Processes

Pertemuan 5 dan 6

Kompetensi Khusus

- Mahasiswa mampu menjelaskan bagaimana proses SO dalam mengatur pengelolaan yang dilakukan oleh sistem komputer (C2, A2)(C1)

Materi

1. Process Concept
2. Process Scheduling
3. Operating on Processes
4. Interprocess Communication
5. IPC in Shared-Memory System
6. IPC in Message-Passing System
7. Communication in Client-Server System

1. Process Concept

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
- A process includes:
  - program counter
  - stack
  - data section

1.1 Process in Memory

A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time.

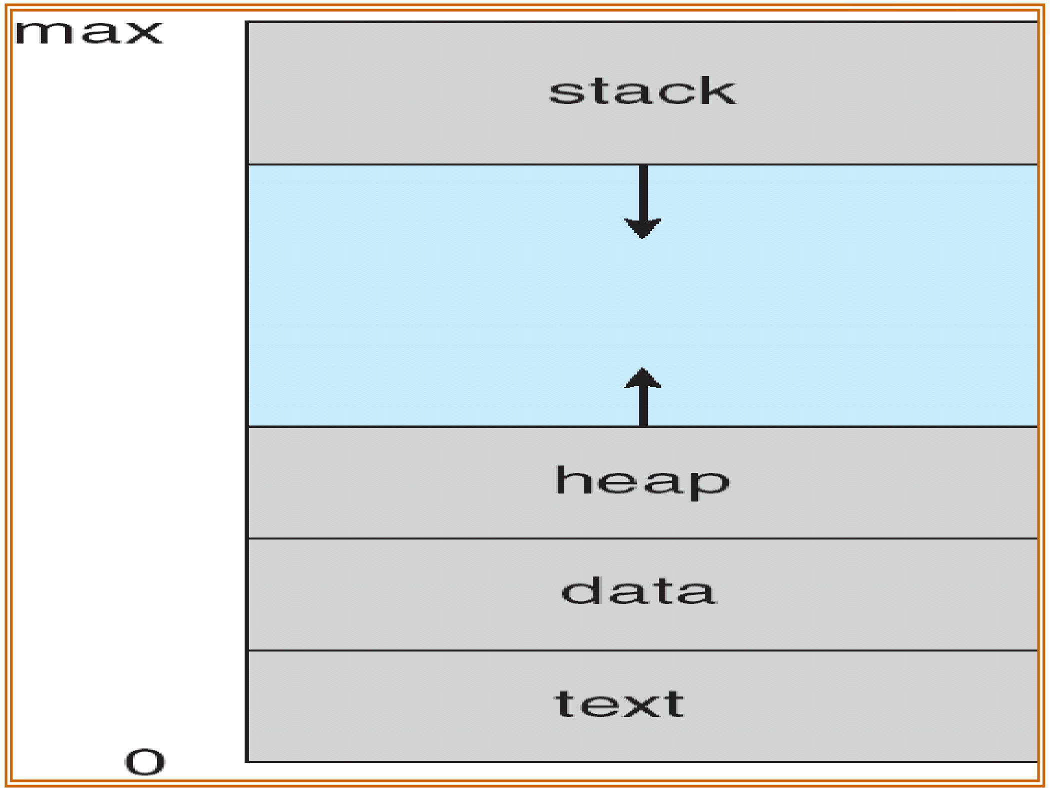


Fig. Process in memory.

1.2 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

1.3 Diagram of 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. The states that they represent are found on all systems, however.

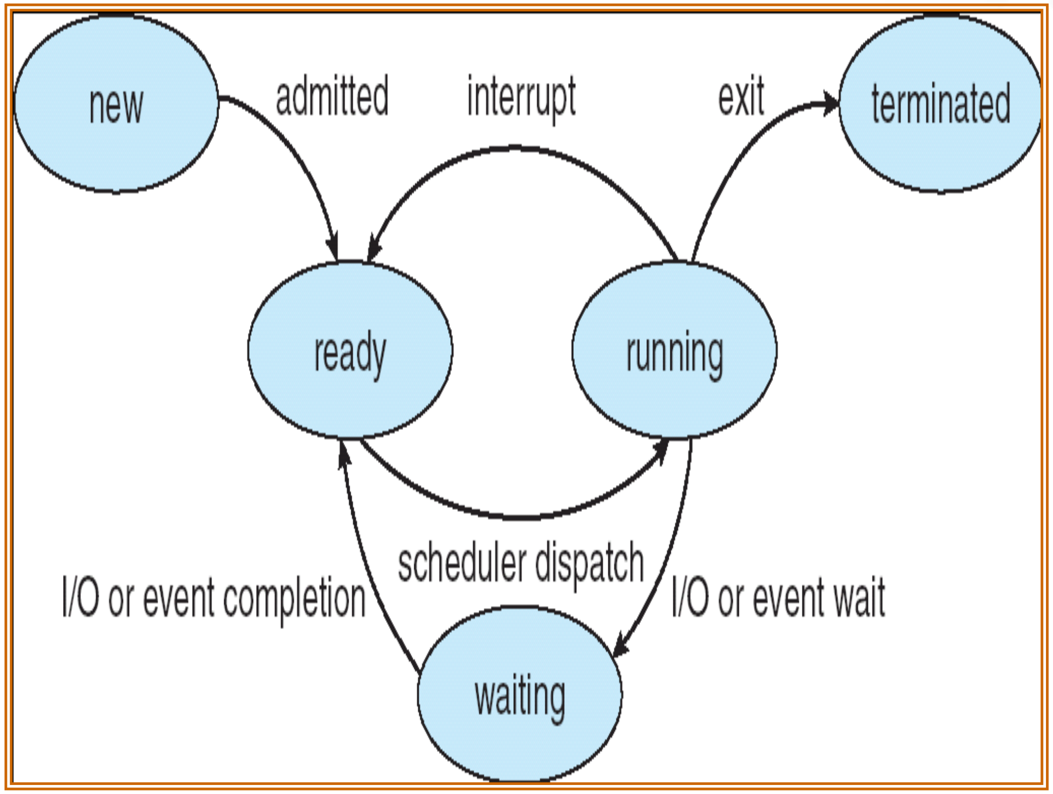


Fig. Diagram of process state

1.4 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

1.4 Process Control Block (PCB) (Cont.)

Each process is represented in the operating system by a process control block (PCB)-also called a task control block. It contains many pieces of information associated with a specific process.

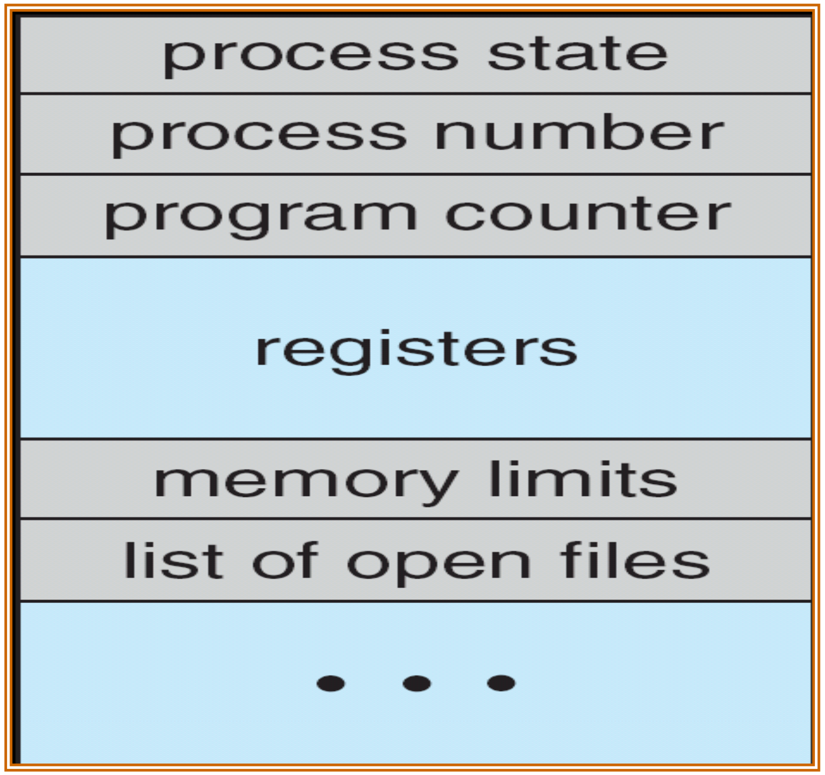


Fig. Process control block (PCB)

1.5 CPU Switch From Process to Process

The process model discussed so far has implied that a process is a program that performs a single thread of execution.

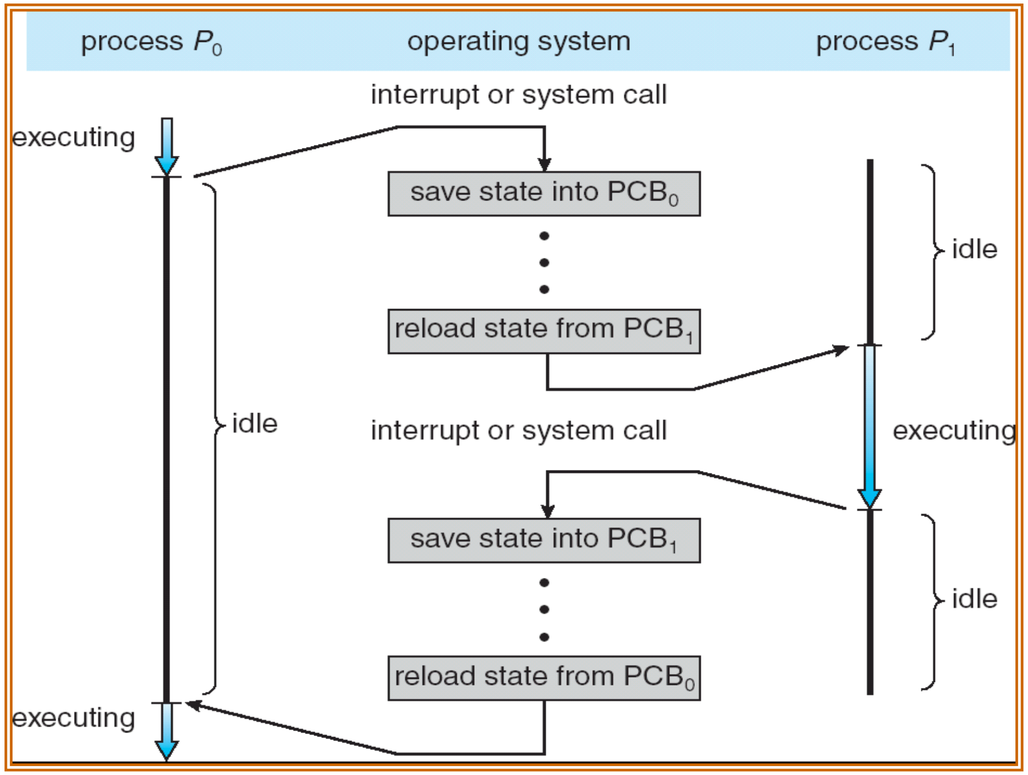


Fig. Diagram showing CPU switch from process to process

Universitas Bunda Mulia

Universitas Bunda Mulia

2. Process Scheduling

2.1 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
- Processes migrate among the various queues

2.2 Ready Queue And Various I/O Device Queues

Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

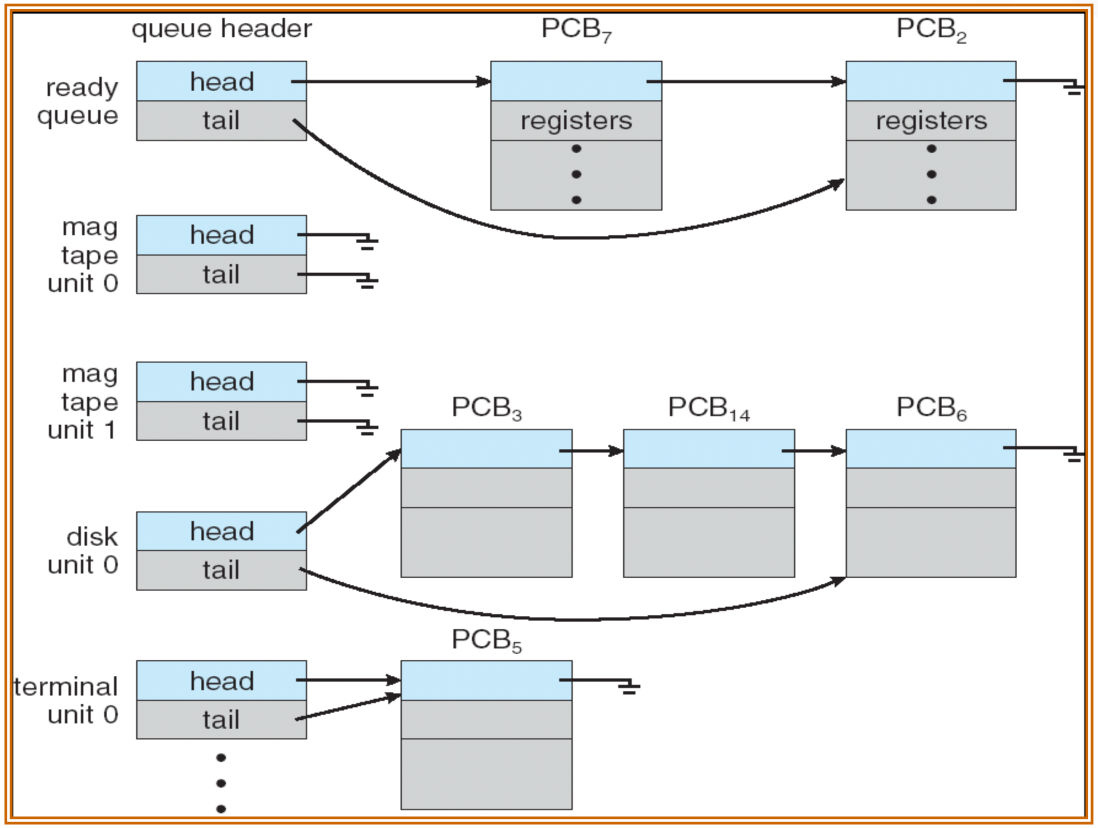


Fig. The ready queue and various I/O device queues.

2.3 Representation of Process Scheduling

Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

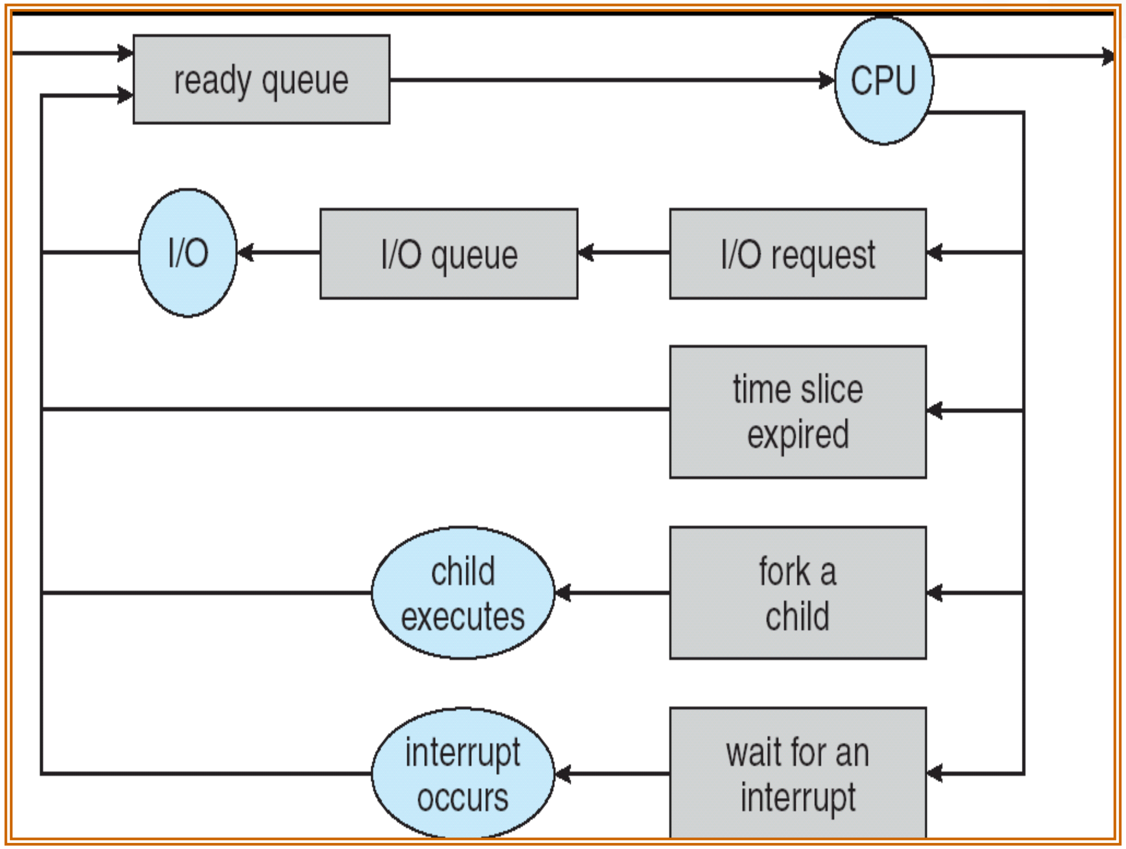


Fig. Queueing-diagram representation of process scheduling.

2.4 Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU

2.4.1 Addition of Medium Term Scheduling

The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.

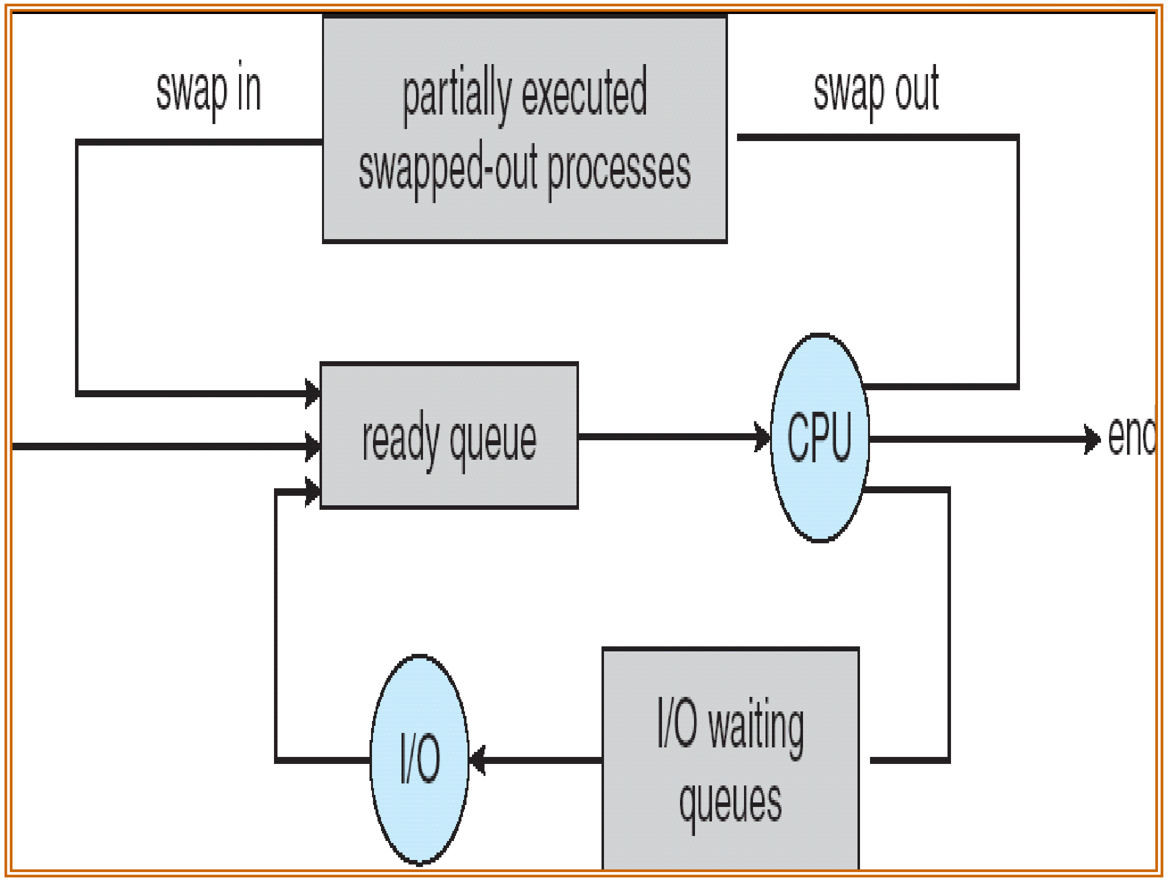


Fig. Addition of medium-term scheduling to the queueing diagram

2.4 Schedulers (Cont.)

- 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
- 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

2.5 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; the system does no useful work while switching
- Time dependent on hardware support

Universitas Bunda Mulia

Universitas Bunda Mulia

3. Operating on Processes

3.1 Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- 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

3.1 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

3.1 Process Creation (Cont.)

As an alternative example, we next consider process creation in Windows. Processes are created in the Win32 API using the CreateProcess () function, which is similar to fork () in that a parent creates a new child process.

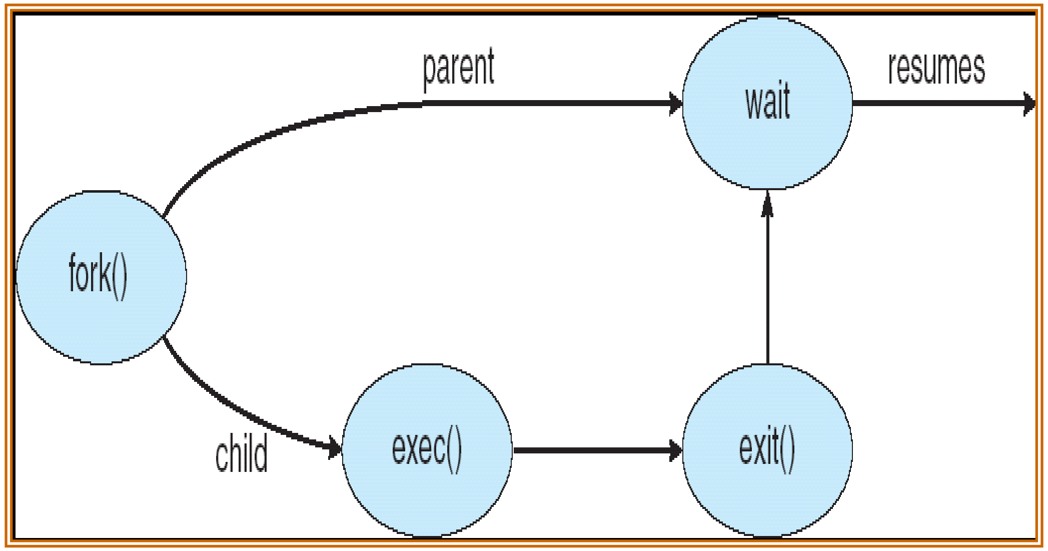


Fig. Process creation

3.2 C Program Forking Separate Process

```
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);
        }
}
```

3.4 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

4. Interprocess Communication

4.1 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

4.2 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

5. IPC in Shared-Memory System

5.1 Bounded-Buffer – Shared-Memory Solution

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

5.2 Bounded-Buffer – Insert() Method

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

Bounded Buffer – Remove() Method

```
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;
{
```

6. IPC in Message-Passing System

6.1 Interprocess Communication (IPC)

- 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 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)

6.2 Implementation Questions

- 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?

6.3 Communications Models

There are two fundamental models of interprocess communication: (1) shared memory and (2) message passing. In the shared-memory model, a region of memory that is shared by cooperating processes is established.

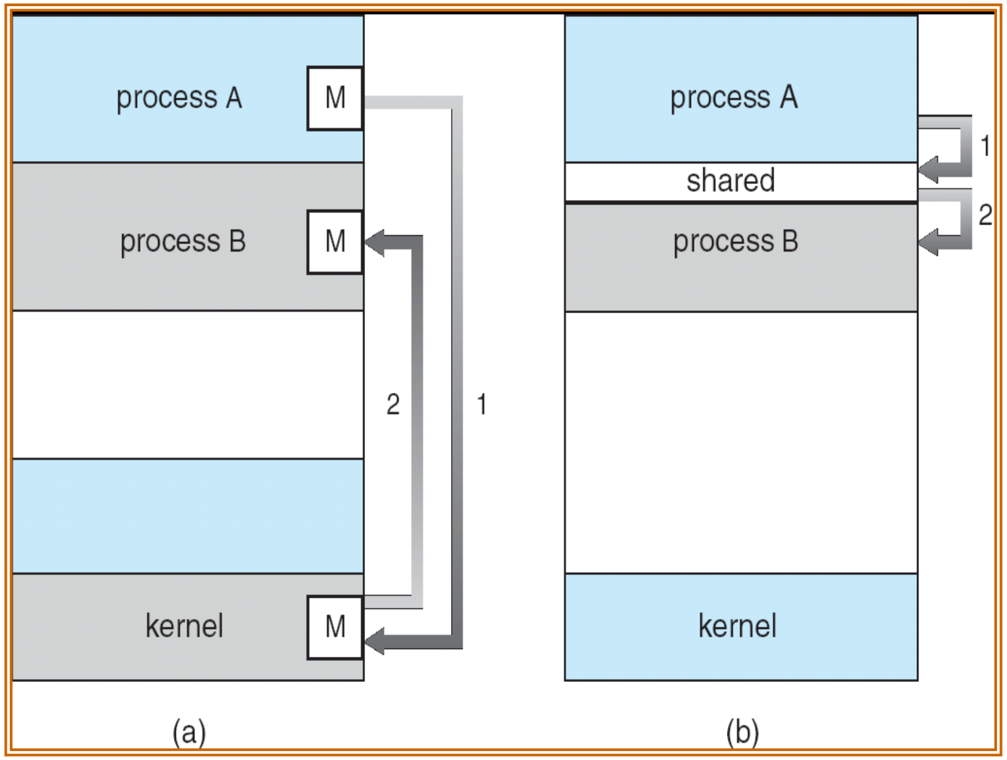


Fig. Communications models. (a) Message passing. (b) Shared memory.

6.4 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

6.5 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

6.5 Indirect Communication (Cont.)

- Operations
  - 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

6.5 Indirect Communication (Cont.)

- 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. Sender is notified who the receiver was.

6.6 Synchronization

- Message passing 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

6.7 Buffering

- Queue of messages attached to the link; implemented in one of three ways
  - 1. Zero capacity – 0 messages
    Sender must wait for receiver (rendezvous)
  - 2. Bounded capacity – finite length of n messages
    Sender must wait if link full
  - 3. Unbounded capacity – infinite length
    Sender never waits

7. Communication in Client-Server System

7.1 Client-Server Communication

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

7.2 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 consists between a pair of sockets

7.2.1 Socket Communication

For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server.

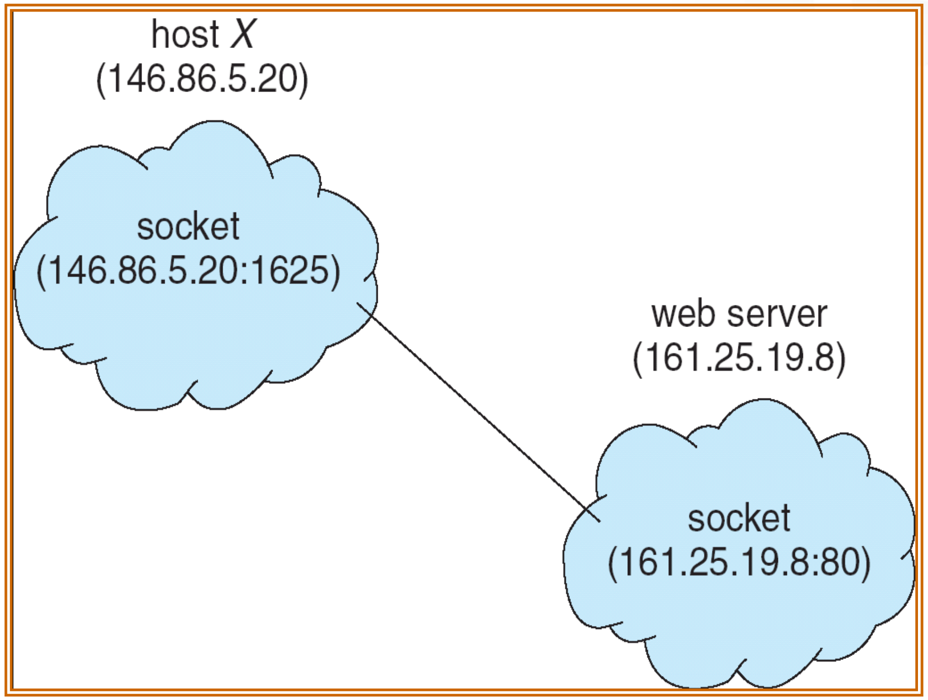


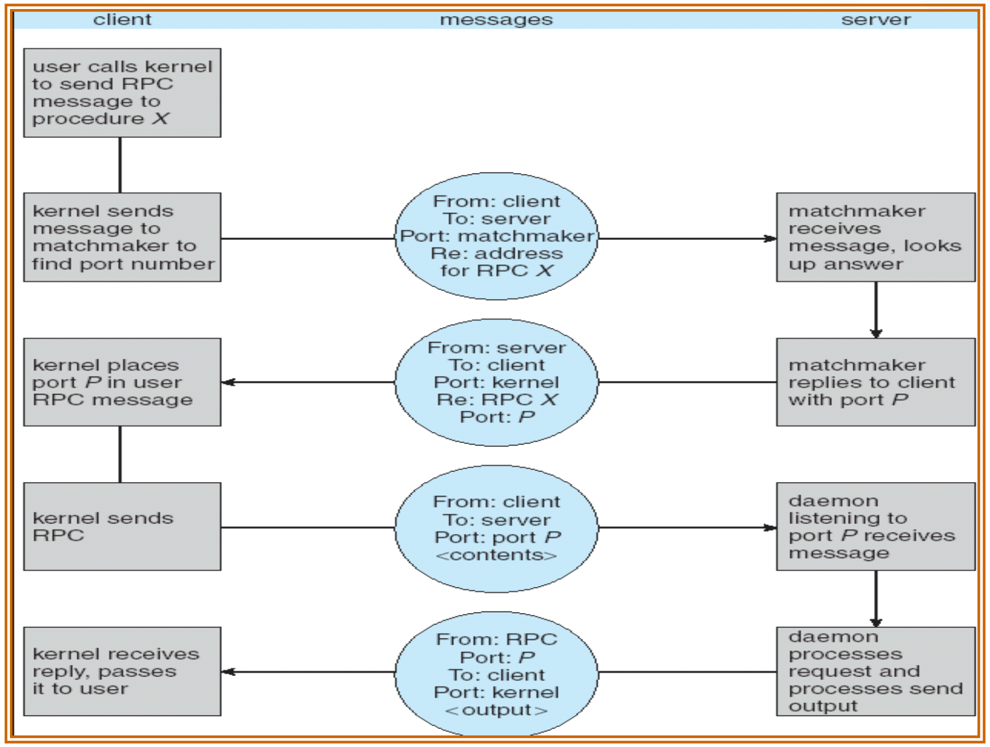
Fig. Communication using sockets.

7.3 Remote Procedure Calls

- Remote procedure call (RPC) abstracts 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.
- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server.

7.3.1 Execution of RPC

The port number is returned, and the RPC calls can be sent to that port until the process terminates (or the server crashes). This method requires the extra overhead of the initial request but is more flexible than the first approach.



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
kernel places port P in user RPC message
From: server To: client Port: kernel Re: RPC X Port: P
matchmaker replies to client with port P
kernel sends RPC
From: client To: server Port: port P <contents>
daemon listening to port P receives message
kernel receives reply, passes it to user
From: RPC Port: P To: client Port: kernel <output>
daemon processes request and processes send output

Fig. Execution of a remote procedure call (RPC).

7.4 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.

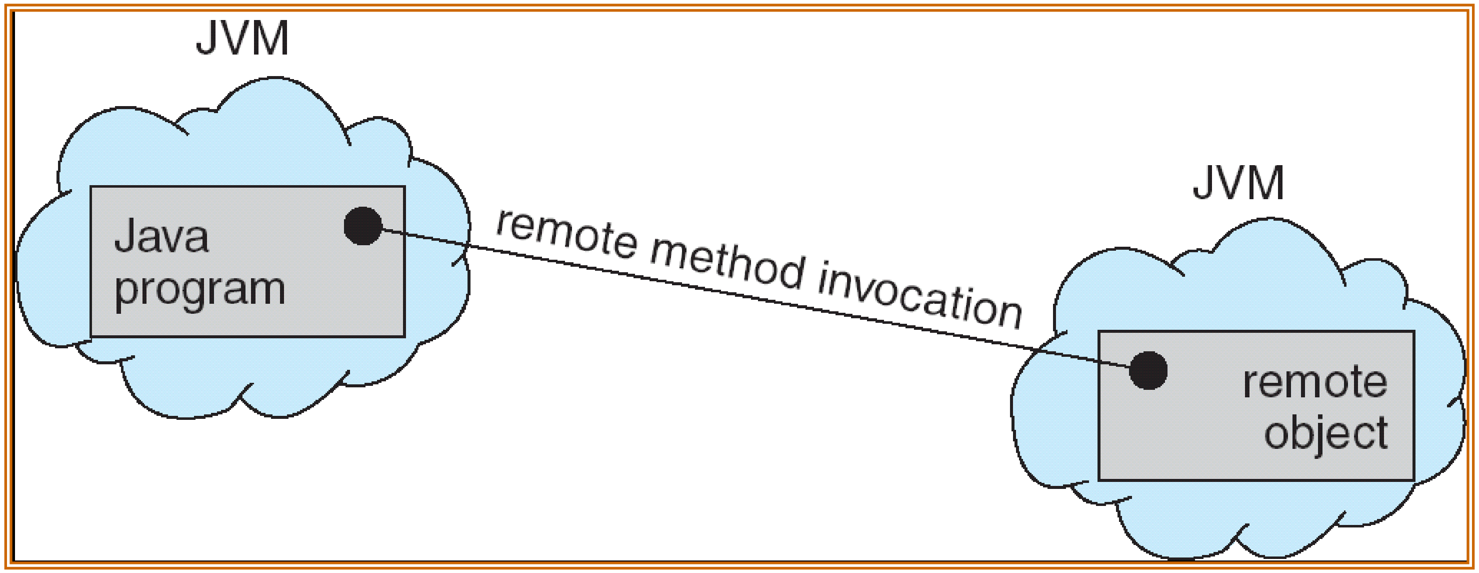


Fig. Remote method invocation

7.4.1 Marshalling Parameters

The actual implementation of someMethod () resides on the server. Once the method is completed, the skeleton marshals the boolean value returned from someMethod () and sends this value back to the client. The stub unmarshals this return value and passes it to the client.

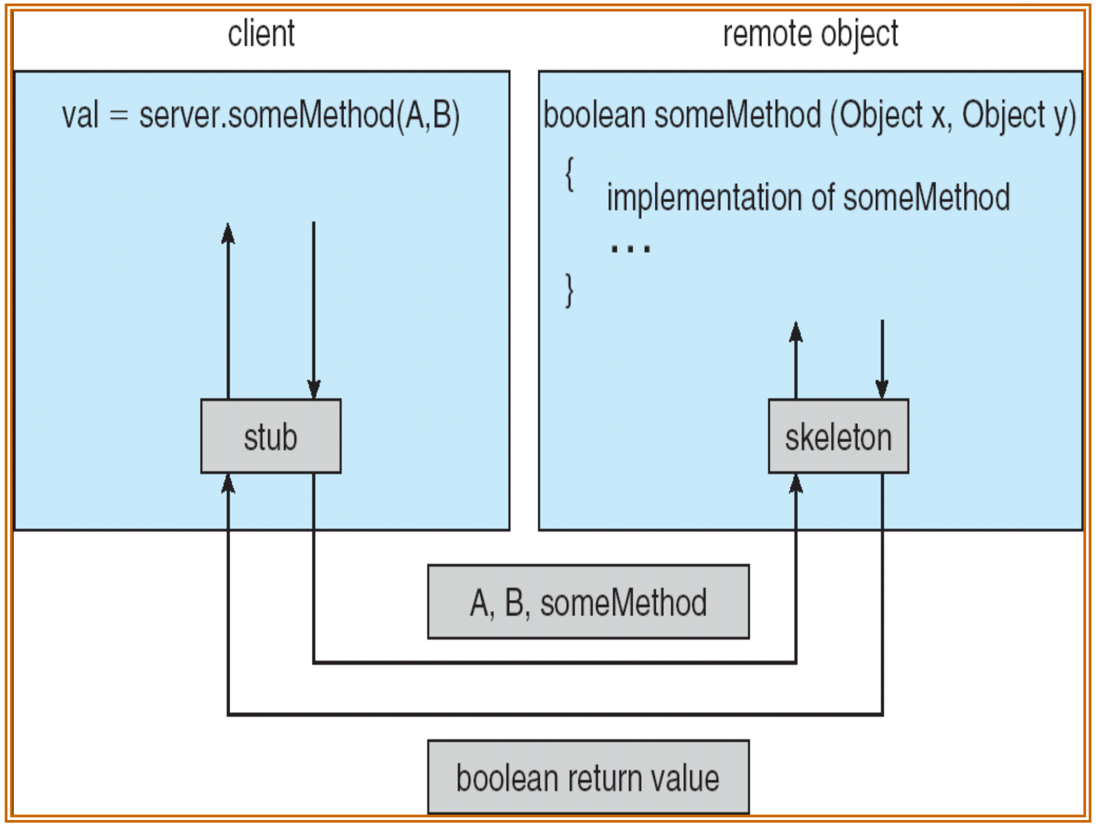

client
remote object
val = server.someMethod(A,B)
boolean someMethod (Object x, Object y)
{
implementation of someMethod
...
}
stub
skeleton
A, B, someMethod
boolean return value

Fig. Marshalling parameters

Summary

- Process State consist of new, running, waiting, ready, and terminated
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
- Client-Server Communication consist of Sockets, Remote Procedure Calls, and Remote Method Invocation (Java)
- A socket is defined as an endpoint for communication

Thank You