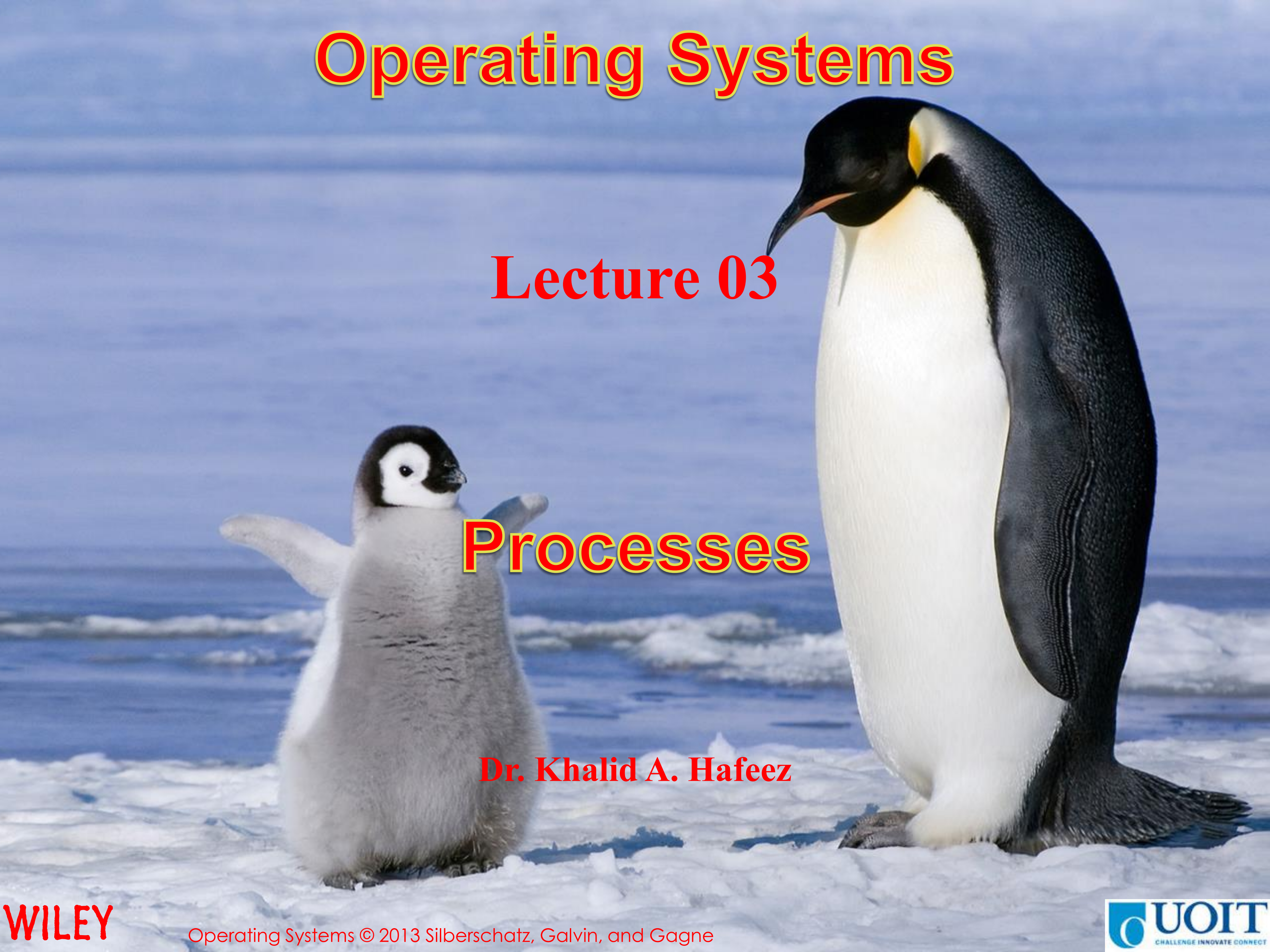


Operating Systems

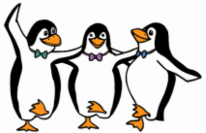
Lecture 03

Processes

Dr. Khalid A. Hafeez

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

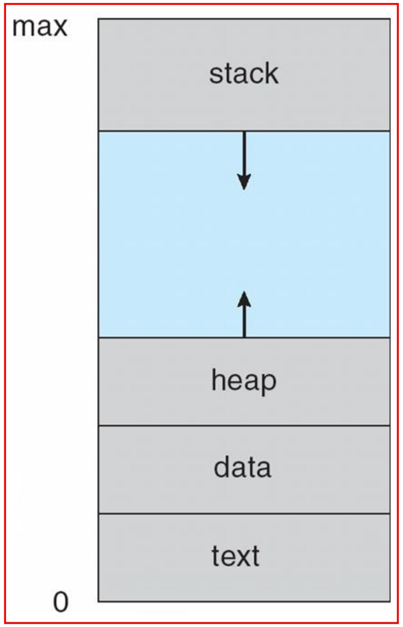
Process Concept

- An operating system executes a variety of programs, what to call all the CPU activities:
  - Batch system: executes **jobs**
  - Time-shared systems: executes **user programs** or **tasks**

- *Textbook uses the terms **job** and **process** almost interchangeably*

- **Process**: it is a program in execution;
  - process execution must progress in sequential fashion

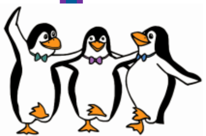
- **The process has multiple parts other than the code:**
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time



Process in Memory

Process Concept

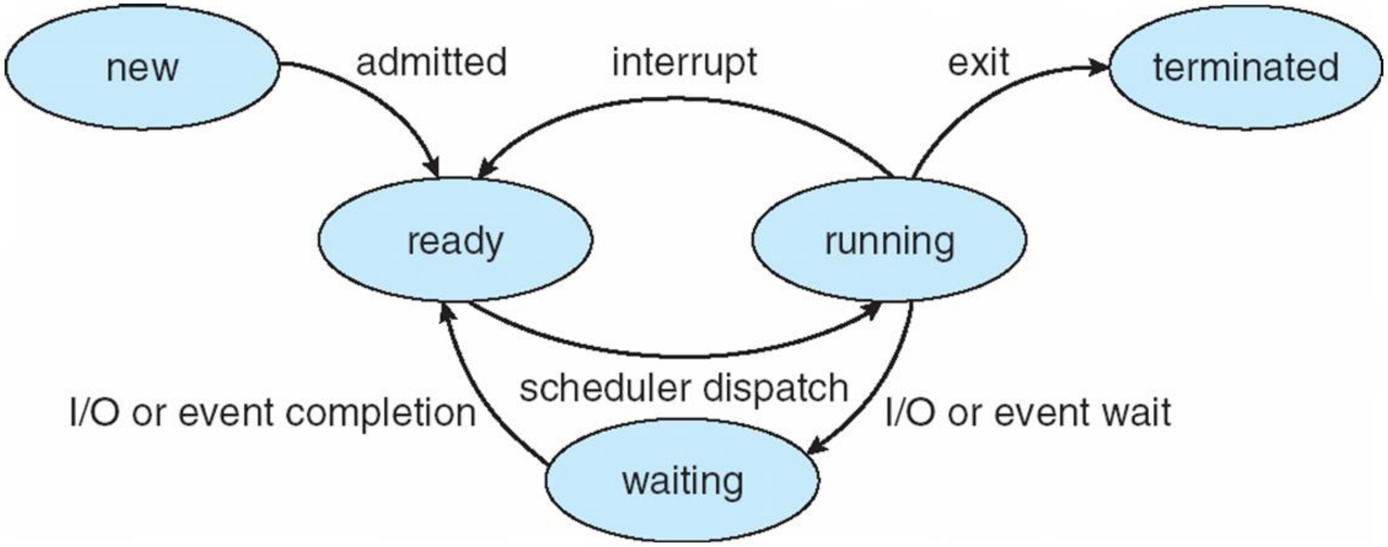
- Program is a ***passive*** entity stored on disk (**executable file**), but the process is an ***active*** entity with program counter pointing to the next instruction o be executed
  - Program becomes process when executable file loaded into memory
- Execution of program can be started via:
  - GUI mouse clicks, or
  - command line entry of its name, etc
- One program can be of several processes
  - Consider multiple users executing the same program
  - Or a user invoking many copies of the web browser



Process Concept

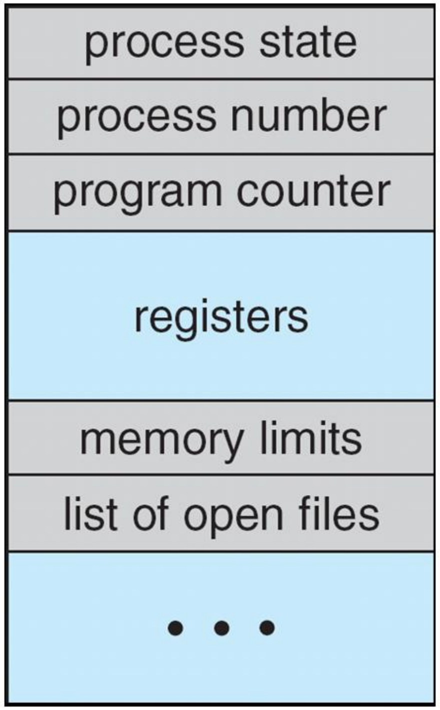
Process State

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



Process Concept

- Process Control Block (PCB)
  - Each process is represented in the OS by a process control block (PCB), also called task control block.
  - PCB contains the following information:
    - Process state: ready, running, waiting, etc
    - Program counter: location of next instruction to be executed
    - 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: amount of CPU used, clock time elapsed since start, time limits, process numbers
    - I/O status information: list of I/O devices allocated to process, list of open files



process state

process number

program counter

registers

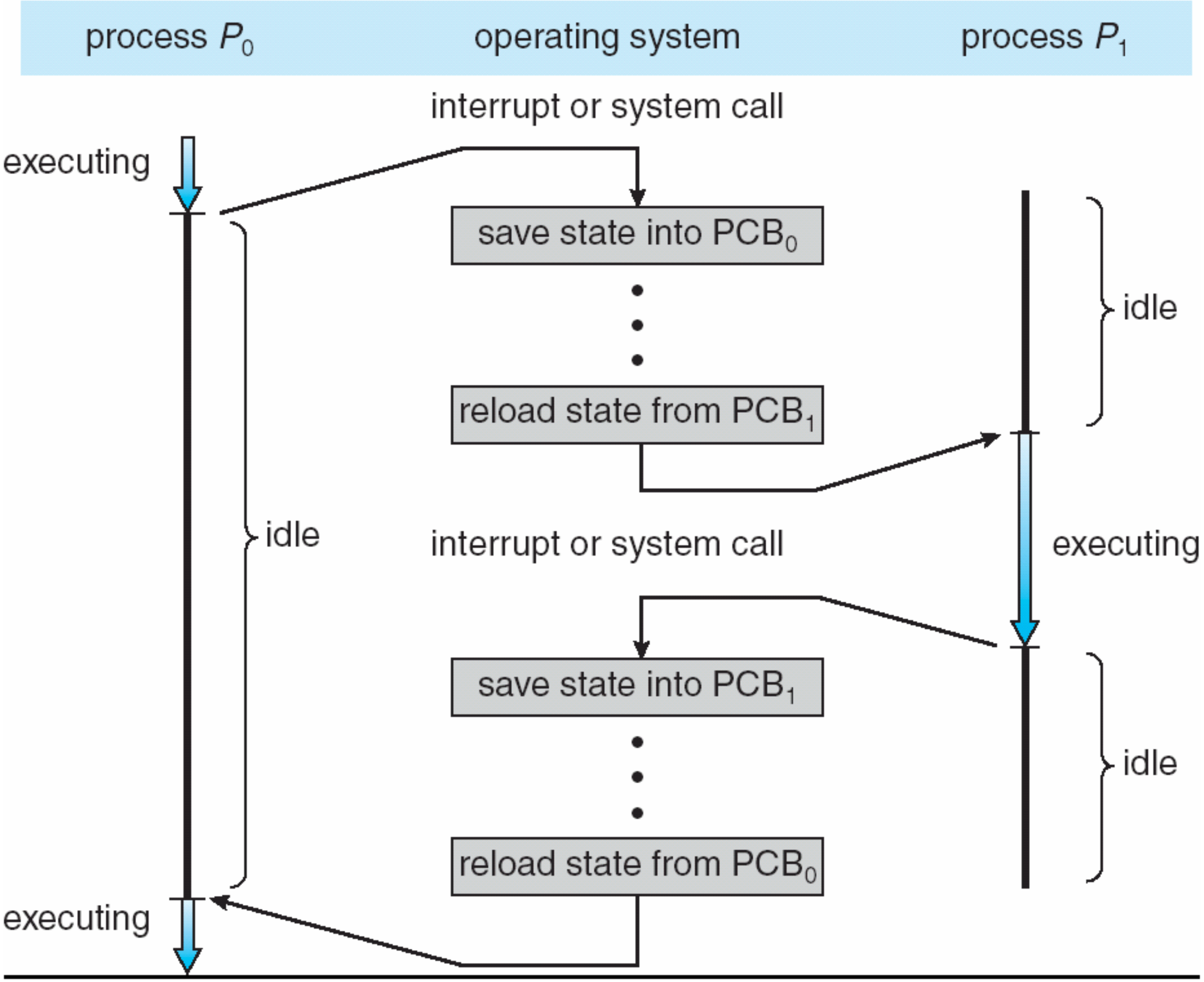
memory limits

list of open files

• • •

Process Concept

- **CPU Switch From Process to Process**



Process Concept

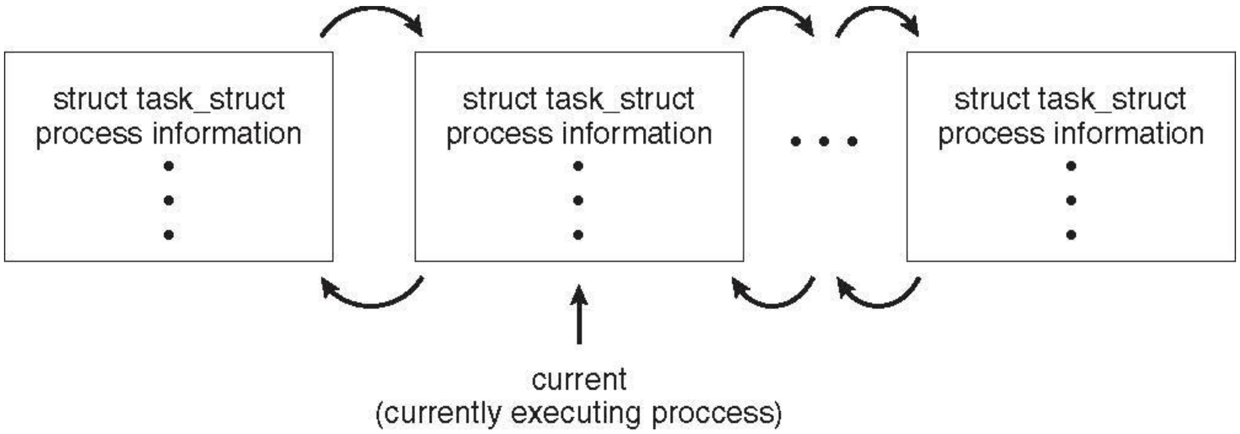
- **Threads**
  - So far, a process is a program that performs a single thread of execution.
  - Most OSes allow a process to have multiple threads of execution and thus to perform more than one task at a time.
    - It is beneficial on multicore systems, where multiple threads can run in parallel.
    - Consider having multiple program counters per process
      - Then multiple locations can execute at once
        - Multiple threads of control -> **threads**
    - The PCB is expanded to include information for each thread, such as multiple program counters

Process Concept

- 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 */
```

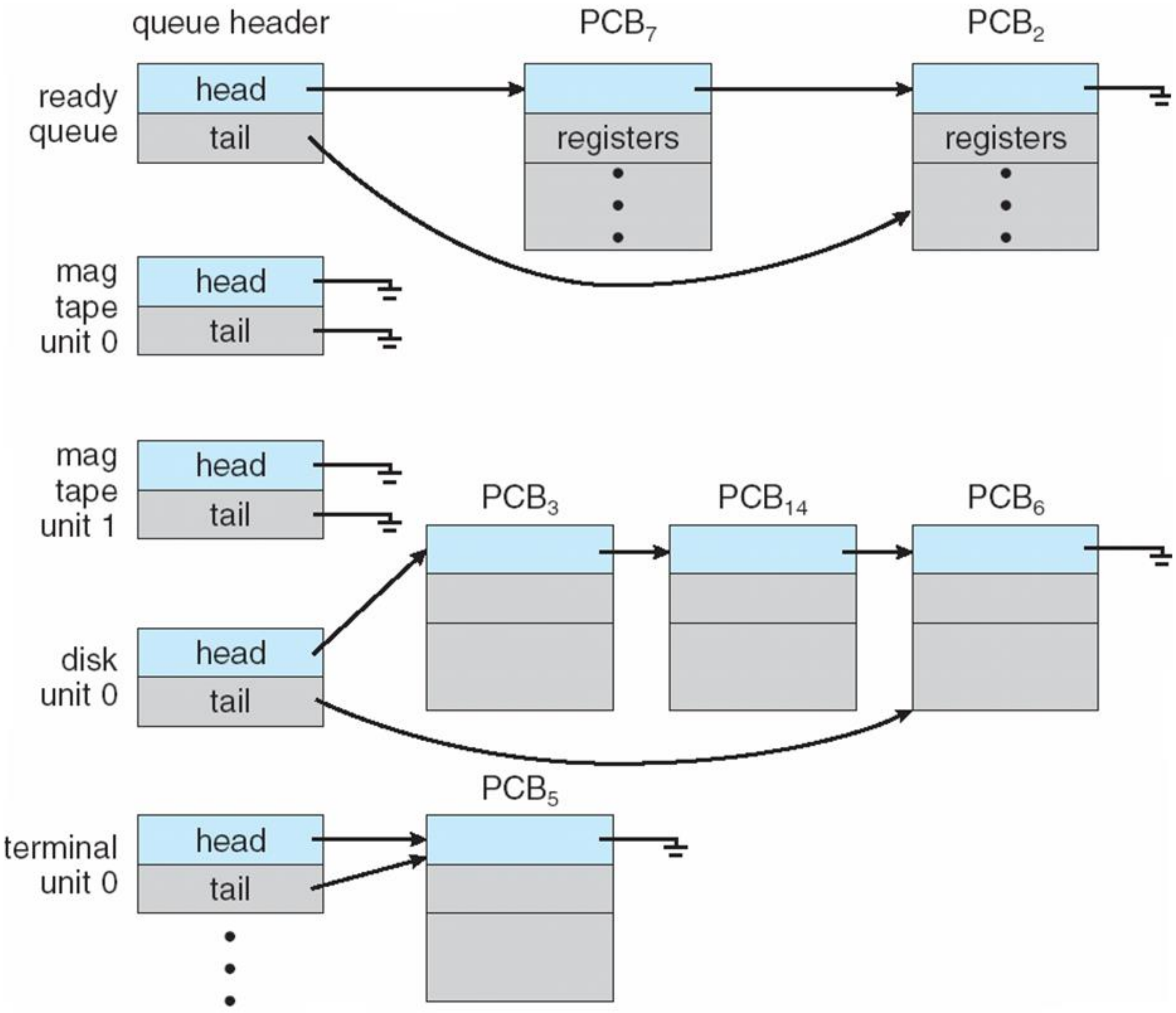


Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes
- For a single-processor system, there will never be more than one running process.
- **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
      - Each device has its own device queue
  - Processes migrate among the various queues

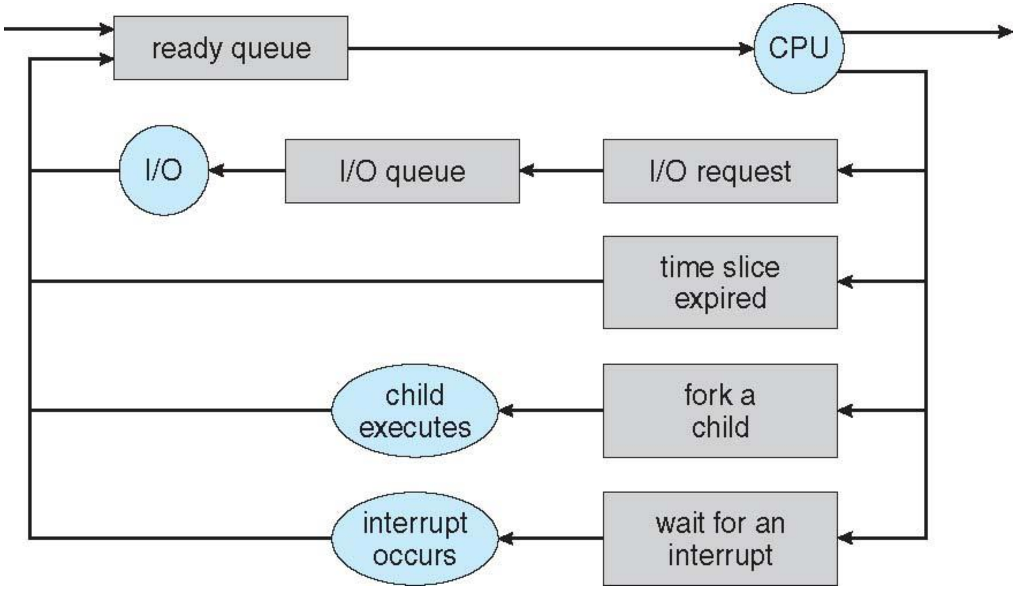
Process Scheduling

- Ready Queue And Various I/O Device Queues



Process Scheduling

- **Queueing diagram:** is a common representation of process scheduling
  - Each rectangular box represents a queue
  - The circles represent the resources that serve the queues,
  - The arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue. It waits there until it is selected for execution, one of several events could occur while executing:
  - The process issues an I/O request, then placed in an I/O queue.
  - The process creates a new child process, wait for the child's termination.
  - The process is removed from the CPU, as a result of an interrupt.



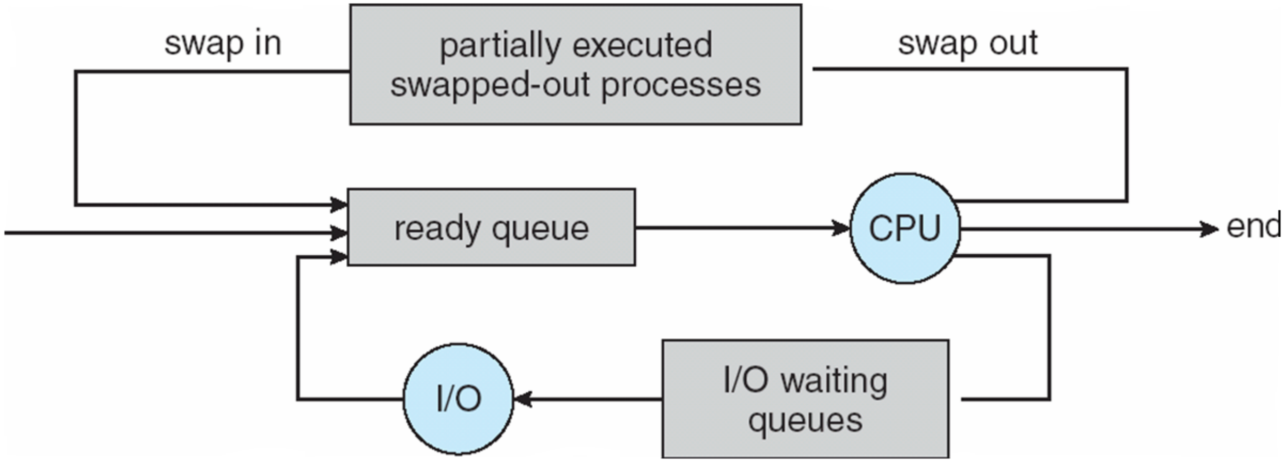
Process Scheduling

Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process from ready processes 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** (the number of processes in memory).
- 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
- The long-term scheduler should select a good ***process mix*** of I/O-bound and CPU-bound processes.

Process Scheduling

- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling
  - **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
    - Remove process from memory, store it on disk, bring it back in from disk to continue execution: this is called **swapping**
      - Swapping may be necessary to improve the process mix



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 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

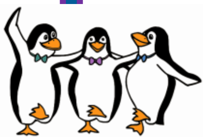
Process Scheduling

- Multitasking in Mobile Systems
  - Some mobile systems (e.g., early version of iOS) allow only one process to run, others are suspended
  - Apple now provides a limited form of multitasking for user applications:
    - 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,

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically

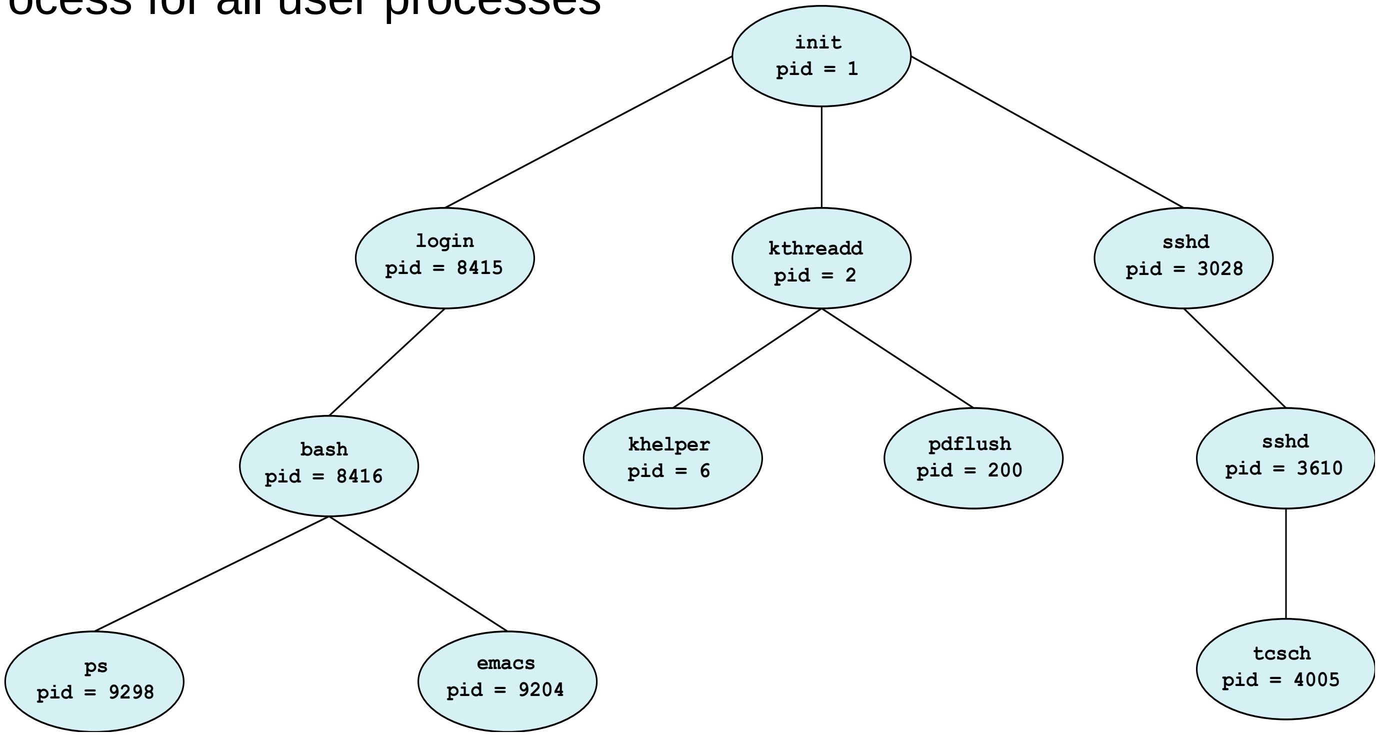


Operations on Processes

Process Creation

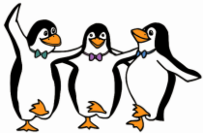
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)

- A Tree of Processes in Linux
  - The ***init*** process (which always has a pid of 1) serves as the root parent process for all user processes



Operations on Processes

- Process Creation
  - Resource sharing options
    - A child process may obtain its resources directly from the OS,
    - A child process may share subset of parent′s resources
      - This prevents any process from overloading the system by creating too many child processes
  - When a process creates a new process, two possibilities for execution exist:
    1. The parent continues to execute concurrently with its children.
    2. The parent waits until some or all of its children have terminated.
  - There are also two address-space possibilities for the new process:
    1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
    2. The child process has a new program loaded into it.

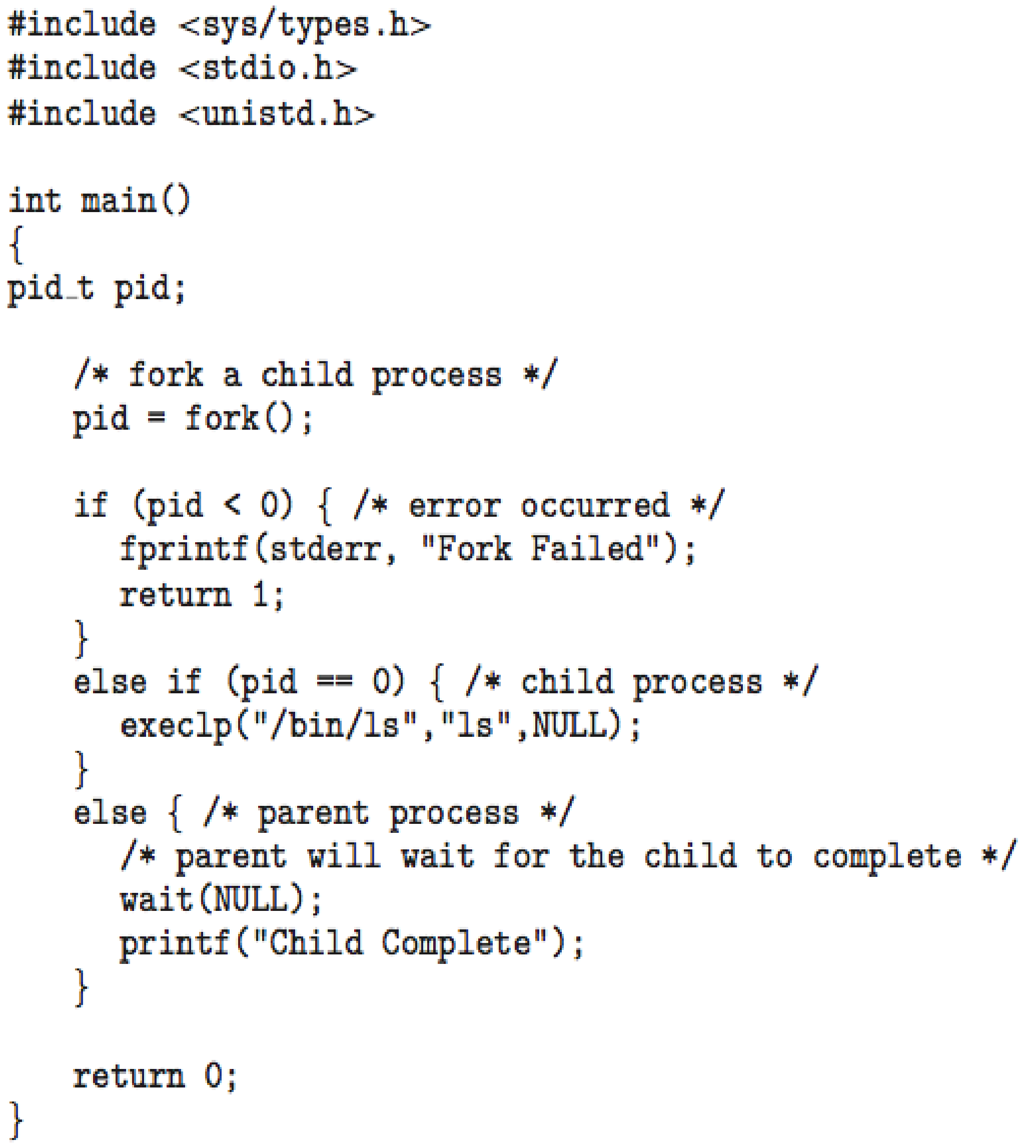


Operations on Processes

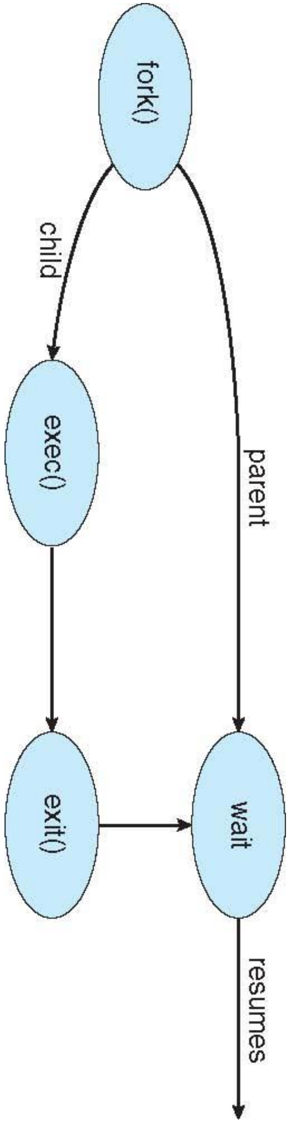
- Process Creation
  - **UNIX examples**
    - *fork():* system call is for creating a new process
      - The new process consists of a copy of the address space of the original process.
      - Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference:
        - the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
    - *exec():* system call which is used after a *fork()* to replace the process' memory space with a new program
      - So the two processes are able to communicate and then go their separate ways.
      - The parent can then create more children; or, if it may issue a wait() system call to move itself off the ready queue until the termination of the child

Operations on Processes

- C Program Forking Separate Process

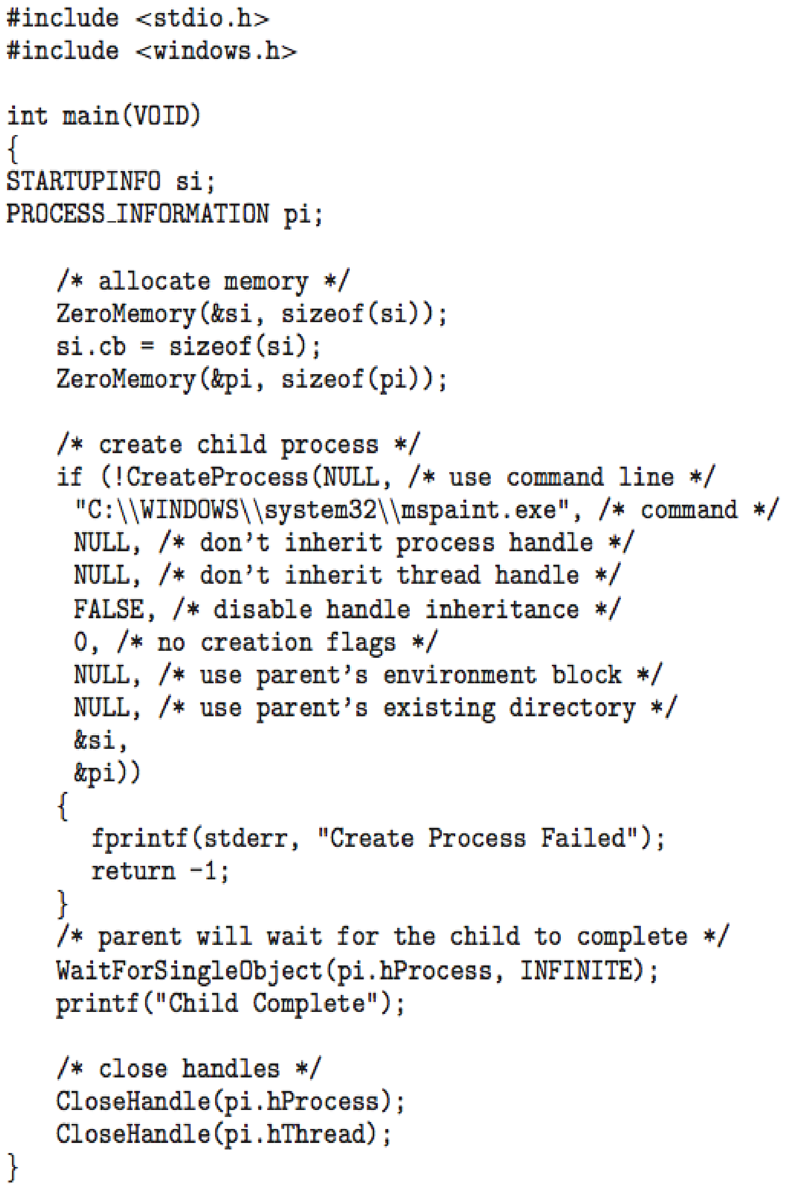


```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

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;
}
```



Operations on Processes

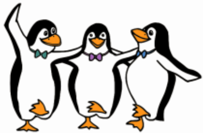
- Creating a Separate Process via Windows API



```
#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);
}
```

Operations on Processes

- 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



Operations on Processes

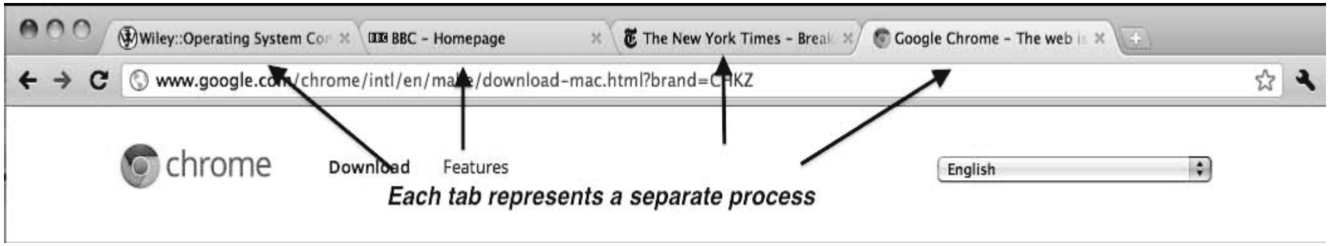
- Process Termination
  - Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
    - This is called **cascading termination.** All children, grandchildren, etc. are terminated.
    - The termination is 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);
```

  - When a process terminates, its resources are deallocated by the OS.
  - But, its entry in the process table must remain there until the parent calls wait().
    - If the parent did not invoke **`wait()`**, the process is called a **zombie**
    - If parent terminated without invoking **`wait()`**, process is an **orphan**
      - The *init* process becomes the parent and issues wait() periodically to clear the orphans

Operations on Processes

- 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


Wiley::Operating System Co
BBC – Homepage
The New York Times – Break
Google Chrome – The web i
www.google.com/chrome/intl/en/make/download-mac.html?brand=CHKZ
chrome
Download
Features
English

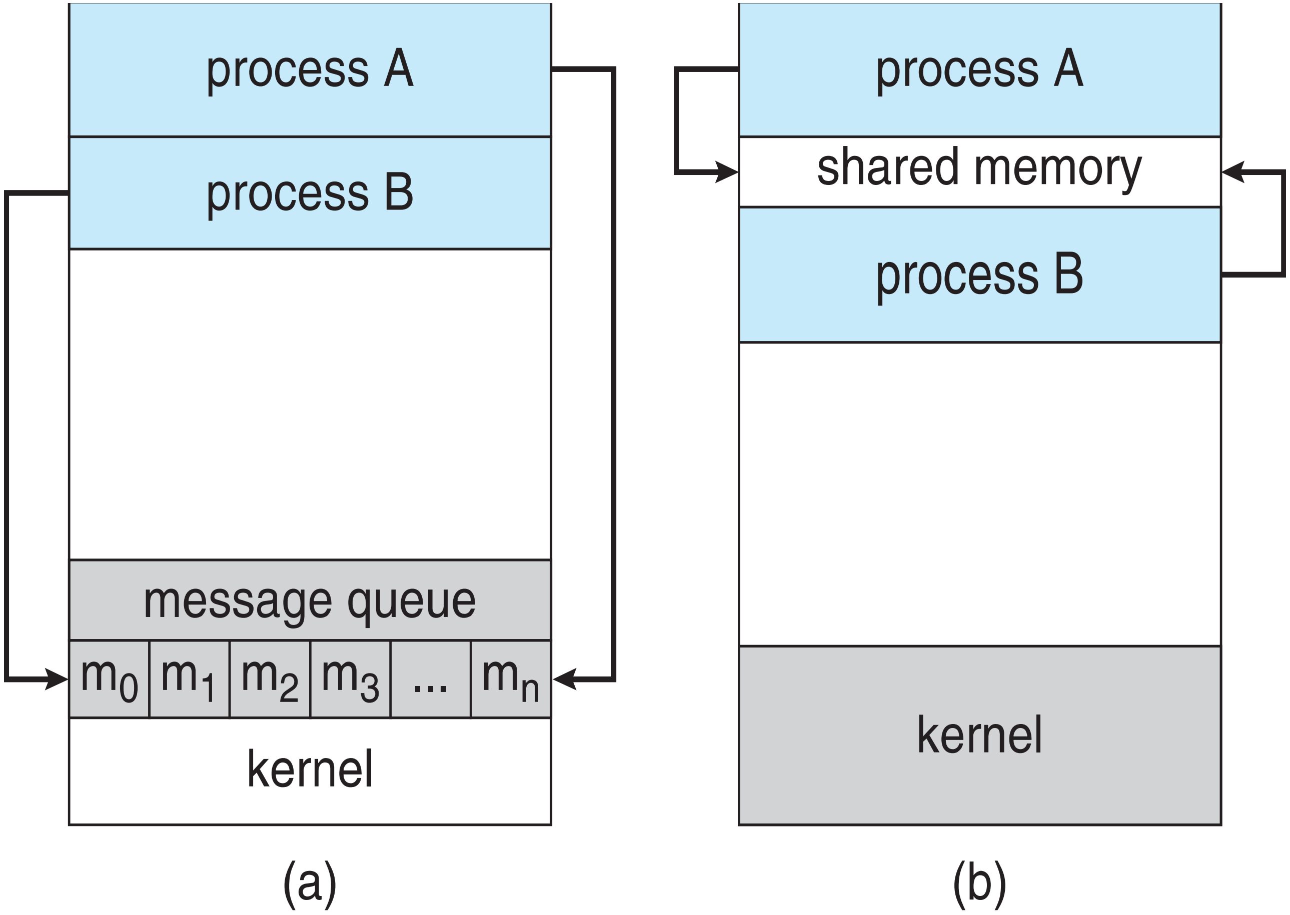
Each tab represents a separate process

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
  - *Independent* process cannot affect or be affected by the execution of another process
  - Cooperating process can affect or be affected by other processes, including sharing data
    - **Reasons for cooperating processes:**
      - Information sharing (e.g. shared file)
      - Computation speedup: break the process into multiple tasks, you need a multicore processor.
      - Modularity: divide the system functions into separate processes or threads
      - Convenience: a user may work on many tasks at the same time
    - Cooperating processes need **interprocess communication** (**IPC**) mechanism to allow them exchange data and information
    - Two models of IPC
      - **Shared memory**
      - **Message passing**

Interprocess Communication

- Communications Models



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

Interprocess Communication

Shared-Memory Systems

- An area of memory shared among the processes that wish to communicate
  - Typically, a shared-memory region resides in the address space of the process creating it.
    - Other processes must attach it to their address space
  - 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.

Interprocess Communication

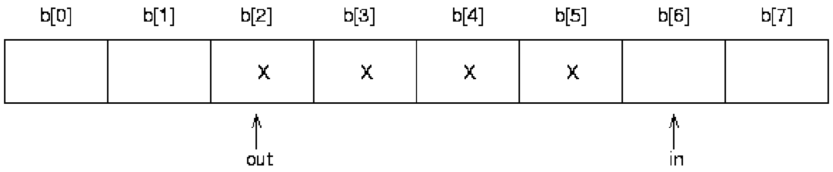
- Shared-Memory Systems
  - **Producer-Consumer Problem**
    - It is a common paradigm for cooperating processes,
      - *producer* process produces information that is consumed by a *consumer* process
      - For example, a compiler may produce assembly code that is consumed by an assembler
      - To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
      - Two types of buffers can be used:
        - **unbounded-buffer** places no practical limit on the size of the buffer
          - The consumer may have to wait for new items, but the producer can always produce new items
        - **bounded-buffer** assumes that there is a fixed buffer size
          - The consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Interprocess Communication

Bounded-Buffer – Shared-Memory Solution

- The shared buffer is implemented as a circular array with two logical pointers: in and out.
  - The variable in points to the next free position in the buffer; out points to the first full position in the buffer.
  - The buffer is empty when in ==out;
  - The buffer is full when ((in + 1) % BUFFER SIZE) == out.
- The following variables reside in a region of memory shared by the producer and consumer processes

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



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

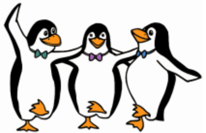
Interprocess Communication

- Bounded-Buffer – Shared-Memory Solution
  - Producer code

```
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 code

```
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

- Message-Passing Systems
  - Mechanism provided by the OS for processes to communicate and to synchronize their actions without sharing the same address space
  - It is useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
    - For example, an Internet chat program
  - Message system – processes communicate with each other without resorting to shared variables
  - Message-passing facility provides at least two operations:
    - **`send`**(*message*)
    - **`receive`**(*message*)
  - The *message* size is either fixed or variable

Interprocess Communication

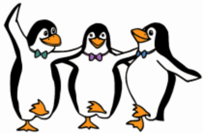
- Message-Passing Systems
  - If processes *P* and *Q* wish to communicate, they need to:
    - Establish a ***communication link*** between them
    - Exchange messages via send/receive
  - Implementation issues:
    - How are links established?
    - Can a link be associated with more than two processes?
    - 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?

Interprocess Communication

- Message-Passing Systems
  - **Implementation of communication link**
    - Physical implementation:
      - Shared memory
      - Hardware bus
      - Network
    - Logical implementation:
      - Direct or indirect communication
      - Synchronous or asynchronous communication
      - Automatic or explicit buffering

Interprocess Communication

- Message-Passing Systems
  - **Naming**: can be direct or indirect communication
    - **Direct communication**
      - Under 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: each process need to know each other identity
        - 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



Interprocess Communication

- Message-Passing Systems
  - **Indirect communication**
    - With indirect communication, messages are sent to and received from mailboxes (also referred to as ports)
      - Each mailbox has a unique id
      - Processes can communicate only if they share a mailbox
    - The send() and receive() primitives are defined as follows:
      - ***send(A, message):*** Send a message to mailbox A.
      - ***receive(A, message):*** Receive a message from mailbox A.
    - 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, with each link corresponding to one mailbox
      - Link may be unidirectional or bi-directional

Interprocess Communication

Message-Passing Systems

- A mailbox that is owned by the operating system is independent and is not attached to any particular process.
- The operating system then must provide a mechanism that allows a process to do the following:
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox
- The process that creates a new mailbox is its owner and is the only one that can receive messages through this mailbox.
  - However, the ownership and receiving privilege may be passed to other processes through appropriate system calls.

Interprocess Communication

Message-Passing Systems

- Mailbox sharing
  - Suppose that processes 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.

Interprocess Communication

Synchronization

- There are different design options for implementing *send()* and *receive()* primitives:
  - **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 resumes operation
      - **Non-blocking receive**: the receiver receives either a valid message, or null message

- **Different combinations possible**
  - If both send and receive are blocking, we have a **rendezvous** between the sender and the receiver.

Interprocess Communication

- Synchronization
  - Producer-consumer becomes trivial when using blocking send() and receive()
    - The producer process using message passing.

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

    - The consumer process using message passing.

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

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

Interprocess Communication

- Buffering
  - Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
  - **Such queues can be implemented in three ways:**
    1. Zero capacity: no messages are queued on a link.
       - Sender must wait for receiver (rendezvous)
    2. Bounded capacity: queue has finite length of *n* messages
       - Sender must wait if link is full, otherwise can resume operation
    3. Unbounded capacity: infinite length
       - Sender never waits

Examples of IPC Systems - POSIX

- Several IPC mechanisms are available for POSIX systems,
  - shared memory and message passing
- POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file
- POSIX Shared Memory
  - Process first creates shared memory segment

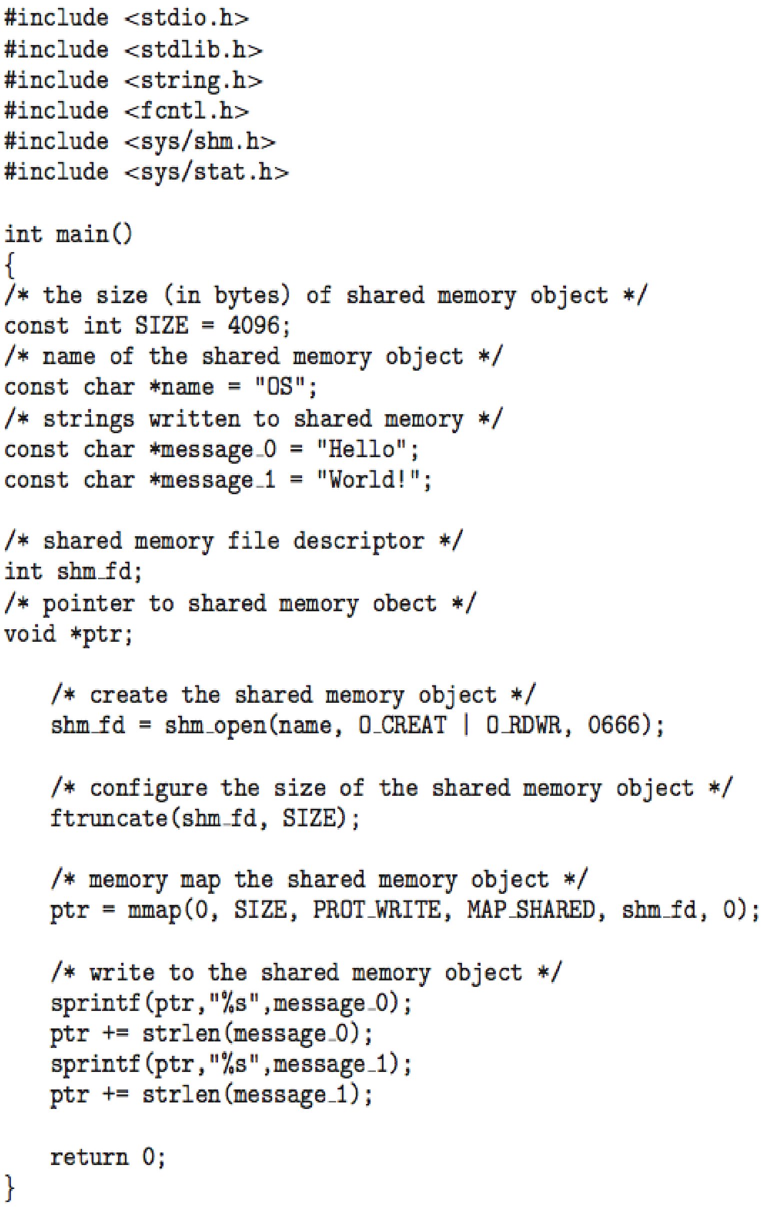
    ```
    shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
    ```
  - Also used to open an existing segment to share it
  - Set the size of the object

    ```
      ftruncate(shm fd, 4096);
    ```
  - Now the process could write to the shared memory

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

Examples of IPC Systems - POSIX

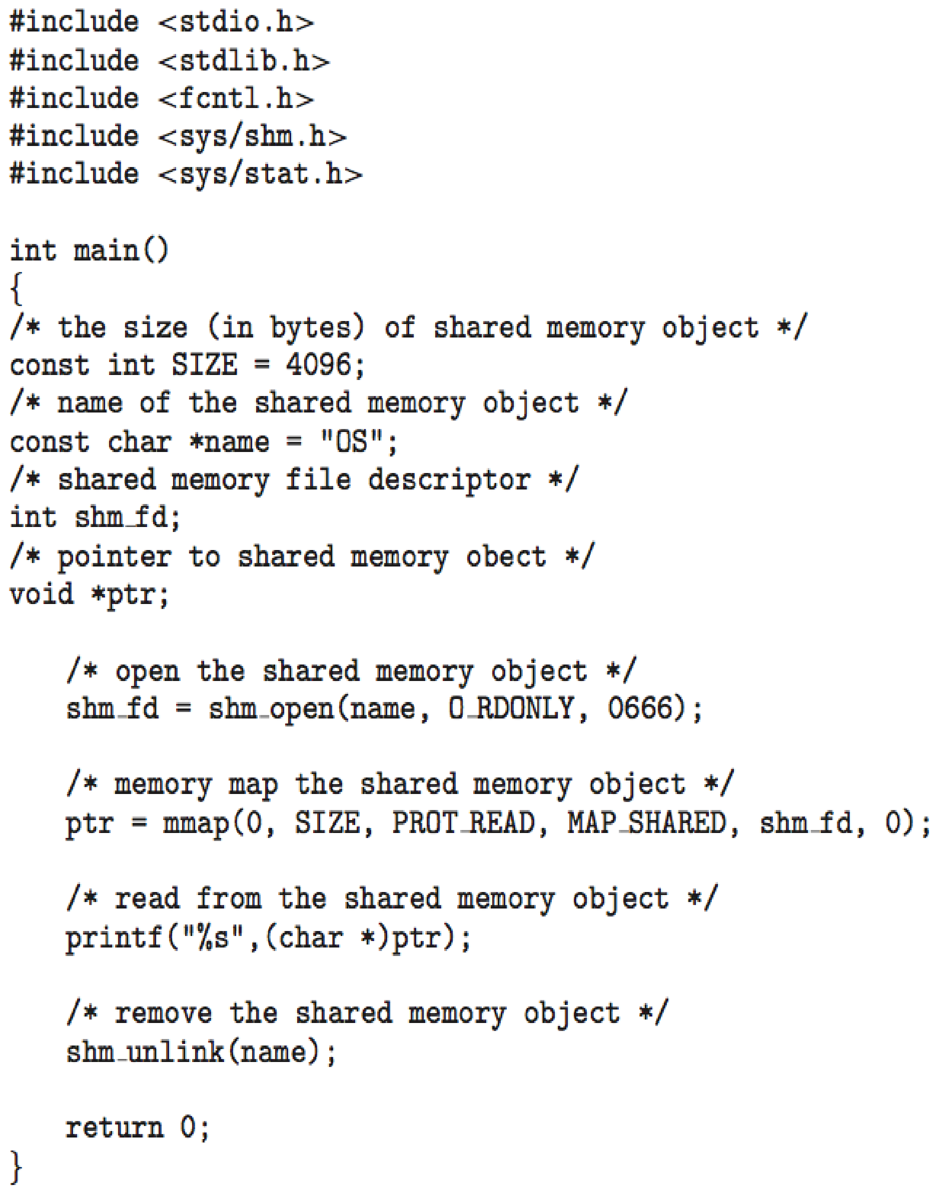
- **IPC POSIX Producer**



```
#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 obect */
void *ptr;

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

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

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

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

   return 0;
}
```

Examples of IPC Systems - POSIX

- 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 obect */
void *ptr;

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

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

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

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

   return 0;
}
```

Examples of IPC Systems - Mach

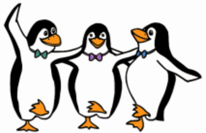
- Mach communication is message based
  - Even system calls are made by messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes needed for commuication, created via

    `port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely until there is room in the mailbox
    - Wait at most *n* milliseconds
    - Return immediately
    - Temporarily cache a message

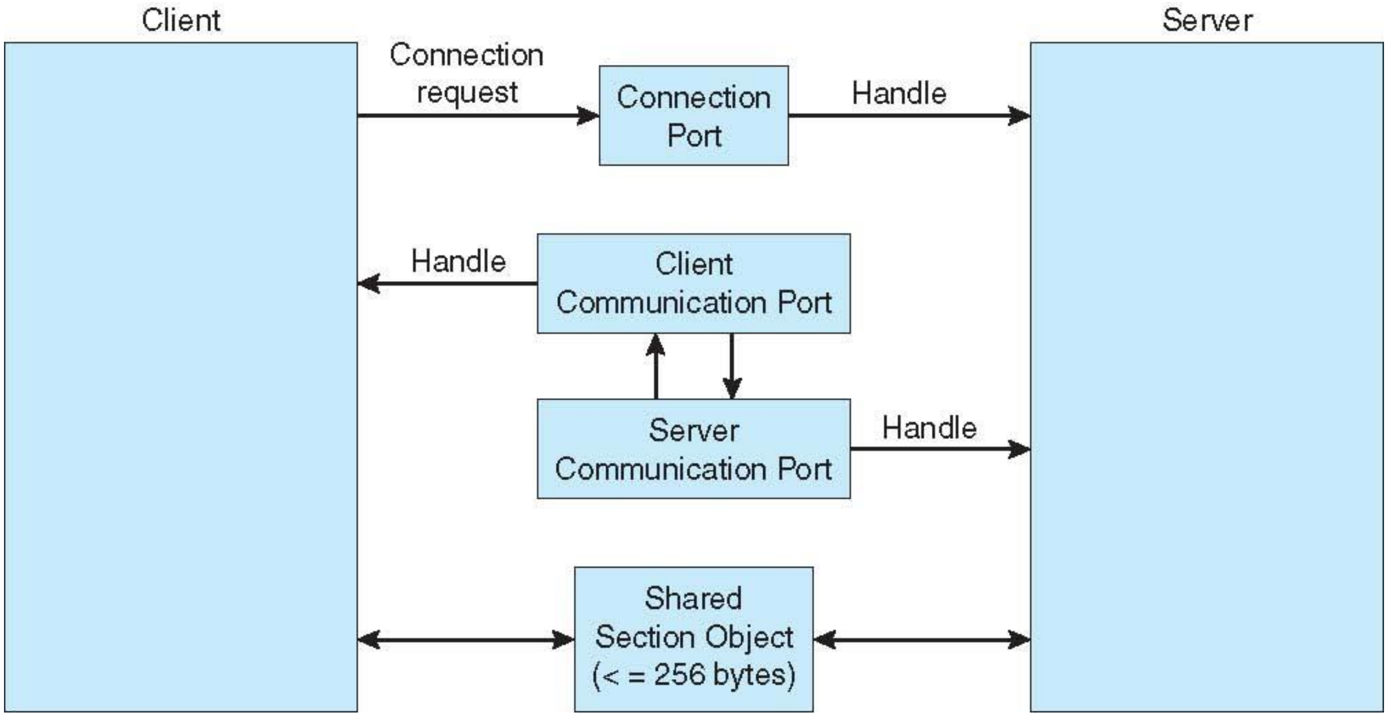
Examples of IPC Systems - Windows

- 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
  - Windows uses two types of ports: connection ports and communication ports.
  - 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.



Examples of IPC Systems - Windows

- **Local Procedure Calls in Windows**


Client
Server
Connection request
Connection Port
Handle
Handle
Client Communication Port
Server Communication Port
Handle
Shared Section Object (< = 256 bytes)

Communications in Client-Server Systems

- The shared memory and message passing can be used for communication in client–server systems
- There are other strategies for communication in client–server systems
  - Sockets
  - Remote Procedure Calls
  - Pipes
  - Remote Method Invocation (Java)

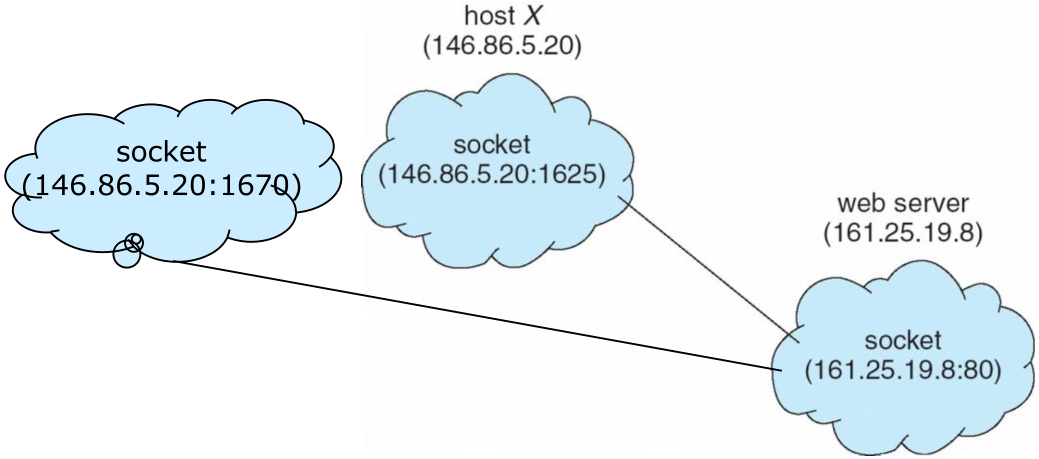
Communications in Client-Server Systems

Sockets

- A **socket** is defined as an endpoint for communication
- A pair of processes communicating over a network employs a pair of sockets
- It is identified by the 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 connections must be unique
- All ports below 1024 are ***well known***, used for standard services
  - FTP server listens to port 21; 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

Communications in Client-Server Systems

- Sockets
  - Sockets use a client–server architecture: The server waits for incoming client requests by listening to a specified port
  - When a client process initiates a request for a connection, it is assigned a port by its host computer.
    - Example,
      - If a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625.
      - The connection consists of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server.

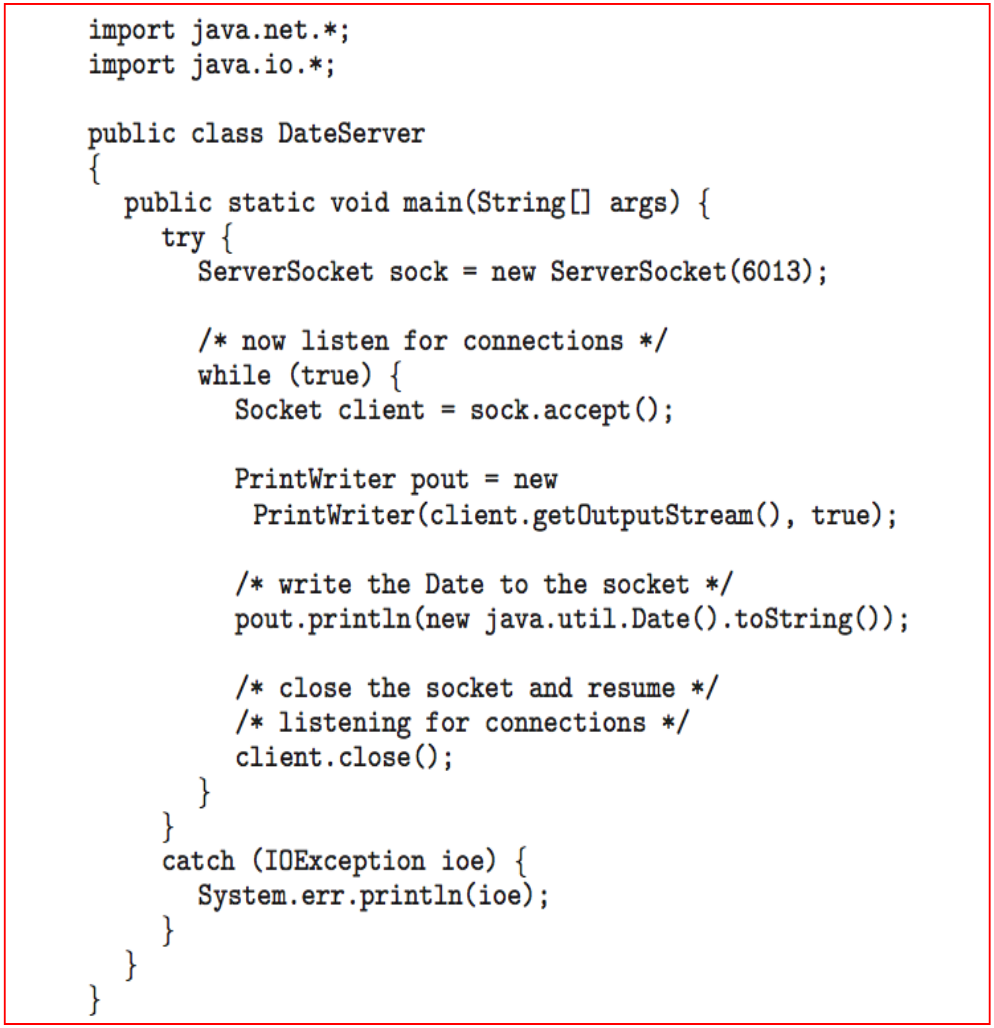


Communications in Client-Server Systems

- Sockets in Java
  - Three types of sockets
    - **Connection-oriented** (**TCP**): implemented with the Socket class
    - **Connectionless** (**UDP**): use the DatagramSocket class
      - `MulticastSocket` class: is a subclass of the DatagramSocket class.
        - data can be sent to multiple recipients
    - **Example**:
      - Date server that uses connection-oriented TCP sockets.
        - The operation allows clients to request the current date and time from the server.

Communications in Client-Server Systems

- Sockets in Java
  - Consider this “Date” server:
  - The operation allows clients to request the current date and time from the server.
  - The server creates a ServerSocket that specifies that it will listen to port 6013. The server then begins listening to the port with the accept() method. The server blocks on the accept() method waiting for a client to request a connection. When a connection request is received, accept() returns a socket that the server can use to communicate with the client.

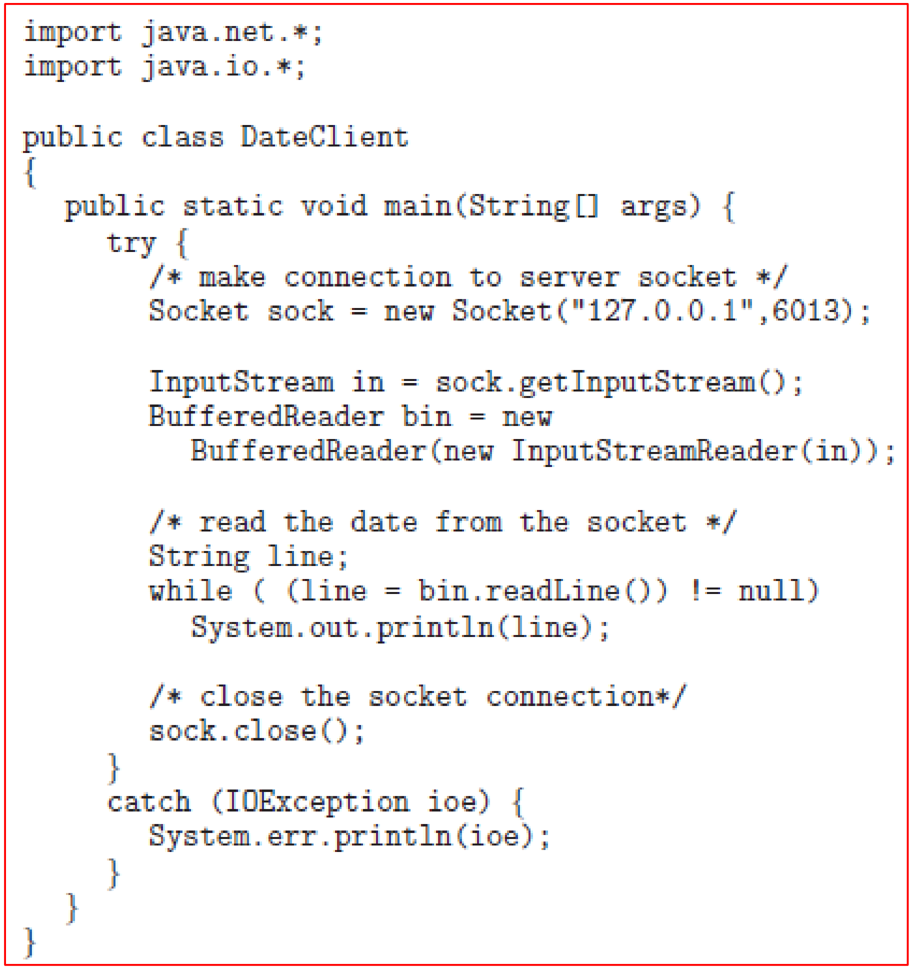


```java
import java.net.*;
import java.io.*;

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);
      }
   }
}
```

Communications in Client-Server Systems

Sockets in Java

- Consider this “Date” client:
- A client communicates with the server by creating a socket and connecting to the port on which the server is listening.
- The client creates a Socket and requests a connection with the server at IP address 127.0.0.1 on port 6013.
- Once the connection is made, the client can read from the socket using normal stream I/O statements.
- Close the connection



```
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);
      }
   }
}
```

Communications in Client-Server Systems

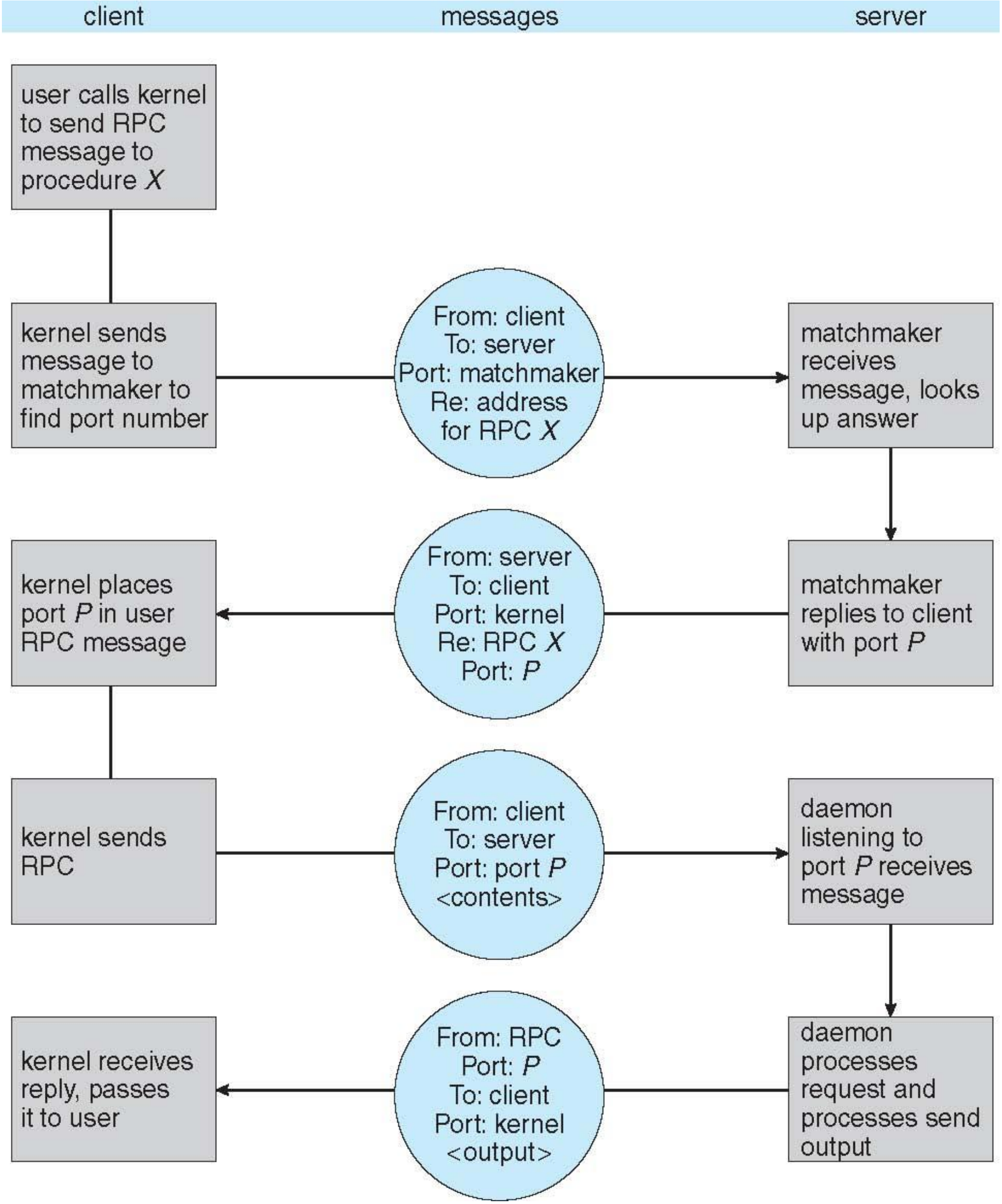
- Remote Procedure Calls (RPC)
  - RPC abstracts procedure calls between processes on networked systems
    - It uses ports for service differentiation
  - A port is a number included at the start of a message packet
  - The RPC system hides the details that allow communication to take place by providing a stub on the client side.
    - When the client invokes a remote procedure, the RPC system calls the appropriate stub
    - 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**)

Communications in Client-Server Systems

- Remote Procedure Calls (RPC)
  - 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

Communications in Client-Server Systems

- Execution of RPC



client

messages

server

user calls kernel to send RPC message to procedure *X*

kernel sends message to matchmaker to find port number

From: client
To: server
Port: matchmaker
Re: address for RPC *X*

matchmaker receives message, looks up answer

matchmaker replies to client with port *P*

From: server
To: client
Port: kernel
Re: RPC *X*
Port: *P*

kernel places port *P* in user RPC message

kernel sends RPC

From: client
To: server
Port: port *P*
<contents>

daemon listening to port *P* receives message

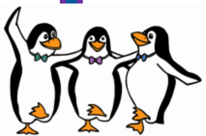
daemon processes request and processes send output

From: RPC
Port: *P*
To: client
Port: kernel
<output>

kernel receives reply, passes it to user

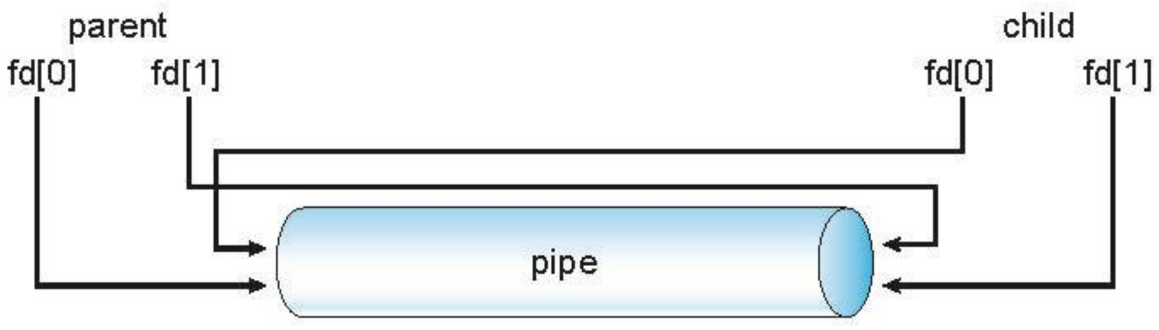
Communications in Client-Server Systems

- Pipes
  - Acts as a conduit allowing two processes to communicate
  - In implementing a pipe, four issues must be considered::
    - Is communication unidirectional or bidirectional?
    - In the case of two-way communication, is it half or full-duplex?
    - Must there exist a relationship (such as, ***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.



Communications in Client-Server Systems

- 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**

Communications in Client-Server Systems

Ordinary Pipes

- On UNIX systems, ordinary pipes are constructed using the function

  pipe(int fd[])

  - It creates a pipe that is accessed through the int fd[] file descriptors: fd[0] is the read-end of the pipe, and fd[1] is the write-end.
  - pipes can be accessed by read() and write() system calls just like files

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define BUFFER SIZE 25
#define READ END 0
#define WRITE END 1
int main(void)
{
char write msg[BUFFER SIZE] = "Greetings";
char read msg[BUFFER SIZE];
int fd[2];
pid t pid;
/* create the pipe */
if (pipe(fd) == -1) {
fprintf(stderr,"Pipe failed");
return 1;
}
/* fork a child process */
pid = fork();
```

```
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
return 1;
}
if (pid > 0) { /* parent process */
/* close the unused end of the pipe */
close(fd[READ END]);
/* write to the pipe */
write(fd[WRITE END], write msg, strlen(write msg)+1);
/* close the write end of the pipe */
close(fd[WRITE END]);
}
else { /* child process */
/* close the unused end of the pipe */
close(fd[WRITE END]);
/* read from the pipe */
read(fd[READ END], read msg, BUFFER SIZE);
printf("read %s",read msg);
/* close the write end of the pipe */
close(fd[READ END]);
}
return 0;}
```

Communications in Client-Server Systems

- 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

- Named Pipes
  - On **UNIX**:
    - Referred to as FIFOs.
    - Once created, they appear as typical files in the file system.
    - A FIFO is created with the mkfifo() system call and manipulated with the ordinary open(), read(), write(), and close() system calls.
    - They are half-duplex. For full duplex use two FIFOs
    - Can be used in the same machine only, otherwise use sockets for inter machine communication
    - Example: ls | more
      - Setting up a pipe (|) between the ls and more commands (which are running as individual processes)
      - The *ls* command serves as the producer, and its output is consumed by the *more* command

Communications in Client-Server Systems

- Named Pipes
  - On **Windows**:
    - Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines.
    - created with the CreateNamedPipe() function, and a client can connect to a named pipe using ConnectNamedPipe().
    - Communication over the named pipe can be accomplished using the ReadFile() and WriteFile() functions.
    - Example: dir | more