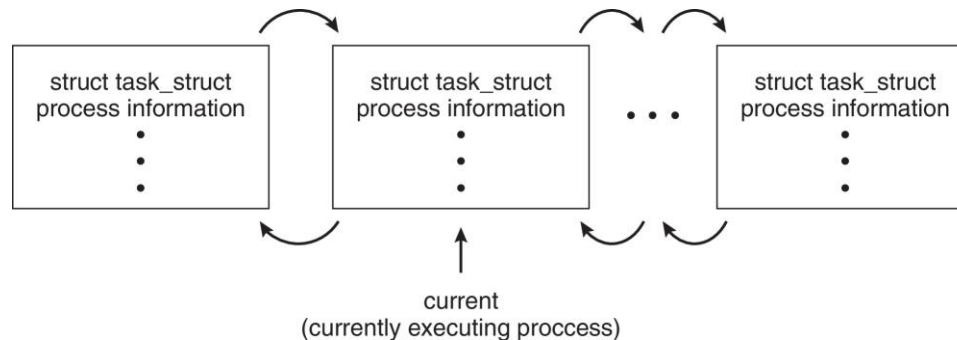

- Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`
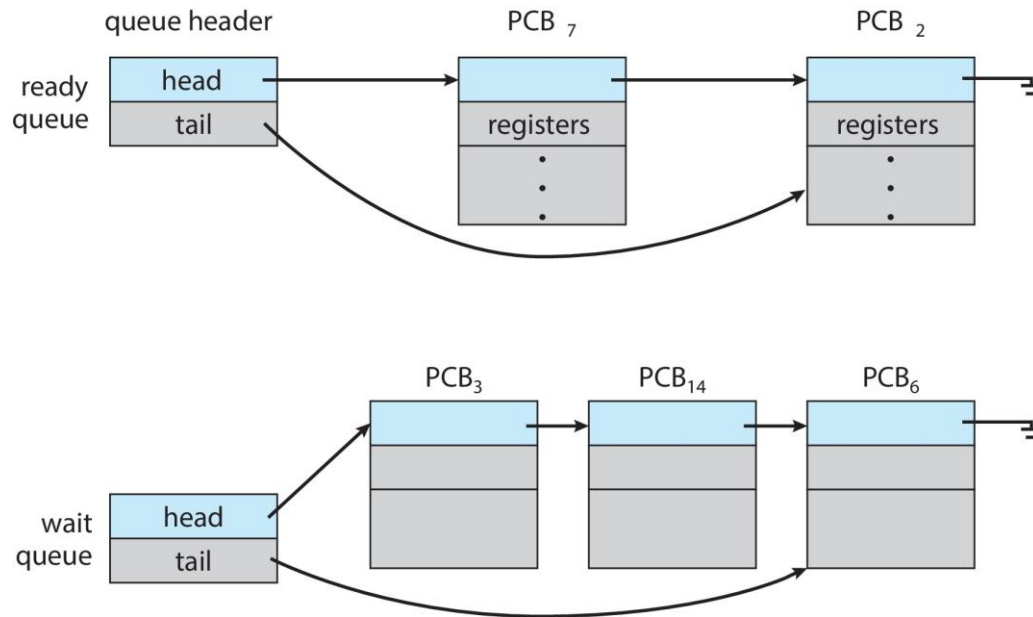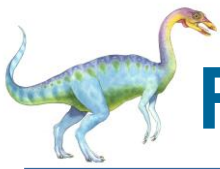
Process Scheduling

- For a system with a single CPU core, there will never be more than one process running at a time.

- In case more processes than cores, excess processes will have to wait until a core is free.

- The number of processes currently in memory is known as the **degree of multiprogramming**.

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

- Goal -- Maximize CPU use, quickly switch processes onto CPU core

- Maintains **scheduling queues** of processes

  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute

  - **Wait queues** – set of processes waiting for an event (i.e., I/O)

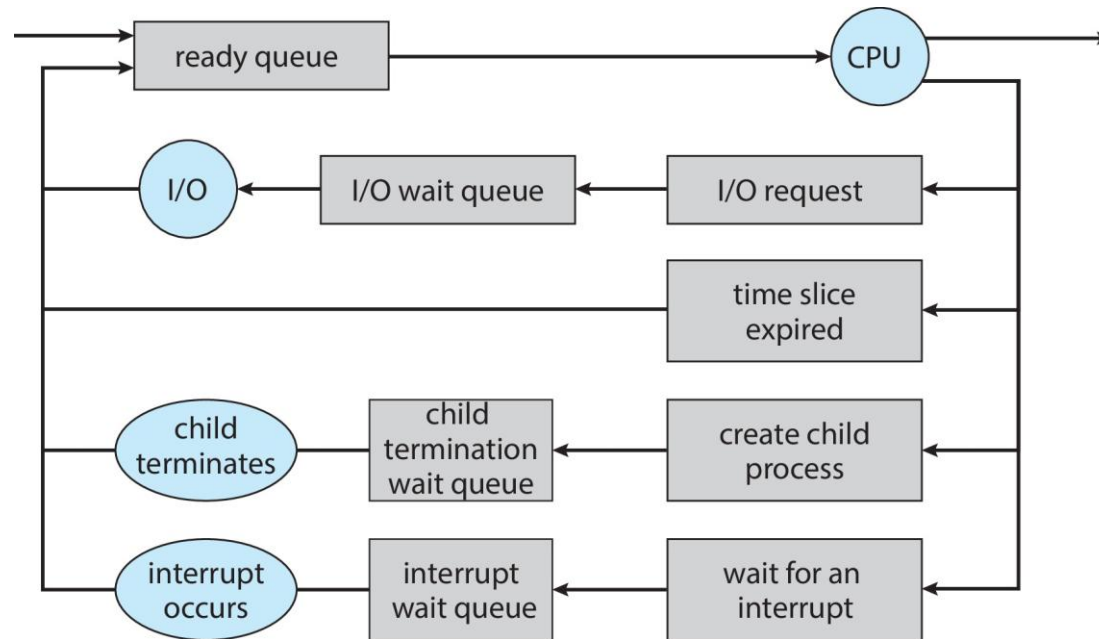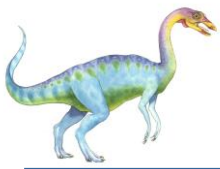 - Processes migrate among the various queues (state diagram!!)

Ready and Wait Queues

# Representation of Process Scheduling

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 pure 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 ➜ A context switch here simply requires changing the pointer to the current register set.
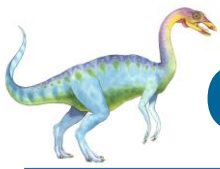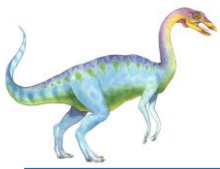
CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.
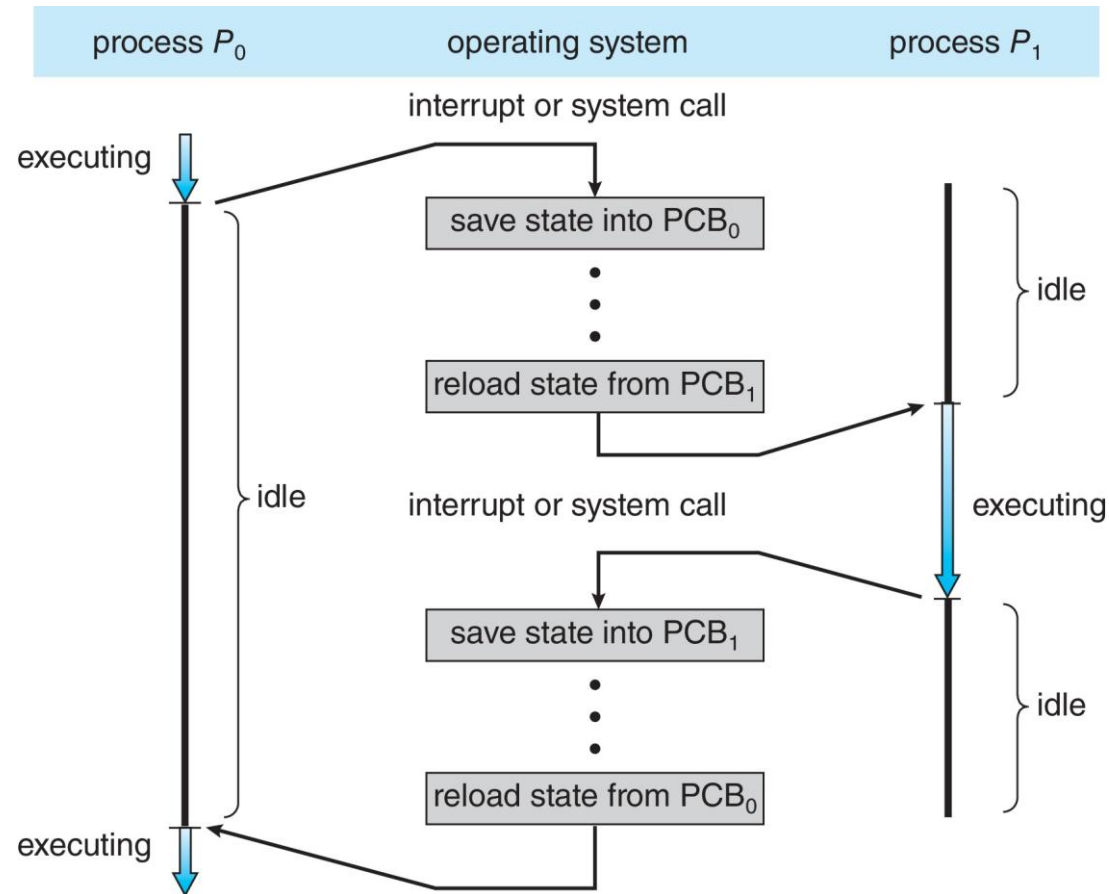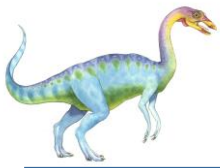
Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended

- Due to limited screen size, 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

Operations on Processes

- System must provide mechanisms for:
  - Process creation
  - Process termination

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes

- **pstree** command displays a tree of all processes in Linux

- 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
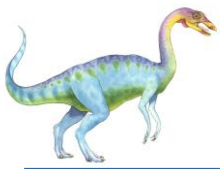
Process Creation (Cont.)

- 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
 - Parent process calls `wait()` waiting for the child to terminate
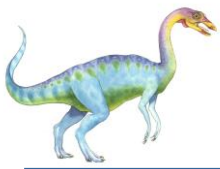
A Tree of Processes in Linux
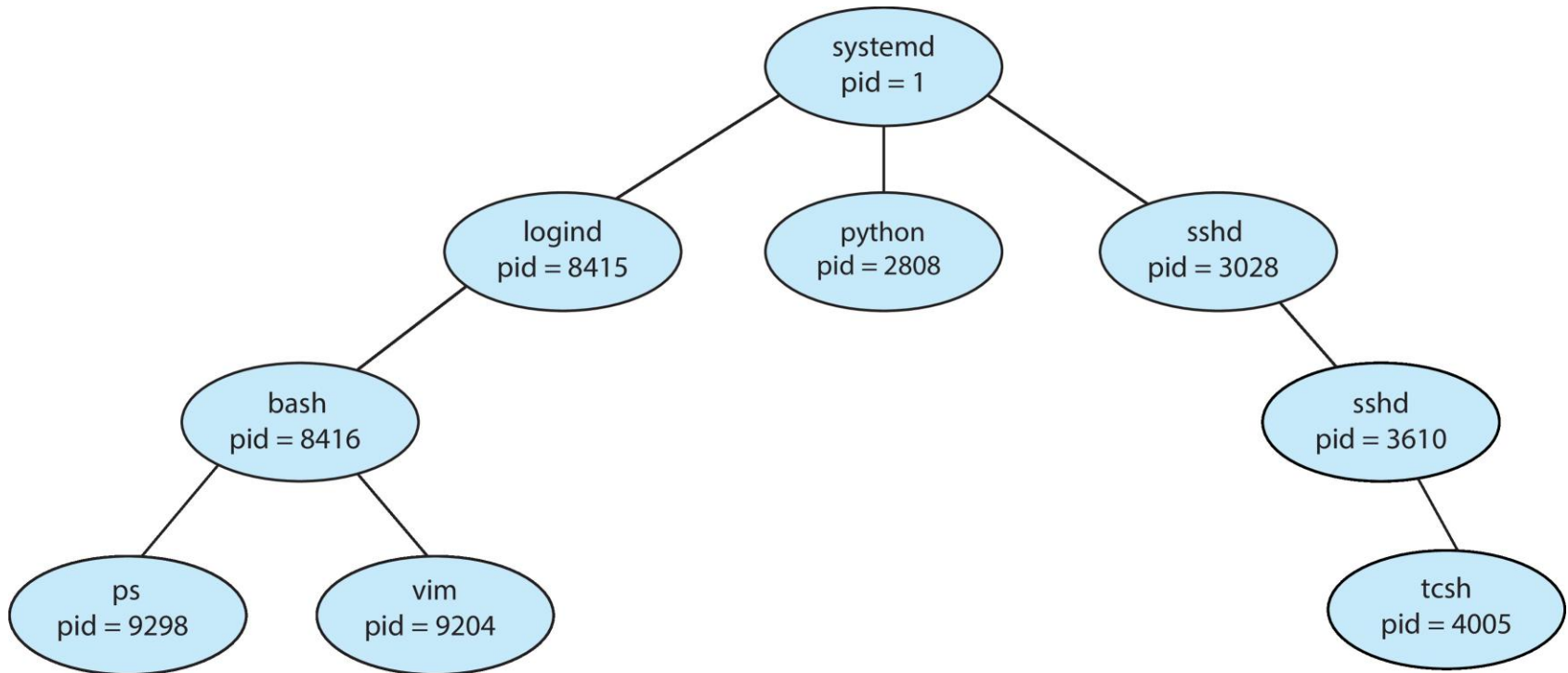
Process Creation (Cont.)

- After fork(), one of the two processes typically uses the exec() to replace the process's memory space with a new program.

- execlp() is a version of the exec() through which child process overlays its address space with the UNIX command /bin/ls

- The parent waits for the child process to complete with the wait()

    - When the child process completes, by either implicitly or explicitly, invoking exit() the parent process resumes from the call to wait()
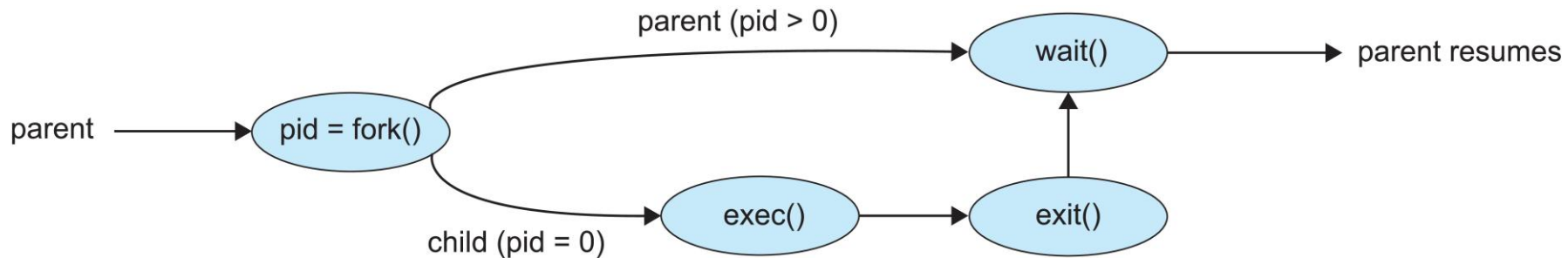
C Program Forking Separate Process

```c
#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;
}
```
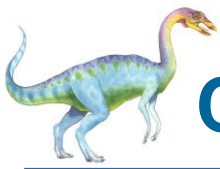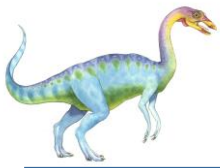
- Read by yourself

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

    ▸ the parent must have a mechanism to inspect the state of its children.

  - 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 (**cascading termination**)

   ▸ Normally initiated by the operating system

Process Termination

- 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);
    ```

- When a process terminates, its resources are deallocated

    - However, its entry in the process table must remain there until the parent calls wait(), because it contains the process's exit status

    - All processes transition to this state when they terminate

- If no parent waiting (did not invoke **wait()**) process is a **zombie**

- If parent terminated without invoking **wait()**, process is an **orphan**
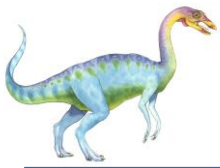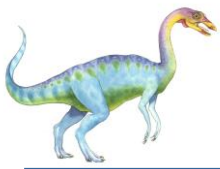
Interprocess Communication

- 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 (allow concurrent access to such information)

  - Computation speedup (we must break it into subtasks)

  - Modularity

- Cooperating processes need **interprocess communication** (**IPC**)
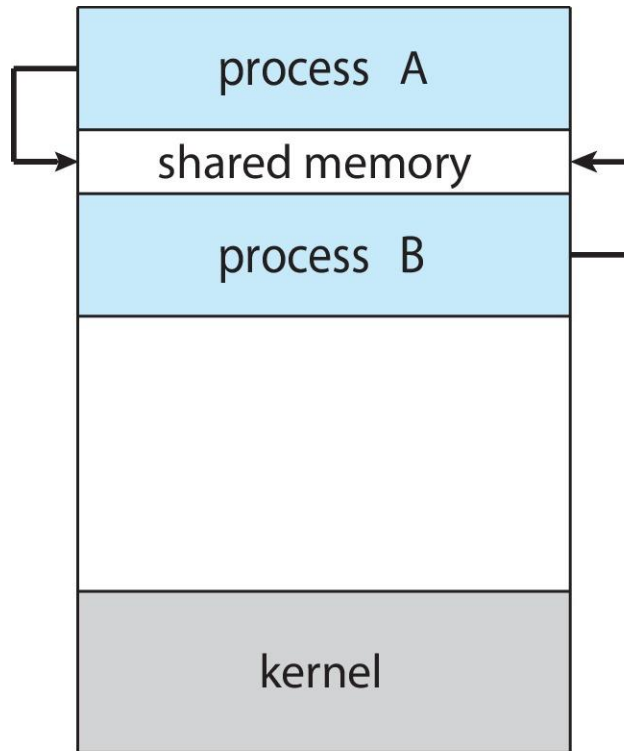
Interprocess Communication

- Two models of IPC

  - **Shared memory**

    - Processes can then exchange information by reading and writing data to the shared region

    - Shared memory can be faster than message passing (Why??)

  - **Message passing**

    - communication takes place by means of messages exchanged between the cooperating processes

    - Useful for exchanging smaller amounts of data, because no conflicts need be avoided
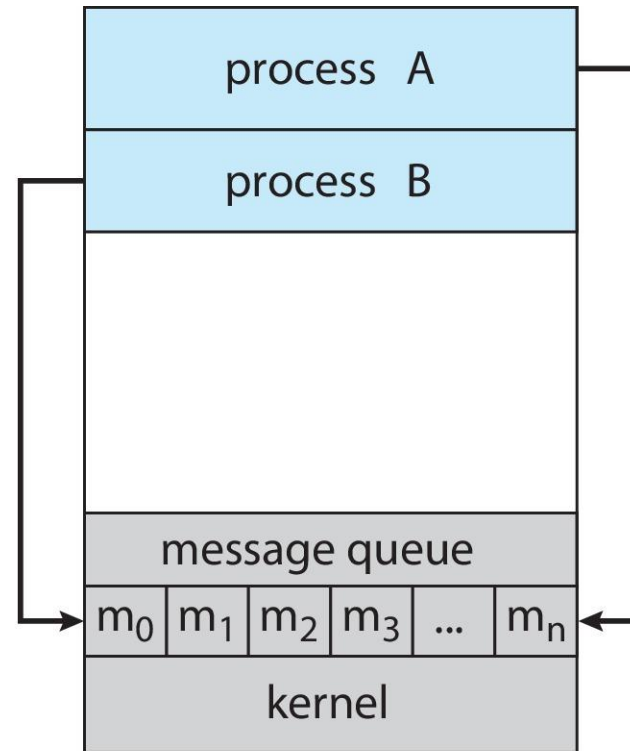
    - easier to implement in a distributed system

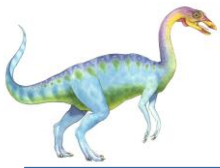# Communications Models

(a) Shared memory.

(b) Message passing.

(a)

(b)

IPC in Shared-Memory Systems

- Requires communicating processes to establish a region of shared memory

- Shared-memory region resides in the address space of the process creating the shared-memory segment.

  - OS tries to prevent one process from accessing another process's memory. Thus, processes agree to remove this restriction

- Exchange information by reading and writing data in the shared areas.

  - The form of the data and the location are determined by these processes and are not under the OS's control.

Producer-Consumer Problem

- Paradigm for cooperating processes:

    - *producer* process produces information that is consumed by a *consumer* process

        ▸ e.g., a compiler may produce assembly code that is consumed by an assembler.

- Two variations:

    - **unbounded-buffer** places no practical limit on the size of the buffer:

        ▸ Producer never waits

        ▸ Consumer waits if there is no buffer to consume

    - **bounded-buffer** assumes that there is a fixed buffer size

        ▸ Producer must wait if all buffers are full

        ▸ Consumer waits if there is no buffer to consume

Bounded-Buffer – Shared-Memory Solution
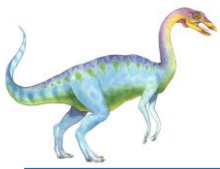
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer Process – Shared Memory

```
item next_produced;

while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
        while (in == out)
                ; /* empty buffer do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;


        /* consume the item in next consumed */
}
```

- Solution is correct, but can only use `BUFFER_SIZE-1` elements

What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.

- We can do so by having an integer `counter` that keeps track of the number of full buffers.

- Initially, `counter` is set to 0.

- The integer `counter` is incremented by the producer after it produces a new buffer.

- The integer `counter` is and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */


        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```

Race Condition

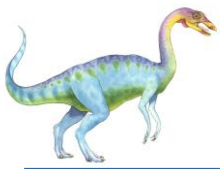- **counter++** could be implemented as

      register1 = counter
   register1 = register1 + 1
      counter = register1

- **counter--** could be implemented as

      register2 = counter
      register2 = register2 - 1
      counter = register2

- Consider this execution interleaving with "count = 5" initially:

      S0: producer execute register1 = counter   {register1 = 5}
      S1: producer execute register1 = register1 + 1 {register1 = 6}
      S2: consumer execute register2 = counter   {register2 = 5}
      S3: consumer execute register2 = register2 – 1 {register2 = 4}
      S4: producer execute counter = register1   {counter = 6 }
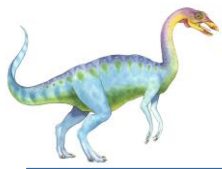   S5: consumer execute counter = register2   {counter = 4}
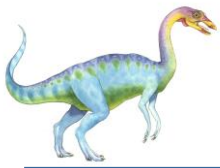
- **Question – why was there no race condition in the first solution (where at most N – 1) buffers can be filled?**
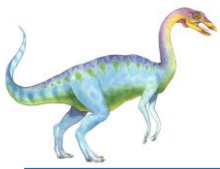
IPC – Message Passing

- 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?

Implementation of Communication Link

- Physical Implementation:
    - Shared memory
    - Hardware bus
    - Network
- Logical Implementation:
    - Direct or indirect
    - Synchronous or asynchronous
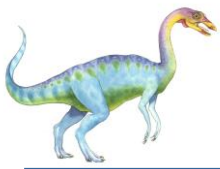   - Automatic or explicit buffering (bounded or unbounded buffer)

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, processes need to know only each other's identity to communicate
    - 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
- Two Variants
    - Both the sender process and the receiver process must name the other to communicate (*symmetry* in addressing)
    - Sender names the recipient; the recipient is not required to name the sender (*asymmetry* in addressing)

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

- Mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

- 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

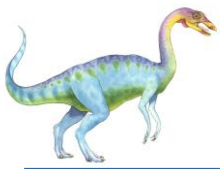 - Link may be unidirectional or bi-directional

Indirect Communication (Cont.)

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete 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 (Cont.)

- 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 depends on which of the following methods we choose

    - Allow a link to be associated with at most two processes

    - Allow only one process at a time to execute a receive operation (How??)

    - Allow the system to select arbitrarily the receiver (may be round robin). Sender is notified who the receiver was.
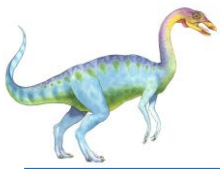
Synchronization

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

Producer-Consumer: Message Passing

- Producer

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

  send(next_produced);
}
```
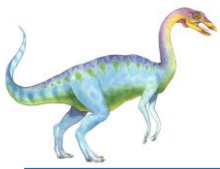
- Consumer

```
message next_consumed;
while (true) {
    receive(next_consumed);

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

- blocking send() and receive() statements

Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.

- Implemented in one of three ways

  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver

  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 – Windows

- Windows provides support for multiple operating environments, or *subsystems and* **application programs** communicate with these subsystems via a message-passing mechanism.

  - application programs are clients of a subsystem server

- Message-passing centric via **advanced local procedure call (ALPC)** facility

  - Only works between processes on the same system

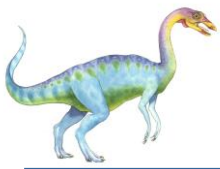 - Uses ports (like mailboxes) to establish and maintain communication channels

Examples of IPC Systems – Windows

- 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.

    - one for client–server messages, the other for server–client messages

  - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

- For small messages (up to 256 bytes), the port's message queue is used as intermediate storage

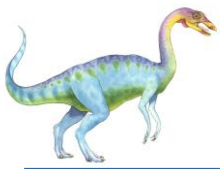- Larger messages must be passed through a **section object**
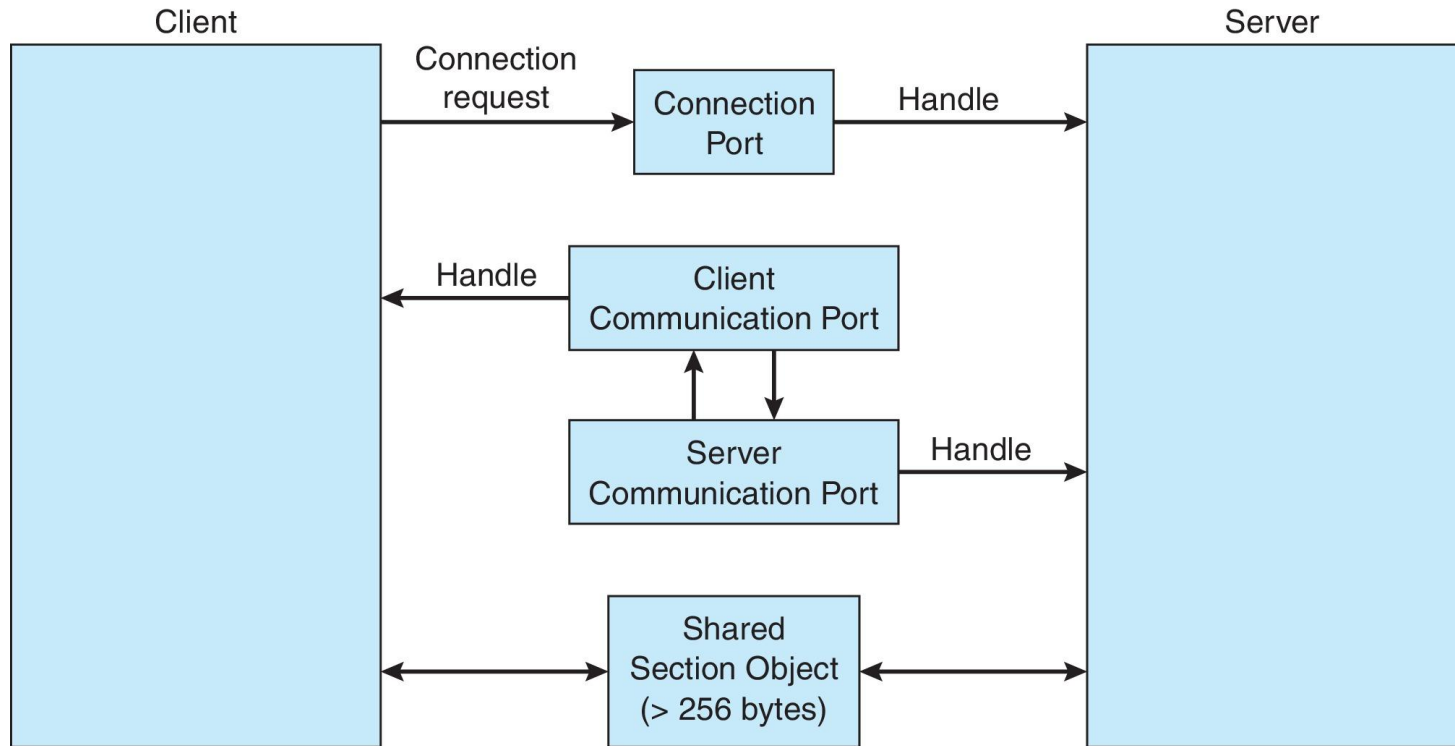
Pipes

- Acts as a channel allowing two processes to communicate

- First IPC mechanisms in early UNIX systems

- 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.

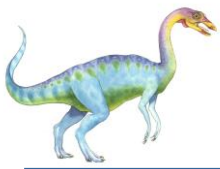- **Named pipes** – can be accessed without a parent-child relationship.

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
  - If two-way communication is required, two pipes must be used

- Require parent-child relationship between communicating processes
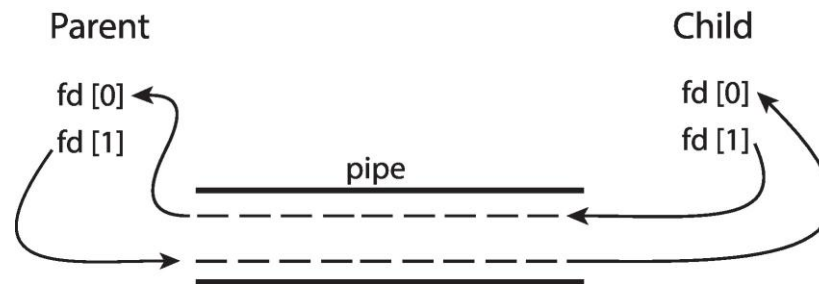
- Windows calls these **anonymous pipes**

- UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary read() and write() system calls
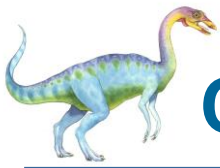
Named Pipes

- 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
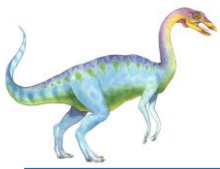
- Provided on both UNIX and Windows systems

Communications in Client-Server Systems
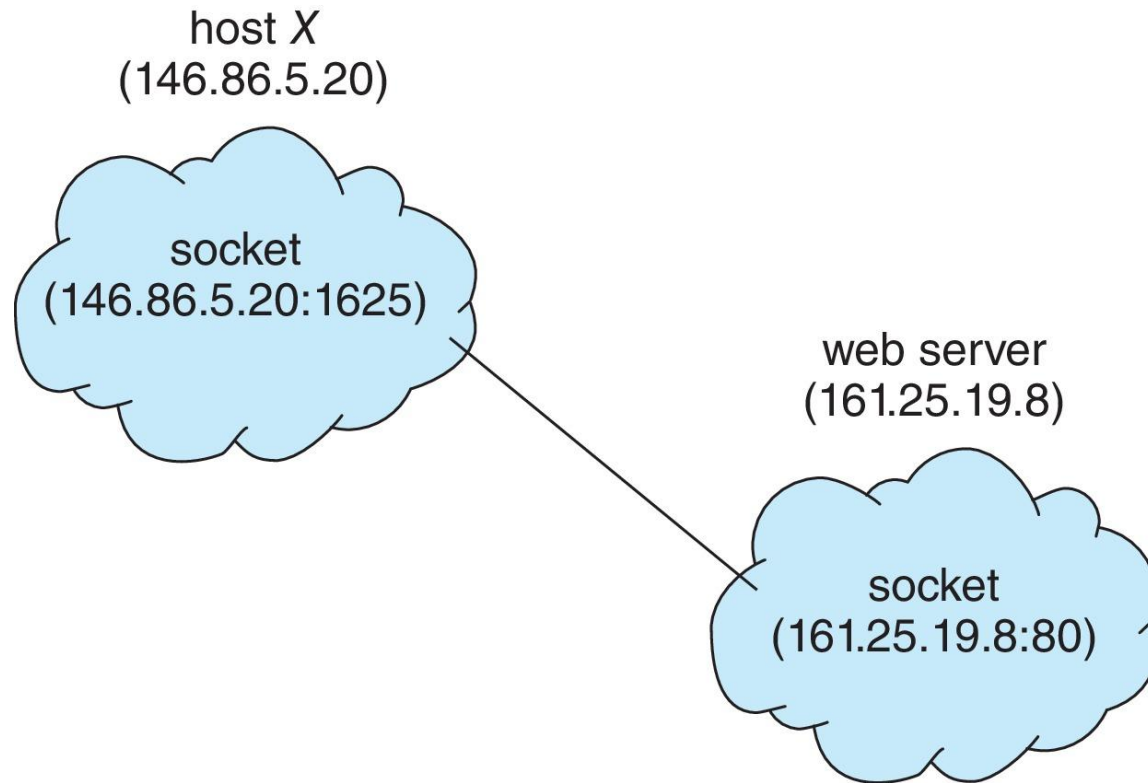
- Sockets
- Remote Procedure Calls

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

  - FTP server listens to port 21; and a web, or HTTP, server listens to port 80

- 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 in Java:

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

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
              PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```
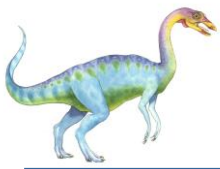
Sockets in Java

The equivalent Date client

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

public class DateClient
{
  public static void main(String[] args) {
    try {
      /* make connection to server socket */
      Socket sock = new Socket("127.0.0.1",6013);

      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      /* read the date from the socket */
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      /* close the socket connection*/
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Remote Procedure Calls

- Read by yourself

End of Chapter 3