

Chapter 3: Processes-Concept





Chapter 3: Processes-Concept

- Overview
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems



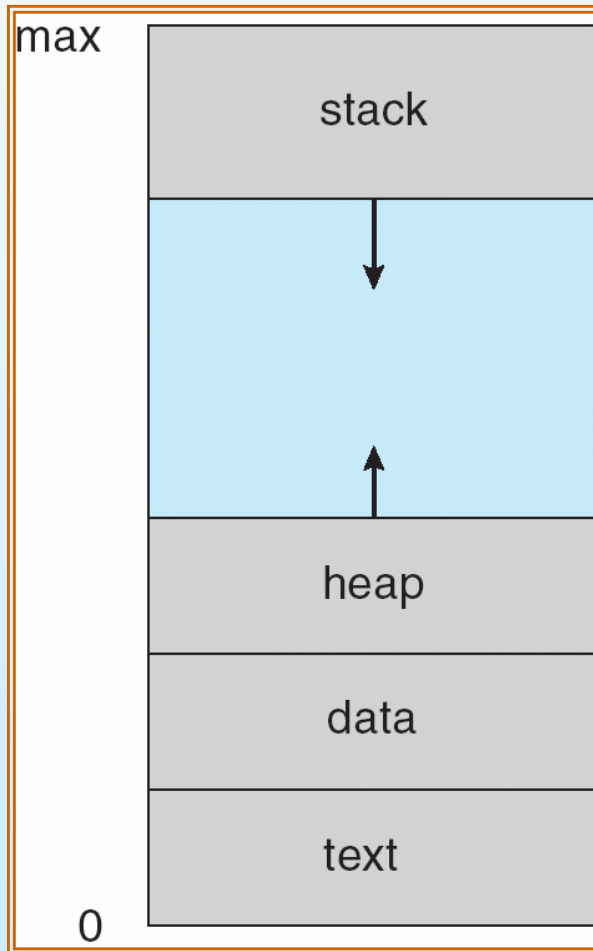


Process Concept

- 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



Process in Memory

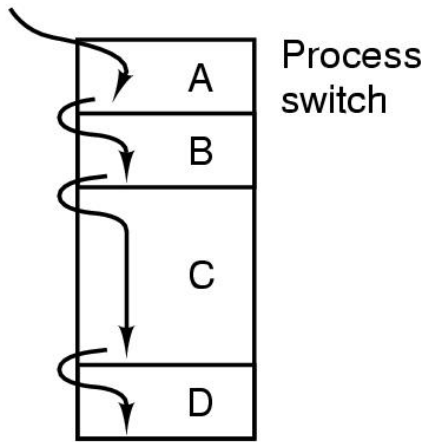


Processes

The Process Model

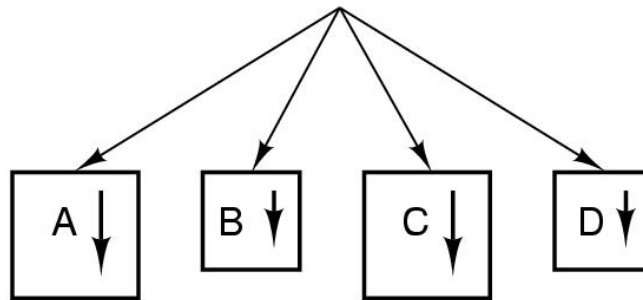


One program counter

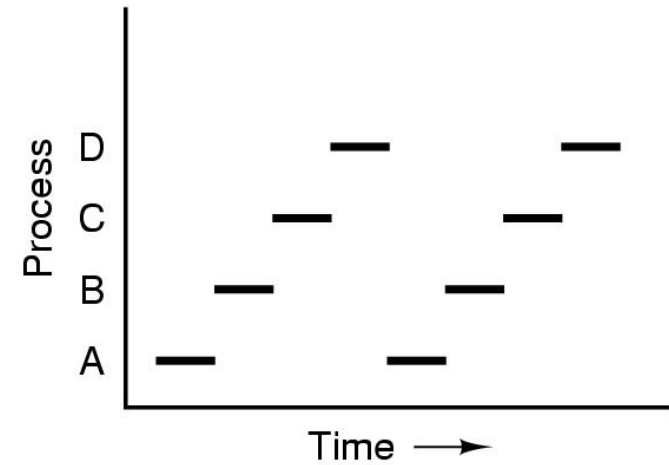


(a)

Four program counters



(b)



(c)

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant



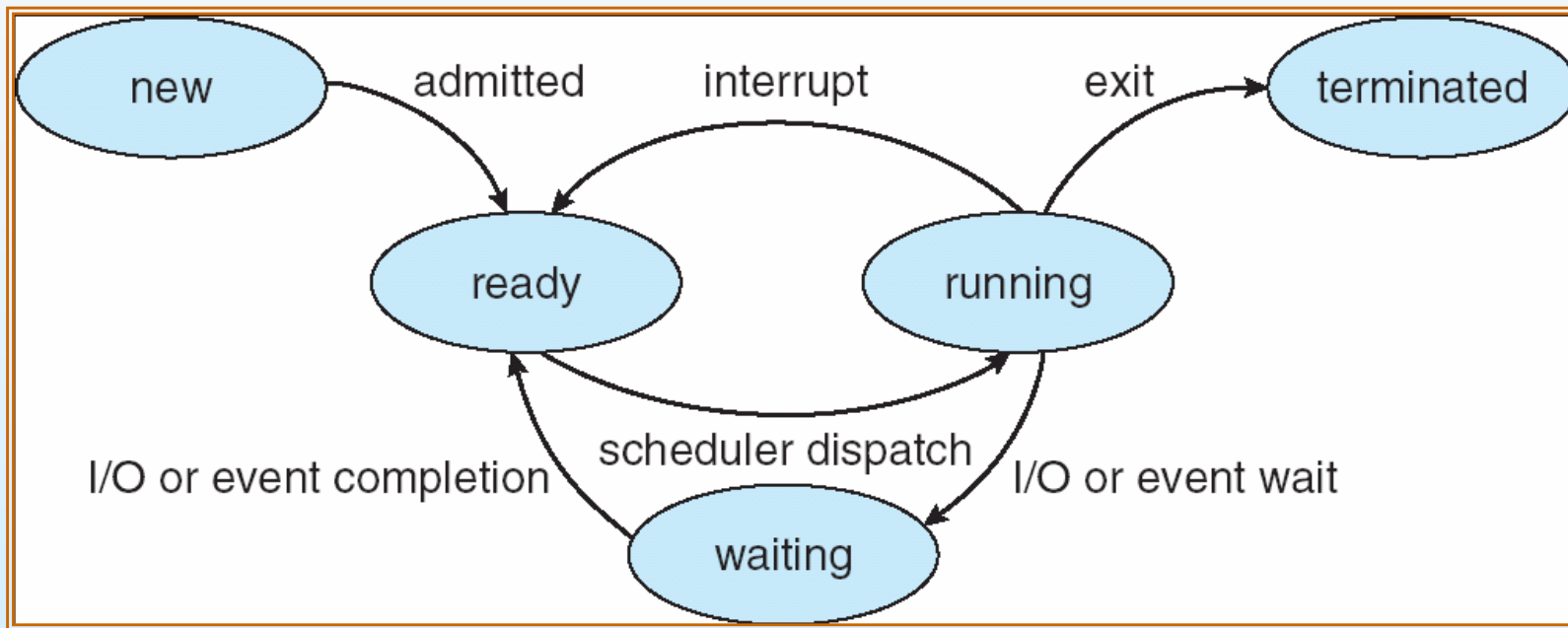


Process State

- 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 processor
 - **terminated**: The process has finished execution



Diagram of Process State





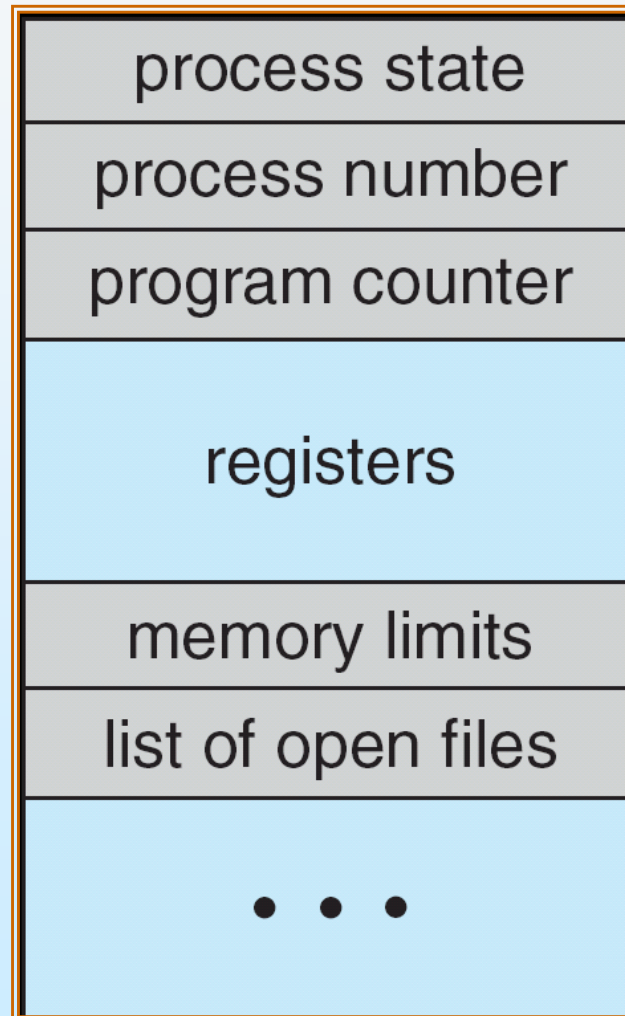
Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



Process Control Block (PCB)





Implementation of Processes

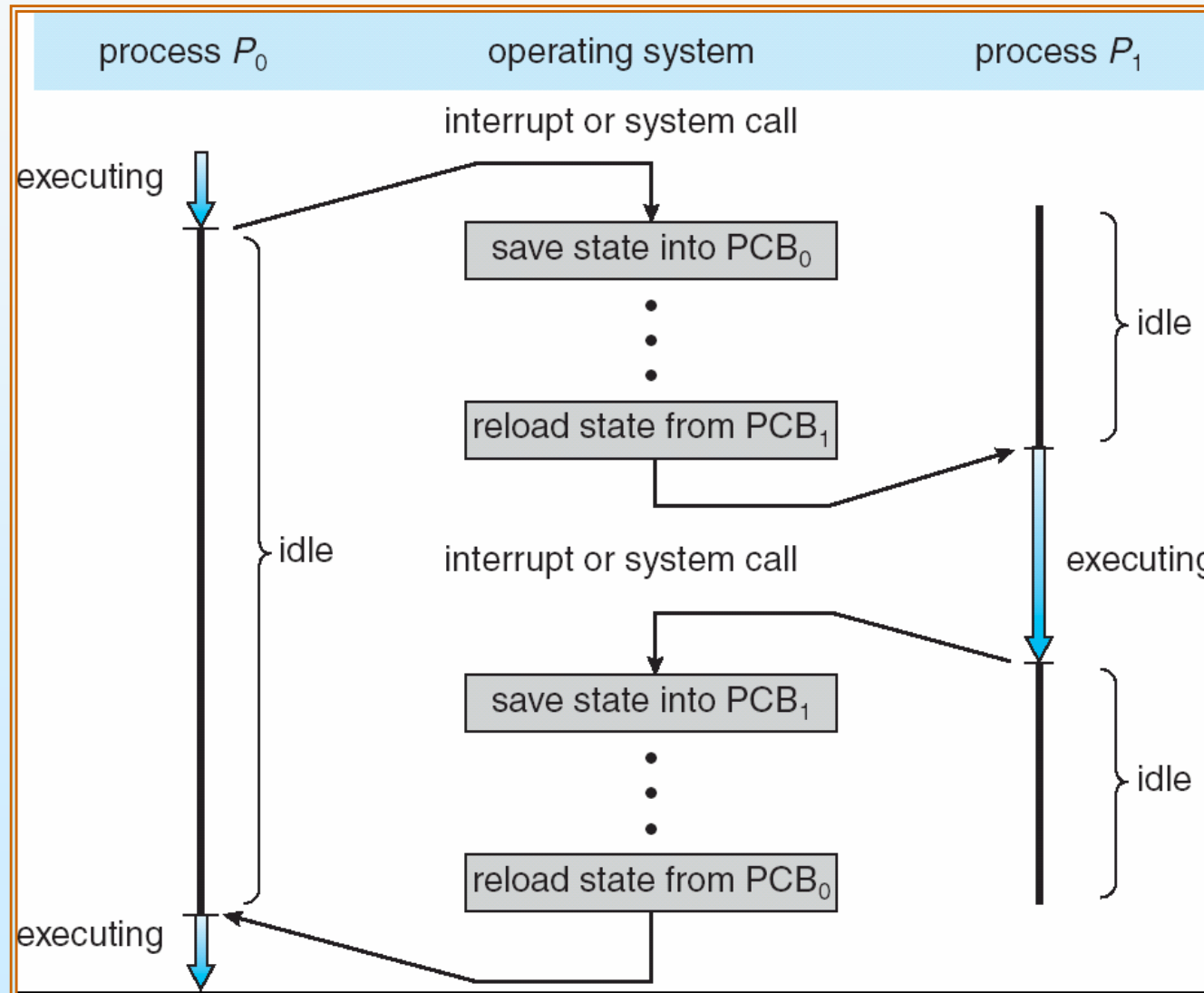
Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
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





CPU Switch From Process to Process





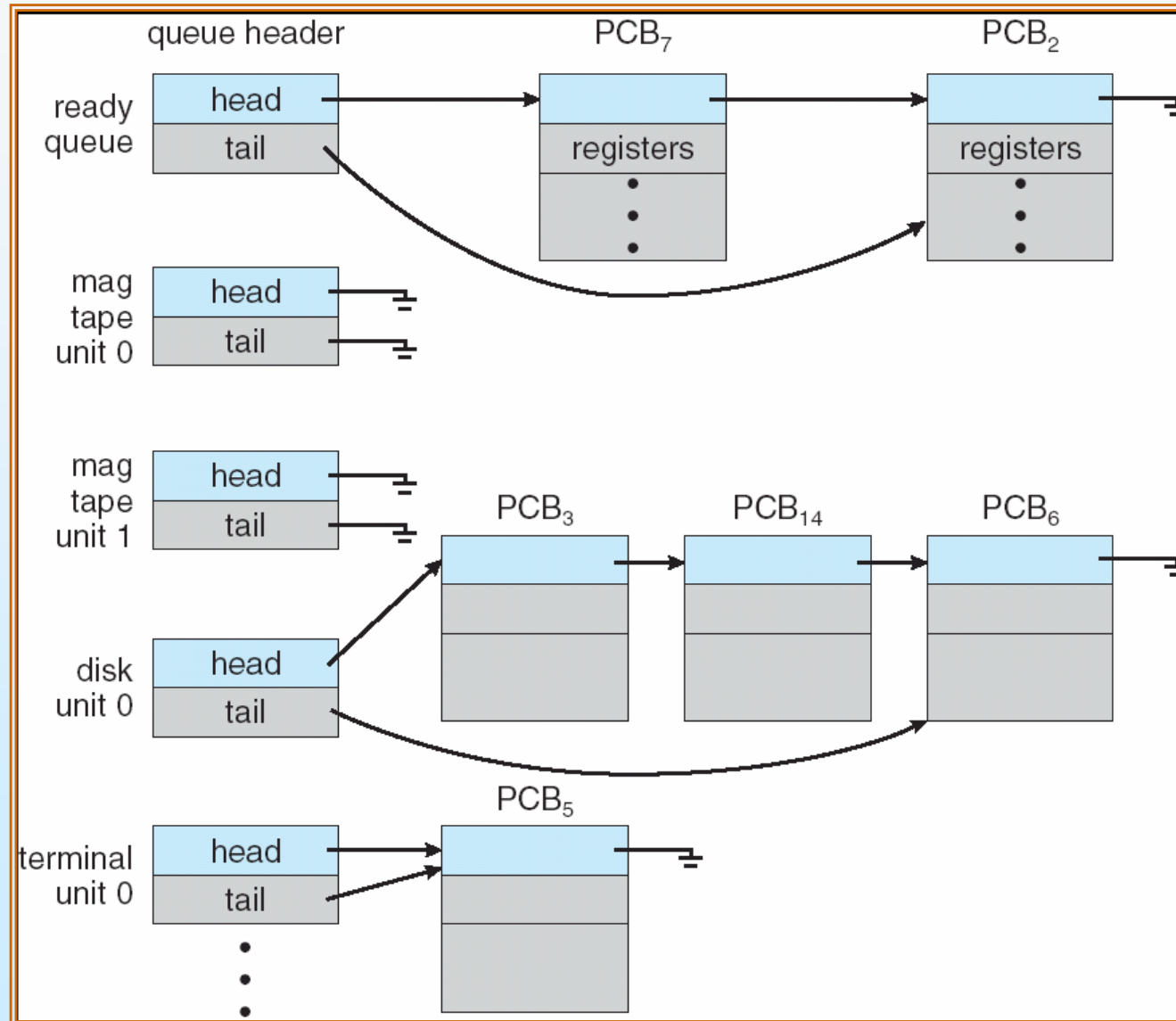
Process Scheduling Queues

- **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
- Processes migrate among the various queues



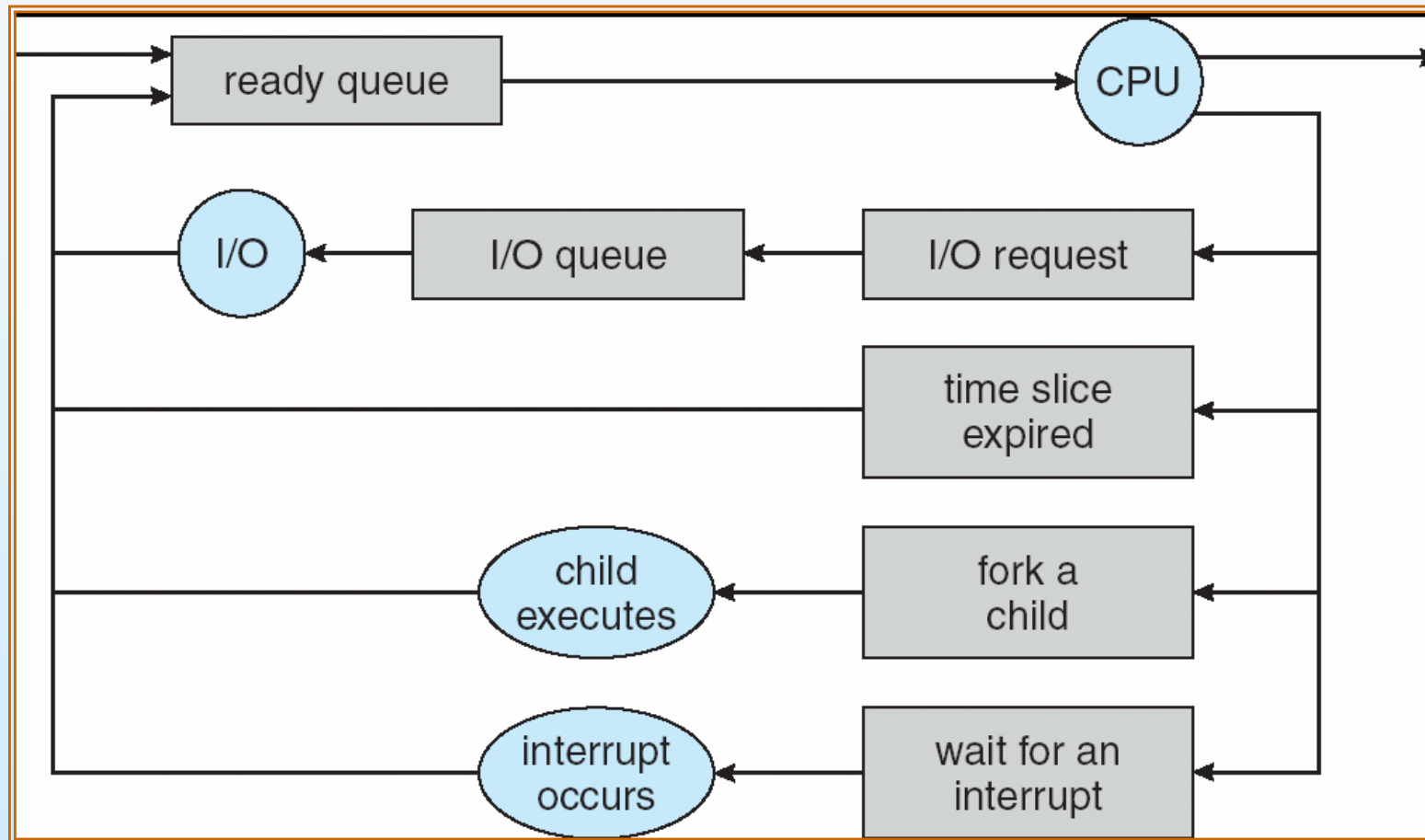


Ready Queue And Various I/O Device Queues

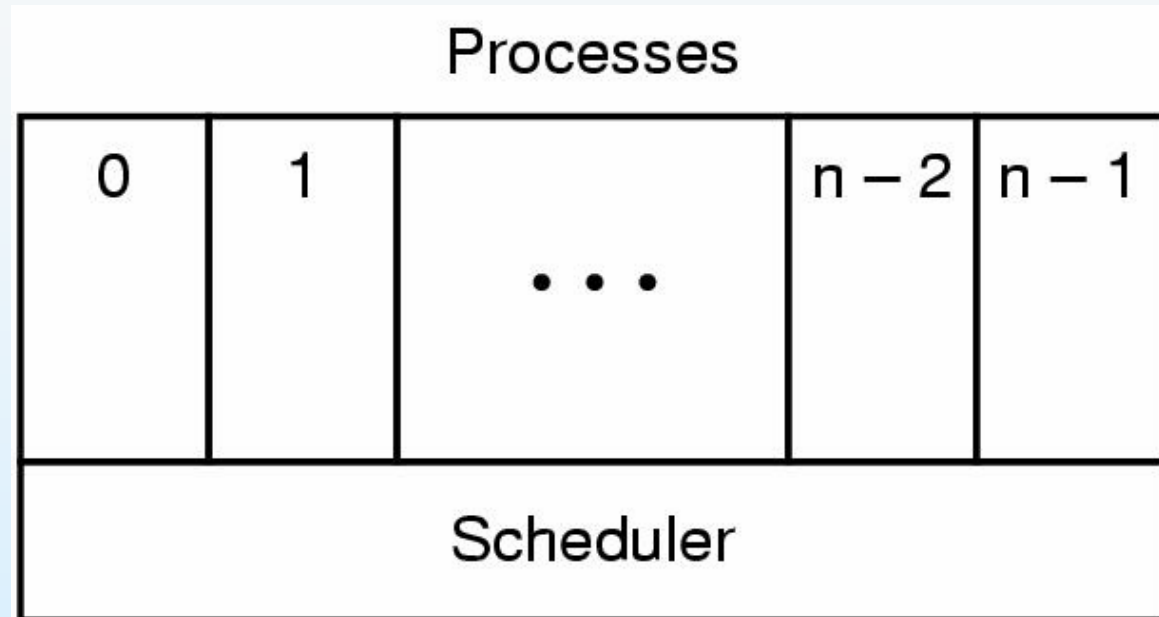




Representation of Process Scheduling



Process and Schedulers



- Lowest layer of process-structured OS
 - handles interrupts, scheduling
- Above that layer are sequential processes





Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought from a mass-storage device into memory for execution
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU





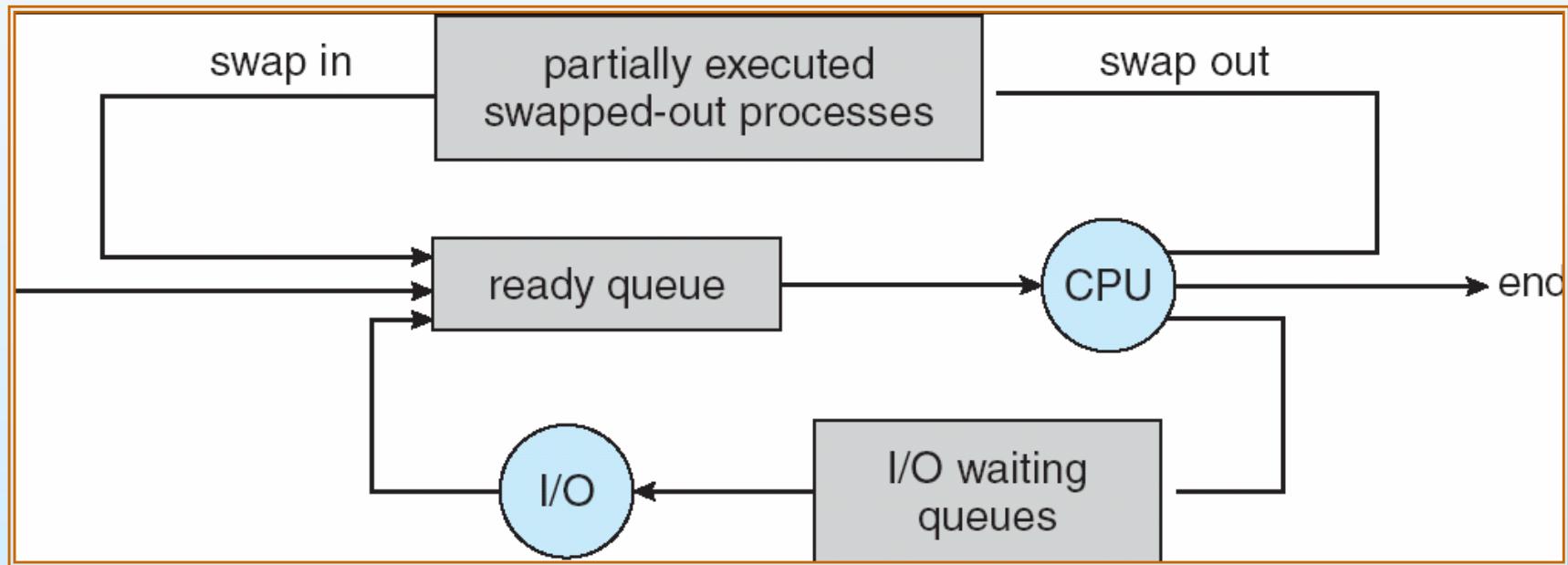
Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) \Rightarrow (must be fast)
- 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





Addition of Medium Term Scheduling





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





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



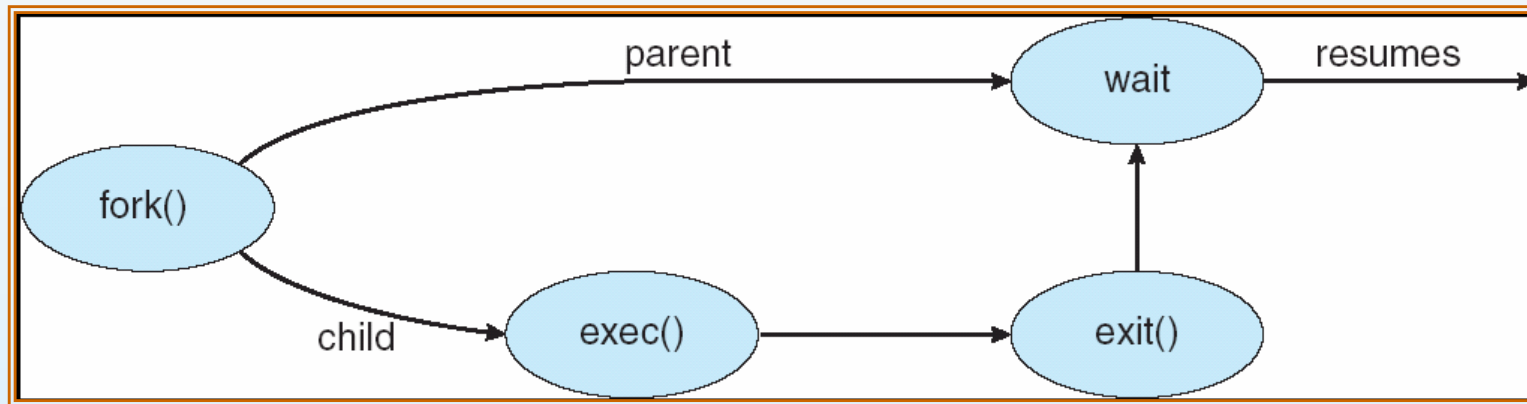


Process Creation (Cont.)

- 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 Creation





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);
    }
}
```





fork()

- The fork() system call will spawn a new child process which is an identical process to the parent except that has a new system process ID.
 - The process is copied in memory from the parent and a new process structure is assigned by the kernel.
 - The return value of the function is which discriminates the two threads of execution.
 - A zero is returned by the fork function in the child's process.
- The environment, resource limits, umask, controlling terminal, current working directory, root directory, signal masks and other process resources are also duplicated from the parent in the forked child process.





```
main() {  
    int return_value;  
    printf("Forking process \n");  
    return_value = fork();  
    printf("The process id is %d and return value is %d \n",  
          getpid(), return_value);  
    execl("/bin/ls/", "ls", "-l", 0);  
    printf("This line is not printed \n");  
}
```

- The Output of this would be:

Forking process

The process id is 6753 and return value is 6754

The process id is 6754 and return value is 0

two lists of files in current directory





wait()

- The parent process will often want to wait until all child processes have been completed. this can be implemented with the wait() function call.
 - wait() return the pid of the child or -1 for an error.
 - The exit status of the child is returned to status_location.

```
int wait (int *status_location)
```

- wait(): Blocks calling process until the child process terminates.
 - If child process has already terminated, the wait() call returns immediately.
 - If the calling process has multiple child processes, the function returns when one returns.





execl(), execlp() – exec() family

- The function call "execl()" initiates a new program in the same environment in which it is operating. An executable (with fully qualified path. i.e. /bin/ls) and arguments are passed to the function.

```
int execl(const char *path, const char *arg1, const char *arg2, ... const char *argn, (char *) 0);
```

```
main() {  
    printf("Files in Directory are: \n");  
    execl("/bin/ls", "ls", "-l", 0);  
}
```

- The routine execlp() will perform the same purpose except that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used.





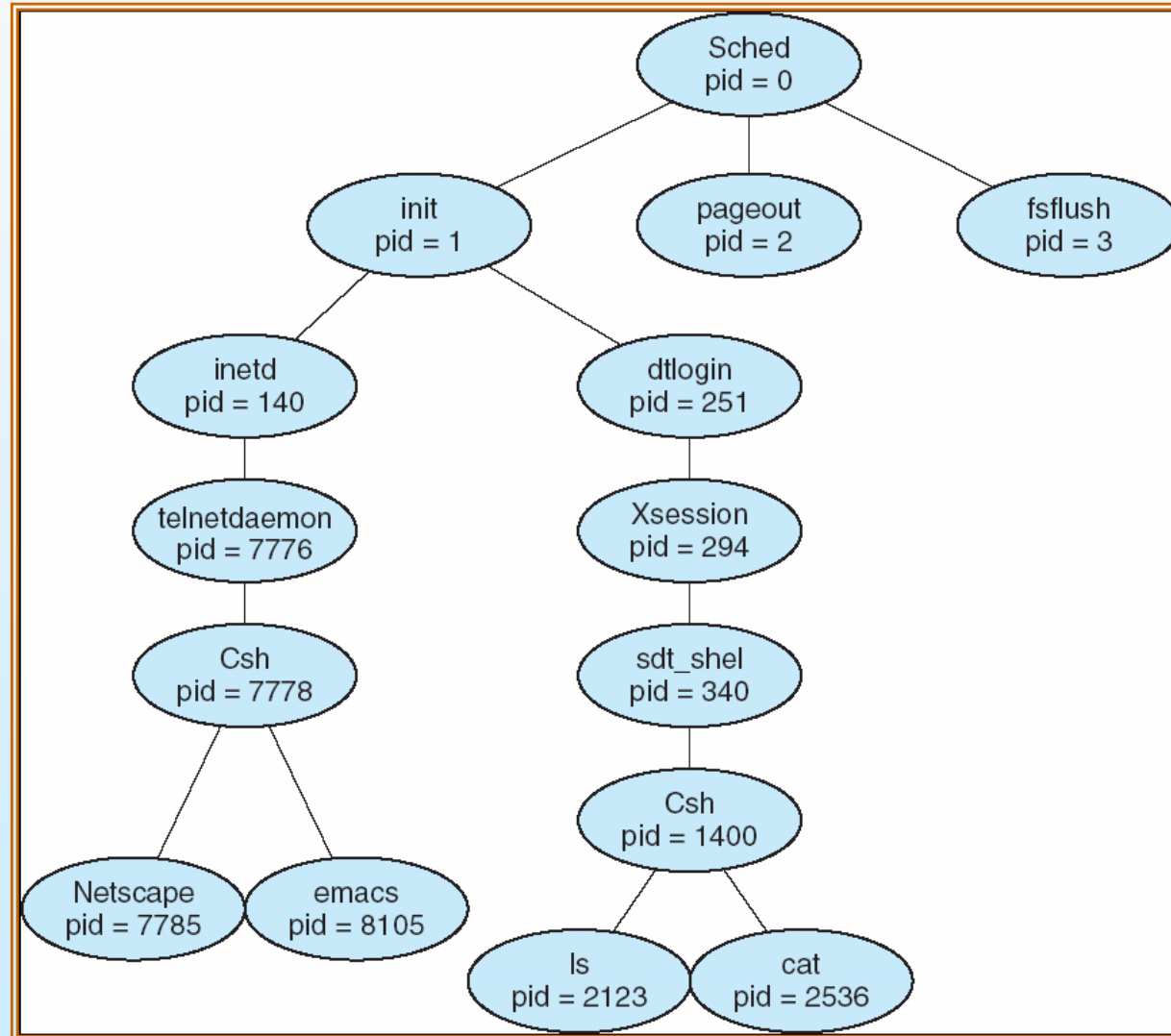
Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
 - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
 - all processes are created equal





A tree of processes on a typical Solaris





Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
 - Output status value from child to parent (via **wait**)
 - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**) for various reasons such as
 - 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*





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





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 – 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;  
}
```





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)





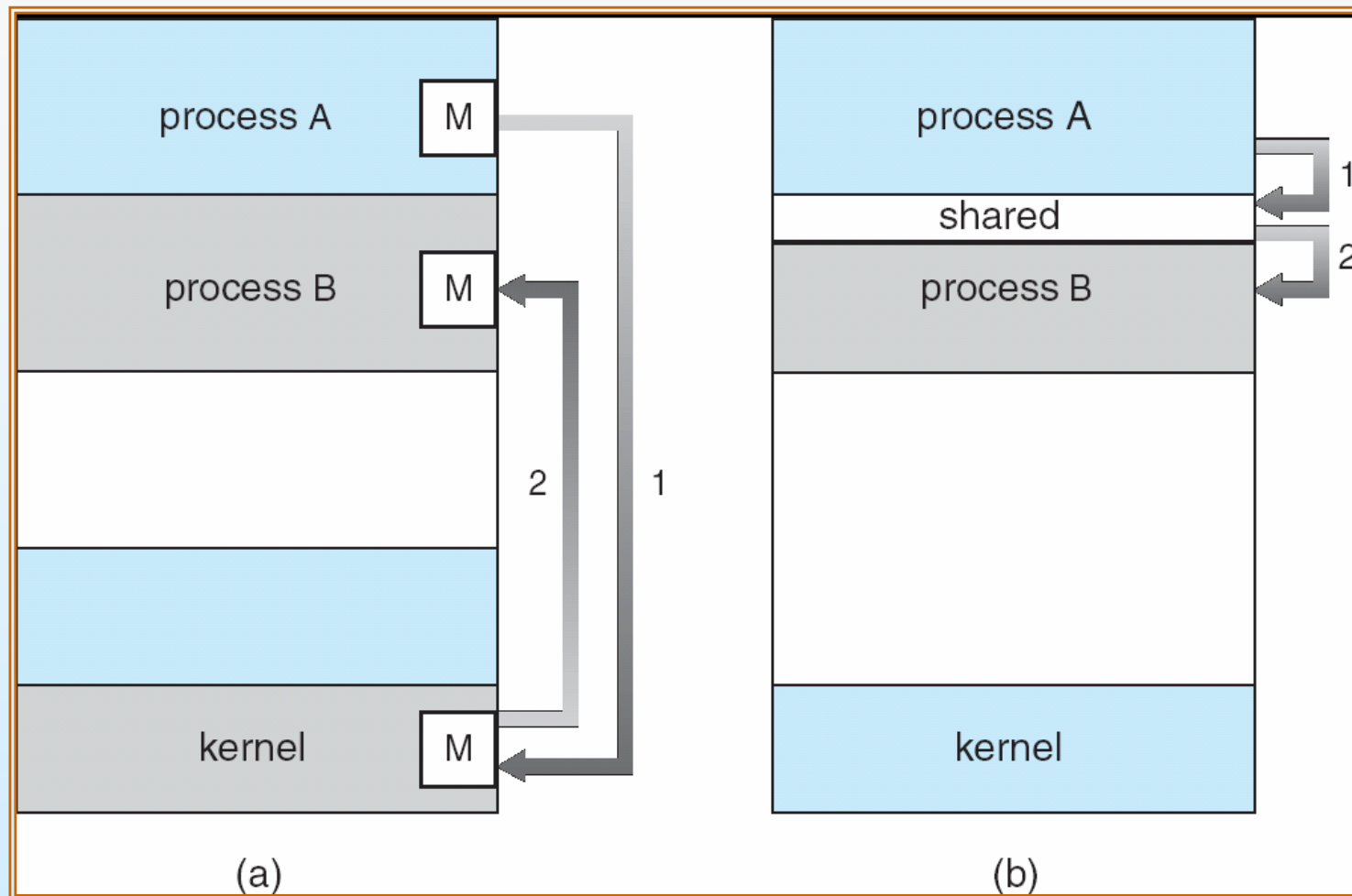
Implementation Questions

- 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?





Communications Models





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





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





Indirect Communication

- 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*





Indirect Communication

- Mailbox sharing
 - P_1 , P_2 , and P_3 share mailbox A
 - P_1 sends; P_2 and P_3 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.





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





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





Client-Server Communication

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

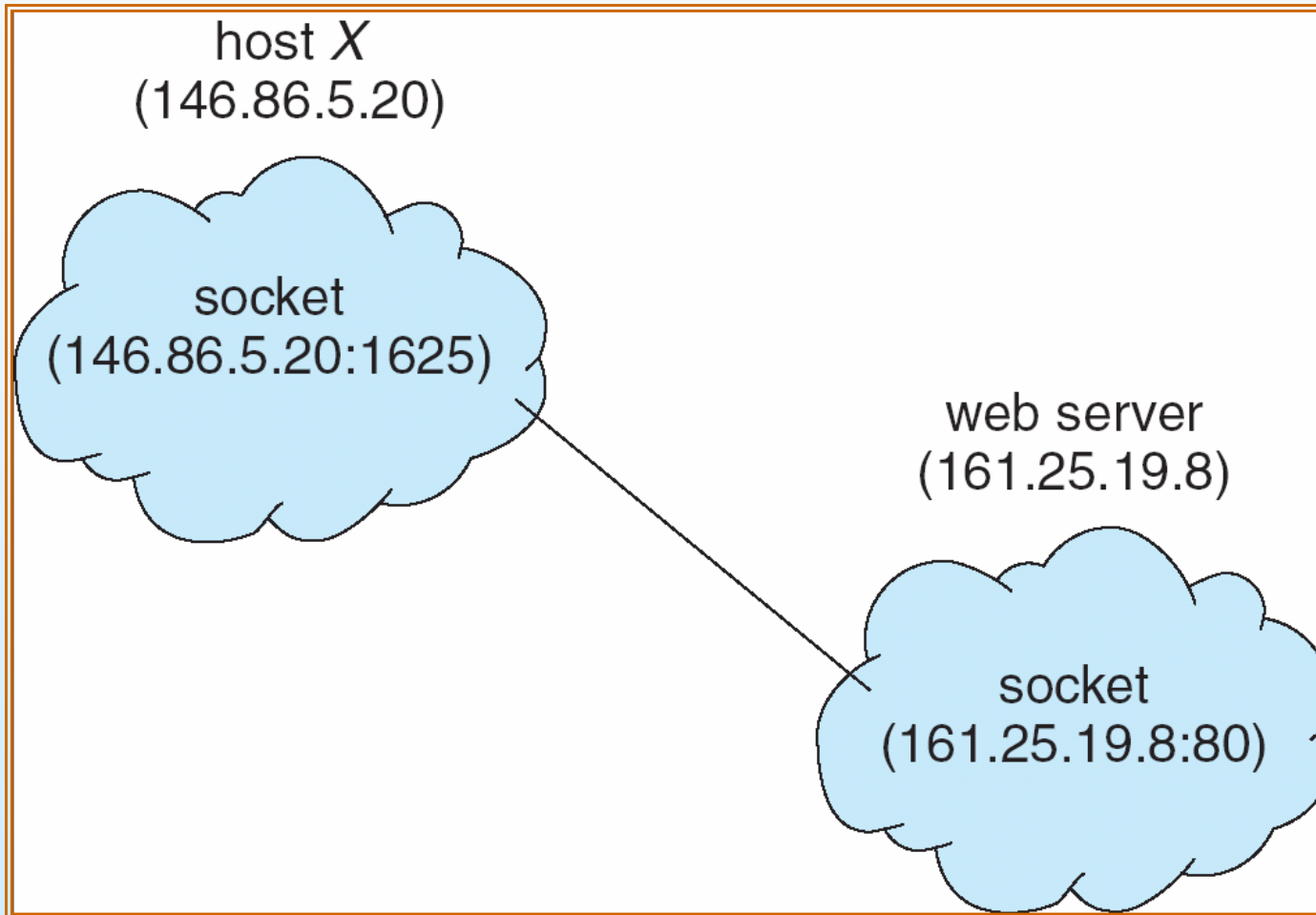


Sockets

- 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 consists between a pair of sockets



Socket Communication





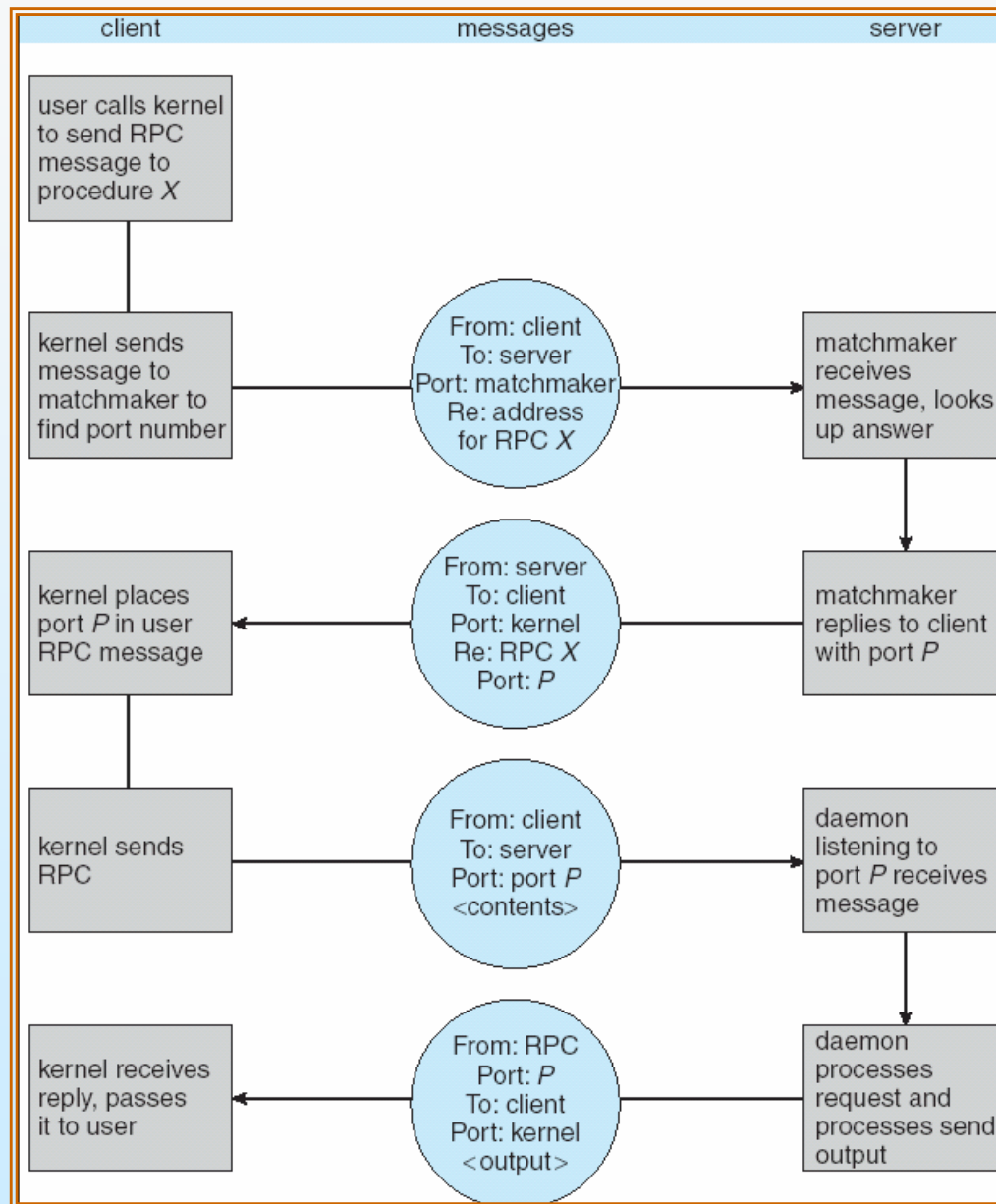
Remote Procedure Calls

- 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 performs the procedure on the server.





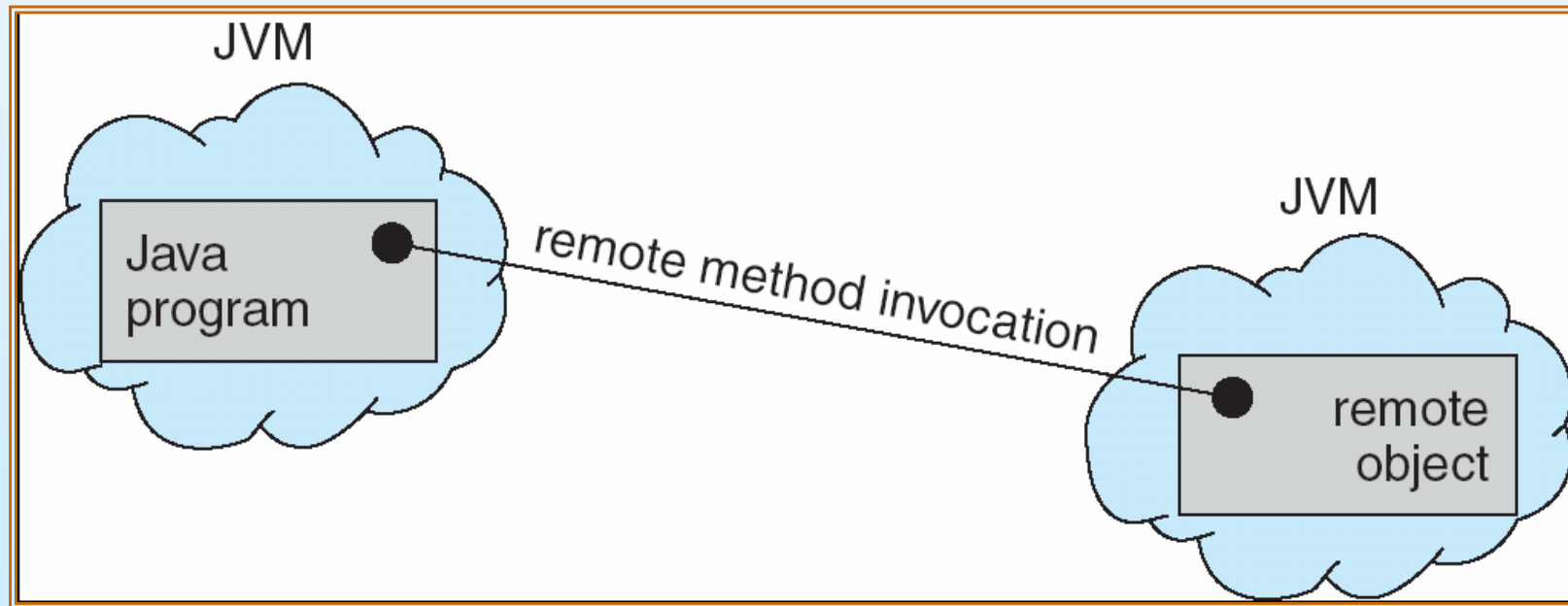
Execution of RPC





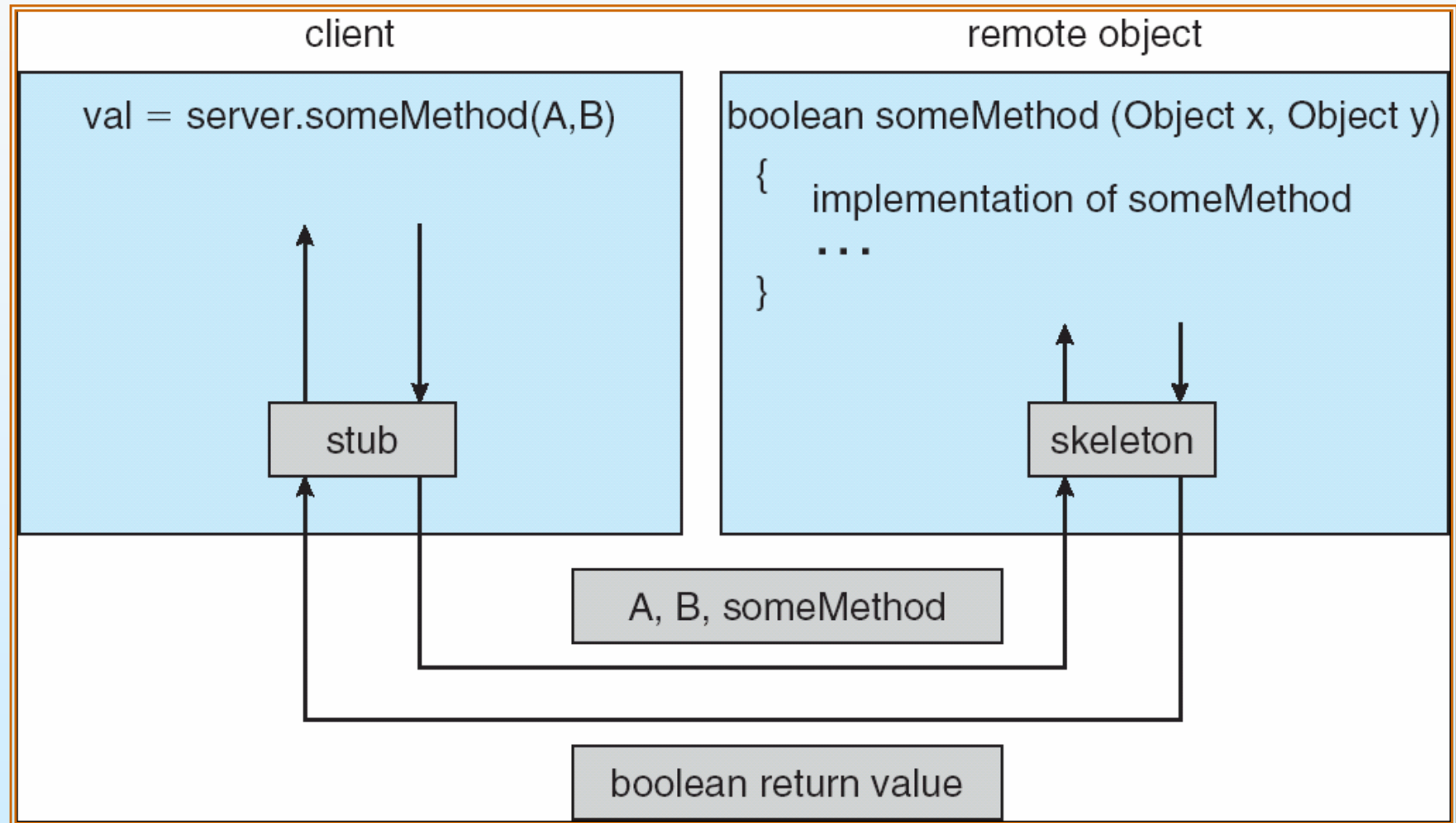
Remote Method Invocation

- 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 of Chapter 3

