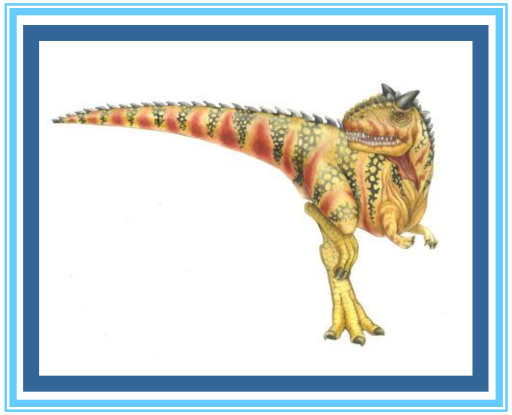
Chapter 3: Processes



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, 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**

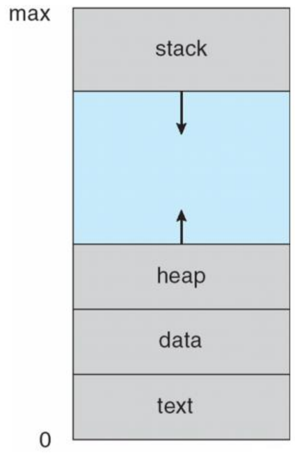
- Textbook uses the terms job and process almost interchangeably
- Process – a program in execution
  -> process execution must **progress** in sequential fashion
- Process has multiple parts

Static info.

  - The program code, also called **text section**
  - Current activity including **program counter**, **processor registers**
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables

Static info.

  - **Data section** containing global variables
  - **Heap** containing dynamically allocated memory during run time (this is not Heap structure)



Process in Memory

Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**) while 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, or depending on OS and its user interfaces
- One program can be run as multiple processes
  - Multiple users executing the same program
  - A user executing the same program multiple times

Process State

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

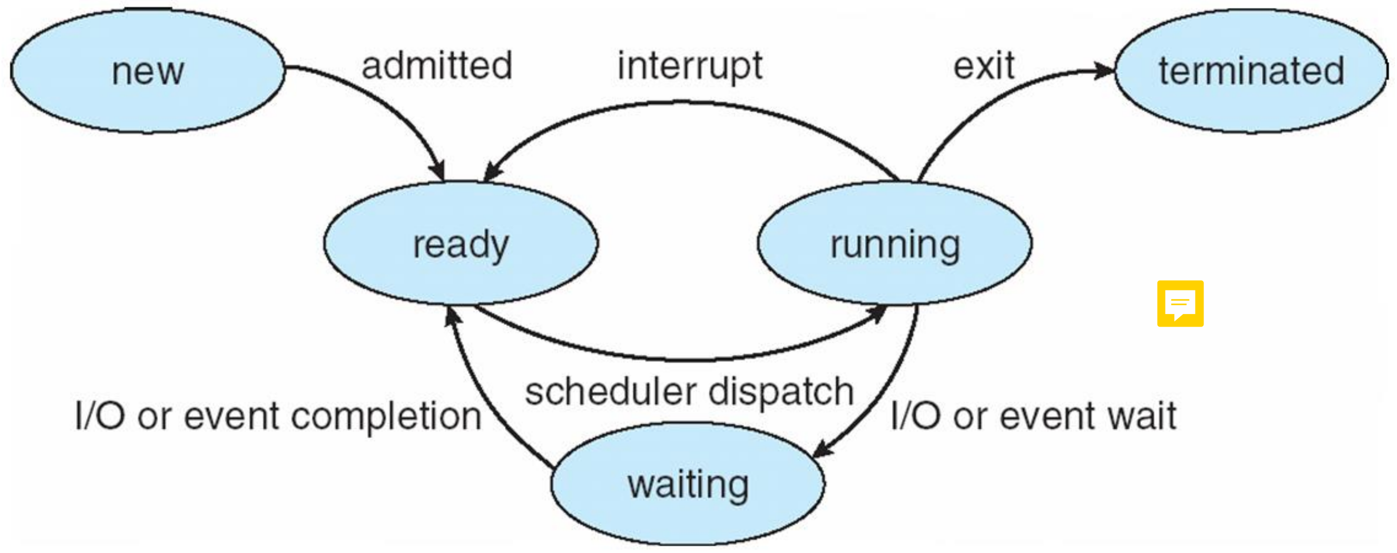
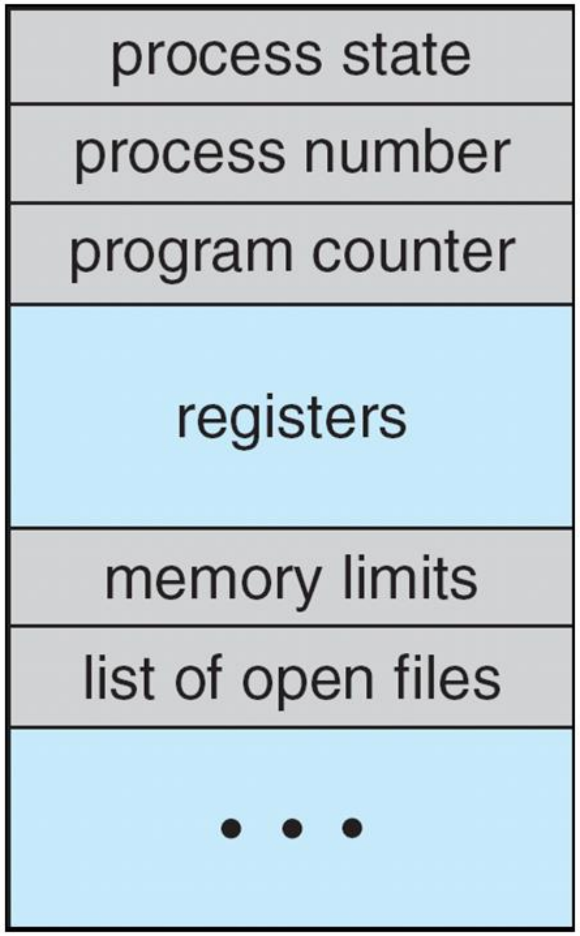


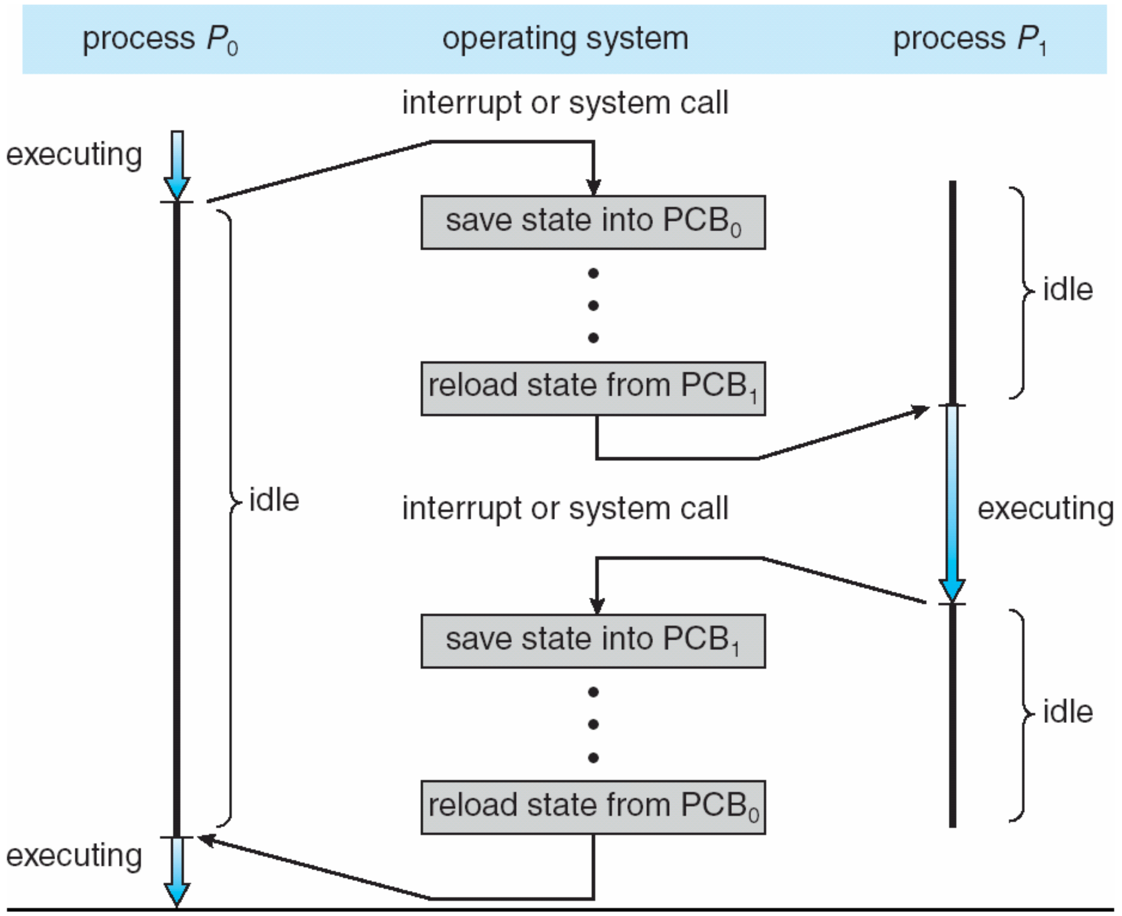
Diagram of Process State

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 execute next
- 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 be executed at once (not simultaneous)

    -> **threads**

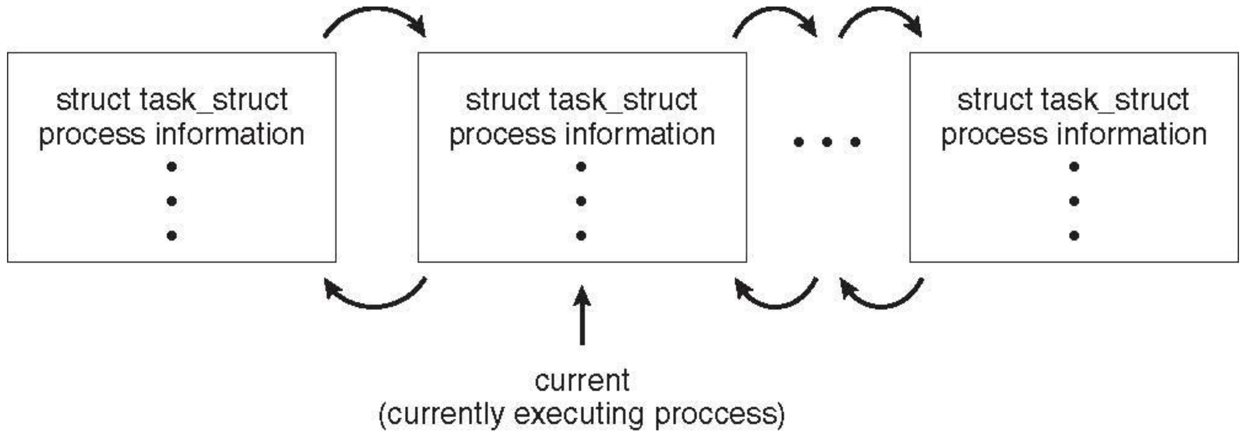
- Must then have storage for thread details, multiple program counters in PCB

  -> See next chapter

Process Control Block in Linux

Represented by the C structure `task_struct`

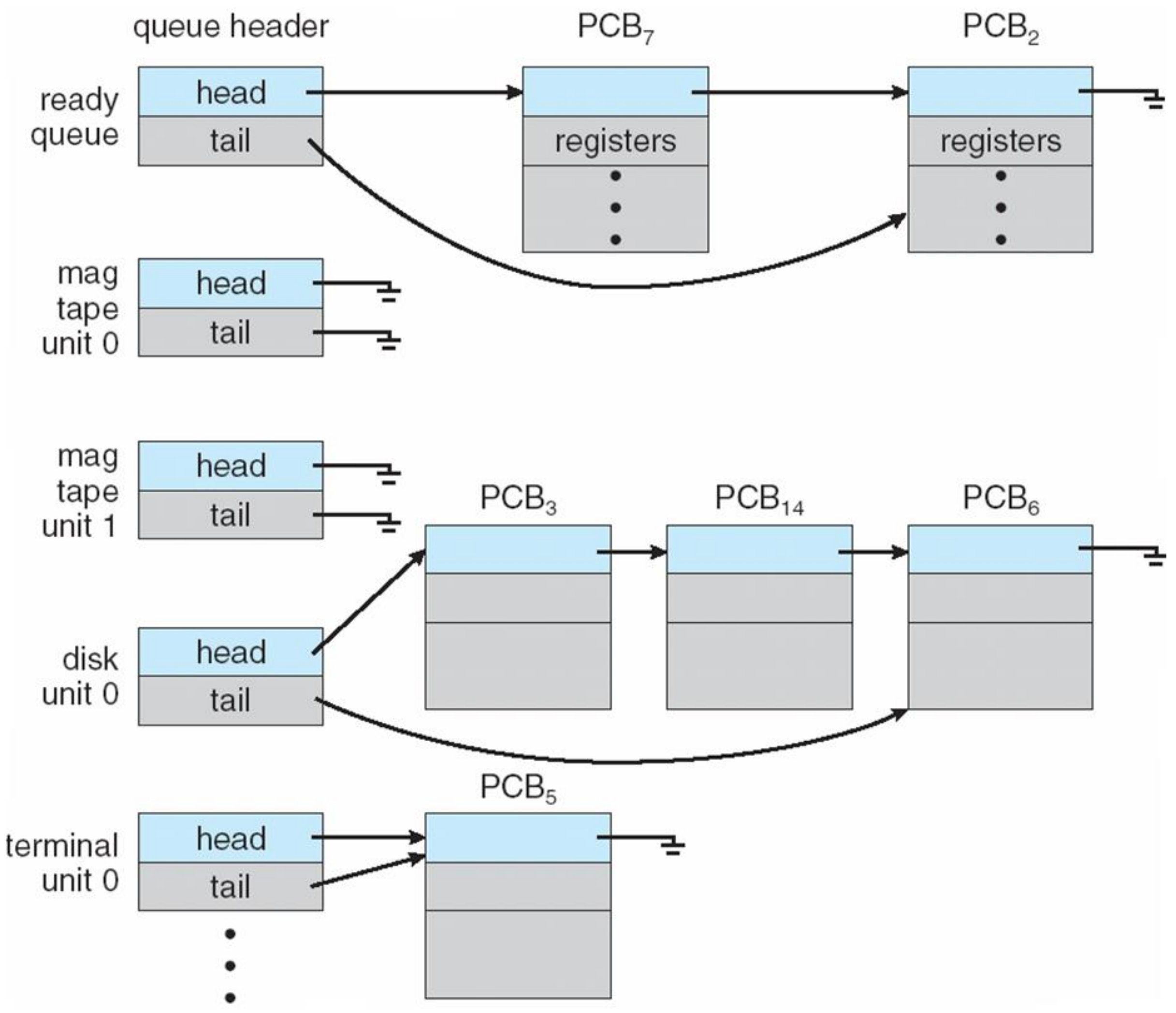
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct 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 executed and waiting for memory allocation
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to running
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues for their whole lifetime

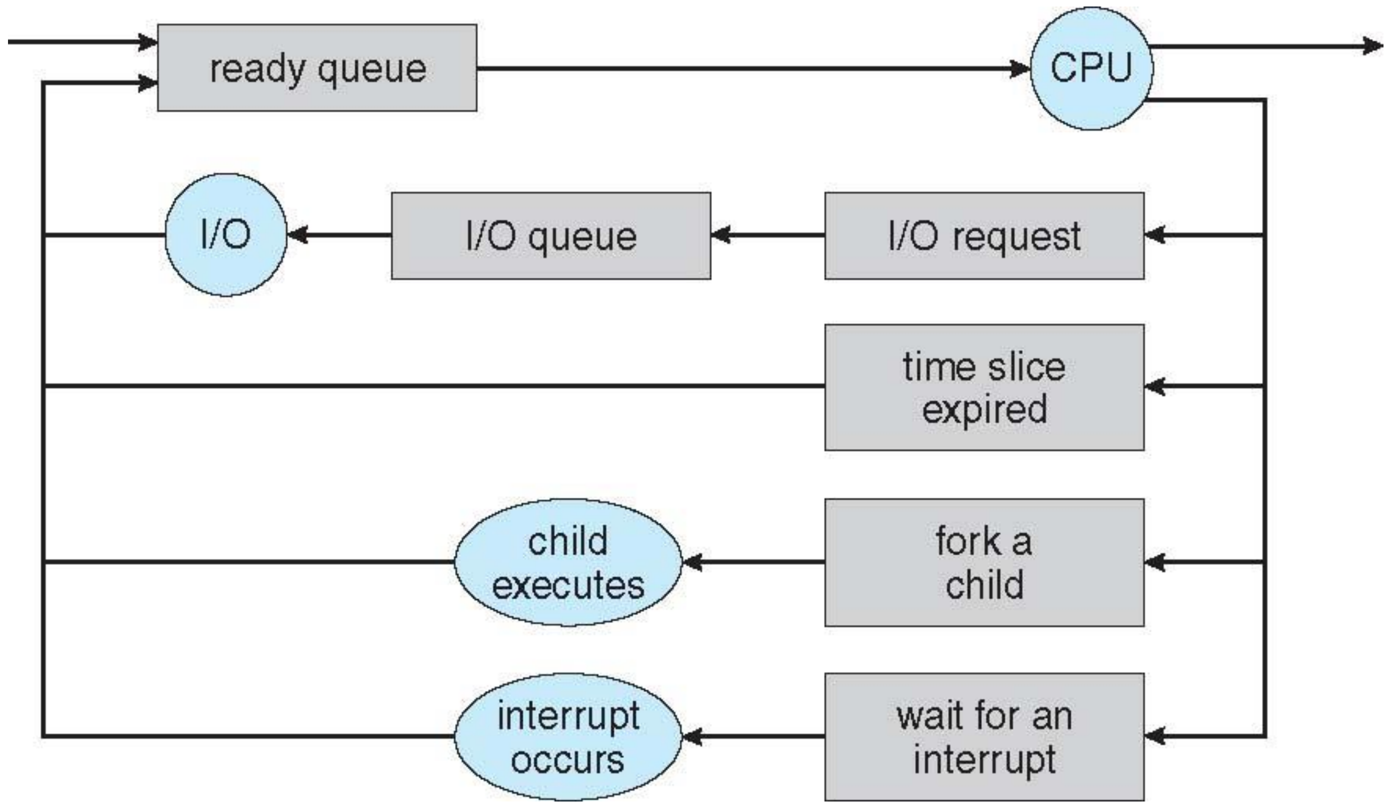
Ready Queue And Various I/O Device Queues



Processes waiting for devices (CPU, tape, disk, terminal, …)

Representation of Process Scheduling

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

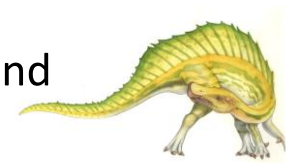


Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Usually being invoked frequently (milliseconds) $\Rightarrow$ (must be fast)

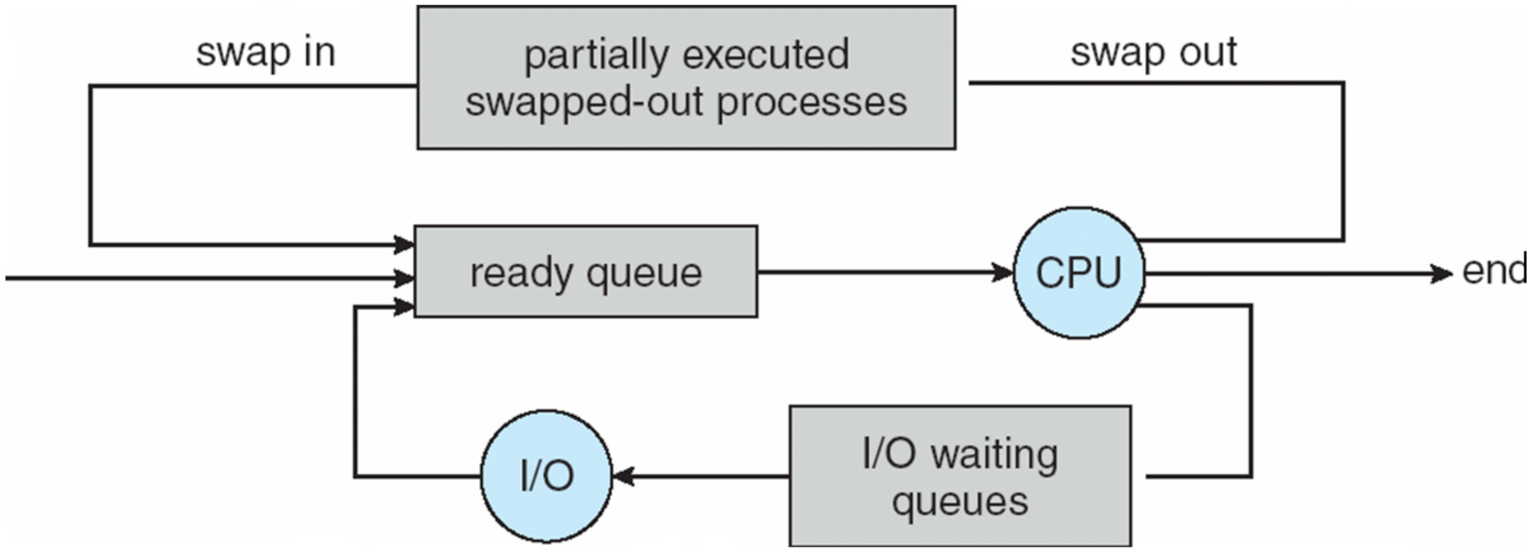
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Usually being invoked infrequently (sec, min) $\Rightarrow$ (can be slow)
  - The long-term scheduler controls the **degree of multiprogramming**

- 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; a few very long CPU bursts
  - Long-term scheduler strives for good process mix of I/O-bound and CPU-bound processes



Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if the degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

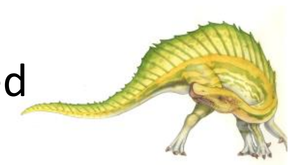


Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one user process to run in foreground, other user processes suspended because of the constraints imposed on mobile devices such as screen real estate limits

- iOS 4 started to provide a limited form of multitasking for user applications
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– running in memory 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 is represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex OS and PCB structure

    -> The longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU

    -> Multiple contexts can be loaded at once, so a process that is about to be run can be pre-loaded before execution

Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

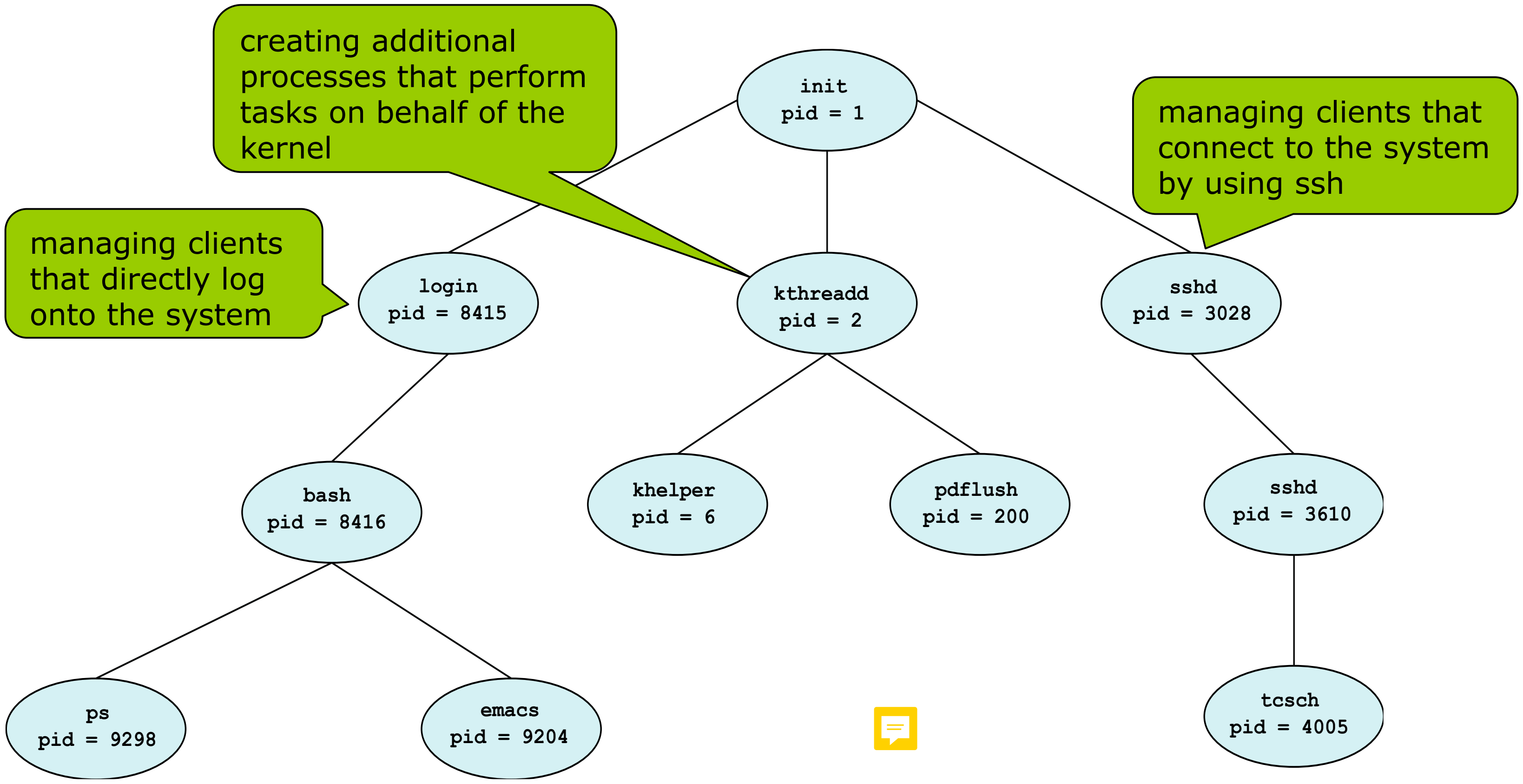
Process Creation

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

- Resource sharing options
  1) Parent and children share all resources
  2) Children share subset of parent's resources
  3) Parent and children share no resources (independent processes)

- Execution options
  1) Parent and children execute concurrently
  2) Parent waits until children terminate

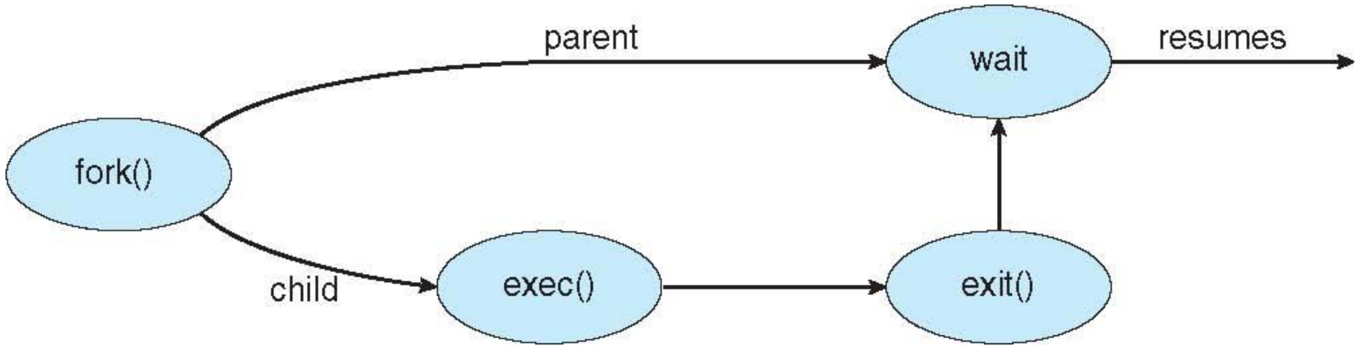
A Tree of Processes in Linux


creating additional processes that perform tasks on behalf of the kernel
managing clients that connect to the system by using ssh
managing clients that directly log onto the system
init pid = 1
login pid = 8415
kthreadd pid = 2
sshd pid = 3028
bash pid = 8416
khelper pid = 6
pdflush pid = 200
sshd pid = 3610
ps pid = 9298
emacs pid = 9204
tcsch pid = 4005

Process Creation (Cont.)

- Address space
  1) Child that is a duplicate of its parent
  2) 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;
}
```

```
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;
```

Parent gets pID of the child process from 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;
```

Child gets 0 from fork()

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);
}
```

loading a specified program ("mspaint.exe") into the address space of the child process at process creation

passed a handle of the child process

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()**); parent waits)
  - 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

    -> Cascading termination: All children, grandchildren, etc. are terminated.

    -> The termination is 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 as:

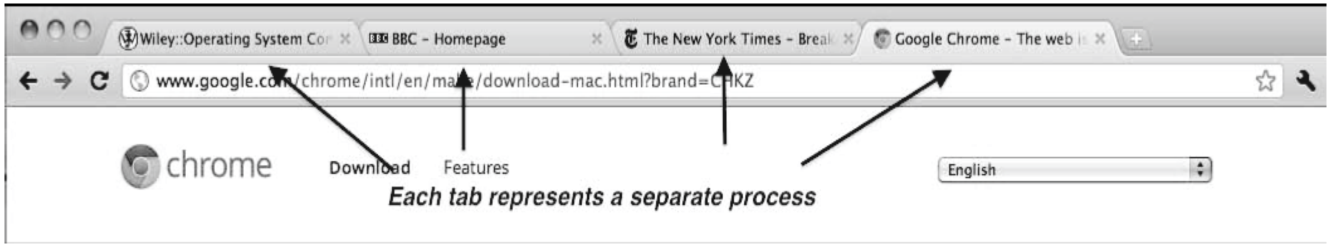
  pid = wait(&status);

- A child process that has terminated, but whose parent has not yet called **wait()** (The parent did not retrieve the child process's status data), is known as a **zombie** process
  - The child process's information is still remained

- If a parent process is terminated without invoking wait, its child process is an **orphan**
  - The **init** process becomes the orphan's parent

Multiprocess Architecture – Chrome Browser

- Many web browsers run as a single process (some still do)
  - If one web site causes trouble, entire browser gets stuck or crashed
- Google Chrome Browser runs with multi-processes in 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 is created for each website opened
    - Runs in **sandbox** (a controlled or emulated section of the system) restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in



Wiley::Operating System Cor
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

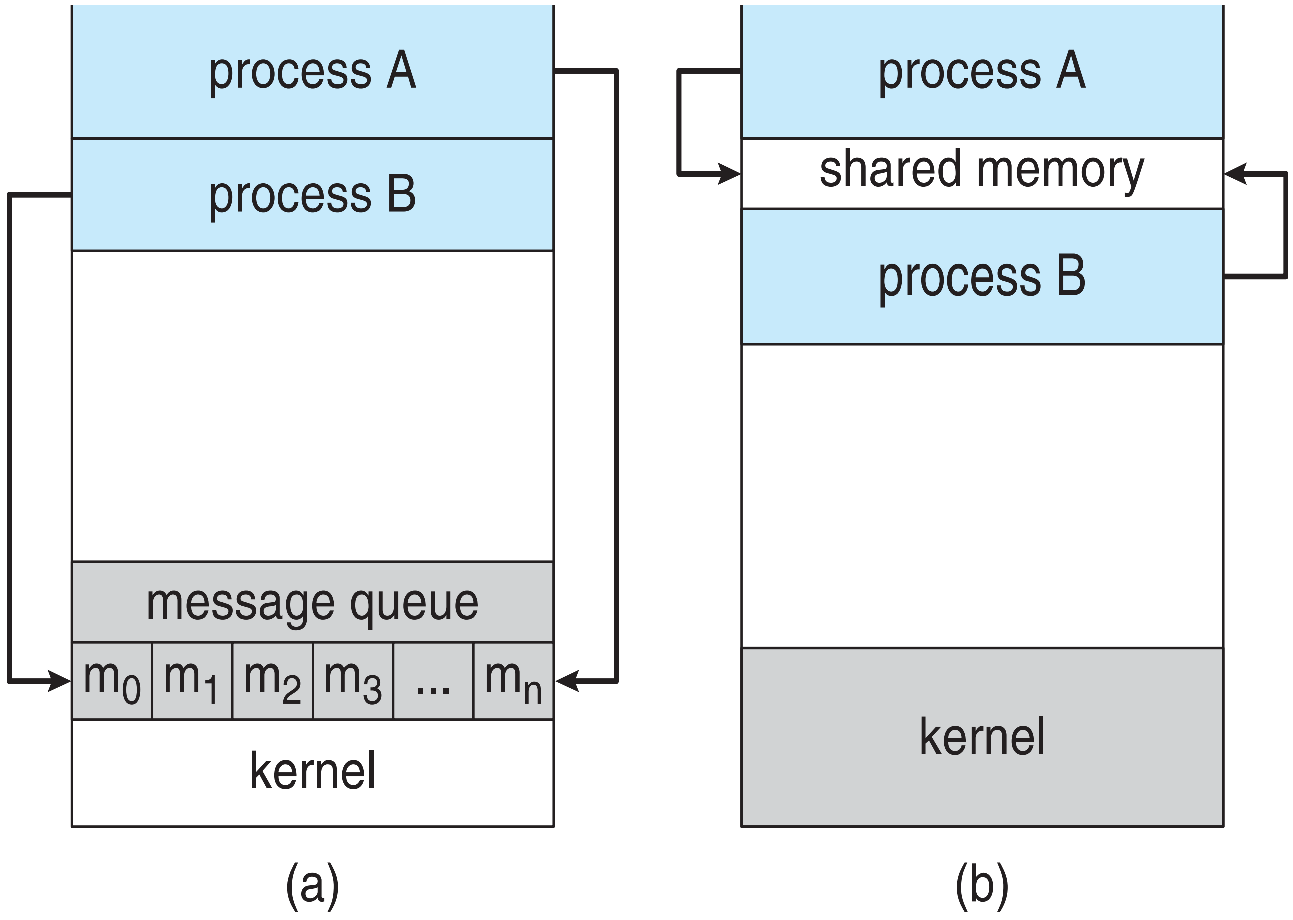
Interprocess Communication

- Processes within a system may be independent or cooperating
- 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**

Communication Models

(a) Message passing (b) shared memory



Producer-Consumer Problem

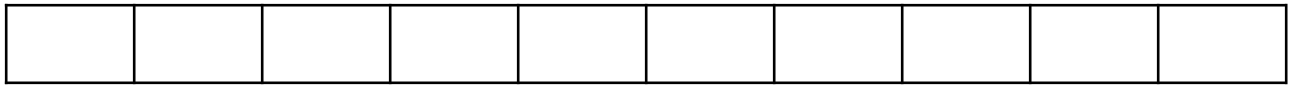
- Paradigm for cooperating processes, a producer process produces information that is consumed by a consumer process
  - **Unbounded-buffer** places no practical limit on the size of the buffer
  - **Bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

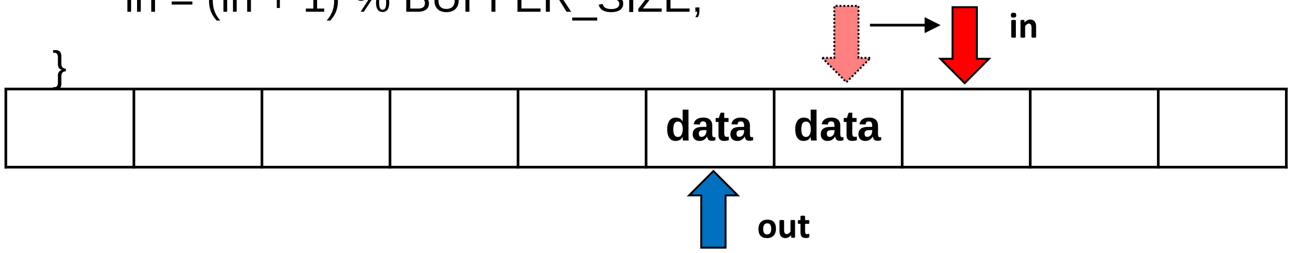
- The solution in the next slides is correct, but can only use **BUFFER_SIZE-1** elements



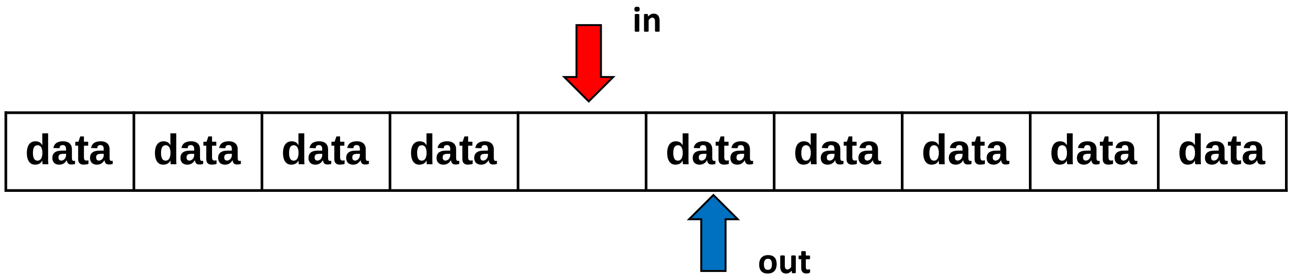
Buffer with size = 10

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;
}
```



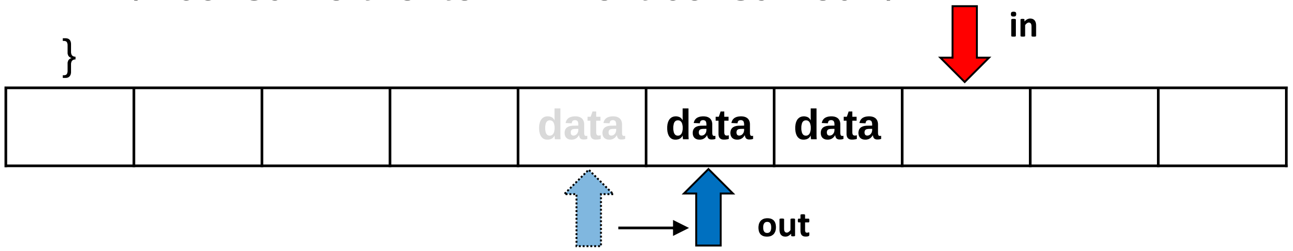
Producer generates an item



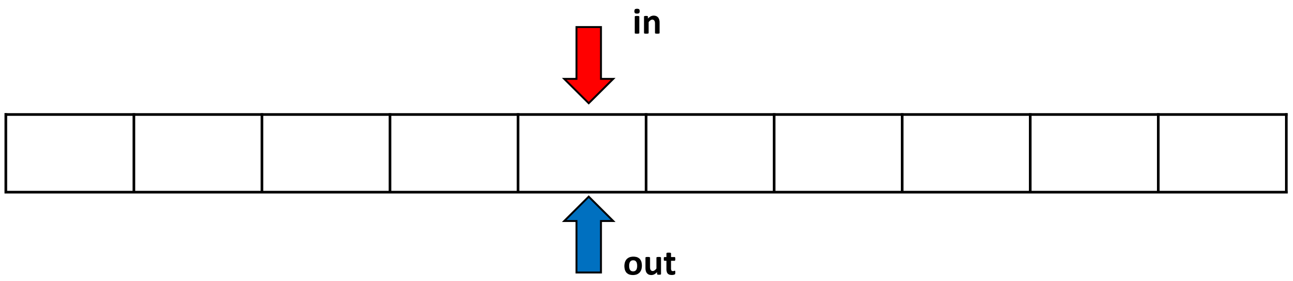
Producer stops producing items

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 */
}
```



Consumer consumes an item



Consumer stops because there are no data available

Interprocess Communication – Shared Memory

- An area of memory is shared among the processes that wish to communicate
- The communication is under the control of the user processes, not by the operating system
- Major issue is to provide a 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 service 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 to establish the links?
    - 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?

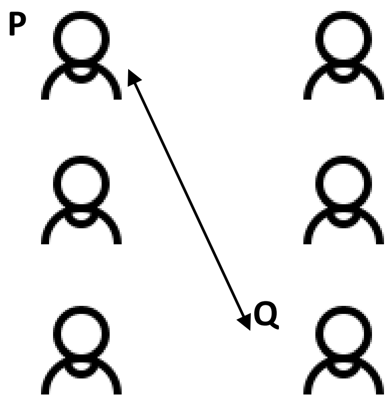
Message Passing (Cont.)

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

Direct Communication

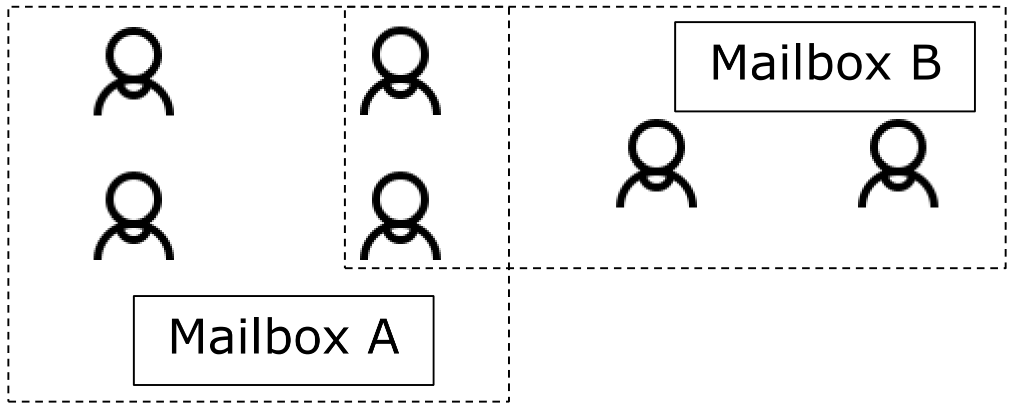
- Processes must name each other explicitly:
  - **send** (P, message) – send a message to process P
  - **receive** (Q, message) – receive a message from process Q

- 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 (or ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link is established only if processes share a common mailbox
  - A link (corresponding to one mailbox) may be associated with many processes
  - Each pair of processes may communicate via several different mailboxes
  - 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:

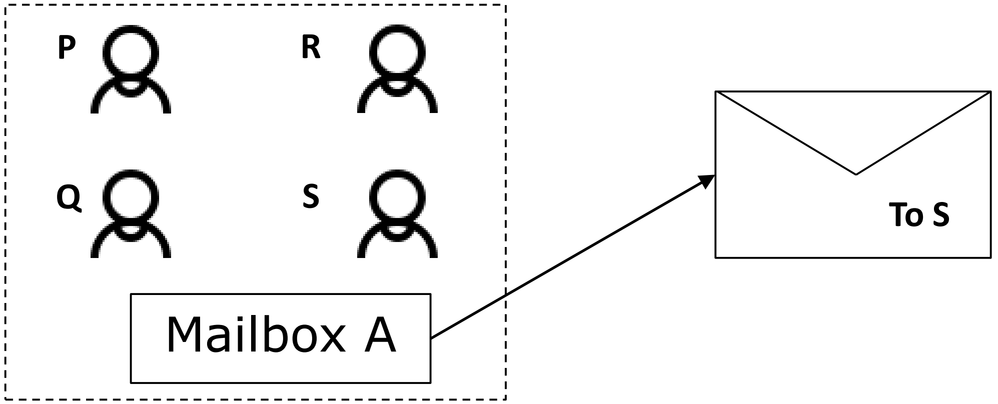
  send(A, message) – send a message to mailbox A

  receive(A, message) – receive a message from mailbox A

Indirect Communication

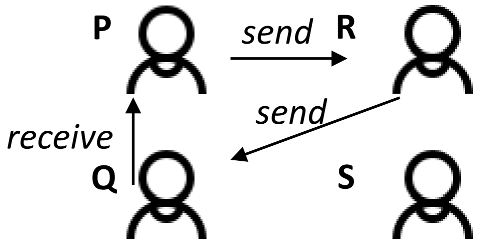
- Mailbox synchronization between sharing processes
  - P1, P2, and P3 share mailbox A
  - P1, sends; P2 and P3 receive
  - Who gets the message?

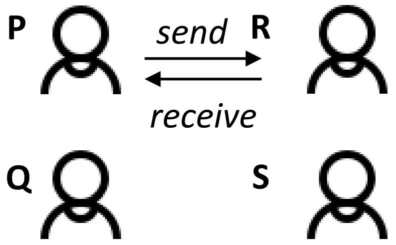
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.



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 either a valid message or a null
- Different combinations possible
  - If both send and receive are blocking in a link, we call it as a **rendezvous** link





Synchronization (Cont.)

- Producer-consumer in blocking send() and receive()

```
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
  - **Zero capacity** – no messages are queued on a link.
    Sender must wait for receiver (rendezvous)
  - **Bounded capacity** – finite length of n messages
    Sender must wait if link full
  - **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);
  - Finally, the mmap() function establishes a memory-mapped file containing the shared-memory object

    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
  - Now the process could write to the shared memory

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

The object (file) pointed by shm_fd becomes accessible through the pointer ptr

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;
}
```

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 messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

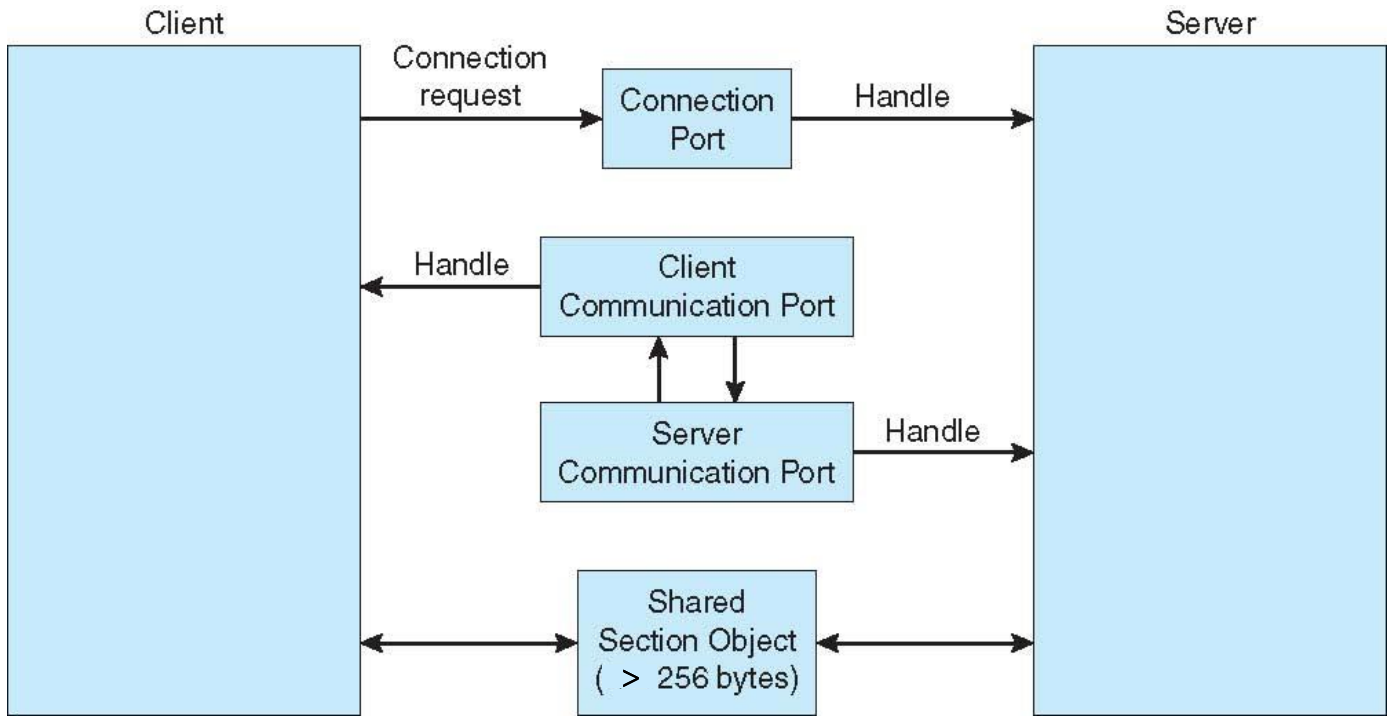
    msg_send(), msg_receive(), msg_rpc()
  - Mailboxes needed for commuication, created via system call

    port_allocate()
  - Send and receive are flexible, for example four options if mailbox is full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily store a message in OS for each mailbox

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**: a client-server message port and a server-client message port
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.
    - Larger messages must be passed through a **section object** created by the server

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)

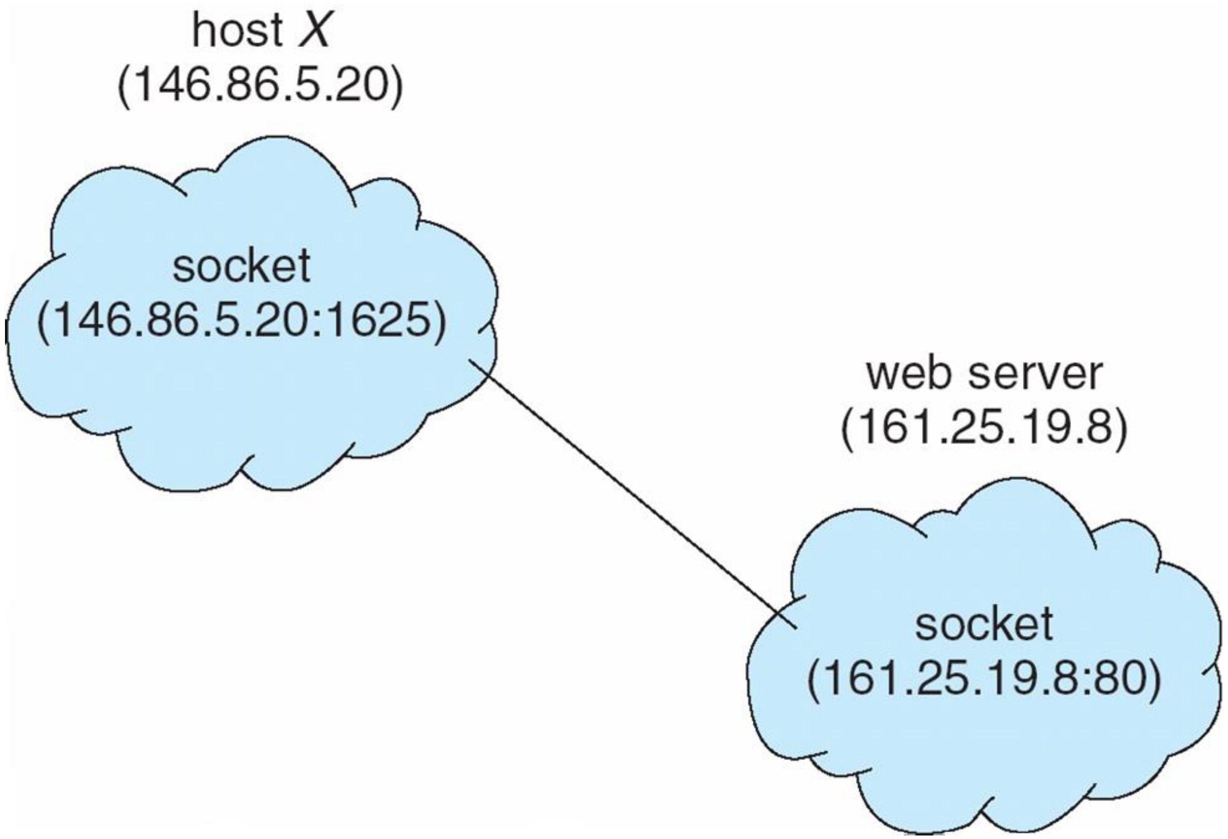
Communications in Client-Server Systems

- 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** -> Port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets
- All ports below 1024 are used or reserved for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running
  - This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol

Socket Communication



Sockets in Java

- Three types of sockets
  - **Connection-oriented** (**TCP**): **Socket** class
  - **Connectionless** (**UDP**): **DatagramSocket** class
  - **MulticastSocket** class– data can be sent to multiple recipients

- Consider this "Date" server:

The server blocks waiting for a client to request a connection.

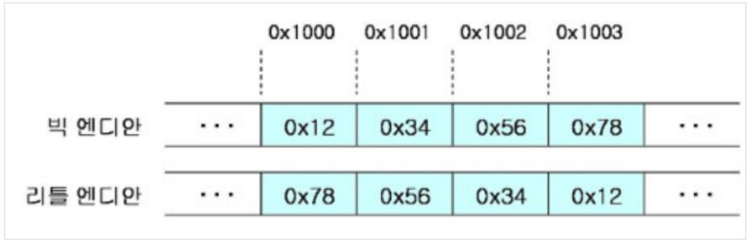
```
import java.net.*;
import java.io.*;

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** – proxy for the actual procedure
- The RPC system, invoked by a client, calls the appropriate stub
- The client-side stub locates the port on the server and **marshals**(**collects**) the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code is compiled from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Remote Procedure Calls (Cont.)

- Machine-independent representation of data known as **External Data Representation** (**XDR**) format to resolve differences like:
  - **Big-endian** and **little-endian**



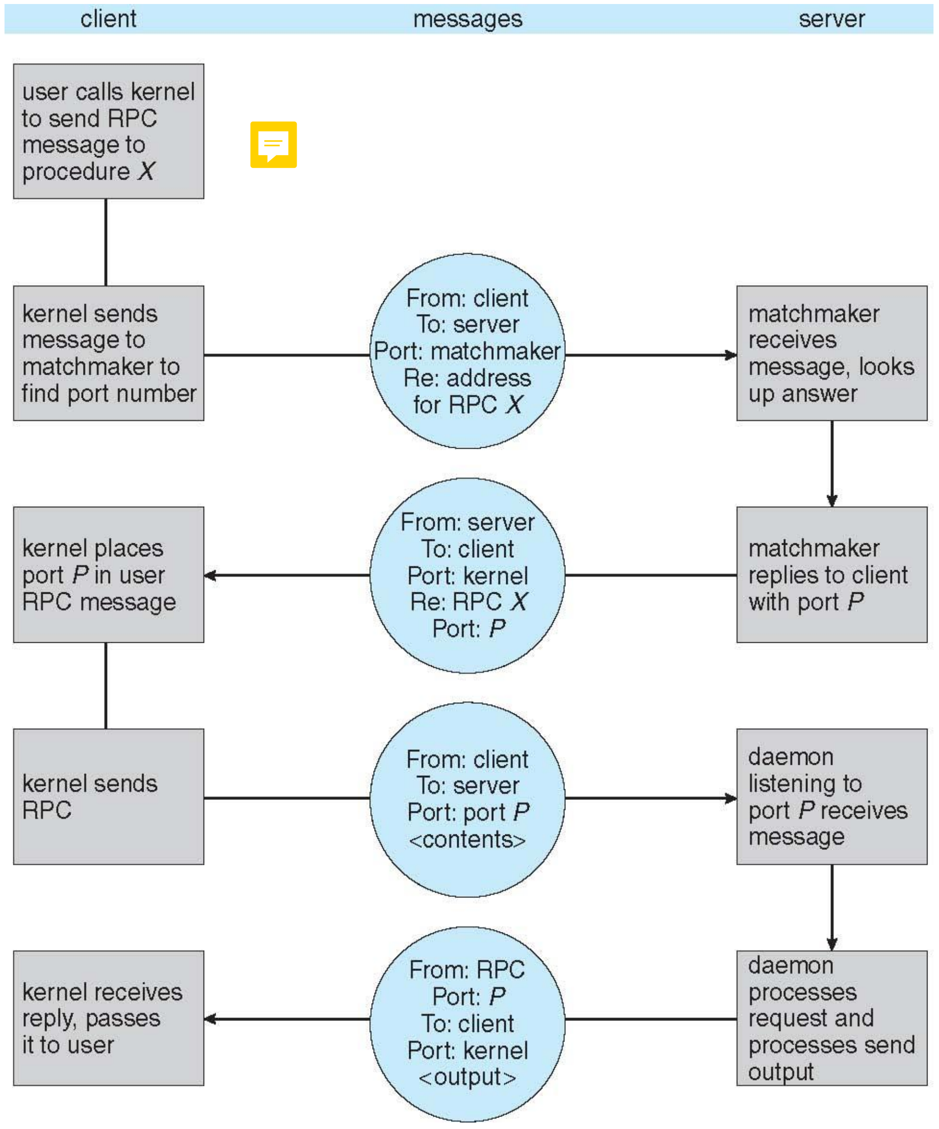
- Remote communication has more failure scenarios than local
  - Policy 1: send messages at most once

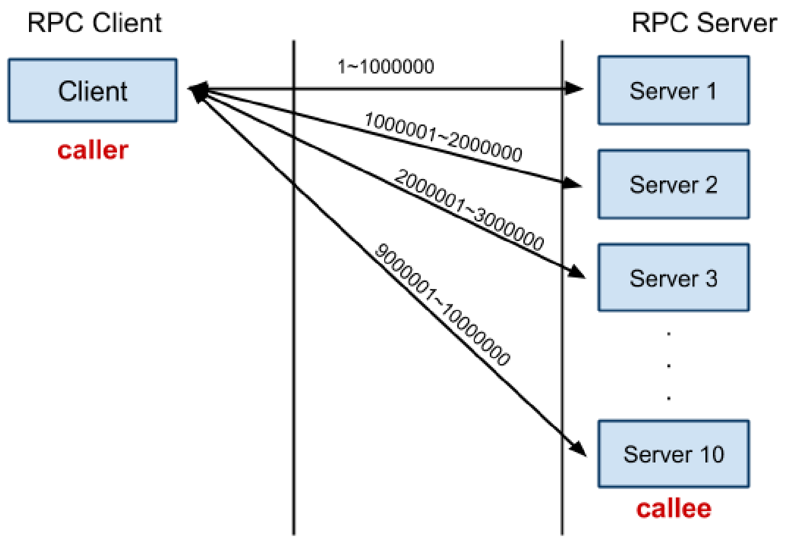
    -> Mechanism: time stamp to each message
  - Policy 2: send messages exactly once

    -> Mechanism: acknowledgement feedback to each message

- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

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



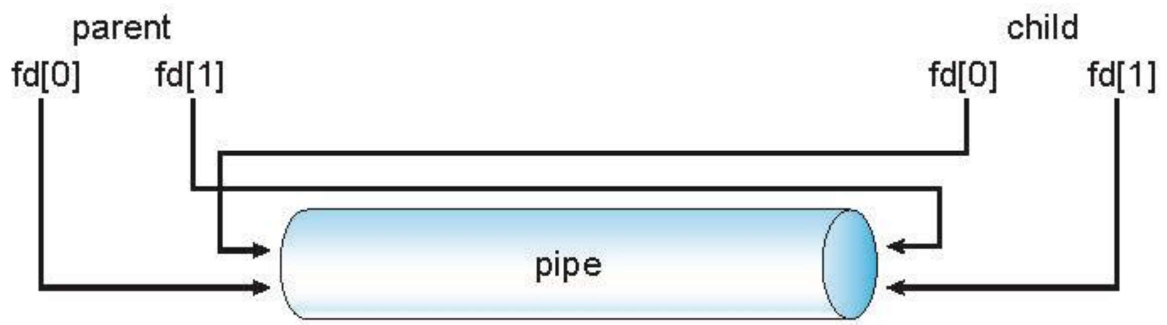
Pipes

- Acts as a conduit allowing two processes to communicate with
- 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, blocking)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- UNIX treats a pipe as a special type of file: read(), write() system calls
- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook

Ex> ls | grep "page"

-> show files that includes "page" texts

Ordinary pipe in UNIX

```
#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;
}
```

Named Pipes

- Named Pipes are more powerful than ordinary pipes (ex> fifo)
- 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