Operating Systems

Processes

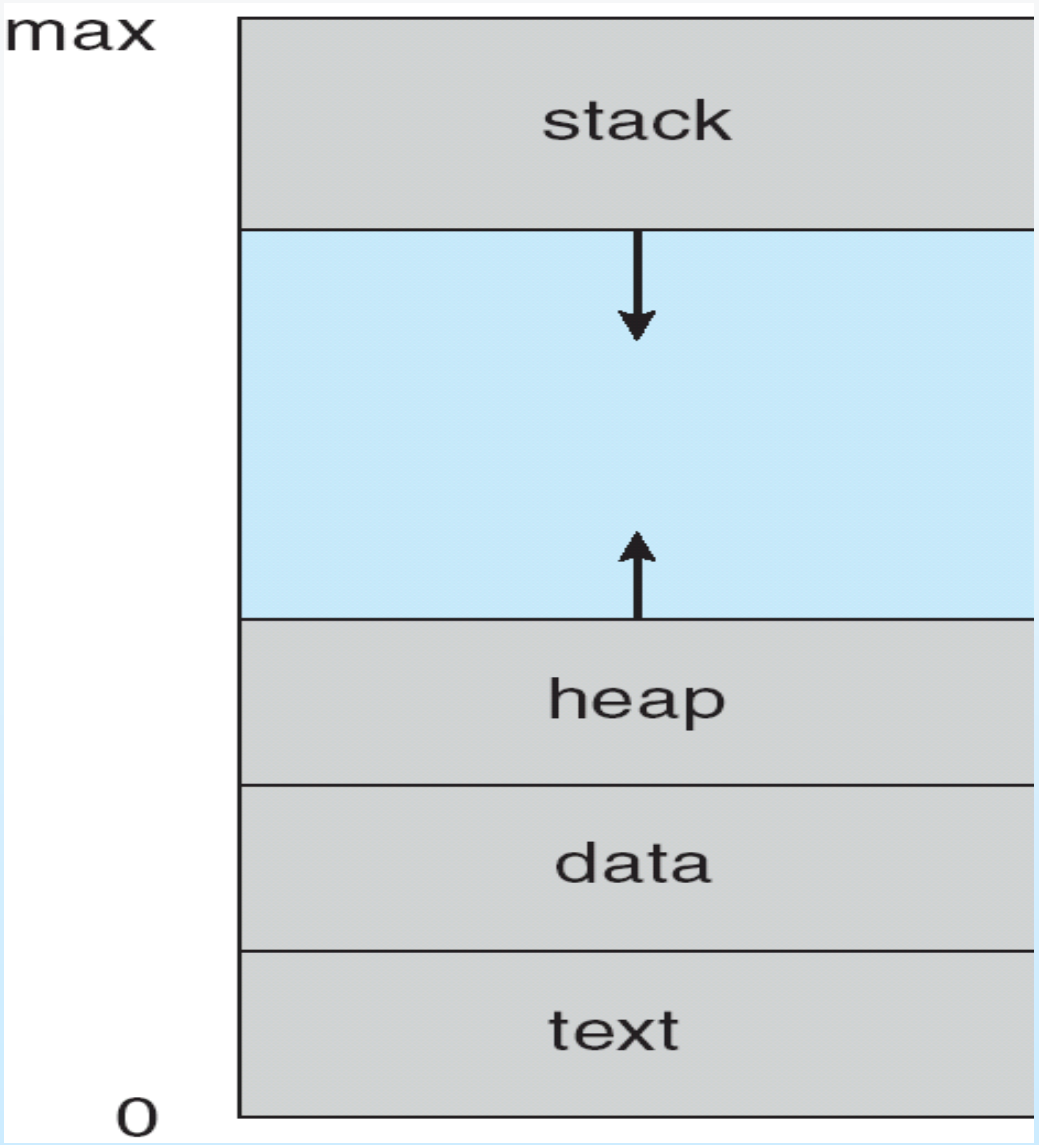
Processes

- Concept
- Scheduling
- Interprocess Communication

Process Concept

- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - Program
  - program counter
  - stack
  - data section

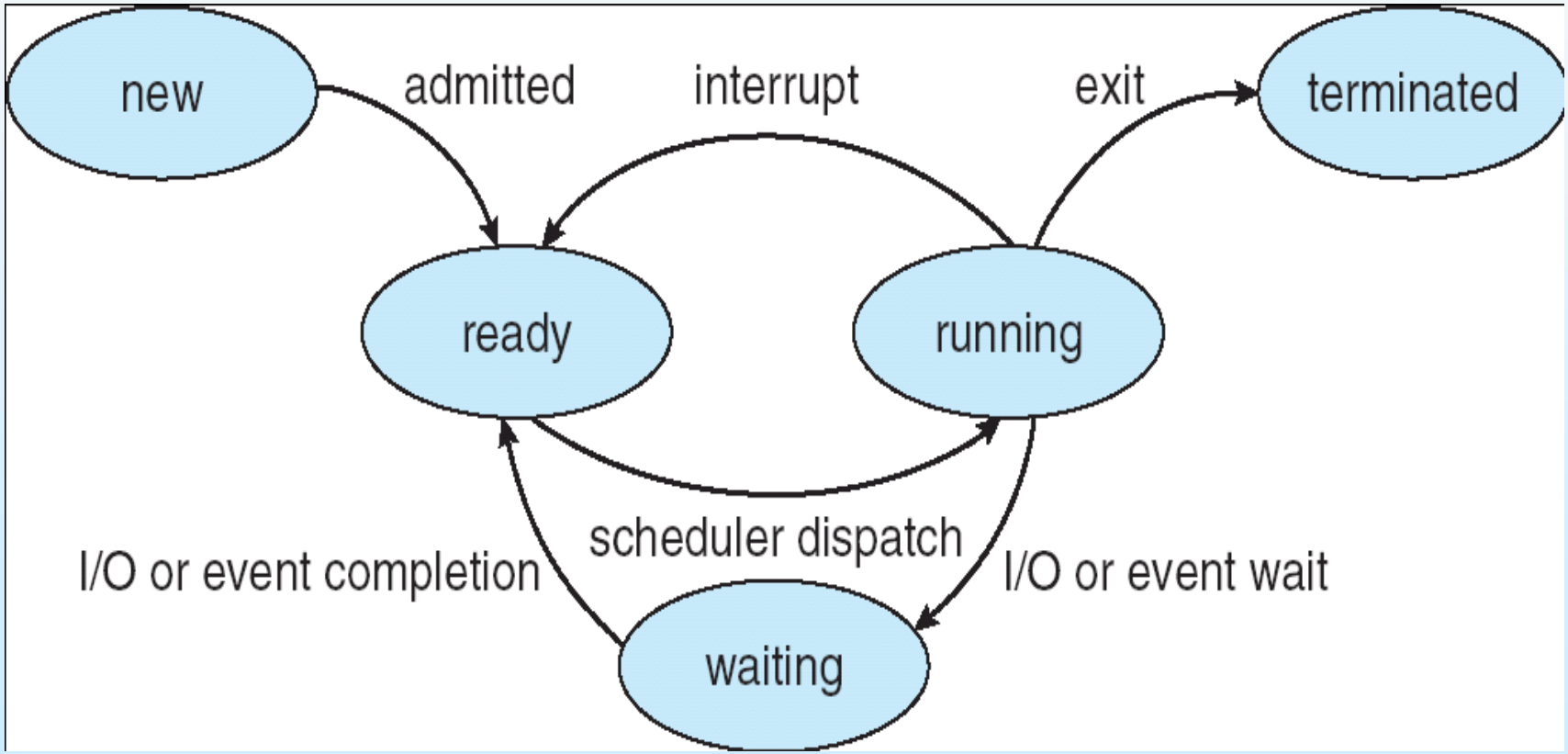
Process in Memory



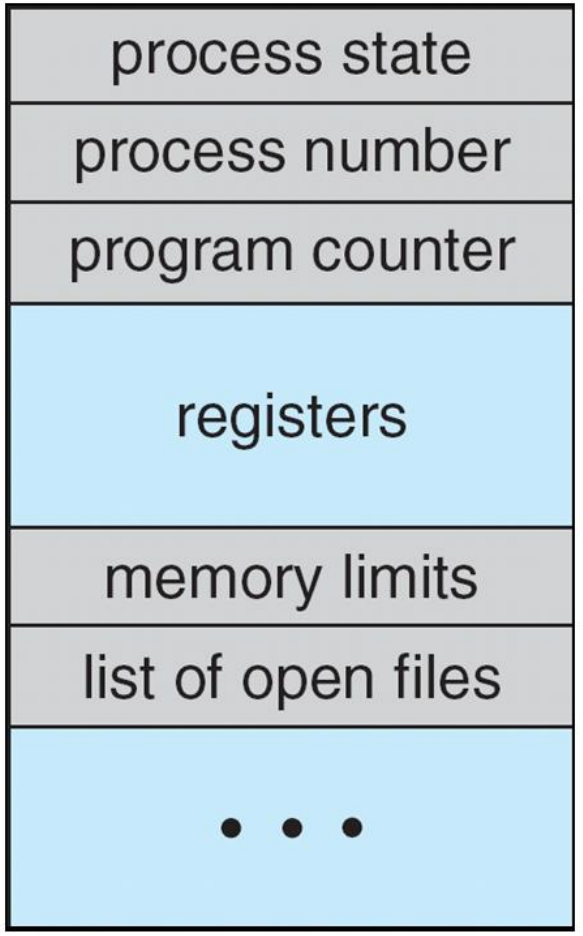
Process State

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

Diagram of Process State



Process Control Block (PCB)

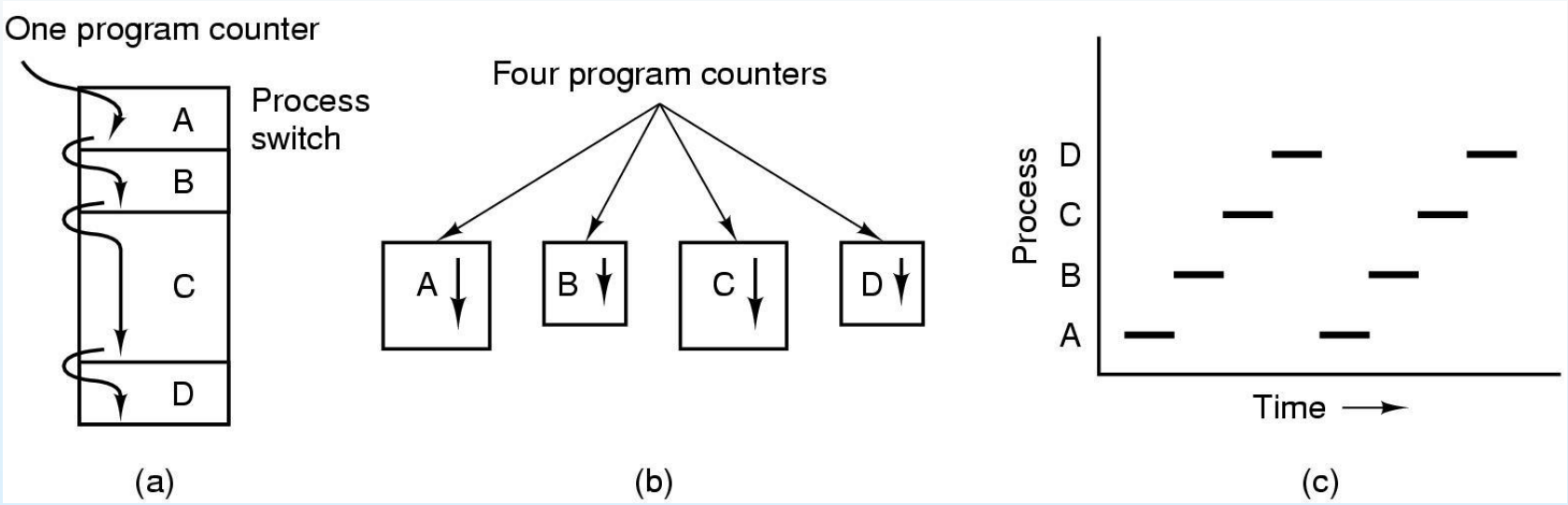


Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information
- Pointers to code and data
- Process identifier (PID)
- Process priority
- File descriptors
- Pointer to the parent of the process
- Pointers to all children of the process
- The processor it is running on

The Process Model: one CPU

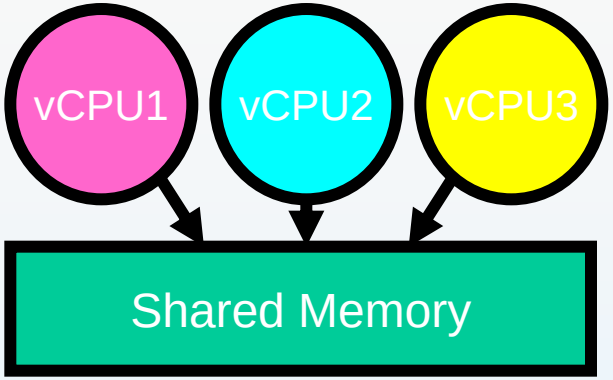


a) Multiprogramming of four programs

b) Conceptual model of 4 independent, sequential processes

c) Only one program active at any instant

How to give illusion of multiple processors?



- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual “CPU” needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How to switch from one virtual CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

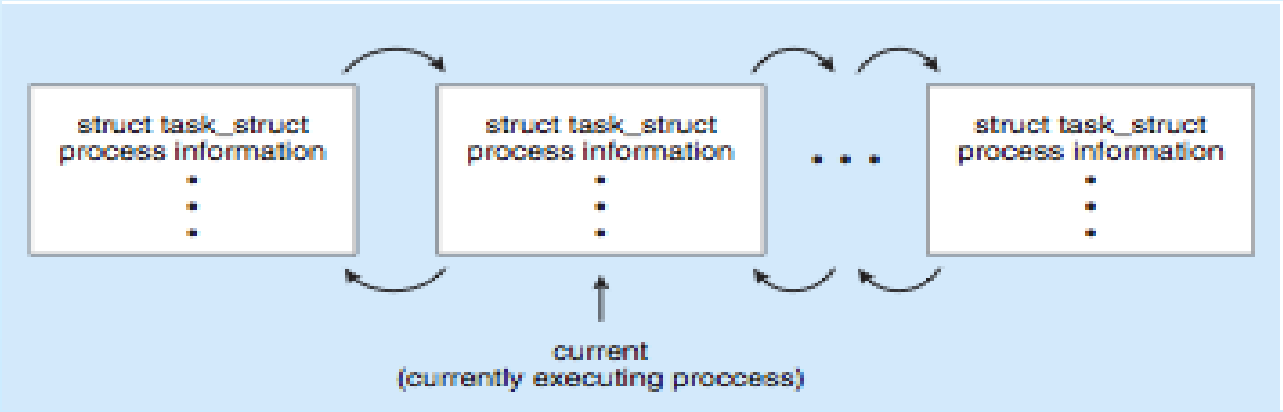
Switching Processes

- CPU is switched between the processes
  - Save the “real world” of the currently active process in process table
  - Restore the “real world” from the process table of the selected process (the one to be executed next)

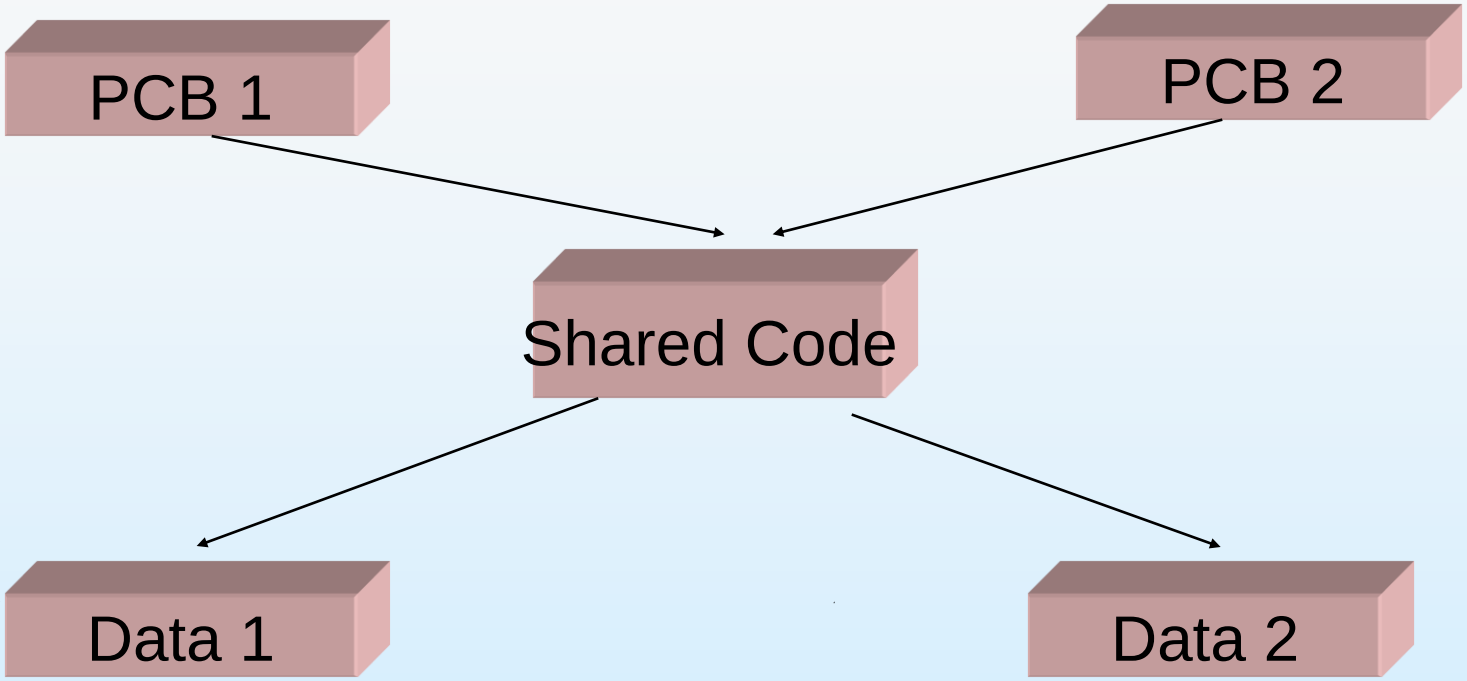
Process Representation in Linux

- 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 */
```



Shared Code



- The shared code must be ***re-entrant*** (ie., it must not modify itself)

Process States: related operations-1

- Create
  - A process is created by an existing process (parent/child); a process is created by a service, or as a new batch job, or an interactive logon
- Terminate
  - Normal completion (exit); External completion (forced completion: by operator, by parent); Internal completion (error, time overrun)
- Schedule
  - A process is selected out of the ready queue (is scheduled) and dispatched to the running state
- Pre-empt
  - The running process is pre-empted because some other process of higher priority has become ready (or it yields)

Process States: related operations-2

- Suspend
  - A process starts a time consuming IO operation, or is swapped out, or makes a spontaneous request to sleep, or is blocked in a synchronization operation. When a process is suspended, it is on some queue, namely the queue of all the processes waiting for the same resource. Processes during their life move from queue to queue, with stays in the running state
- Continue
  - The inverse of Suspension: the resource requested becomes available (timer, IO completes, unblocking)

Implementation of Processes: Typical Process Table

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Fields of a process table entry

Process Scheduler

- Scheduler:
  - The lowest operating system layer handles interrupts and does scheduling
  - The higher layers maintain all other process related tasks, such as accounting, logging, etc.

Implementation of Process Switch

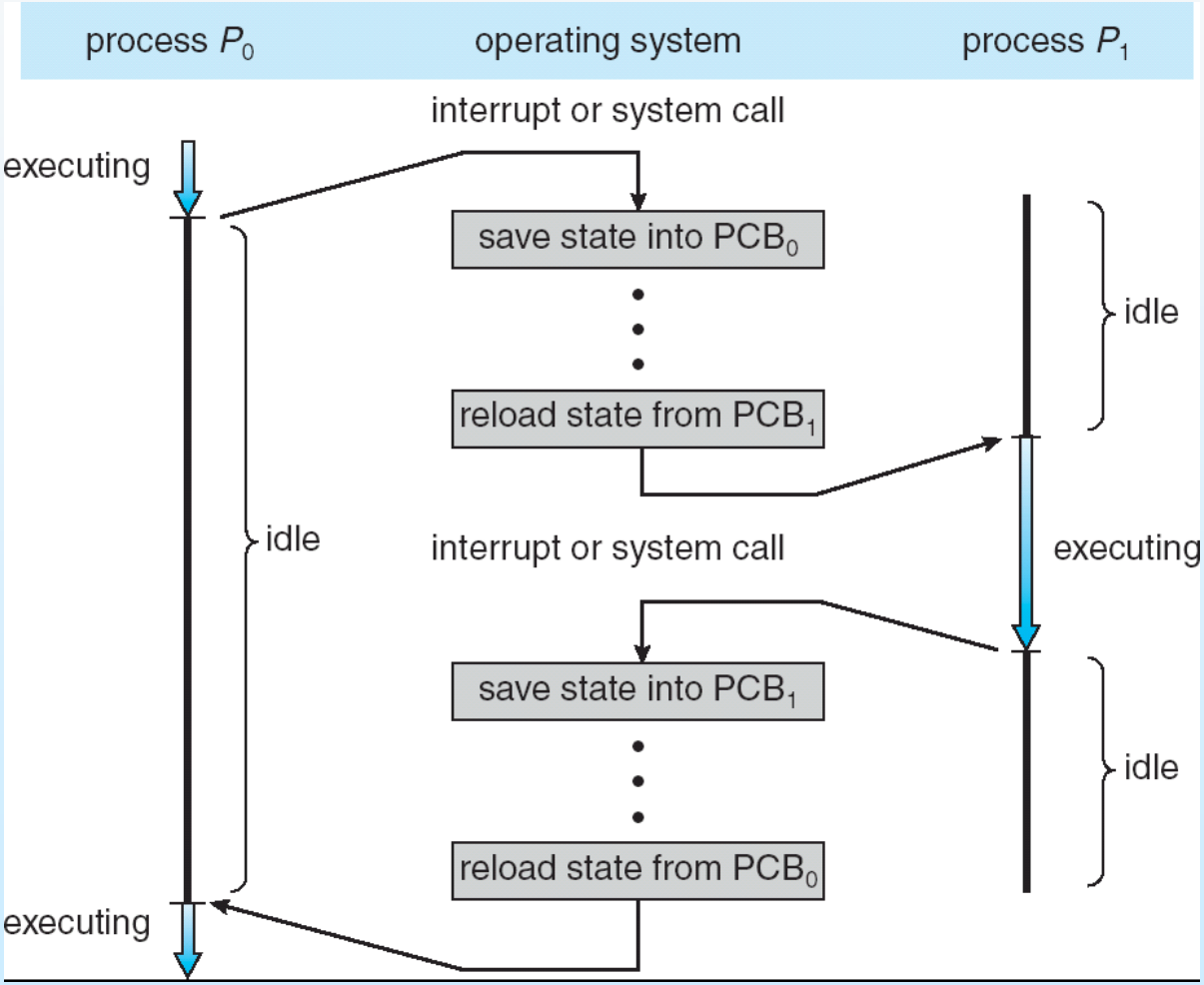
1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Skeleton of what lowest level of OS does when an interrupt occurs

Context Switching Overhaed

- 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
- Context-switch time is overhead; the system does no useful work while switching
- Time is dependent on hardware support and its speed

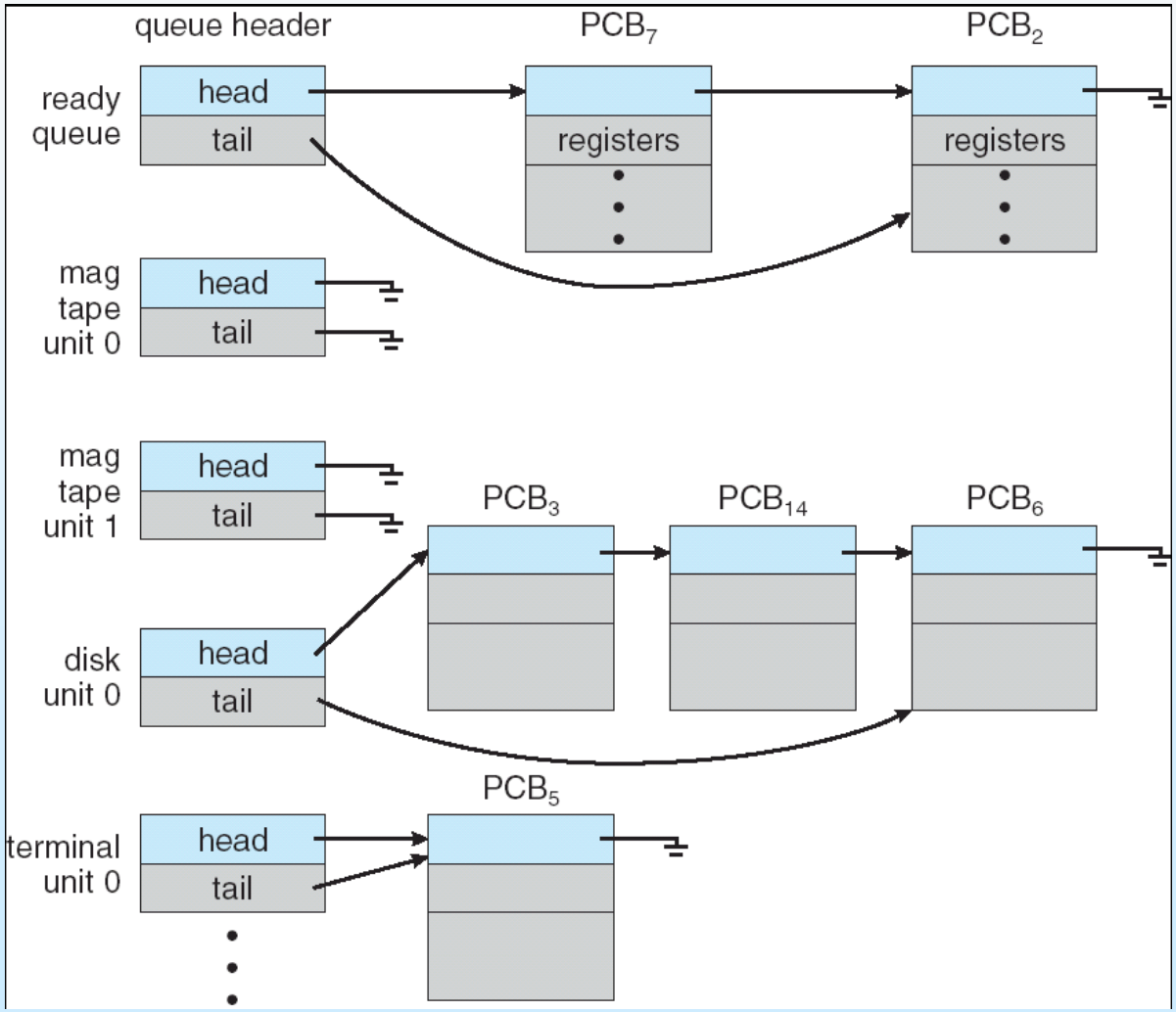
CPU Switch From Process to Process



Process Queues

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

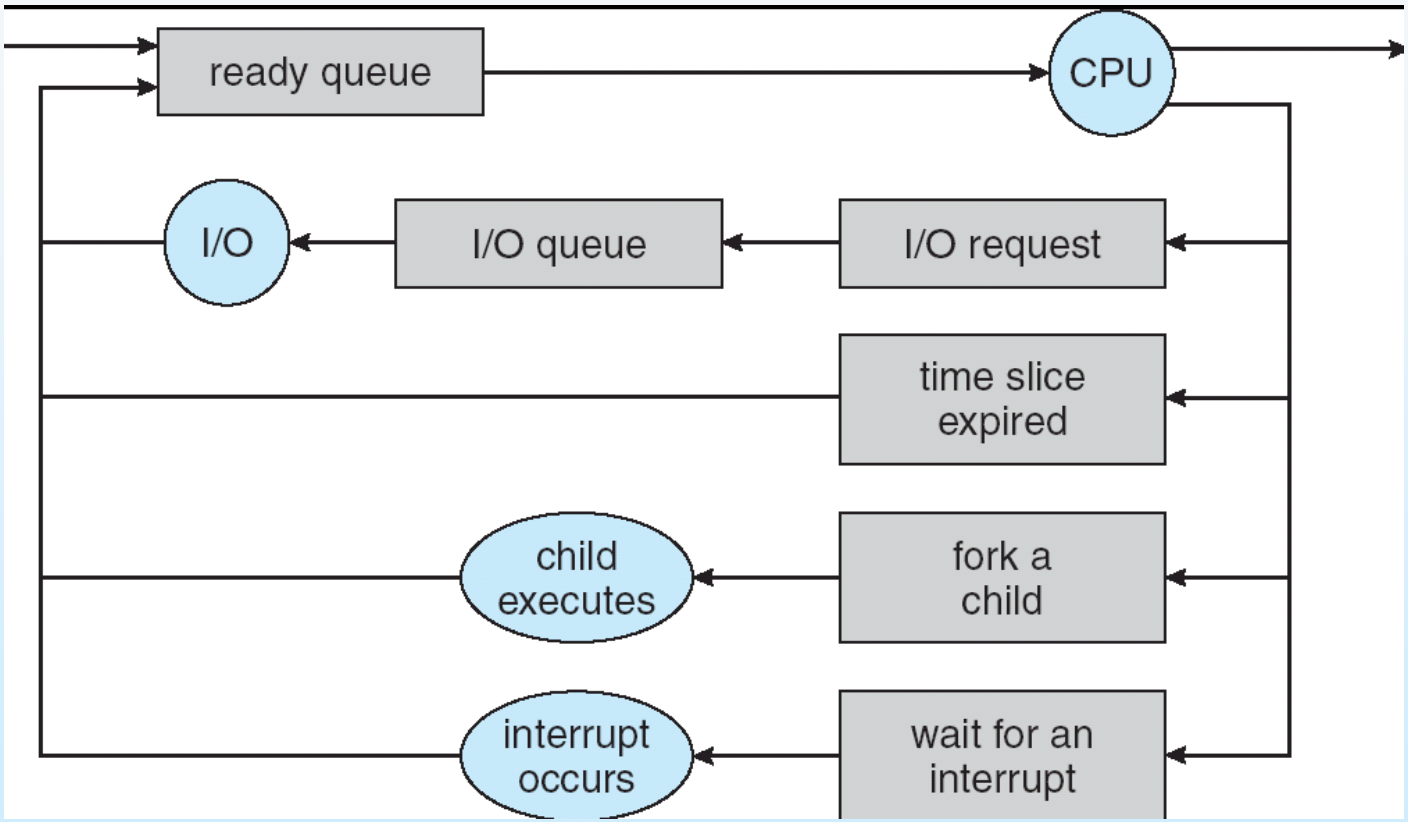
Process Queues



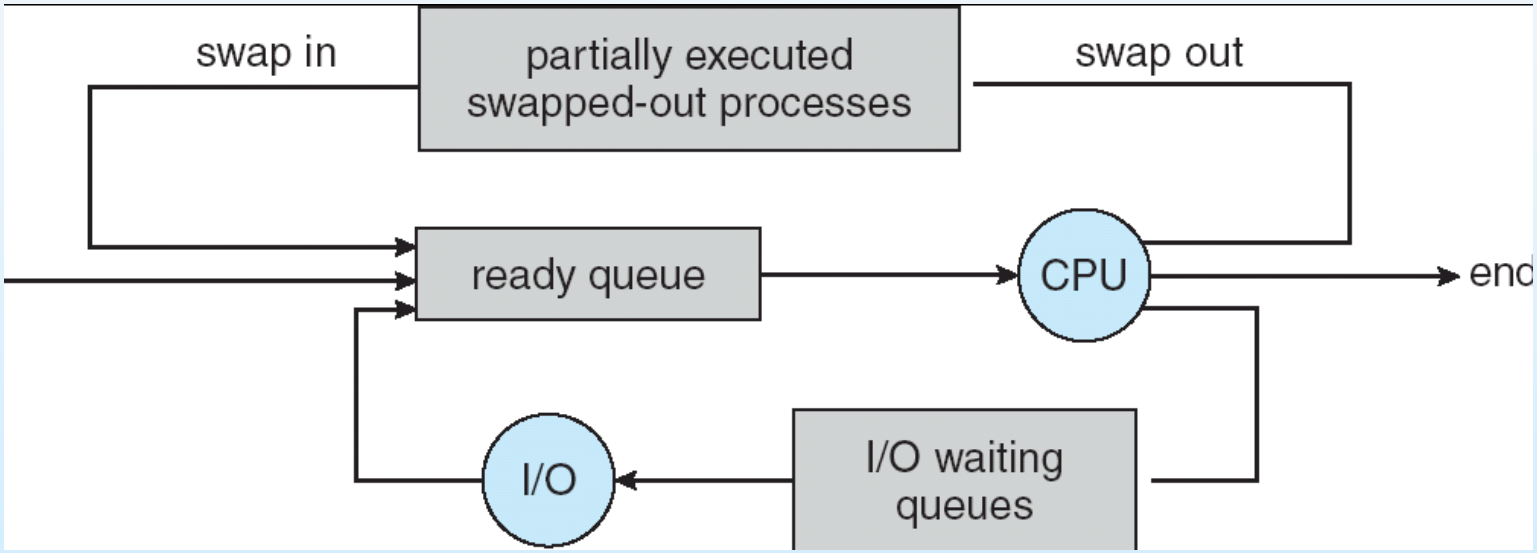
Scheduling Levels

- **Short-term scheduler** (or CPU scheduler) – selects which process in he ready queue should be executed next and allocates CPU
- **Mid-term scheduler:** Select which partially executed process should be brougt into the ready queue.
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue

Representation of Short Term Process Scheduling



Graphical Representation of Medium Term Scheduling



Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- Mid-term scheduler is invoked if the degree of multiprogramming falls beyond certain threshold.
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many **short CPU bursts**
  - **CPU-bound process** – spends more time doing computations; few very **long CPU bursts**

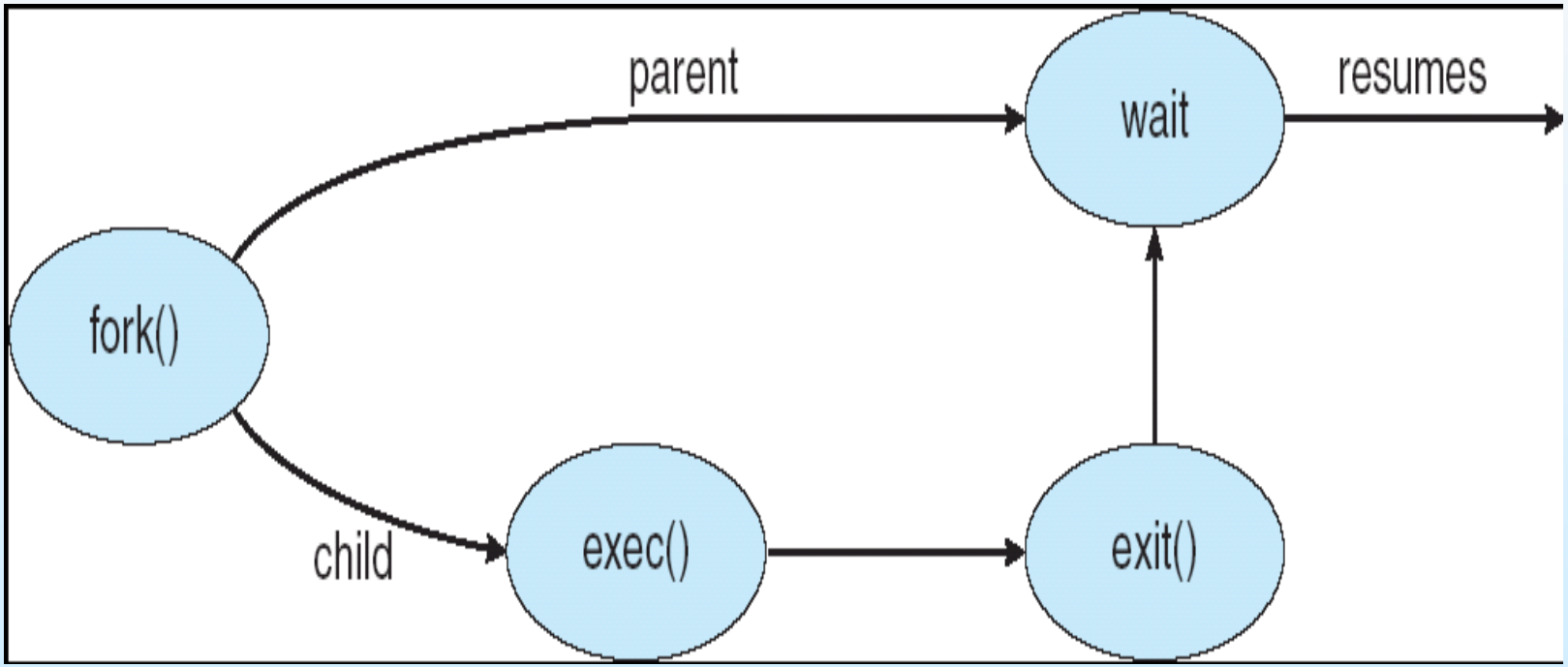
Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Children share subset of parent's resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child is duplicate(Clone) of parent Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process, which is duplicate of the parent in terms of program in execution

A typical parent child relationship



Process Termination

- Process executes last statement and asks the operating system to delete it (**`exit()`**)
  - Output data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**`abort()`**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**
- Wait for termination, returning the pid:

```
pid t pid; int status;

pid = wait(&status);
```

- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 categories
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript, new one 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

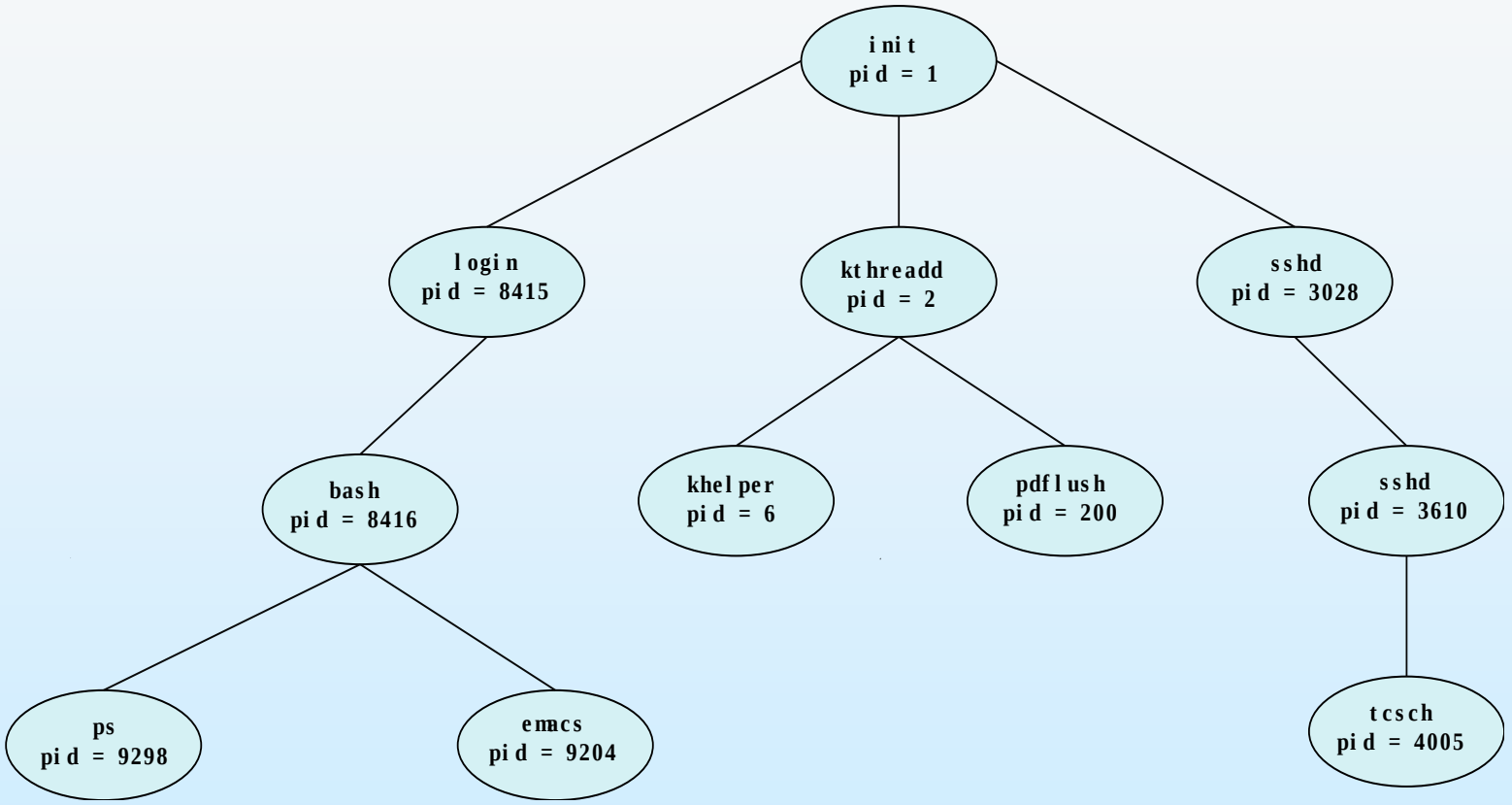
How to Create New Processes? Fork & Exec

- fork
  - create a new process that is a copy of current one
- exec
  - change the program that a process is executing
- Some Oss combine the two into one system call…

fork() system call

- Using fork() system call the parent process divides itself into two identical processes.
- When a process forks, a complete copy of the executing program is made into the new process.
- This new process (which is a child of the parent) has a new process identifier (PID).
- The fork() function returns the child's PID to the parent, while it returns 0 to the child, in order to allow the two identical processes to distinguish one another.
- The parent process can either continue execution or wait for the child process to complete.

A Tree of Processes in Linux



C Program example: fork()

```
int main()
{
Pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-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");
            exit(0);}}
```

execxx() Family of System Calls

- The exec family functions of Unix-like operating systems cause the running process to be completely replaced by the program passed as an argument to the function.
- As a new process is not created, the process identifier (PID) does not change, but the *data, heap* and *stack* of the original process are replaced by those of the new process.
- Exec() family functions: execl, execle, execlp, execv, execve, and execvp,
- The new process image inherits the current environment variables.

C Program: exec() family system Calls

- **Prototypes of exec() family system calls:**
- The functions are declared in the unistd.h header for the POSIX standard.

```
int execl(char const *path, char const *arg0, ...);

int execle(char const *path, char const *arg0, ..., char const * const
*envp);

int execlp(char const *file, char const *arg0, ...);

int execv(char const *path, char const * const * argv);

int execve(char const *path, char const * const *argv, char const *
const *envp);

int execvp(char const *file, char const * const *argv);
```

Example C programs for exec system calls-1

- **execl()**
- **The following example executes the ls command, specifying the pathname of the executable (/bin/ls) and using arguments supplied directly to the command.**

```
#include <unistd.h>

int ret;

...

ret = execl ("/bin/ls", "ls", "-l", (char *)0);
```

Example C programs for exec system calls-2

- **execle()**
- **The following example is similar to Using execl(). In addition, it specifies the environment for the new process image using the env argument.**

```
#include <unistd.h>

int ret;

char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };

...

ret = execle ("/bin/ls", "ls", "-l", (char *)0, env);
```

Example C programs for exec system calls-3

- **execlp()**
- **The following example searches for the location of the ls command among the directories specified by the PATH environment variable.**

```
#include <unistd.h>

int ret;

...

ret = execlp ("ls", "ls", "-l", (char *)0);
```

Example C programs for exec system calls-4

- **execv()**
- **The following example passes arguments to the ls command in the cmd array.**

```
#include <unistd.h>

int ret;

char *cmd[] = { "ls", "-l", (char *)0 };

...

ret = execv ("/bin/ls", cmd);
```

Example C programs for exec system calls-5

- **execve()**
- **The following example passes arguments to the ls command in the cmd array, and specifies the environment for the new process image using the env argument.**

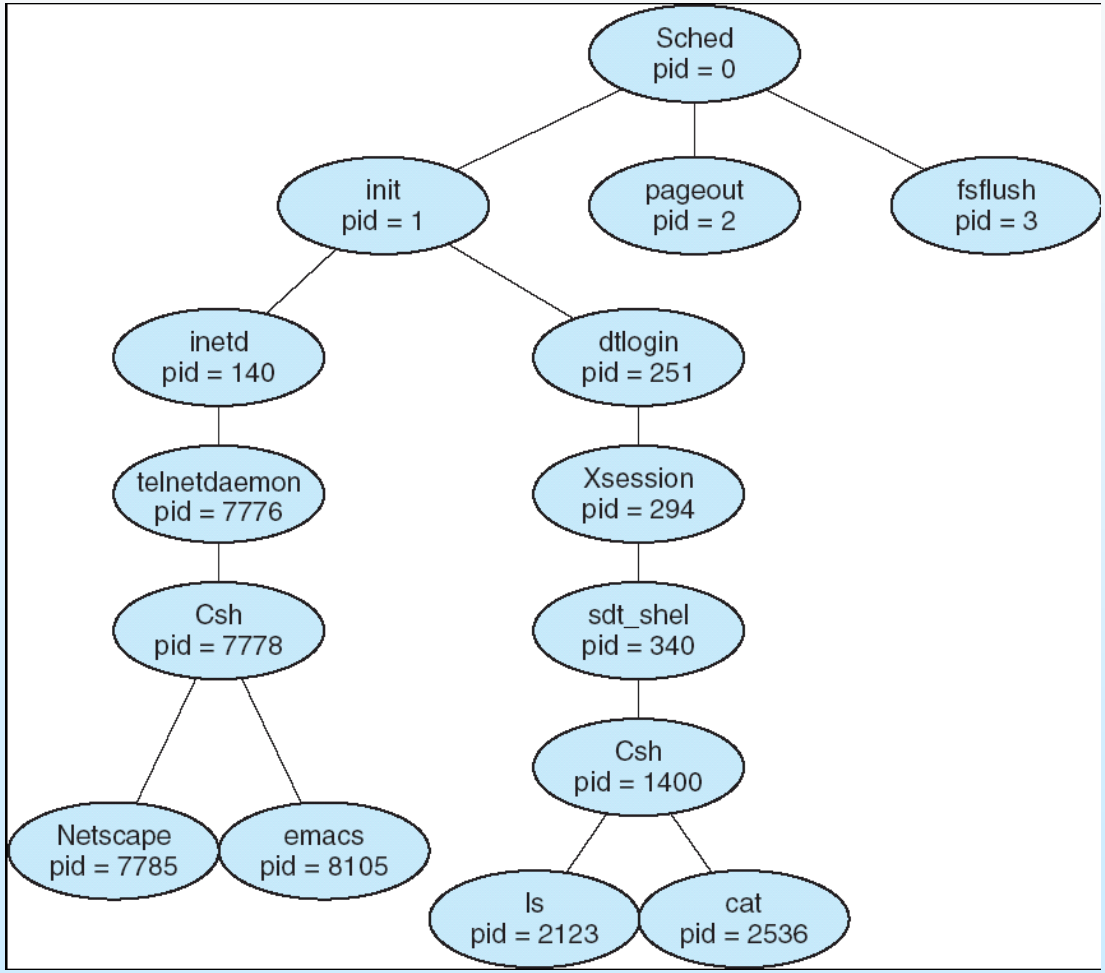
```
#include <unistd.h>

int ret;

char *cmd[] = { "ls", "-l", (char *)0 };

char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };

...

ret = execve ("/bin/ls", cmd, env);
```

Example C programs for exec system calls-6

- **execvp()**
- **The following example searches for the location of the ls command among the directories specified by the PATH environment variable, and passes arguments to the ls command in the cmd array.**

```
#include <unistd.h>

int ret;

char *cmd[] = { "ls", "-l", (char *)0 };

...

ret = execvp ("ls", cmd);
```

A tree of processes on a typical Solaris OS



Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

Process Cooperation

Meaning of Process Cooperation

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

Producer-Consumer Problem

- Cooperation must be supported by the operating systems.
- A buffer need to be implemented between the producer and consumer to hold the data to be exchanged.
- There are two bufering methodologies
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size. This is a realistic approach.

  However, the right size of buffer is important. We should not have excessive or insufficient buffer .

Bounded-Buffer Management: Shared-Memory Solution

- Shared data: C Language like Implementation

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

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

Bounded-Buffer: Insert() Method: producer process

```
while (true) {
   /* Produce an item */
   while (((in + 1) % BUFFER SIZE)  == out)
     ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
{
```

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

Bounded-Buffer:
Remove() Method: consumer process

```
while (true) {
     while (in == out)
          ; // do nothing -- nothing to consume

     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
return item;
 {
```

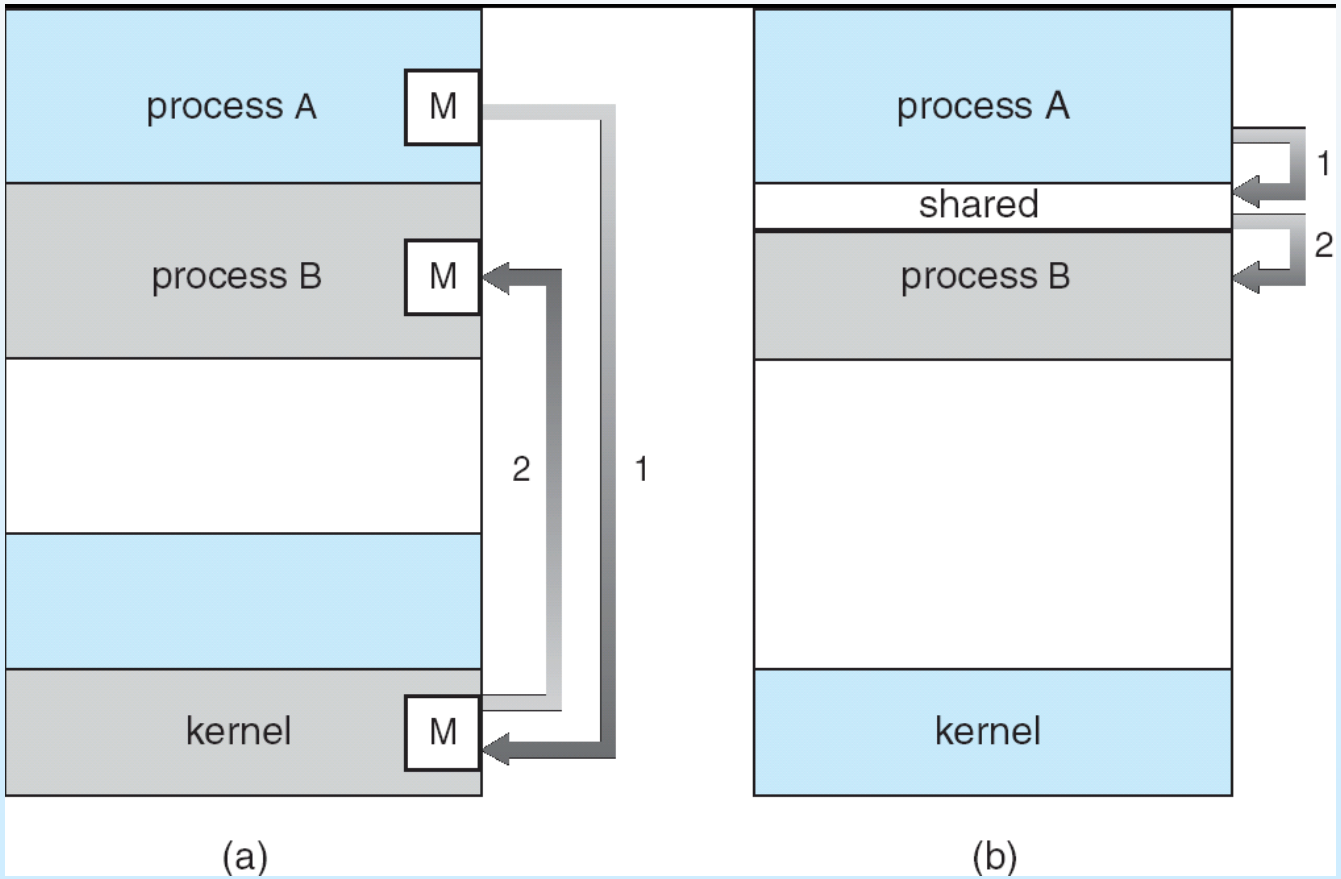
Inter-Process Communication

- Buffer sharing is IPC.
- IPC must have necessary mechanisms for processes to communicate and to synchronize.
- There are two main paradigms for IPC
  - Shared memory based: Buffer sharing is an example to this.
  - Message based: no shared memory

IPC: message based

- Messaging provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- This is need to be facilitated by the OS
- How?

IPC Models: (a) kernel level buffer, (b) user level buffer



Shared Memory Communication in UNIX

Pipe() system call: pipe();

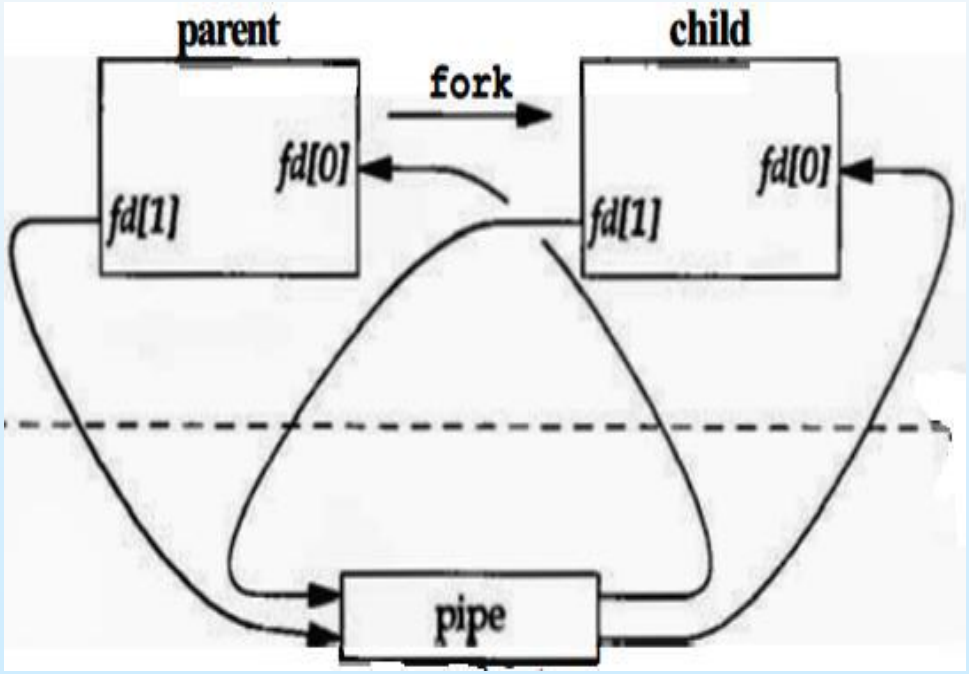
PROTOTYPE:
int pipe(int fd[2]);
RETURNS: 0 on success
-1 on error: errno = EMFILE (no free descriptors)
EMFILE (system file table is full)
EFAULT (fd array is not valid)

•Two file descriptors are created: fd[0] i for reading, fd[1] for writing

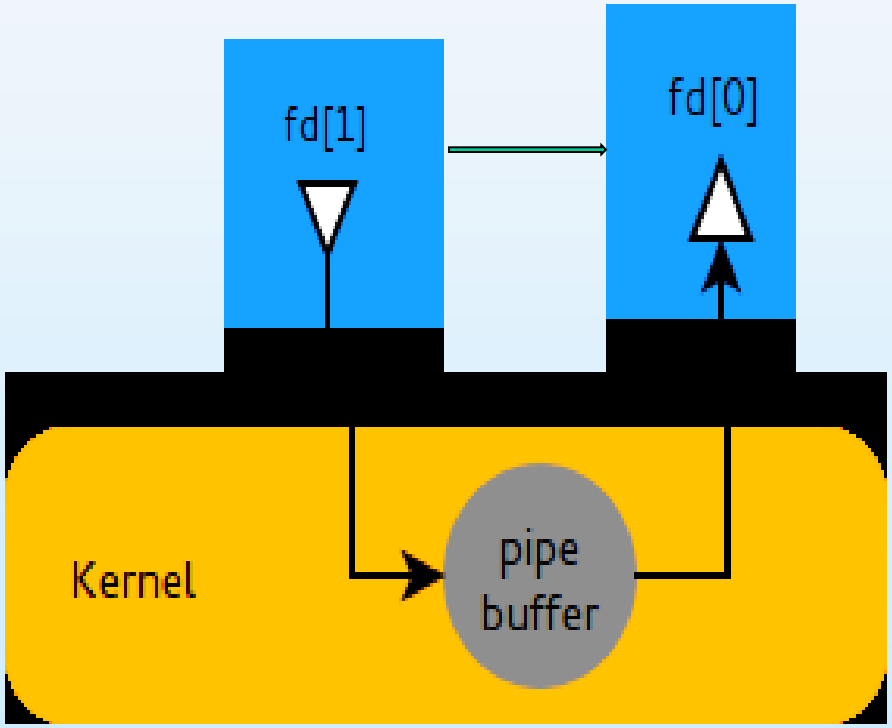
•the “**pipe**“ function is defined in the <**unistd.h**>

Chaotic use of pipe()!

- Let a **process** create a **pipe,** then a **child**, then both processes share expected to communicate
- Their **writes** are merged in the pipe, **reads** are intermixed! This is a chaotic situation.



Example on Correct use of pipe



A full C Program on Pipe() system call

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"

 *****************************************************************************
 MODULE: pipe.c
 *****************************************************************************/

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

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

Redirection in pipe()

- **Redirection** of an **input** or an **output** descriptor is done by **overwriting** it with a new file descriptor;
- Practical Example:
  - redirection of **stdout** to the pipe's output descriptor, or
  - redirection of **stdin** to the pipe's input descriptor.

dup() system call

- Using dup() system call, the file descriptors are duplicated, such that two descriptors will have the same affect.
- The dup() system call gets the lowest-numbered unused descriptor.
- Thus, dup() can be used to redirect standard I/O (STDOUT, STDIN) as well. If needed. They firse closed, before the use of dup().

SYSTEM CALL: dup();

PROTOTYPE: int dup(int oldfd);
RETURNS: new descriptor on success
-1 on error: errno = EBADF (oldfd is not a valid descriptor)
EBADF (newfd is out of range)
EMFILE (too many descriptors for the process)

- the old descriptor is not closed. Both may be used interchangeably

dup2()

- **Dup2() function is also** defined in the <**unistd.h**> header
- It uses the new file descriptor provided by the user, for redirection
- Format:

  int dup2 (source fd, target fd)

  the call returns **–1** on error
- Example:

  if (**dup2 (fd[1], stdout) < 0)**

  perror (“redirection err”);
- The effect of the above is the same with the following:

  close(1);

  if (**dup (fd[1]) < 0)**

  perror (“redirection err”);

Message Based Direct Communication

- Processes must name each other explicitly: For example, for process P and Q to communicate:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Such a communication link need to be implementated by OS

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
- Mailboxes can be created OS
- 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 mailboxes
  - Such communication may be unidirectional or bi-directional
- Example syntax

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

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

IPC Synchronization

- IPC-Inter Process Communication may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

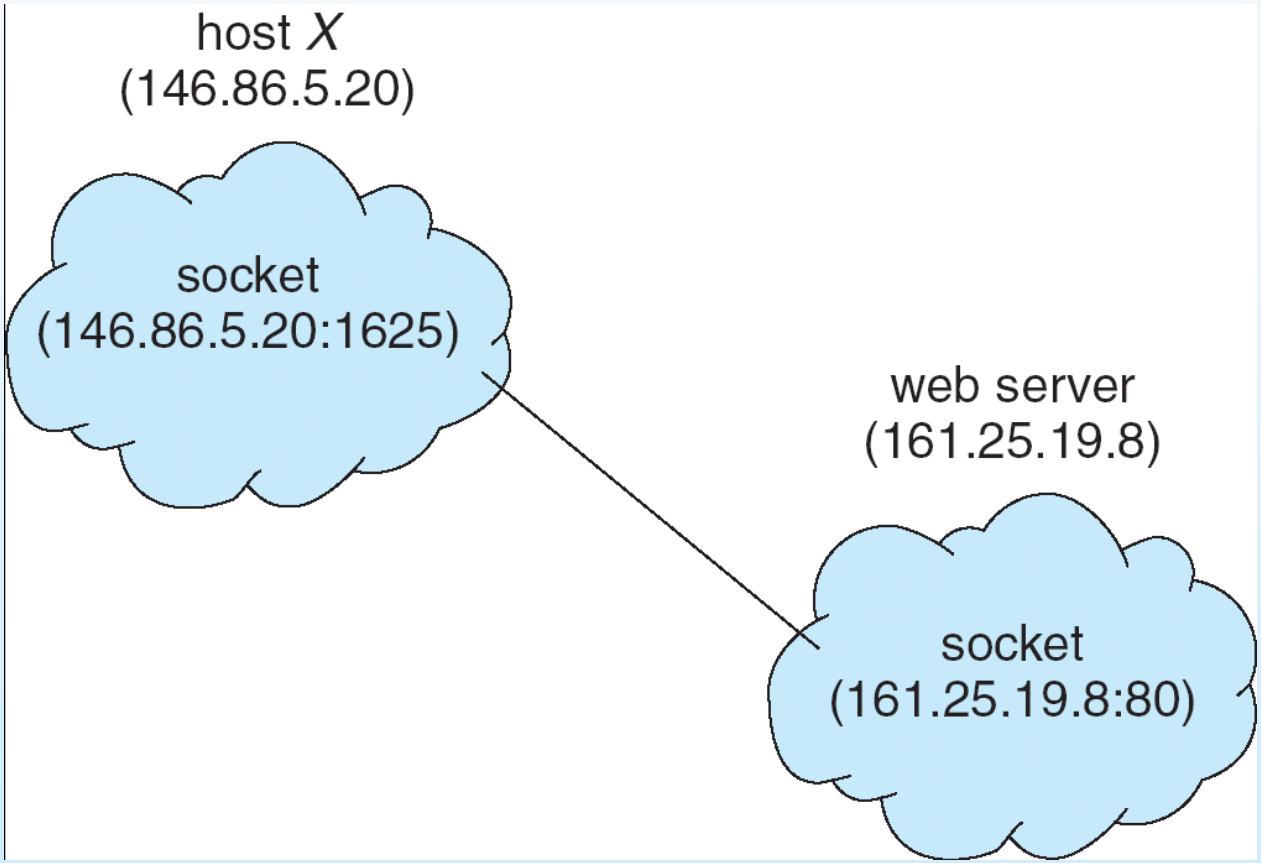
Client-Server Communication

- Sockets (universal)
- Remote Procedure Calls (classical UNIX domain IPC)
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket "**161.25.19.8:1625**" refers to port **1625** on host **161.25.19.8**
- Communication is between a pair of sockets: receiver, sender.

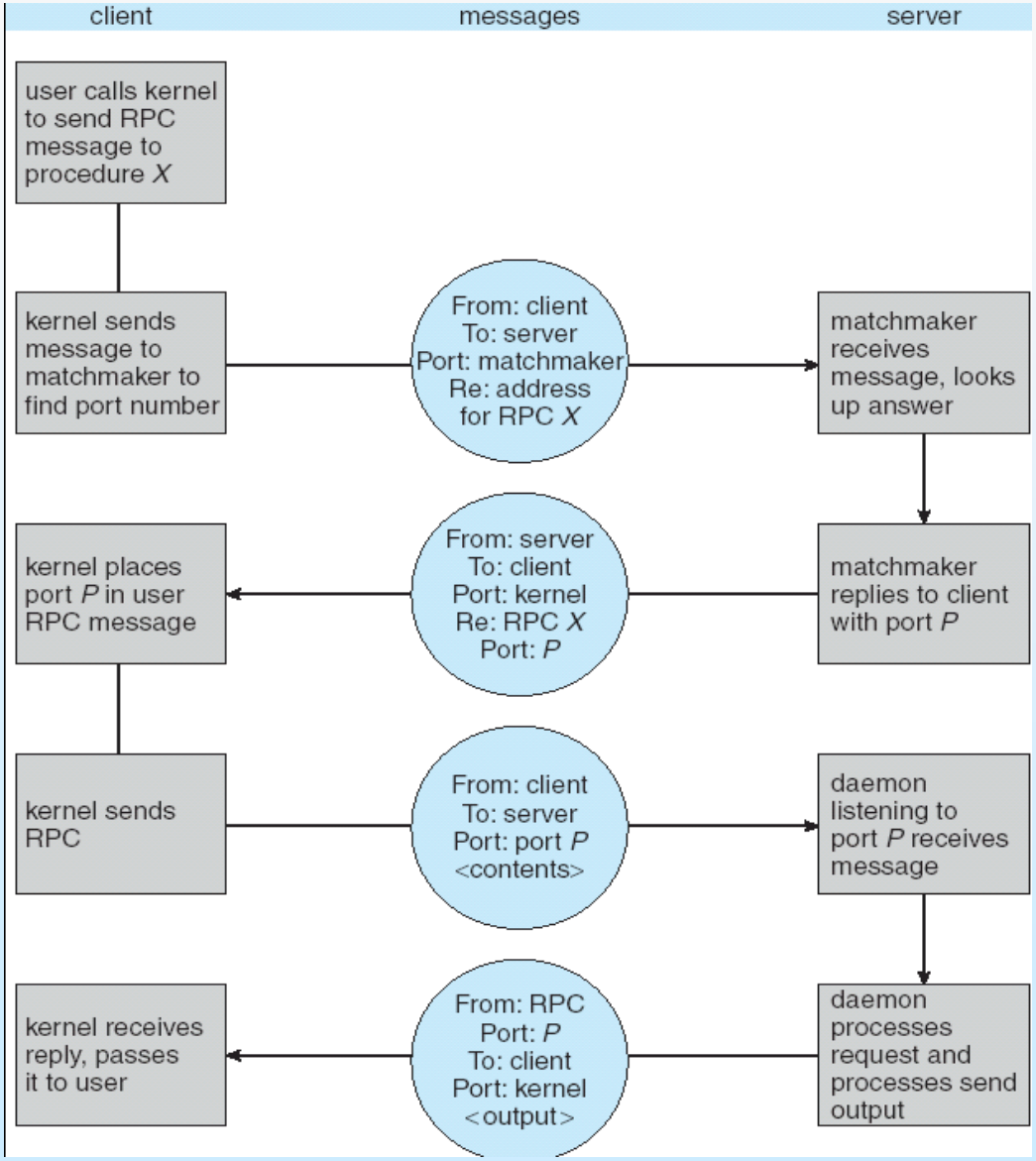
Socket Communication



Remote Procedure Calls

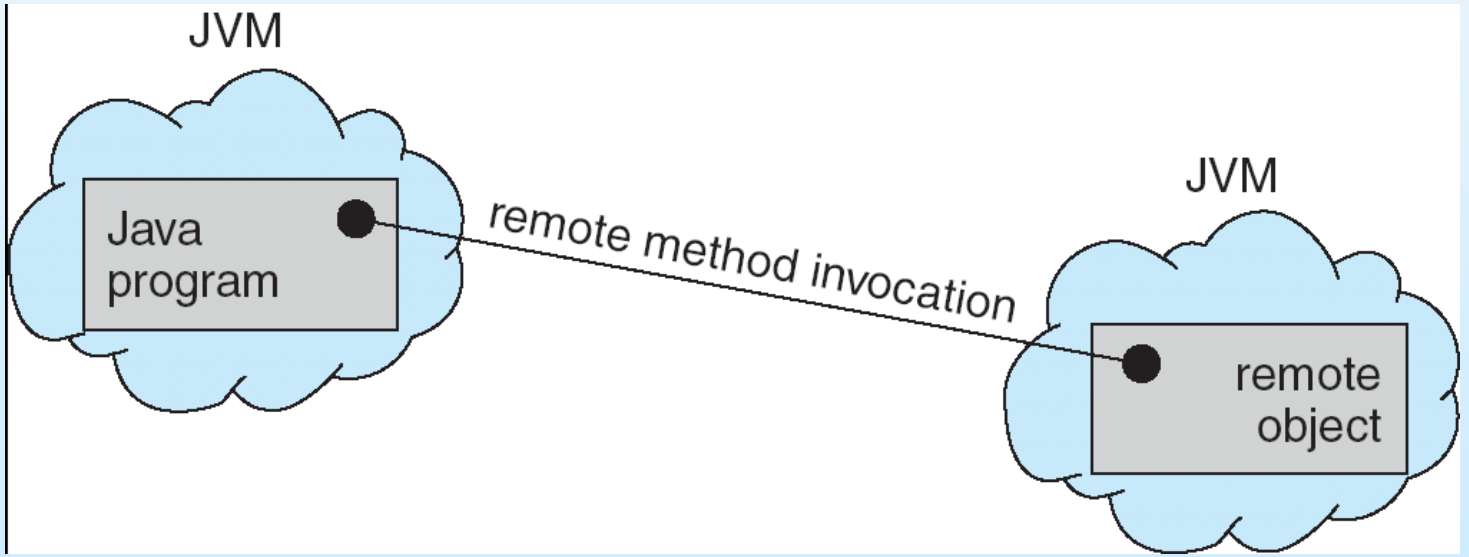
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- A proxy concept is realized through so called stubs:
  - **Client side stub acts as a proxy** for the actual procedure on the server, it locates the server and *marshals* the parameters.
  - **Server-side stub** receives the message from the client stub, unpacks the marshaled parameters, and performs the procedure on the server.

Execution of RPC

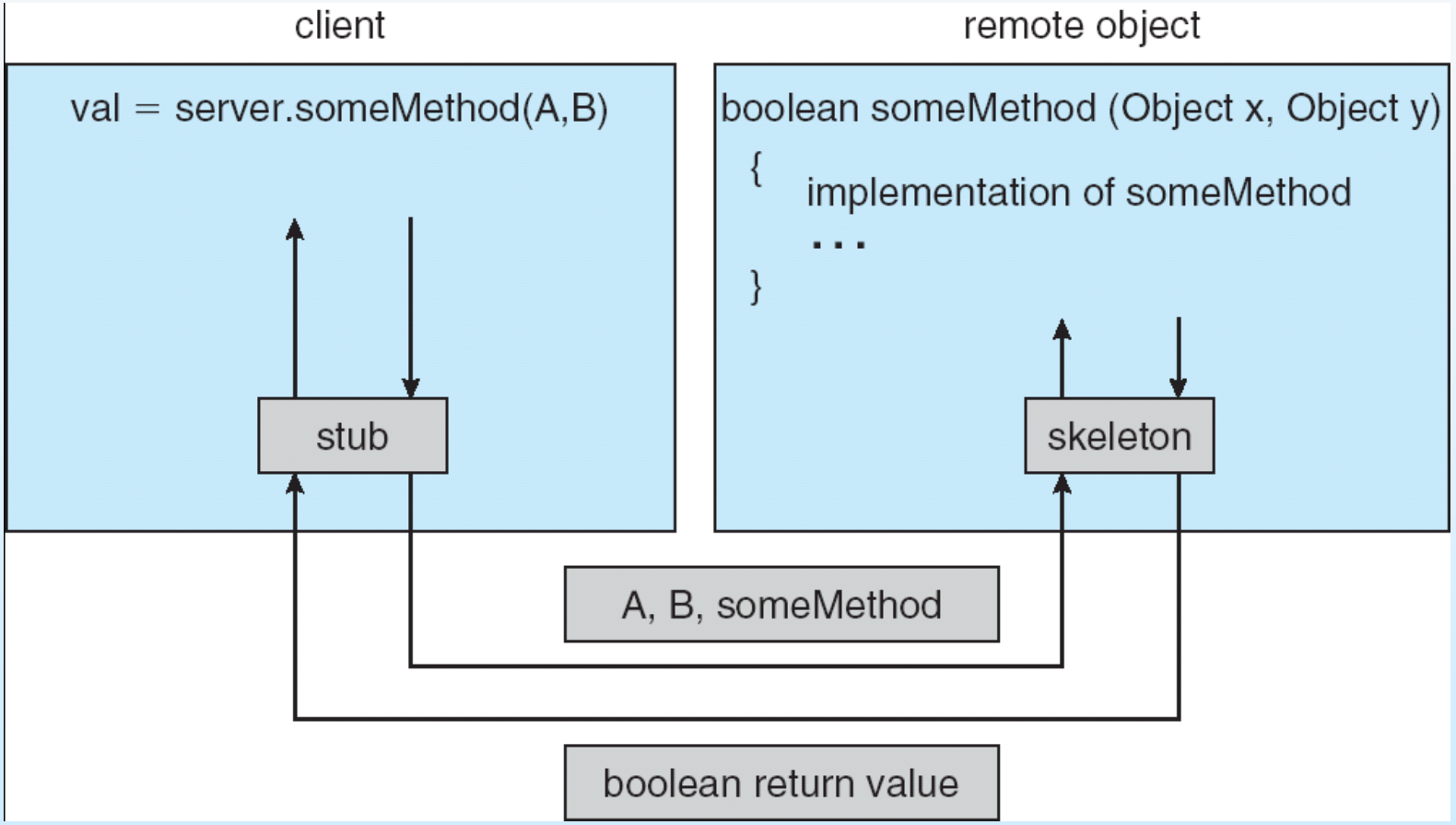


Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.



Marshalling Parameters



END