Process Management

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems

4.1 Process Concept

- An operating system executes a variety of programs:
    - Batch system executes – jobs
    - Time-shared systems – has user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably.

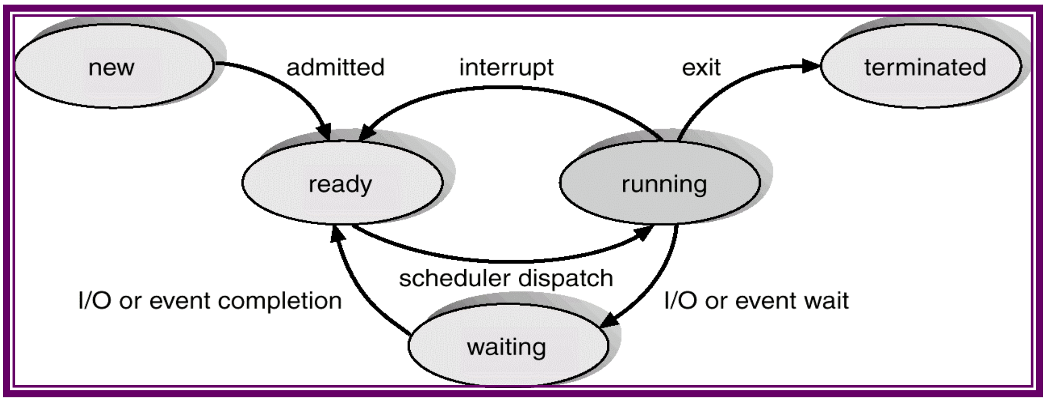
The Process

- **Process** – is a program in execution; process execution must progress in sequential fashion.
- A process includes:
    - program counter and the content of the processor's register .
    - stack – contains temporary data
    - data section – contains global variables

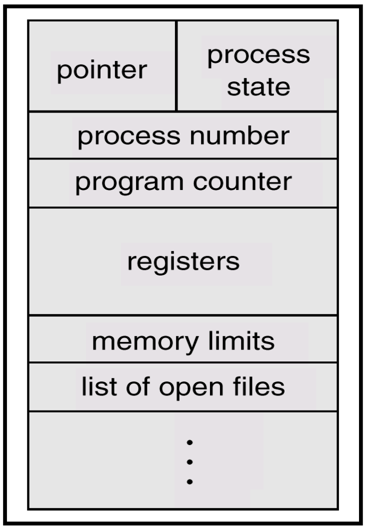
Process State

- As a process executes, it changes *state*. The **state** of a process is defined in part by the current activity of that process. Each 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.
    - **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)



Each process is represented in the operating system by a **process control block**.
information associated with each process.

- Process state – new, ready, running …
- Program counter – address of next Instruction to be executed.
- CPU registers – accumulators … etc
- CPU scheduling information – pointer To scheduling queue … etc
- Memory-management information – Value of base, limit, page table, segment table
- Accounting information – amount of CPU and real time used, time limits .. etc
- I/O status information – list of I/O devices allocated, list of open files .. etc

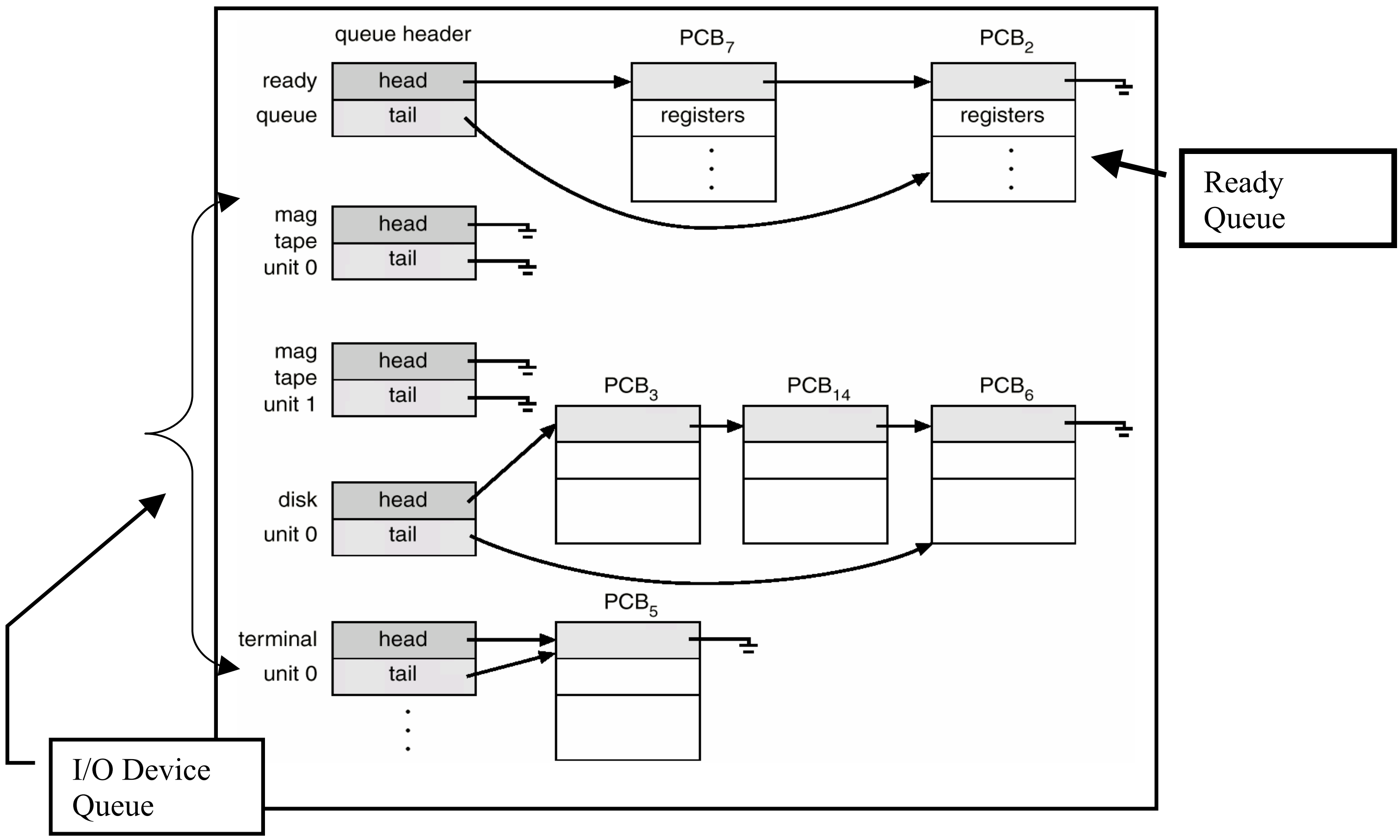
4.2 Process Scheduling

- The objective of multiprogramming is to have some process running at all the time. >>> max CPU utilization
- The objective of time-sharing is to switch the CPU among processes so frequently so that the user can interact with each program.
- The uniprocessor system can have only one running process.

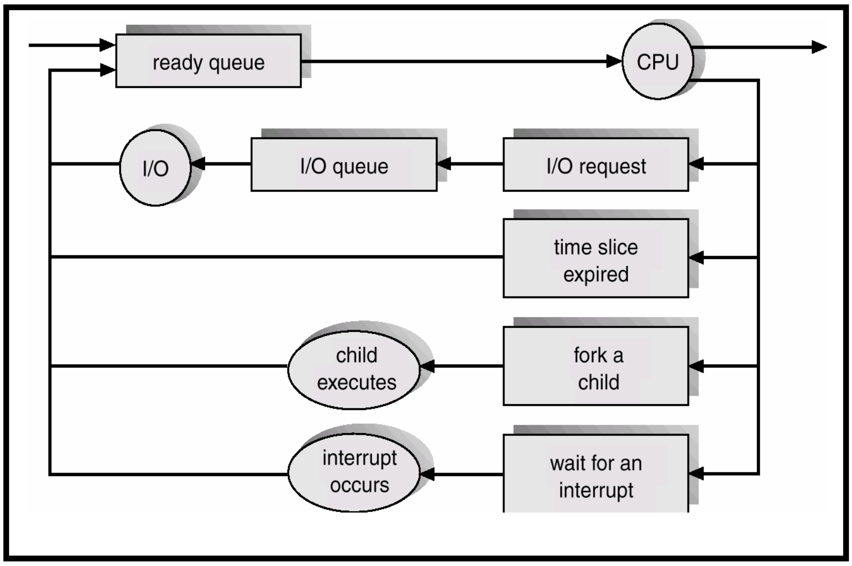
Process Scheduling Queues

- **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.
- Process migration between the various queues throughout its life time

Ready Queue And Various I/O Device Queues



Queuing Diagram - Representation of Process Scheduling



A new process is initially placed in the ready queue. It waits until it is selected for execution. Once it is executed, one of the several things might happen :-

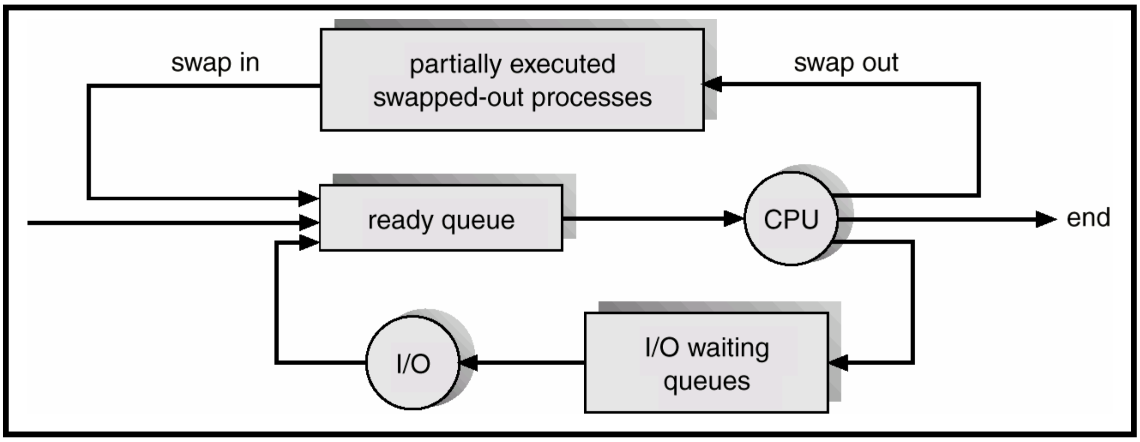
- The process could issue an I/O request, then it is placed in I/O queue.
- The process could create new subprocess and wait until it terminates.
- The process could be removed from the CPU as a result of interrupt ands put back in the ready queue

Schedulers

- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into memory for execution..
- **Short-term scheduler** (or **CPU scheduler**) – selects which process from the ready queue should be executed next and allocates CPU for it.
- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).
- The long-term scheduler controls the *degree of multiprogramming – 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. WHY ???????????**

Addition of Medium Term Scheduling

On some systems, long-term scheduler is absent or minimal.



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.
- Context-switch time is overhead (time consuming); the system does no useful work while switching.

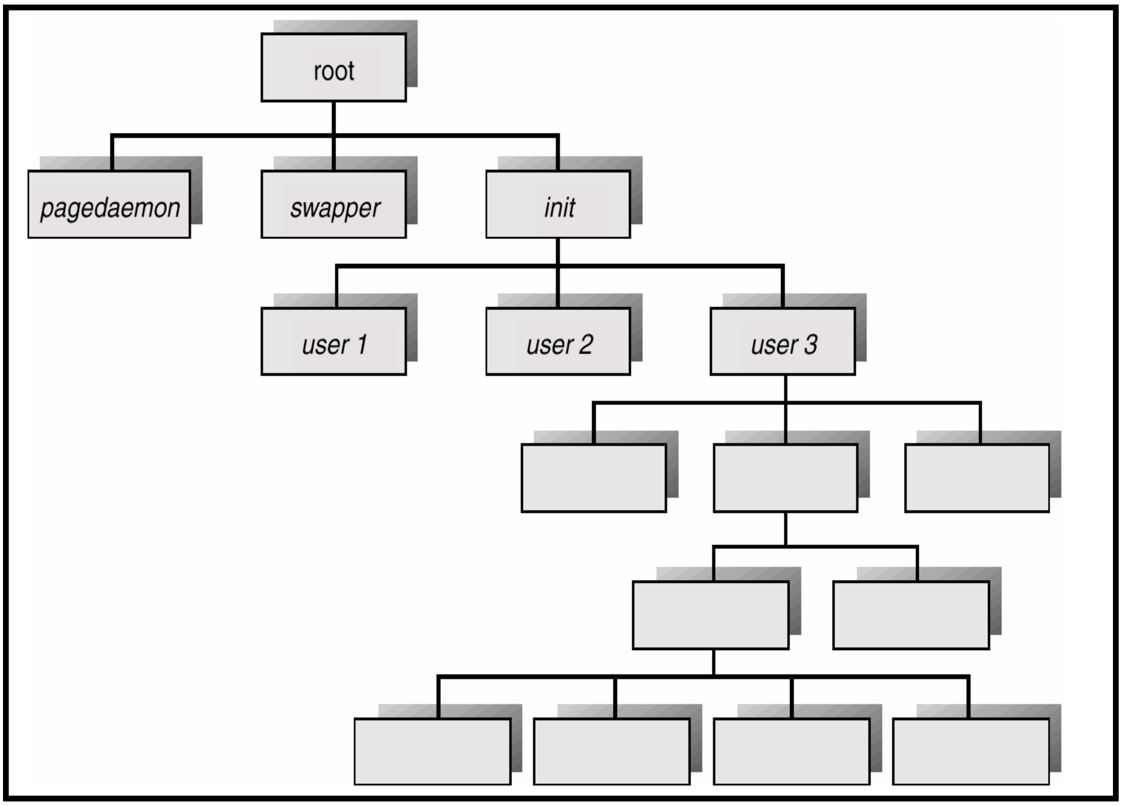
4.3 Operation on Processes.

The processes in the system can execute concurrently, and they must be created and deleted dynamically, thus the operating system must provide a mechanism for process creation and termination.

Process Creation

- A process may create several new processes during the course of execution.
- The creating process is called **parent** process, new process is called **children** of that process, thus , parent process create children processes, which, in turn create other processes, forming a tree of processes.
- Processes will need certain resources to accomplish its tasks.
- Resource sharing
  - 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.

A tree of processes on a typical UNIX System



Process Termination

- A process terminates when it finishes executing its final statement **(normal ending**) and ask the operating system to delete it by using **exit system call**.
  - Process may return data to its parent process (via the **wait** system call)
  - All the resources of the process – including virtual memory, open files, I/O buffers – are deallocated by operating system.
- Parent may terminate execution of children processes (via **abort** system call) for the following reasons :-
  - Child has exceeded its usage of some of the allocated resources.
  - The task that is assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - On such a system, if a process terminates (either normally or abnormally) then all its children must be terminated as well. (**Cascading termination**).

4.4 Cooperating Processes

The concurrent processes executing in the operating system may be either :-

- ***Independent*** process cannot affect or be affected by the execution of another process (such as processes that do not share data with any other processes).
- ***Cooperating*** process can affect or be affected by the execution of another process
- **Advantages** of process cooperation
    - Information sharing (several users may be interested in the same file)
    - Computation speed-up (break big tasks into sub tasks each of which is executed separately – need parallel processing machine).
    - Modularity (divide the system functions into separate processes).

Producer-Consumer Problem

- Paradigm for cooperating processes, ***producer*** process produces information that is consumed by a ***consumer*** process (such a print program produces characters that are consumed by the printer driver).
- To allow the producer and the consumer to run concurrently, we must have available **buffer** of items that can be filled by a producer and can be emptied by a consumer.
- The producer and the consumer must be synchronized so that the consumer does not try to consume an item that has not yet been produced.
- **Buffer** is provided by the operating system through the use of **Interprocess-communication (IPC).**

    - ***unbounded-buffer*** places no practical limit on the size of the buffer (produce can always produce new items).
    - ***bounded-buffer*** assumes that there is a fixed buffer size (Consumer must wait if buffer is empty, produce must wait if buffer is full)

4.5 Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions without sharing the address space.
- It is useful in a distributed environment where the communicating processes reside on separate machines (such as **chat** programs in WWW - **messenger** or **yahoo**).
- Best provided by message-passing system

Message-Passing System

- The function of message-passing system is to allow processes to communicate with each other without resorting to shared variables. (communicating between users).
- IPC 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 of communication link .
  - Should not be concerned with the physical implementation of the link (e.g., shared memory, hardware bus), however the logical implementation (e.g., logical properties – direct or indirect , fixed size or variable size messages .. etc)

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 when processes want to communicate..
  - A link is associated with exactly one pair of communicating processes.
  - Between each pair there exists exactly one link.
  - The link may be unidirectional, but is usually bi-directional

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.
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link may be unidirectional or bi-directional.

- A mailbox may be owned by a process or by an operating system.
- Mailboxes that are owned by a process are terminated when the process itself is terminated. Any process that send a message to this mailbox must be notified that the mailbox no longer exist.

- Operations on mailboxes that are owned by the operating system
    - create a new mailbox
    - 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

- Mailbox sharing
    - *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 (not both). Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous (** sending process is blocked until the message is received by the receiving process or a mailbox **)**
- **Non-blocking** is considered **asynchronous (** the process sends a message and resumes operation **)**
- **send** and **receive** primitives may be either blocking or non-blocking.

Buffering

Whether the communication is direct or indirect, messages reside in a temporary queue. Queue of messages are implemented in one of three ways.

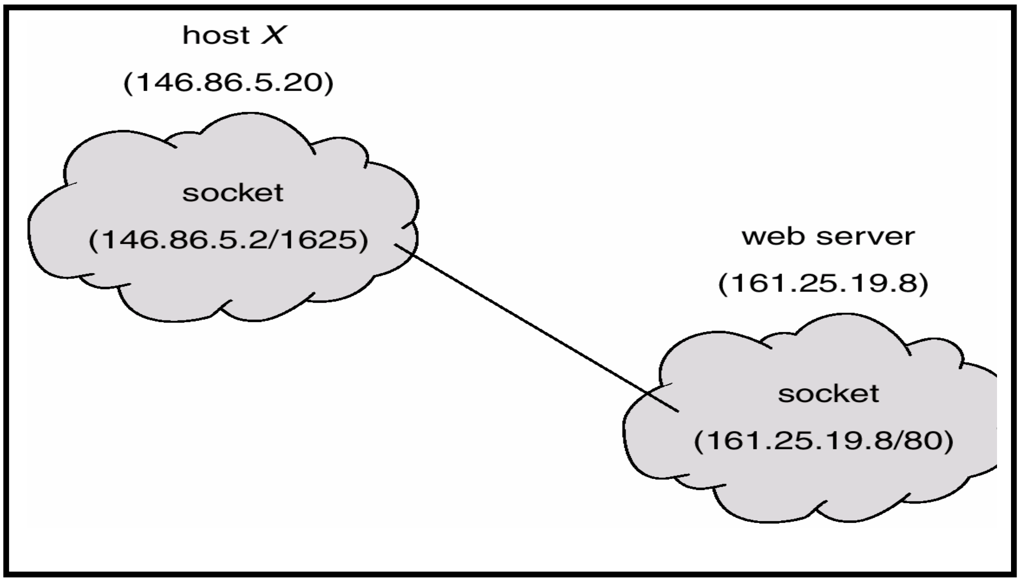
1. Zero capacity – 0 messages waiting. (sender must block until the recipient receives the message) (**rendezvous**).
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full.
3. Unbounded capacity – infinite length
   Sender never waits.

4.6 Client-Server Communication

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an *endpoint for communication*.
- A pair of processes communicating over a network employs a pair of sockets
- Concatenation of IP address and port is socket identification.
- Sockets use client-server architecture.
- The server waits for incoming client request by listening to a specific port.
- Once the request is received, the server accepts a connection from the client socket to complete the connection
- Servers implementing specific service (such as telnet , ftp .. etc) listen to well known ports (telnet server listen to port 23, ftp server listen to port 21, web server (http) server listen to port 80).
- All posts below 1024 are considered well known and used to implement standard services.
- Client host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening in port 80) at address 161.25.19.8, host X may be assigned port 1625 which is greater that 1024.

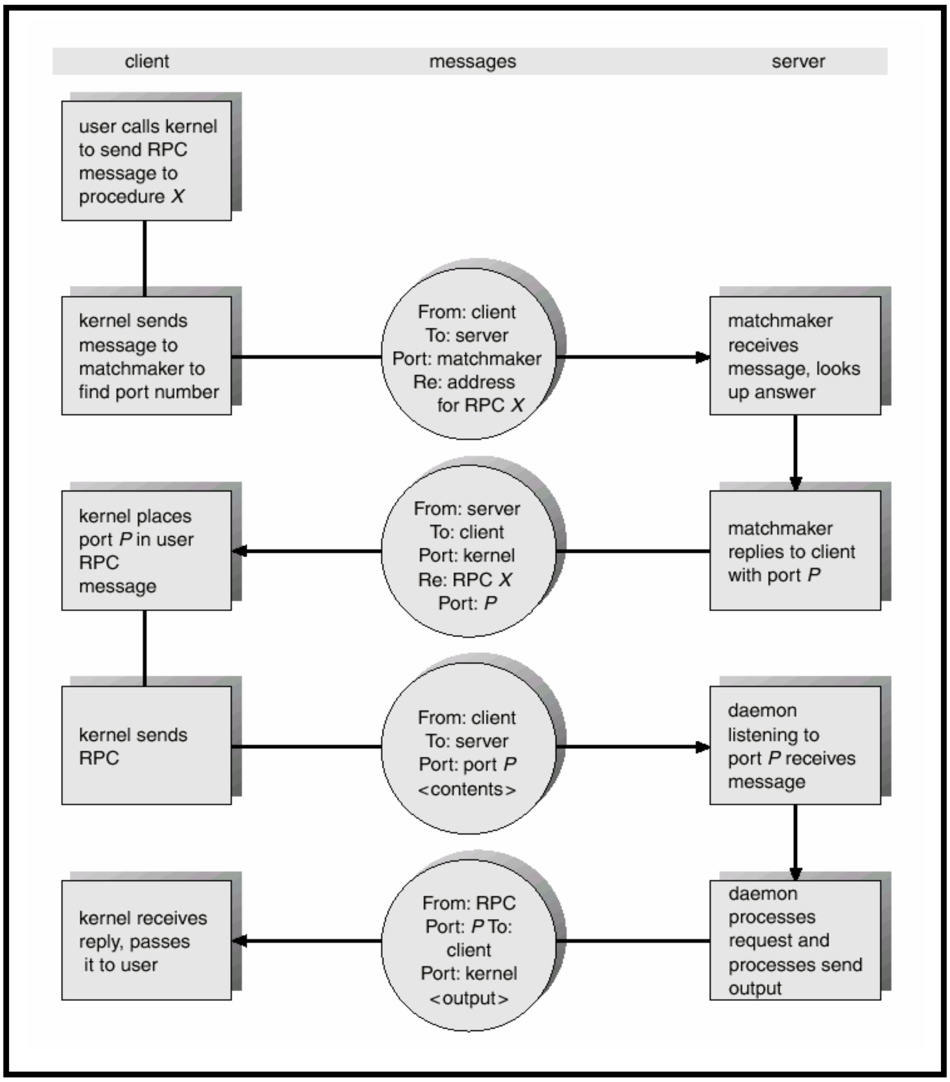


- All connections must be **unique**. If another process also on host X wants to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625.

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- Message-based communication scheme is used.
- **Stubs** – client-side proxy for the actual procedure on the server.
- 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.

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.
- Objects are considered remote if they reside in a different Java Virtual Machine (JVM).
- A **stub** is a proxy for a remote object. It resides with the client.
- A client invokes a remote method, the stub for the remote object is called.
- The client-side stub is responsible for creating a **parcel** consisting the name of the method to be invoked on the server and the marshalled parameters for the method.
- The stub send the parcel to the server.
- The **skeleton** for the remote object receives the parcel.
- The skeleton is responsible for unmarshalling the parameters and invoking the desired method on the server.
- The skeleton then marshalls the return value (or exception if any) into a parcel and return this parcel to the client.
- The stub unmarshalls the return value and passes it to the client.

Remote Method invocation