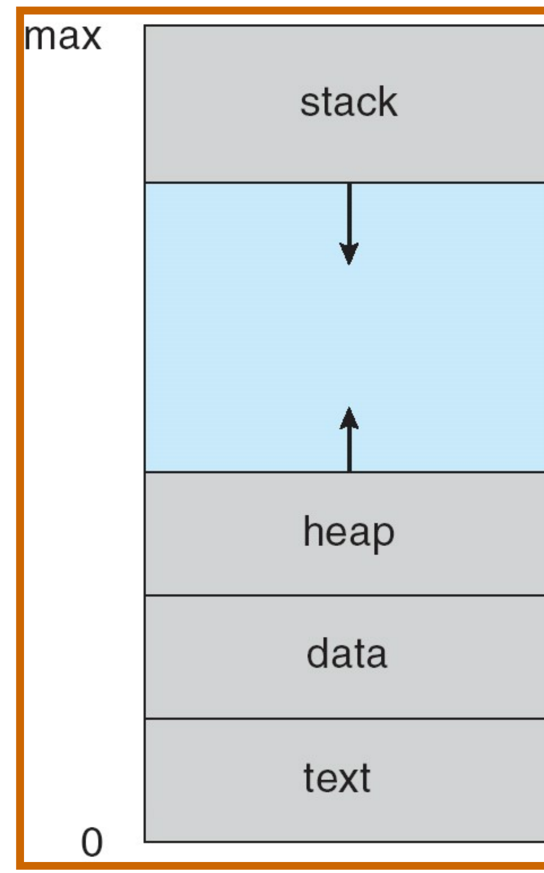
Process Management

What is a Process?

- Process – a program in execution
- A process includes:
  - program counter + registers
  - Memory contents
    - stack
    - data section
    - heap

A Process In Memory...

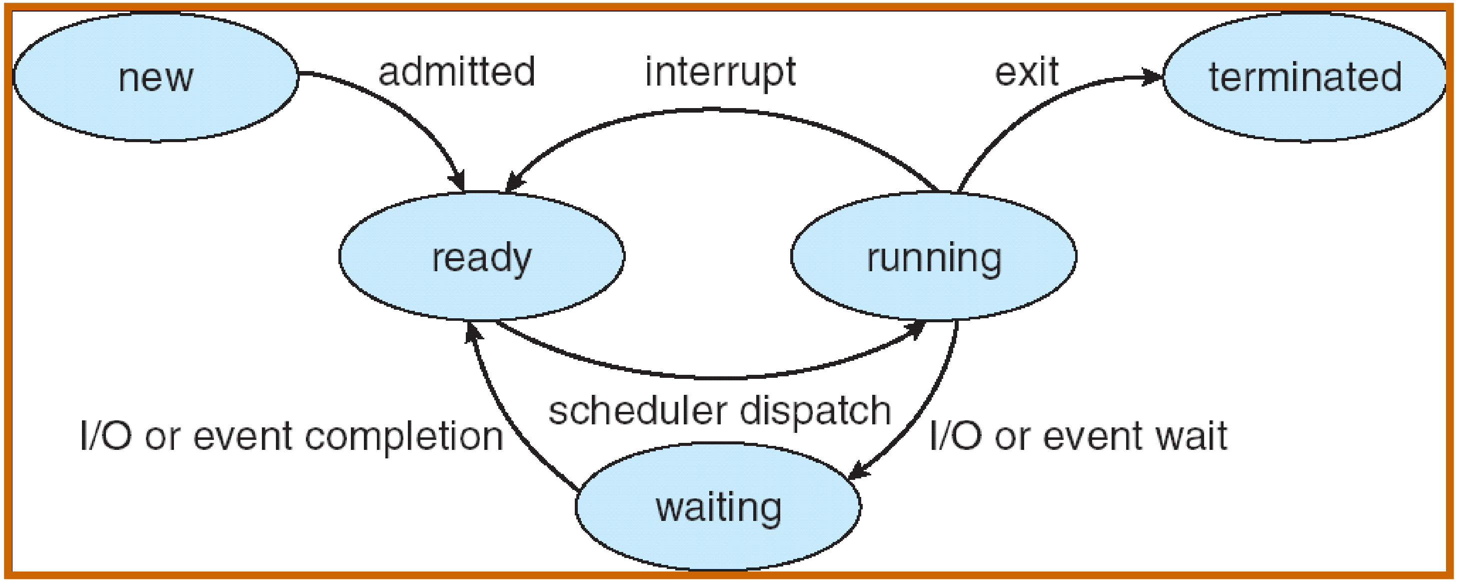
- Stack:
  - Local variables declared inside functions.
  - Function Parameters
- Heap
  - Dynamically allocated memory.
- Data
  - Global variables etc.
- Text
  - Program instructions



Process State

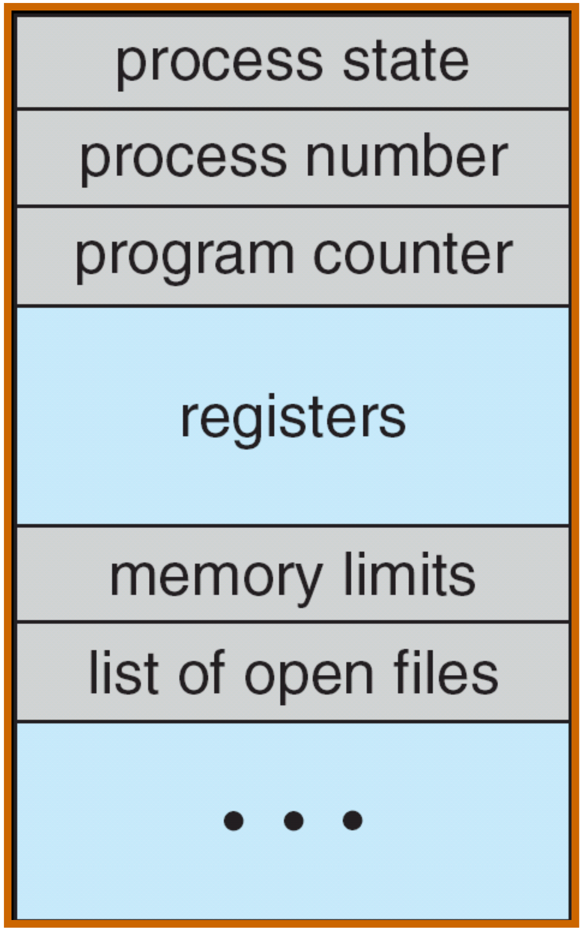
- As a process executes, it changes state
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to CPU
  - **terminated**: The process has finished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

Process State Diagram



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

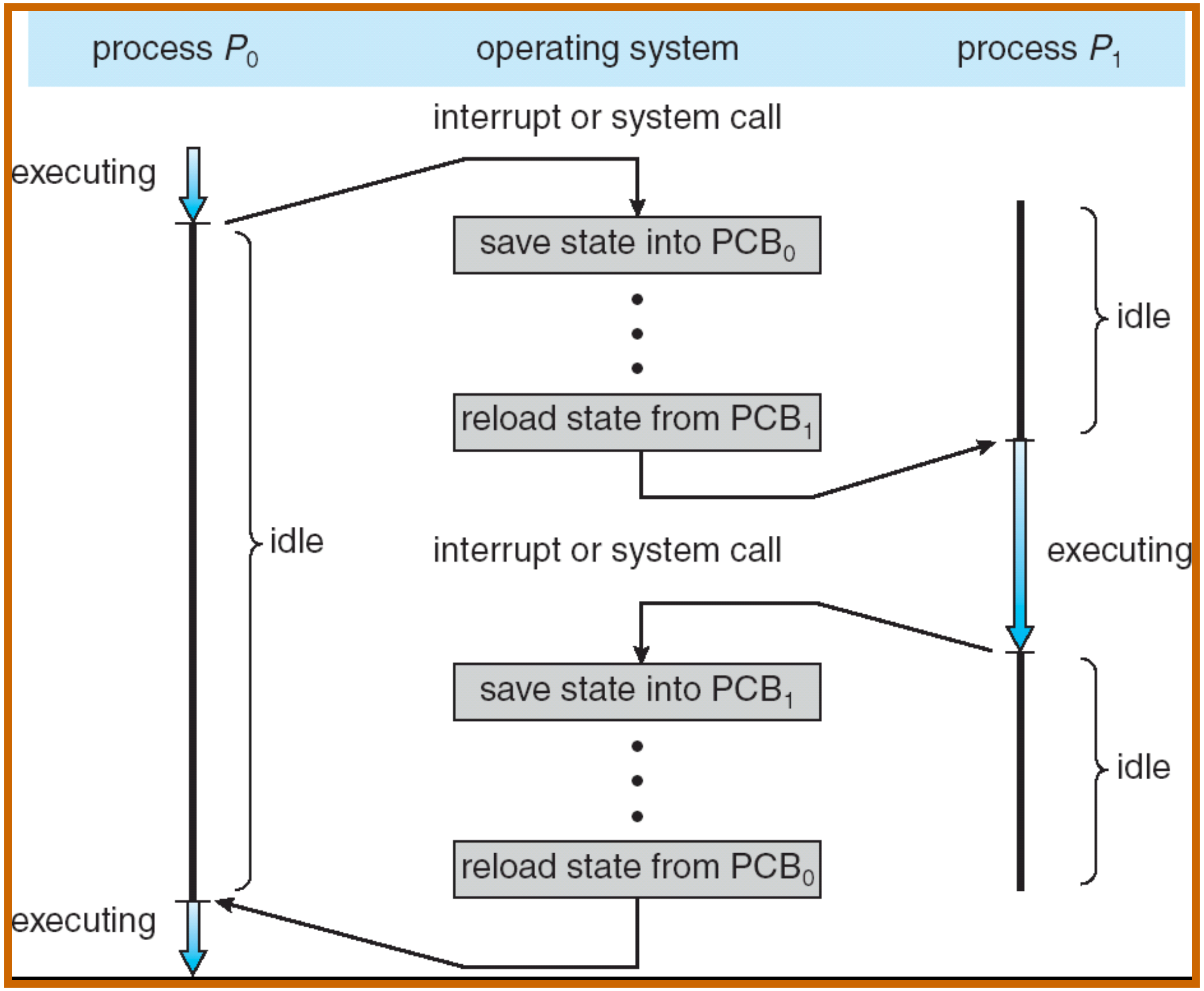


- Thinking ahead: Why Registers?

Context Switch

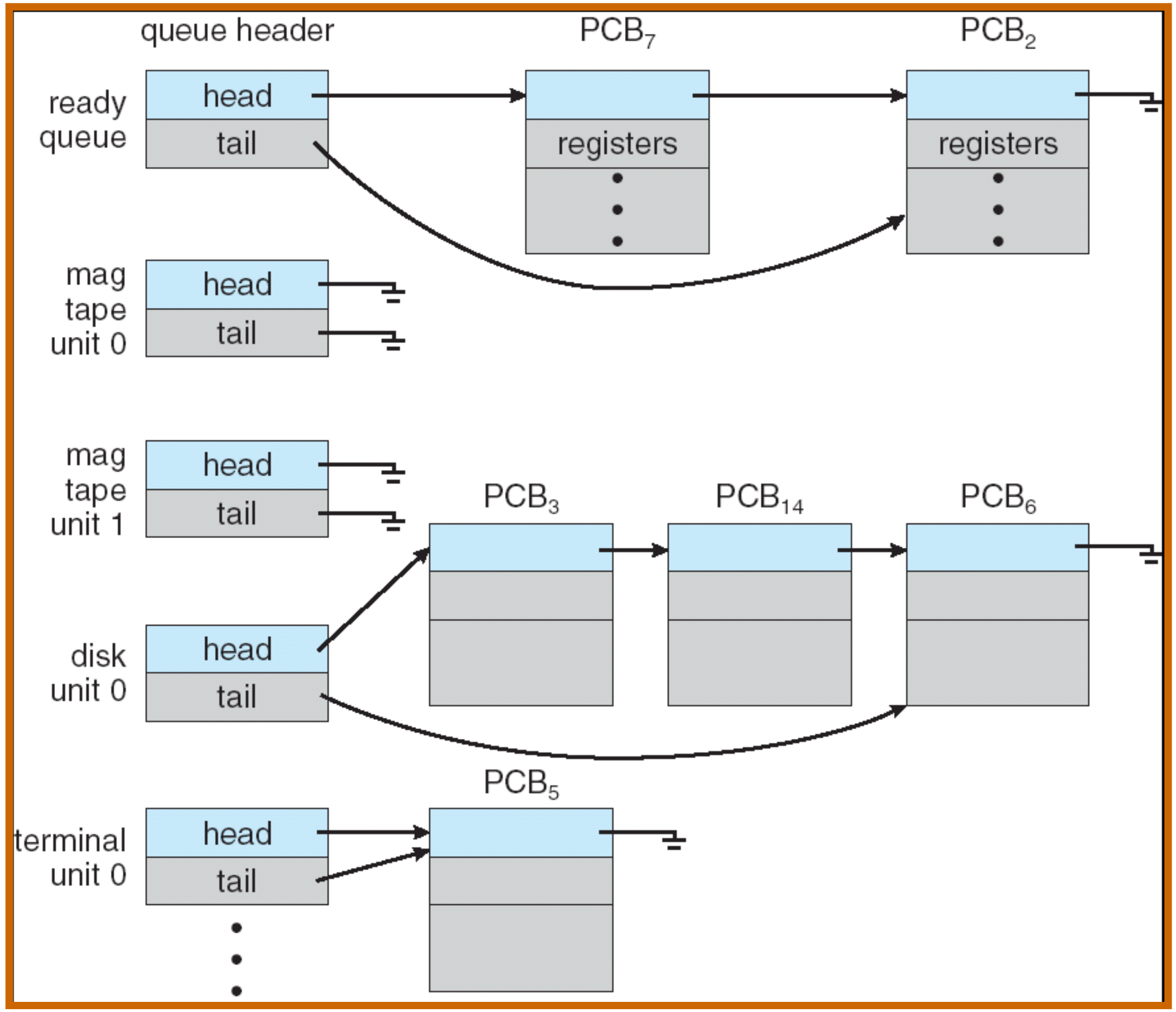
- When CPU switches to another process, the system must:
  - Save the state of the old process.
  - Load the saved state for the new process.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.

Context Switching Example



Ready Queue and Device Queues

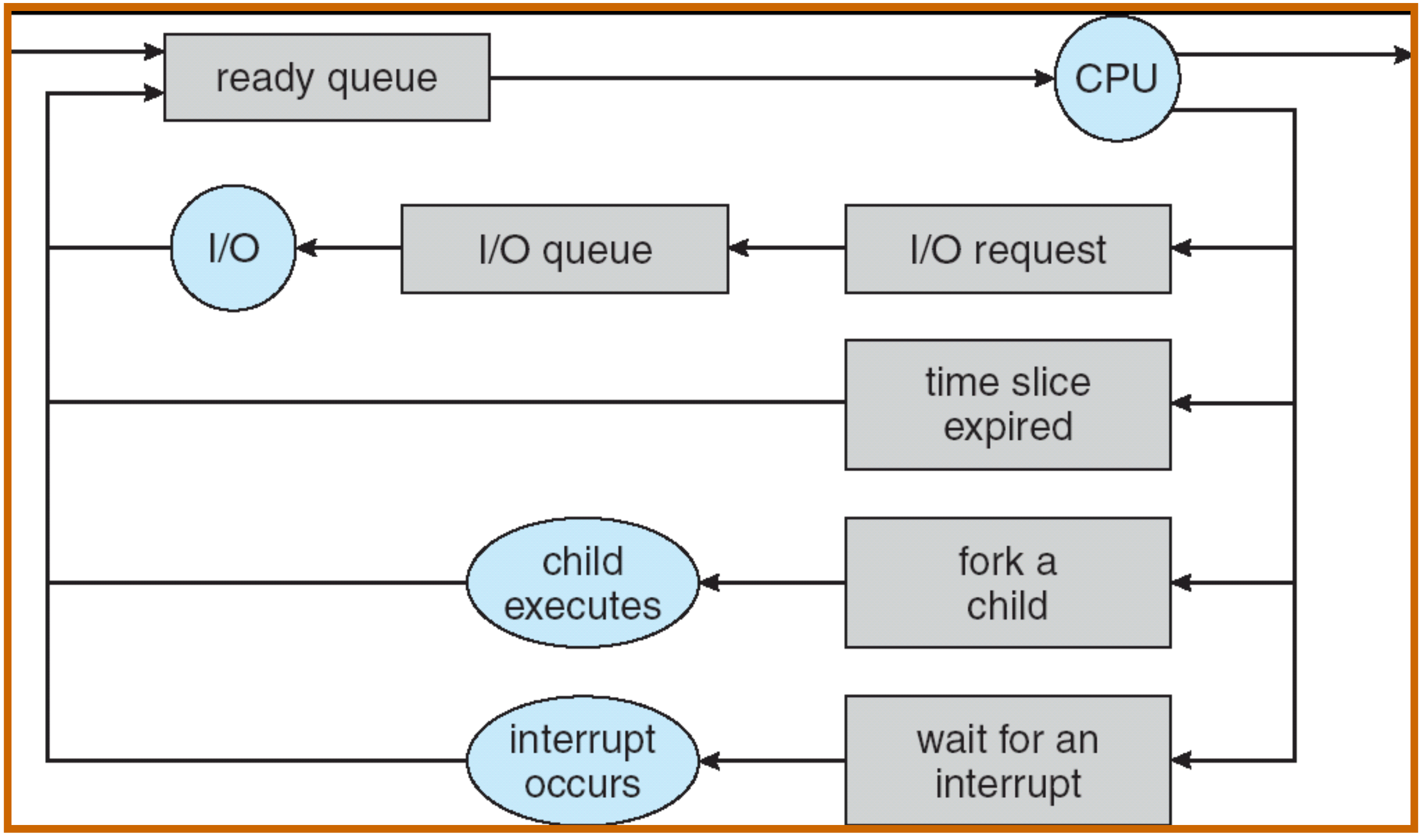
- Scheduler pulls ready processes from the ready queue.
- Waiting processes are moved to an appropriate wait queue.



Scheduler

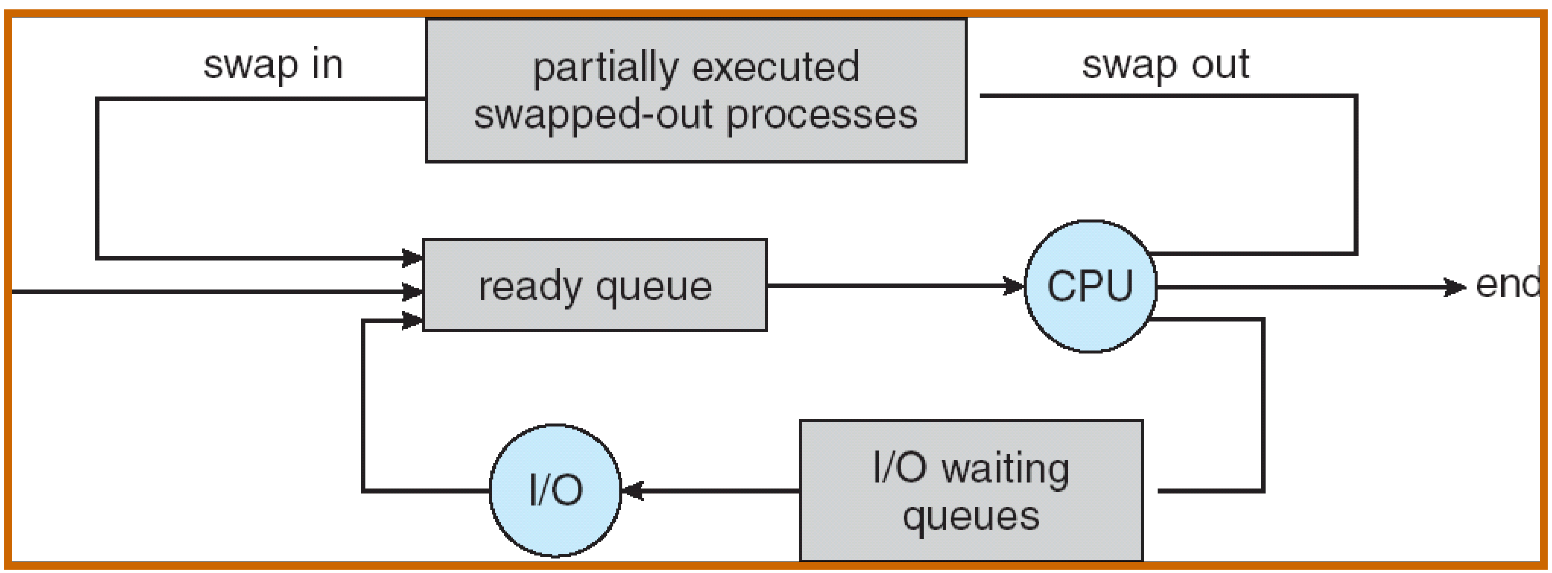
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - More of an issue for large compute servers.
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Process Scheduling



Medium Term Scheduler

- Swap out some processes if insufficient resources.
- This can avoid memory thrashing.



Schedulers

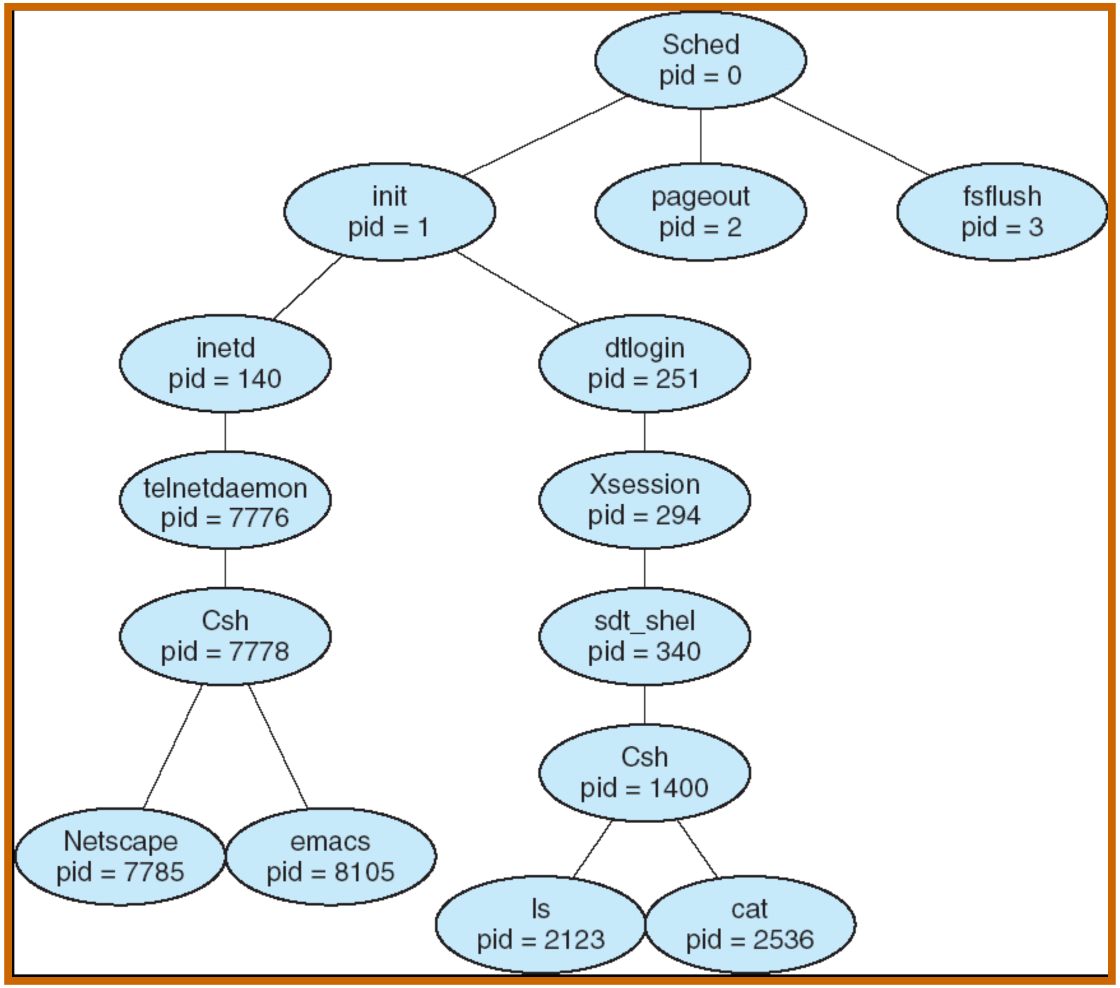
- **Short-term scheduler** is invoked frequently (milliseconds) => (must be fast)
- **Long-term scheduler** is invoked infrequently (seconds, minutes) => (may be slow)
- The long-term scheduler controls the degree of multiprogramming
- Processes can be described as either:
  - **I/O-bound process** – 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 processes create children processes, which create other processes, forming a tree.
- Resource sharing possibilities:
  - Parent and children share all resources.
  - Children share subset of parent’s resources.
  - Parent and child share no resources.
- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

Process Tree

- In Linux: ps - ef shows all processes with id's and parent id's.
- Graphical version:
  - `gnome-system-monitor`



Process Address Space

- Address space possibilities:
  - Child duplicate of parent.
  - Child has a program loaded into it.
- UNIX examples
  - fork system call creates new process.
  - exec system call used after a fork to replace the process’ memory space with a new program.

Unix Example...

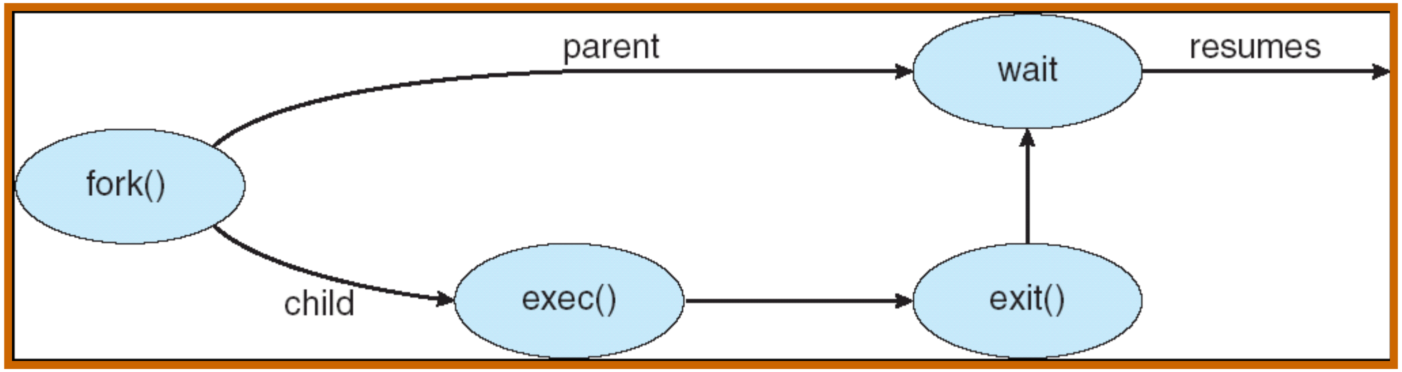
```
int main()
{
  pid_t  pid;
   pid = fork();

   if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        exit(-1);
   }

   else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
   }

   else { /* parent process */
        wait (NULL);/* wait for child to terminate */
        printf ("Child Complete");
        exit(0);
   }
}
```

fork/exec Diagram



Process Termination

- Process executes last statement and asks OS to delete it (**exit**).
  - Output data from child to parent (via wait).
  - Process' resources are deallocated by OS.
- Parent may terminate execution of child processes (**kill**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - If parent is exiting (sometimes).

Interprocess Communication

- **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

Shared Memory IPC

- Shared memory region exists in the address space of one process.
- Other processes attach that region to their own address space

Producer Consumer Problem

- Paradigm for cooperating processes.
- Producer process produces information that is consumed by a consumer process.
  - **unbounded-buffer** places no limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size
- Book gives a solution to the bounded-buffer version that does not use explicit synchronization mechanisms.

Bounded Buffer Solution

- Shared data

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

item buffer[BUFFER_SIZE];
int in = 0;  /* next free position position */
int out = 0; /* first full position */
```

Bounded Buffer Solution

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

```
while (true) {
    while (in == out)
        ; /* do nothing -- nothing to consume */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER SIZE;
}
```

Unix Shared Memory Example

```
int main(int argc, char *argv[]) {
      int status;
      int pid;

      char* shared_mem;
      int segment_id;
      int size = 4096;
      segment_id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR);

      pid = fork();  /* CONTINUED...*/
```

- `shmget` creates a shared memory region, and returns its ID.

Unix Shared Memory Example

```
if (pid==0) { /* CHILD PROCESS */
   shared_mem = (char*) shmat(segment_id, NULL, 0);
   int i = 0;
   int j = 0;
   for (i = 0; i< 100000;i++){
     for (j = 0; j< 10000; j++) {;}
     sprintf(shared_mem, "aaaaaaaaaaaaaaaa\n");
     printf("%s", shared_mem);
   }
}
```

- `shmat` attaches the shared memory to the process address space.

Unix Shared Memory Example

```
      if (pid > 0) { /* PARENT PROCESS */
        shared_mem = (char*)shmat(segment_id, NULL, 0);
        int i=0;
        int j=0;
        for (i = 0; i < 100000; i++){
          for (j = 0; j < 10000; j++){;}
          sprintf(shared_mem, "bbbbbbbbbbbbbbbb\n");
          printf("%s", shared_mem);
        }
    }
}
```

- Anything wrong with our example?

Message Passing

- Message passing facility provides two operations:
  - **send**(message) – message size fixed or variable
  - **receive**(message)
- If 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?

Direct Communication

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

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.or bi-directional.

Indirect Communication

- Operations:
  - Create a new mailbox.
  - Send and receive messages through mailbox.
  - Destroy a mailbox.
- Mailbox can belong to a particular process, or to the OS.
- Primitives are defined as:
  - **send**(A, message) – send a message to mailbox A.
  - **receive**(A, message) – receive a message from mailbox A.

Synchronization

- Message passing may be either blocking or non-blocking.
- Blocking is considered synchronous.
  - **Blocking send** - sender blocks until message is received
  - **Blocking receive -** receiver blocks until a message is available
- Non-blocking is considered asynchronous.
  - **Non-blocking send** - sender sends the message and continues.
  - **Non-blocking receive** - receiver receive a valid message or null.

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  - **Zero capacity** – 0 messages
    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

Example: Mach

- Mach micro-kernel developed at CMU between 1985 to 1994.
- Descendants of Mach:
  - Apple's Os X.
    - Darwin is an amalgam of FreeBSD and Mach.
  - GNU HURD / GNU Mach.
- Mach makes extensive use of message passing.
- Even system calls are made by sending messages to a special kernel mailbox.

Mach Messages

- Relevant system calls:
  - `msg_send()`, `msg_receive()`
    - Messages are composed of:
      - Header, including message size and return address.
      - Variable sized data segment.
  - `msg_rpc()` - Remote Procedure Call (more in a second)
  - `port_allocate()` - create a mailbox.
    - Defaults to a queue size of 8.
  - `port_status()` - check number of pending messages.

Mach Messaging Properties

- Mailboxes belong to OS.
- Message passing can be blocking or non-blocking.
  - Possible to set a timeout value.
- Buffers have bounded capacity.

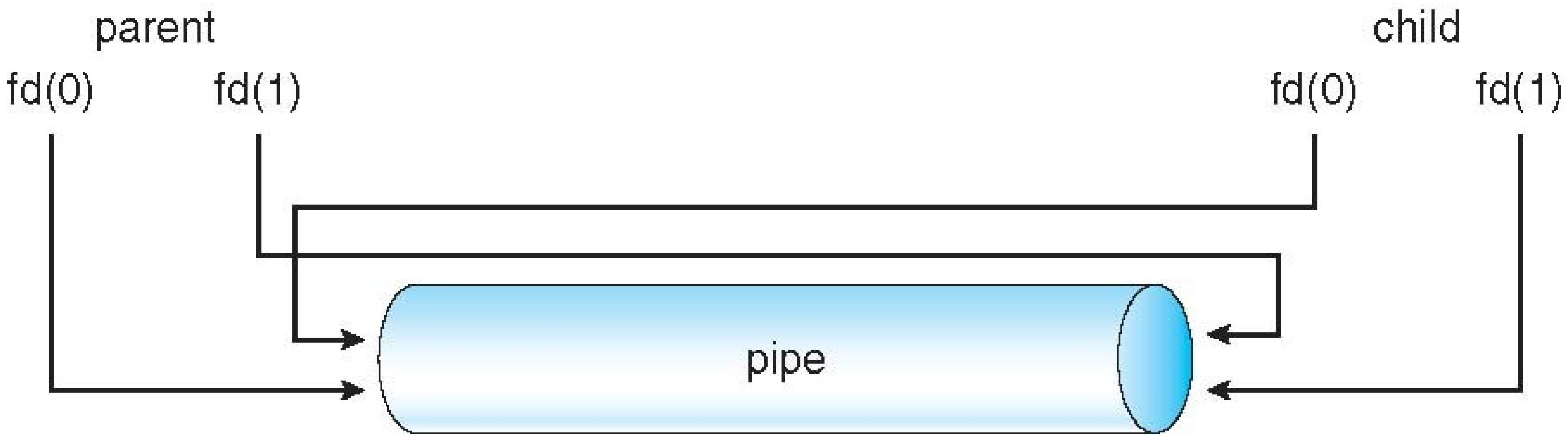
Pipes

- File-like IPC mechanism
  - Ordinary (unnamed) pipes
  - Named pipes, also known as fifo's

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)
- Require parent-child relationship between communicating processes
- Created in Linux using the `pipe` system call.

Ordinary Pipes



Named Pipes

- Named pipes appear to be regular files, but exist only for IPC
- 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
- In Linux, created with the `mkfifo` system call

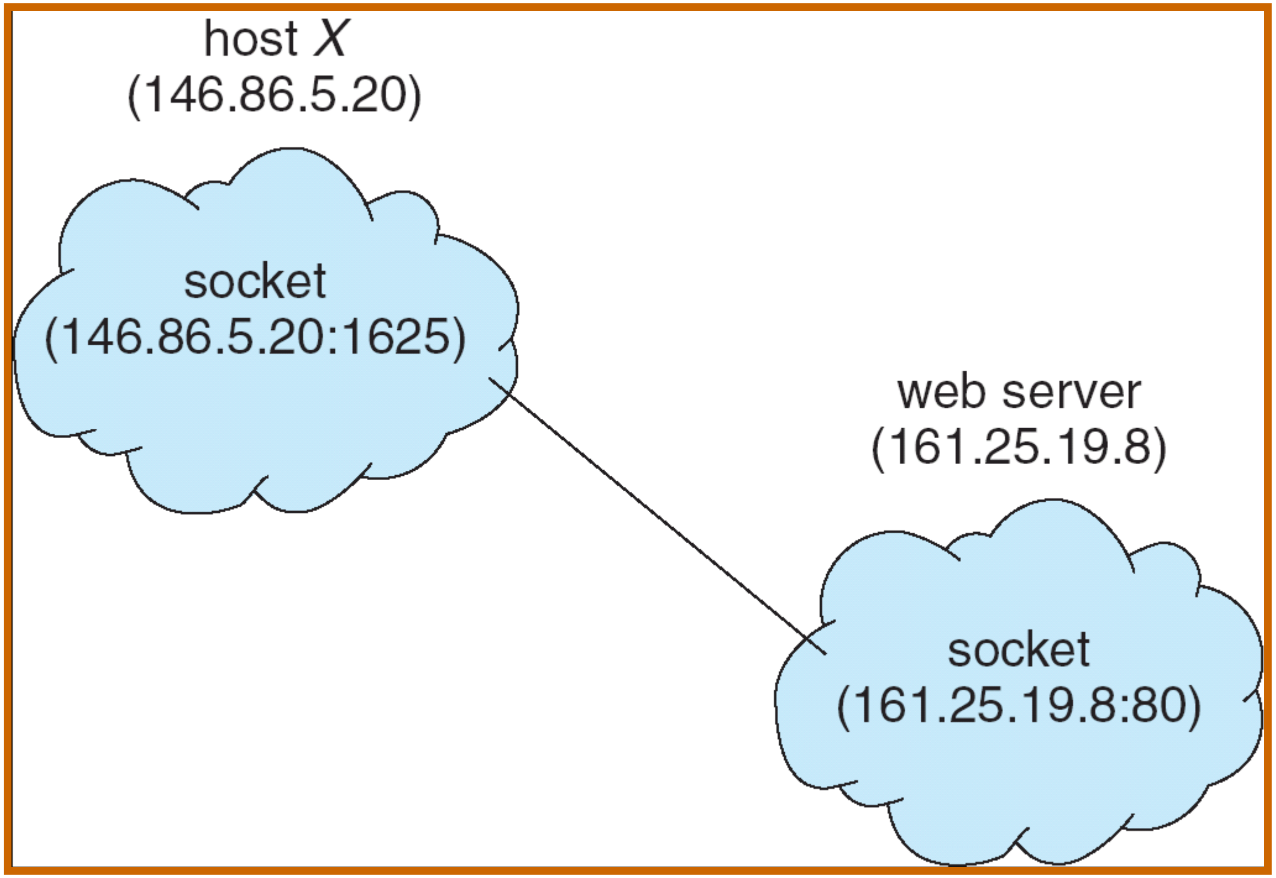
Client Server Communication

- IPC mechanisms we've seen so far are mostly intra-computer.
- It may also be necessary for processes to communicate across a network.
- Let's look at:
  - Sockets.
  - Remote Procedure Calls.
  - Remote Method Invocation (Java).
- Note that the inter/intra distinction is not hard and fast.

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 occurs between a pair of sockets.

Sockets


host X
(146.86.5.20)
socket
(146.86.5.20:1625)
web server
(161.25.19.8)
socket
(161.25.19.8:80)

Socket Code Examples

- Let's look at the Java example from the book...
- We'll also look at a C example...
  - (From Beej's Guide to Network Programming)

From the Linux socket man Page

socket(2) creates a socket, **connect**(2) connects a socket to a remote socket address, the **bind**(2) function binds a socket to a local socket address, **listen**(2) tells the socket that new connections shall be accepted, and **accept**(2) is used to get a new socket with a new incoming connection. **socketpair**(2) returns two connected anonymous sockets (only implemented for a few local families like PF_UNIX)

send(2), **sendto**(2), and **sendmsg**(2) send data over a socket, and **recv**(2), **recvfrom**(2), **recvmsg**(2) receive data from a socket. poll(2) and select(2) wait for arriving data or a readiness to send data. In addition, the standard I/O operations like **write**(2), **writev**(2), **send-file**(2), **read**(2), and **readv**(2) can be used to read and write data.

getsockname(2) returns the local socket address and **getpeername**(2) returns the remote socket address. **getsockopt**(2) and **setsockopt**(2) are used to set or get socket layer or protocol options. **ioctl**(2) can be used to set or read some other options.

close(2) is used to close a socket. **shutdown**(2) closes parts of a full duplex socket connection

Remote Procedure Calls

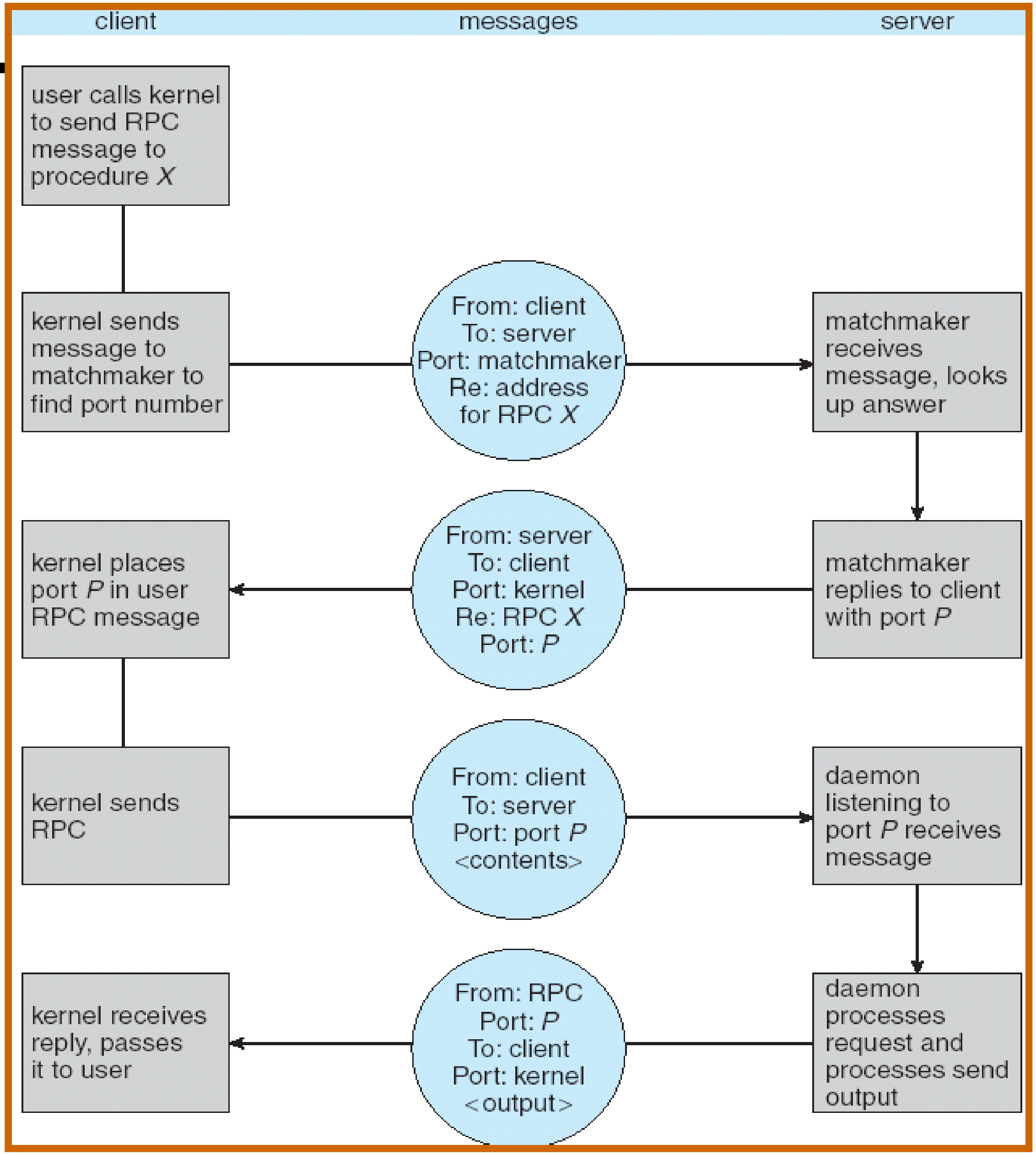
- RPCs abstract procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and **marshalls** the parameters.
  - Handles hardware differences in data representations.
- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server.

Example: NFS

- An example of a system that is implemented using RPC is the NFS networked file system.
- `man rpc` to get the scoop on the remote procedure call library that comes with Linux.

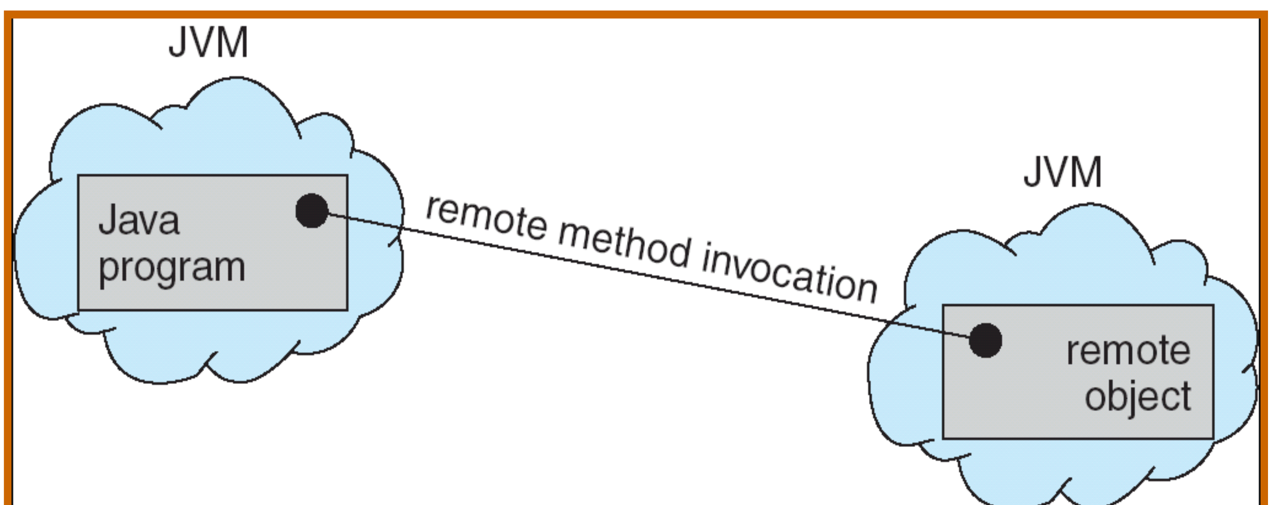
RPC Example

- Note: RPC does note necessarily need explicit kernel support.

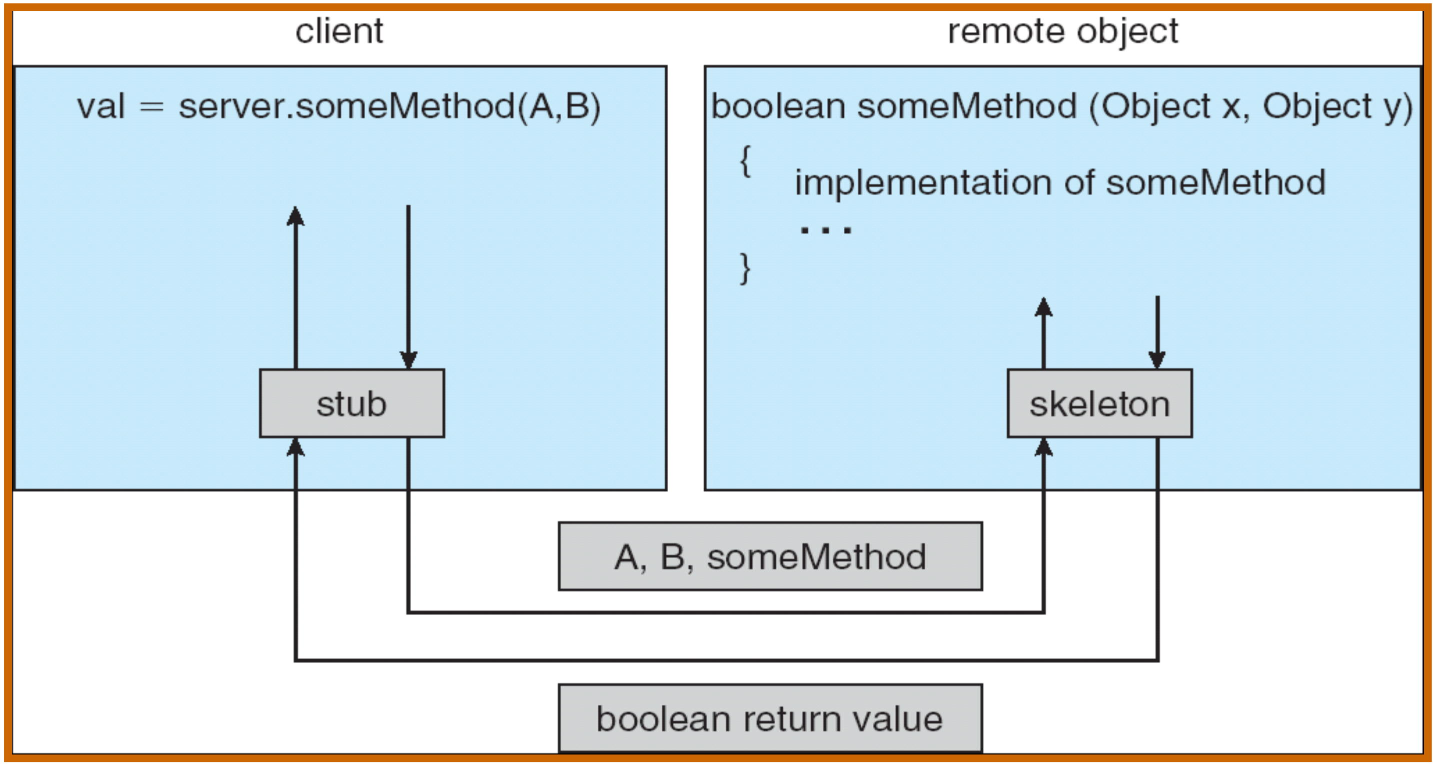


Java RMI

- 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.
- Objects that will be passed as arguments must implement the java.io.Serializable interface.



Marshalling Parameters in RMI



- Check out http://java.sun.com/products/jdk/rmi/ for the whole story.

Acknowledgments

- Portions of these slides are taken from Power Point presentations made available along with:
  - Silberschatz, Galvin, and Gagne. Operating System Concepts, Seventh Edition.
- Original versions of those presentations can be found at:
  - http://os-book.com/