Chapter 2: Processes

Chapter 2: Processes

- Process Concept
- Process States
- Process Control Block (PCB)
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

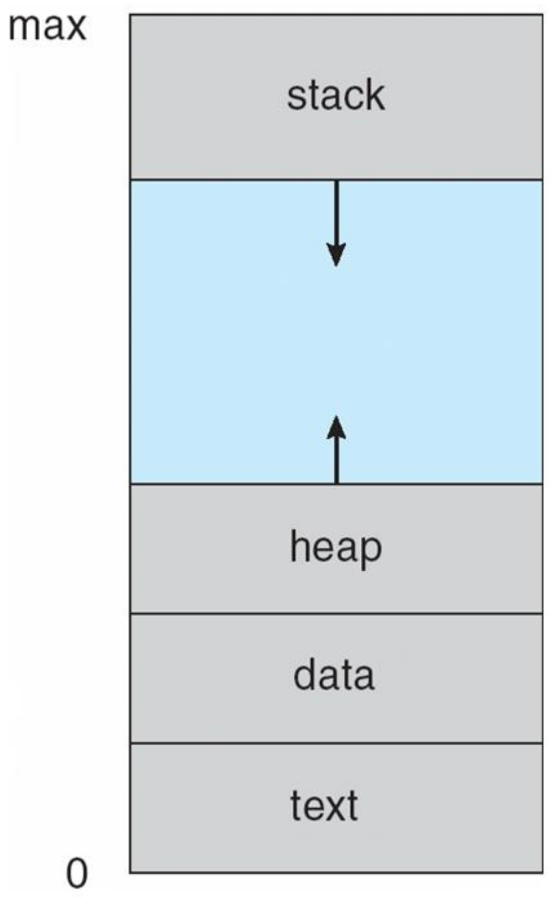
Introduction

- A process is a **program in execution**.
- It includes:
    - Program code (text section)
    - Data (variables, heap, stack)
    - Current activity (program counter, CPU registers).
- Example: Running *MS Word* → Program on disk becomes a *process* in memory.

Concept of a Process

- Difference between **program** and **process**:
  - Program = passive (on disk).
  - Process = active (running in memory).
- A single program can have multiple processes.
- Example: Opening 3 Chrome windows → 3 processes of Chrome.

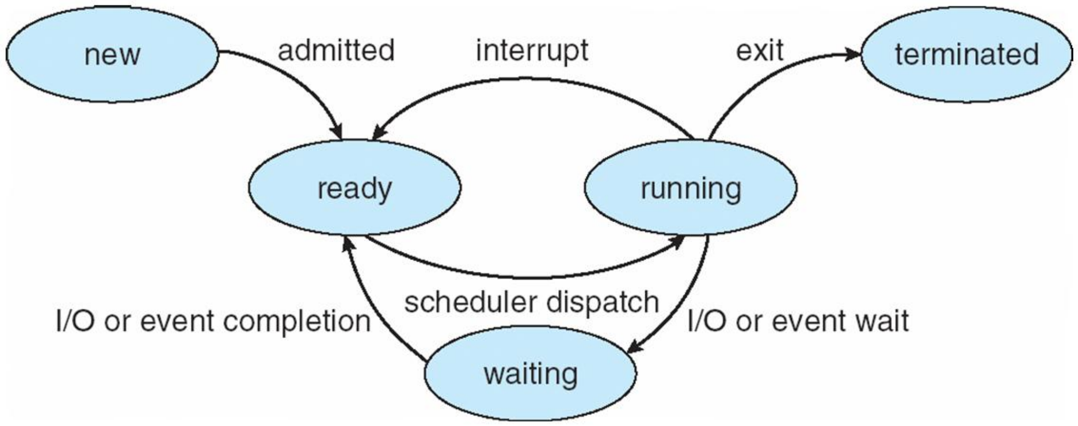
Process in Memory



Process States

- **New:** Process is being created.
- **Ready:** Waiting for CPU.
- **Running:** Currently executing.
- **Waiting:** Waiting for I/O or event.
- **Terminated:** Finished execution.
- Example: While downloading a file, the browser may switch between **Running** (CPU active) and **Waiting** (I/O pending).

Diagram of Process State



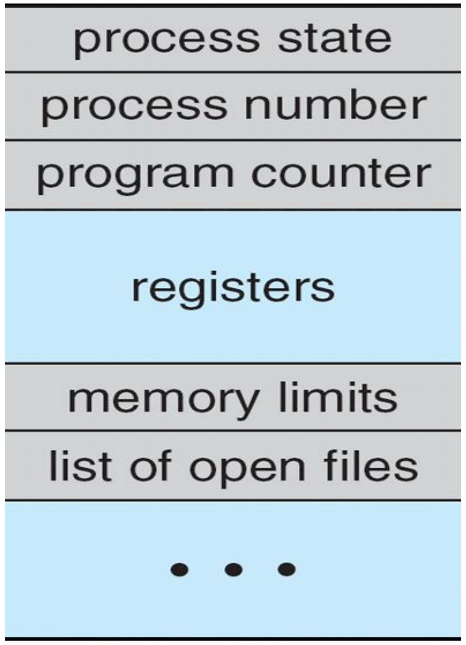
Concurrent Processes

- Multiple processes running at the same time (time-shared).
- The CPU switches rapidly between them → illusion of parallelism.
- Example: Listening to music 🎵 while typing a Word document 📝 and downloading a file 📥 .
- In multi-core systems: true parallelism possible.

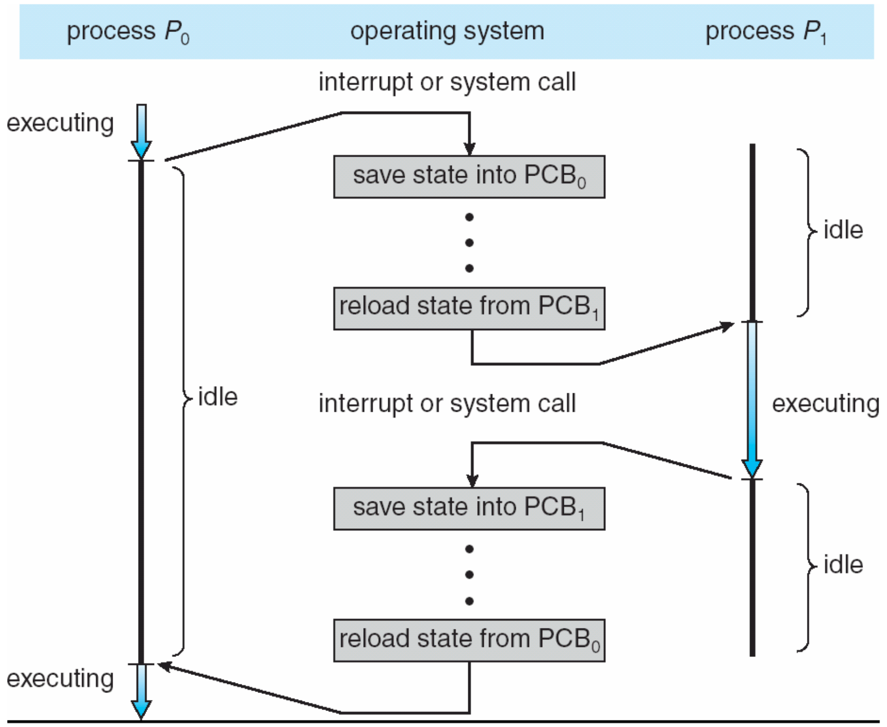
Process Control Block (PCB)

Information associated with each process
(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- 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



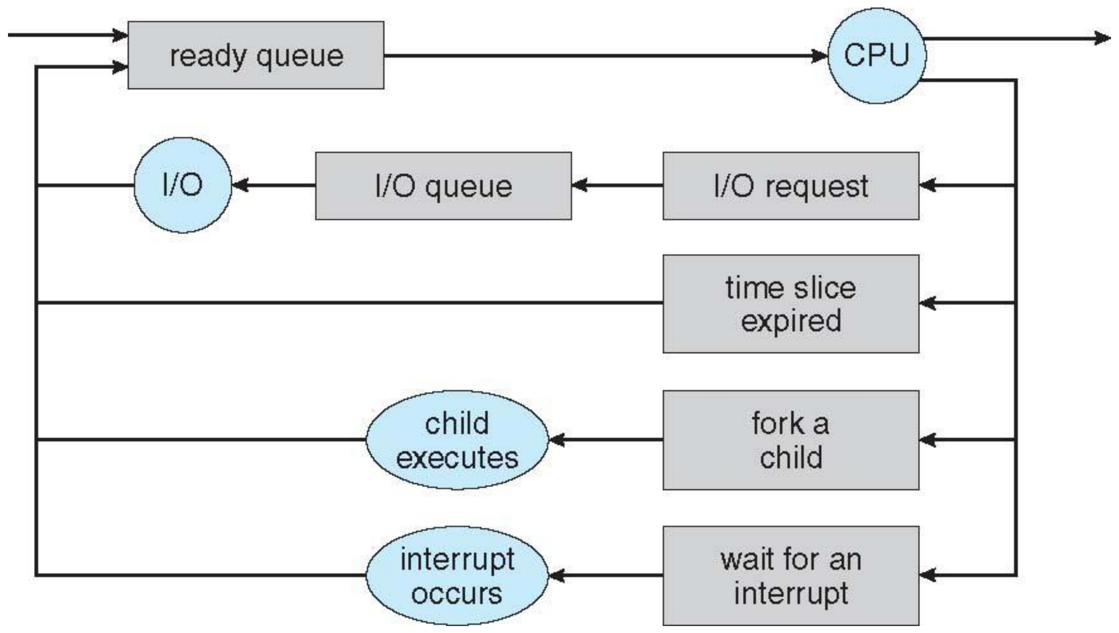
CPU Switch From Process to Process



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 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

Representation of Process Scheduling

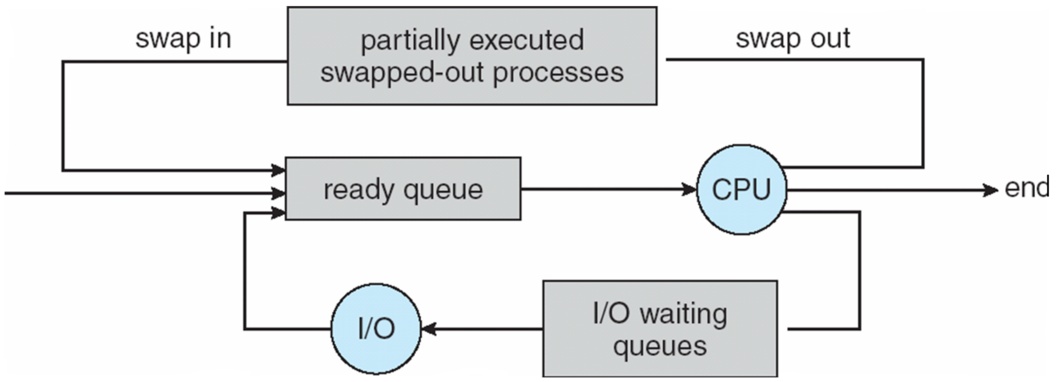


Queueing diagram represents queues, resources, flows

Schedulers

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

Addition of Medium Term Scheduling



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

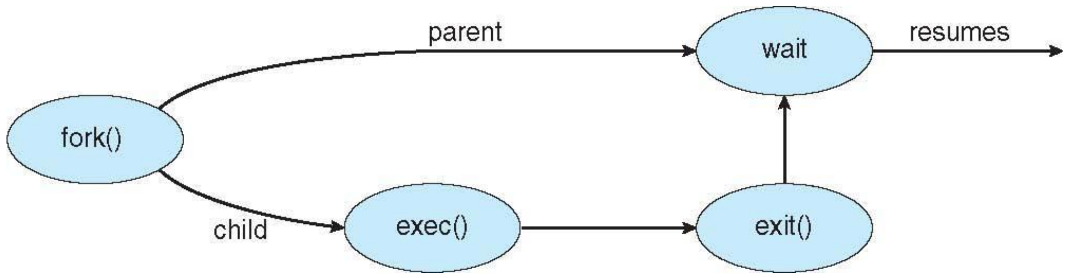
Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ⍰ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ⍰ multiple contexts loaded at once

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)



- Address space
  - 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

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit)**
  - Returns status data from child to parent (via **wait()**)
  - Process’ resources are deallocated by operating system
- Parent may terminate the execution of children processes (**abort)**
  - 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

Interprocess Communication

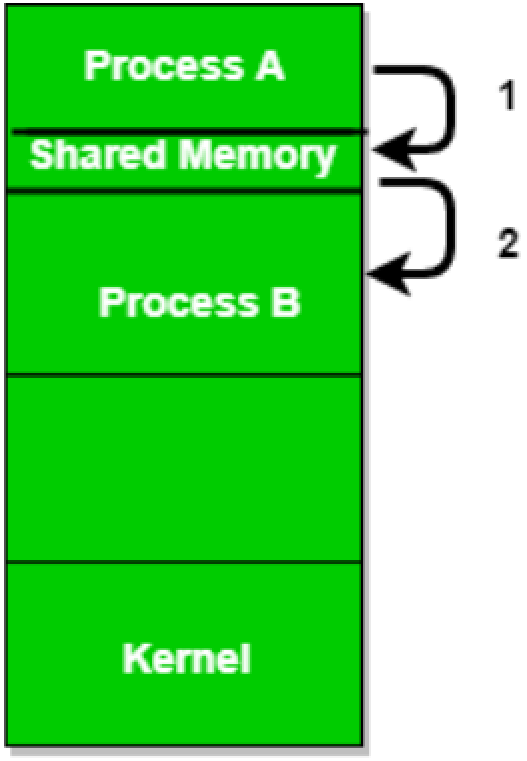
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Cooperating Processes

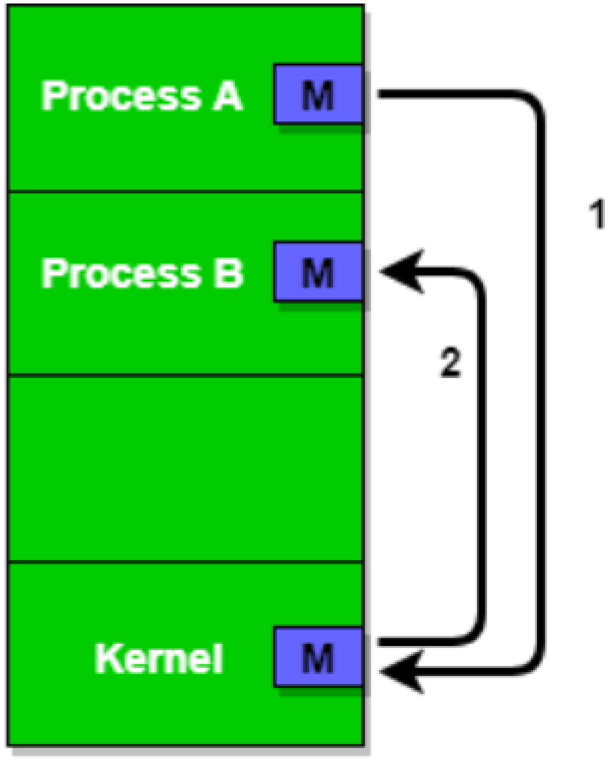
- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Communications Models

(a) Shared memory



(b) Message passing.



Producer-Consumer Problem

- Paradigm for cooperating processes, *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;
```

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

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;
}
```

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 */
}
```

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 facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation of communication link
  - Physical (e.g., shared memory, hardware bus)
  - Logical (e.g., logical properties(Synchronous or asynchronous))

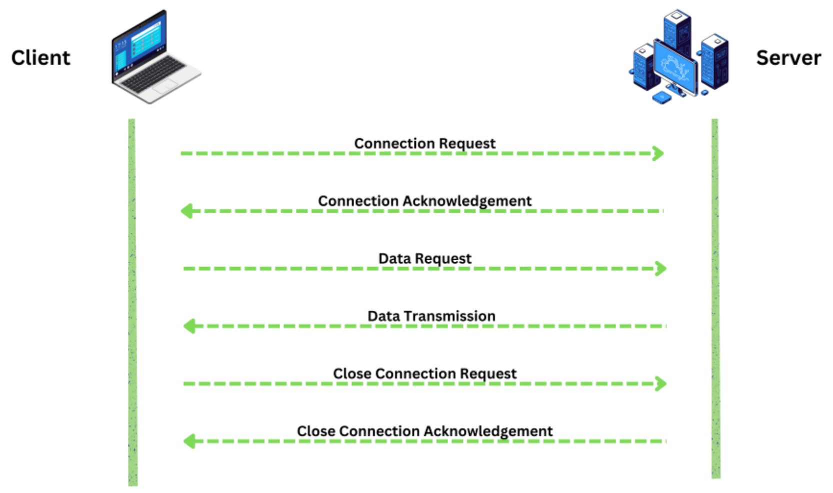
Direct Communication

- 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

Communications in Client-Server Systems

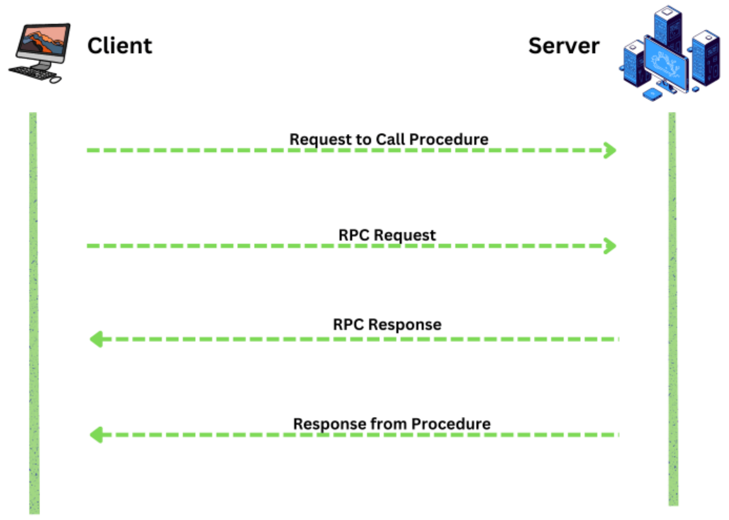
- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)

Sockets



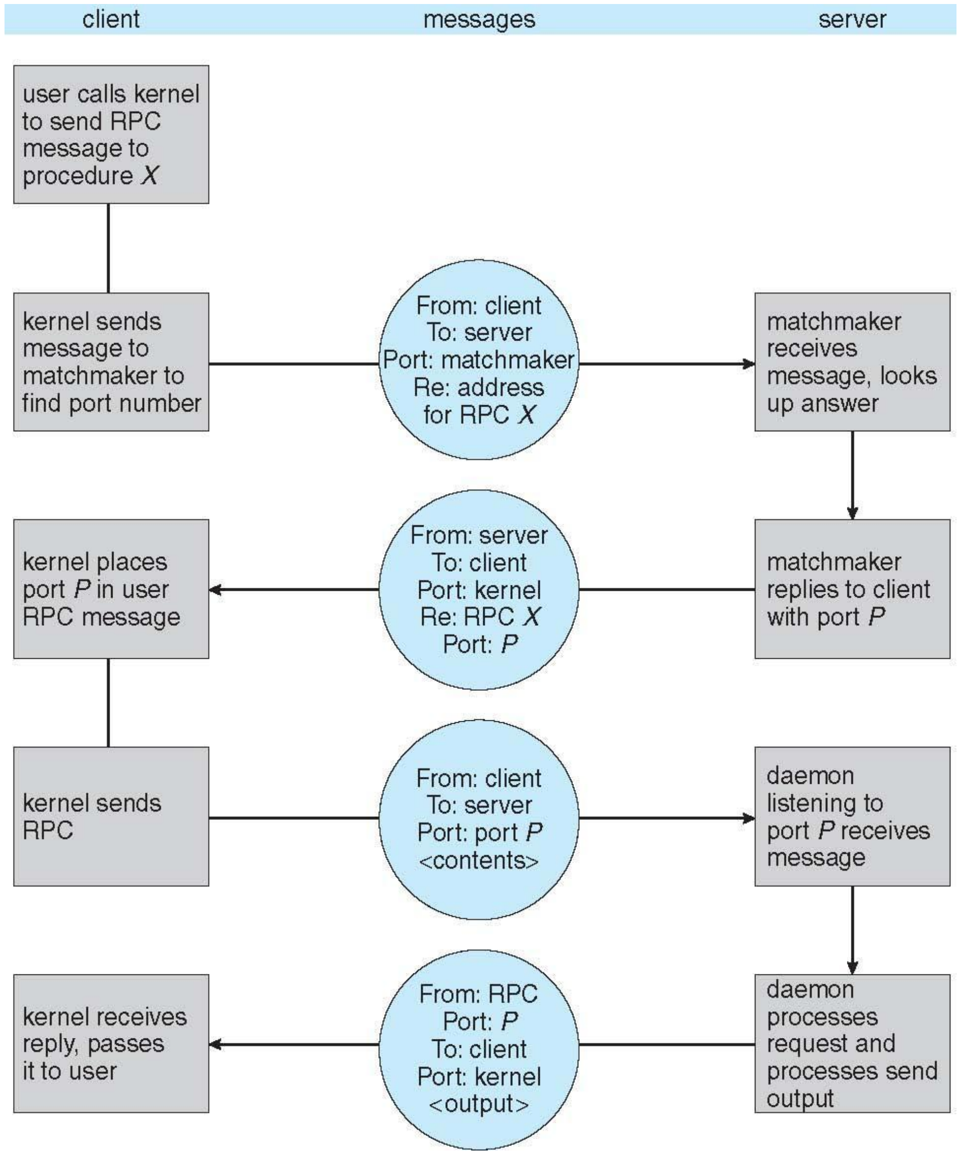
- A **socket** is defined as an endpoint for communication

Remote Procedure Calls



- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.

Execution of RPC


client
messages
server
user calls kernel to send RPC message to procedure *X*
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC *X*
matchmaker receives message, looks up answer
matchmaker replies to client with port *P*
From: server To: client Port: kernel Re: RPC *X* Port: *P*
kernel places port *P* in user RPC message
kernel sends RPC
From: client To: server Port: port *P* <contents>
daemon listening to port *P* receives message
daemon processes request and processes send output
From: RPC Port: *P* To: client Port: kernel <output>
kernel receives reply, passes it to user

End of Chapter 2