操作系统原理及应用

李　伟

xchlw@seu.edu.cn

计算机科学与工程学院、软件学院
江苏省网络与信息安全重点实验室

Chapter 3   Processes

Outline

- **Process Concept**

- **Process Scheduling**

- **Operations on Processes**

- **Interprocess Communication**

- **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 Concept

- **Process – a program in execution; process execution must progress in sequential fashion.**

- **A process includes**
  - **text section (program code)**
  - **program counter** ⟶ **current activity**
  - **contents of the processor's registers**
  - **Heap-stack — temporary data (function parameters, return address, local variables)**
  - **data section — global variables**

Process in Memory

max
stack
↓
heap ↑
heap
data
text
0
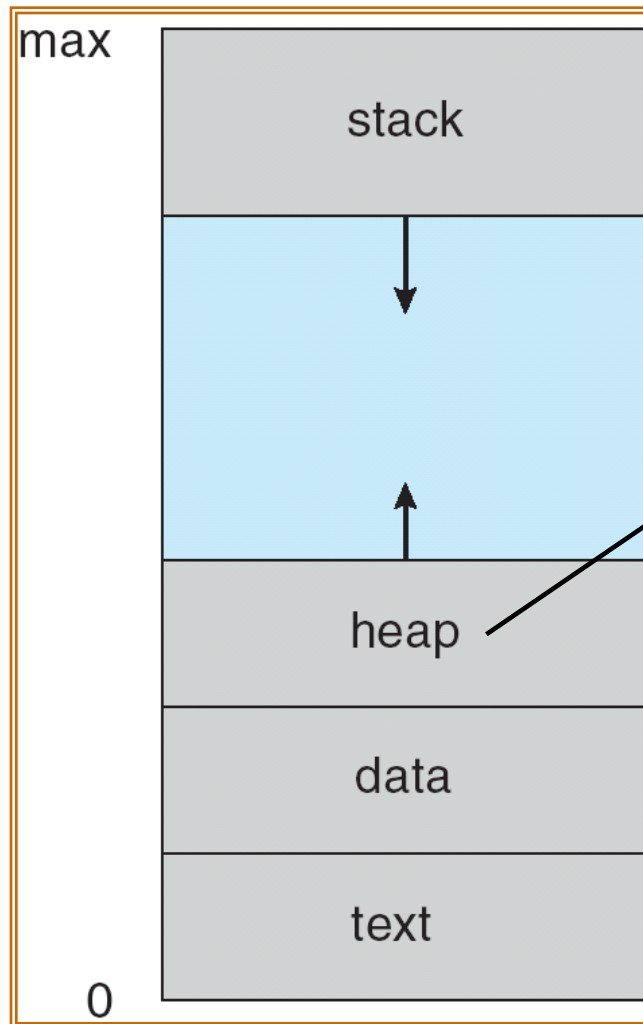
Heap (堆) is memory that is dynamically allocated during process run time.

Process in Memory

```
//main.cpp
int a = 0;        ←——— 数据段
char *p1;         ←——— 数据段
main()
{   int b;        ←——— 栈
    char s[] = "abc";      ←——— 栈
    char *p2;     ←——— 栈
    char *p3 = "123456";   ←——— 栈
    p1 = (char *)malloc(10);      ←——— 堆
    p2 = (char *)malloc(20);      ←——— 堆
}
```

Characteristic of Process

- **Dynamic (动态性)**

- **Independency (独立性)**

- **Concurrence (并发性)**

- **Structure (结构化)**

作业1

进程和程序是两个密切相关的概念，请阐述它们之间的区别和联系。

Process State

- **As a process executes, it changes *state***
  - **New**: The process is being created.
  - **Ready**: The process is waiting to be assigned to a processor.
  - **Running**: Instructions are being executed.
  - **Waiting**: The process is waiting for some event to occur.
  - **Terminated**: The process has finished execution.

Diagram of Process State
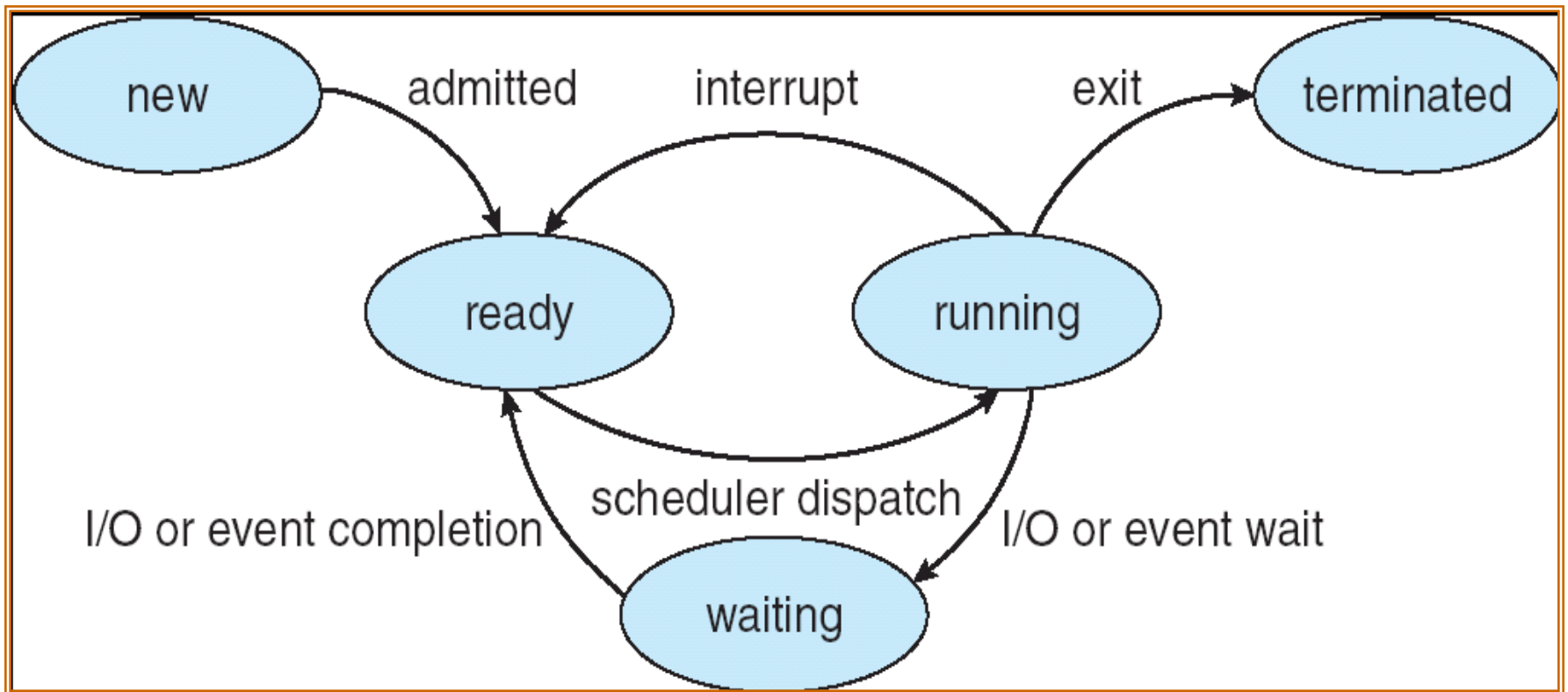
Exercise

- 在一个只有单处理机（不考虑多核）的操作系统中，进程有运行、就绪、等待三个基本状态。假如某时刻该系统中有10个用户进程并发执行，且CPU为非核心态情况下，试问：
  - 这时刻系统中处于运行状态的用户进程数最多有几个？最少有几个？
  - 这时刻系统中处于就绪状态的用户进程数最多有几个？最少有几个？
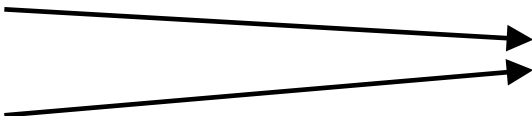 - 这时刻系统中处于等待状态的用户进程数最多有几个？最少有几个？

作业2

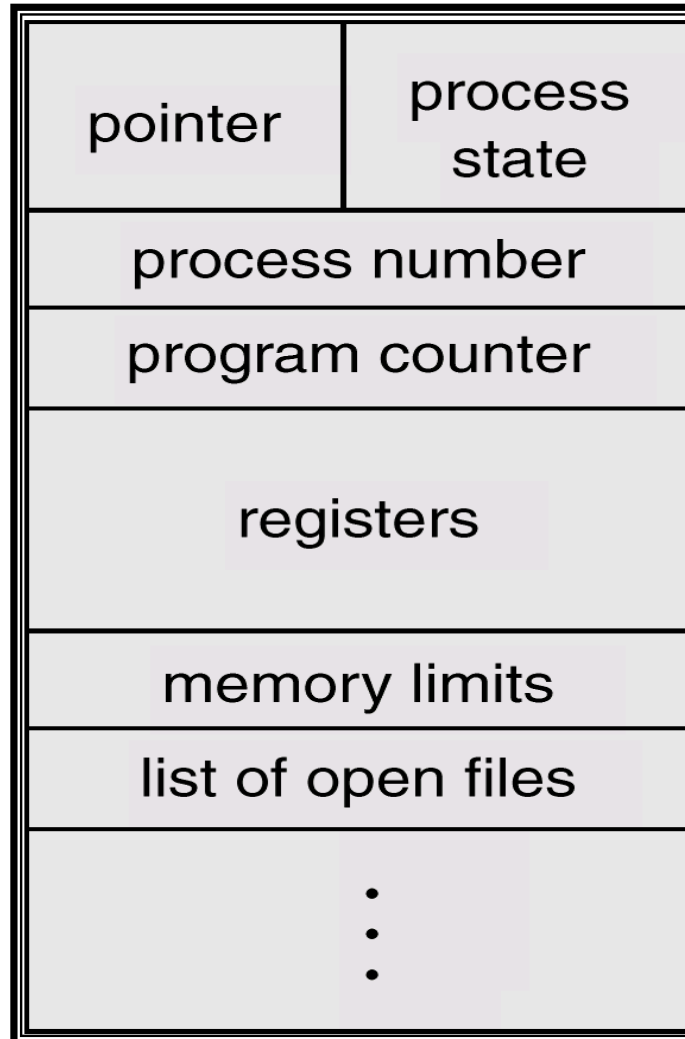 画出进程在就绪、运行和等待三个基本状态之间的状态转换图，并简述发生相应状态转换的原因。

Process Control Block (PCB)

- **Containing the information associated with a specific process**
  - **Process state**
  - **Program counter**
  - **CPU registers**
  - **CPU scheduling information**
  - **Memory-management information**
  - **Accounting information**
  - **I/O status information**
  - **… …**

Must be saved when an interrupt occurs

Process Control Block (PCB)

| pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
$\vdots$	

Outline

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

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

Ready Queue And Various I/O Device Queues
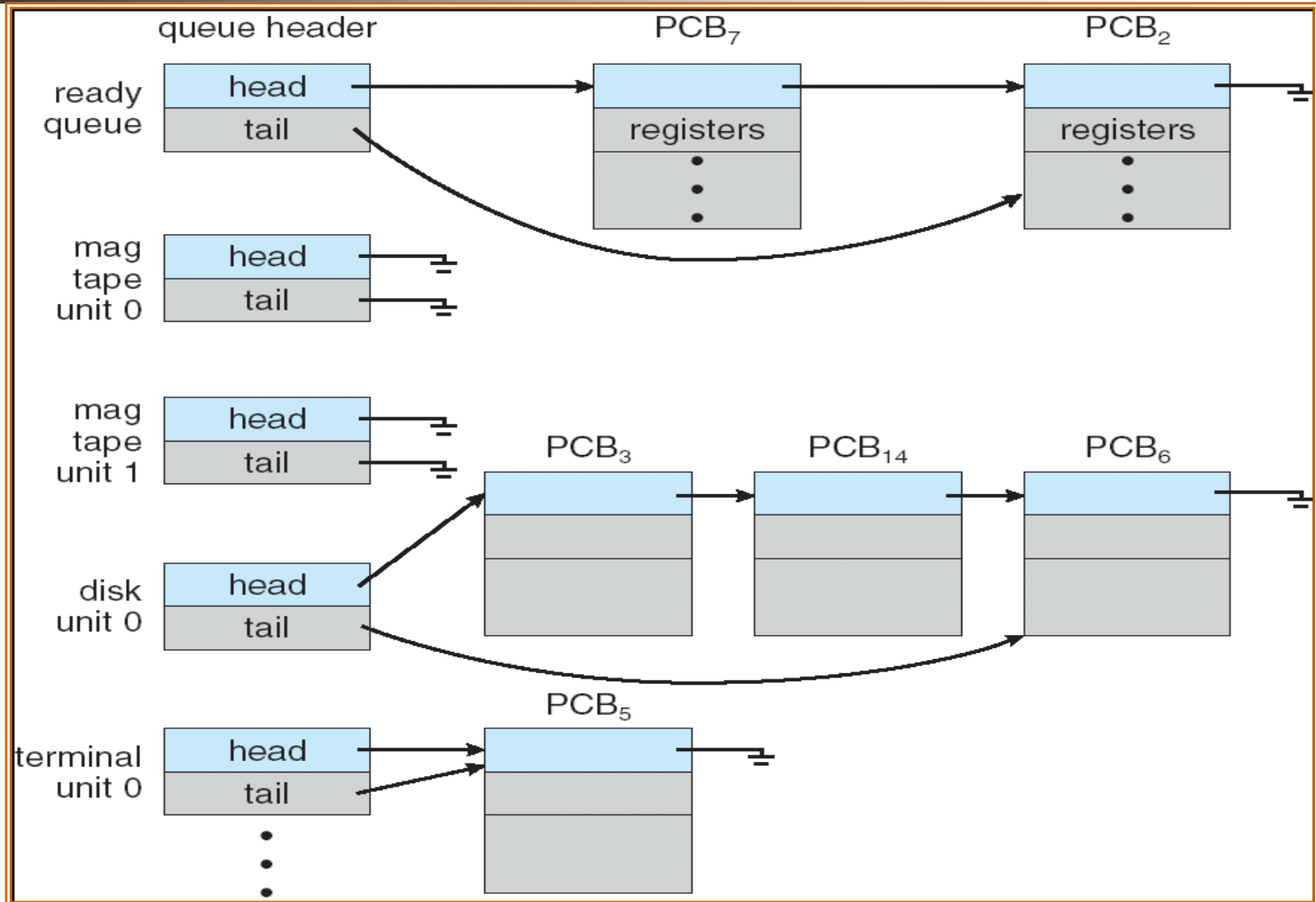
# Representation of Process Scheduling
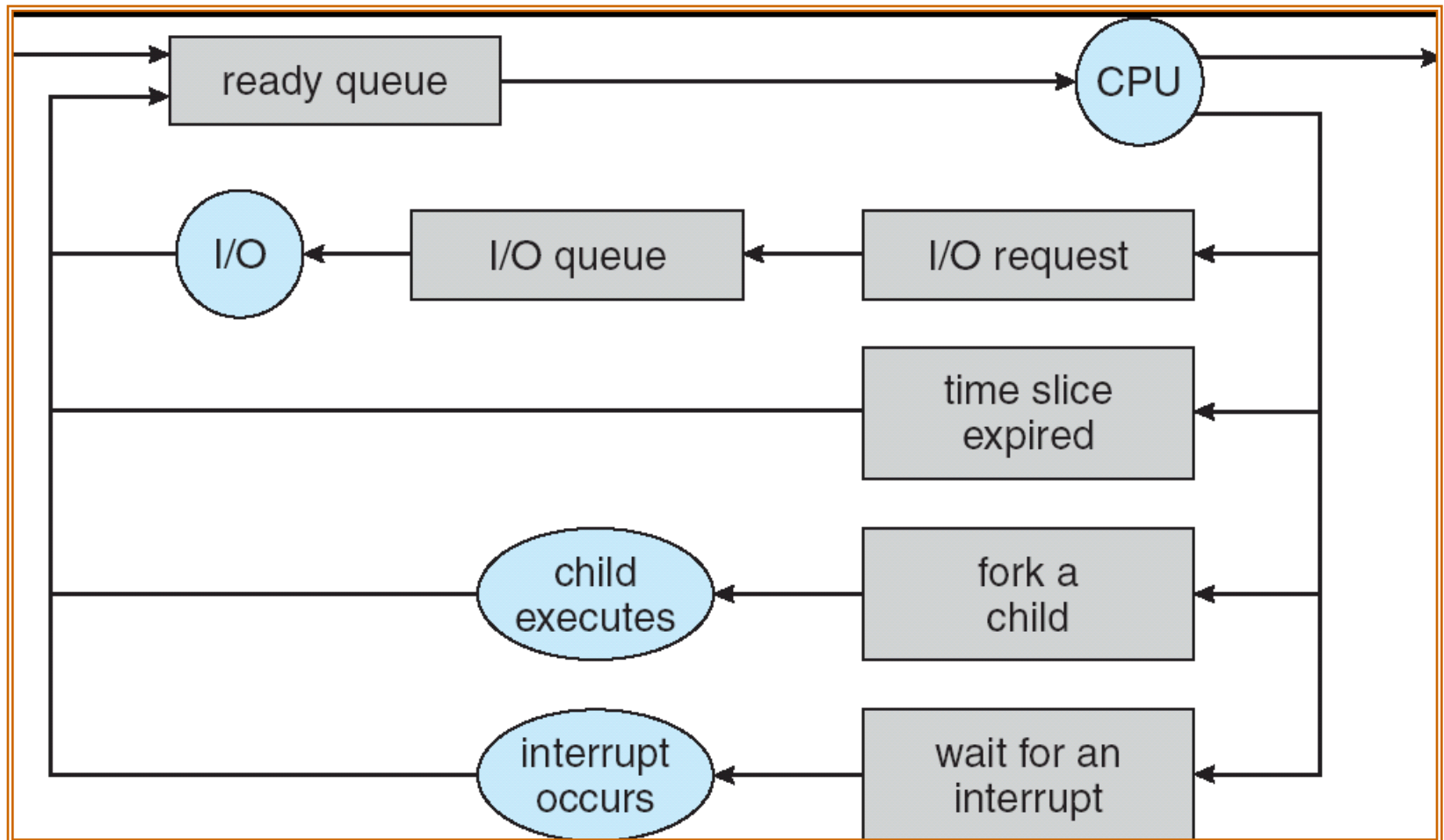
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be loaded into memory for execution.

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU.

- **Medium-term scheduler** – remove processes from memory and reintroduce them into memory later **(swapping).**

Schedulers

- **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* (the number of processes in memory).**

Medium-Term Scheduling
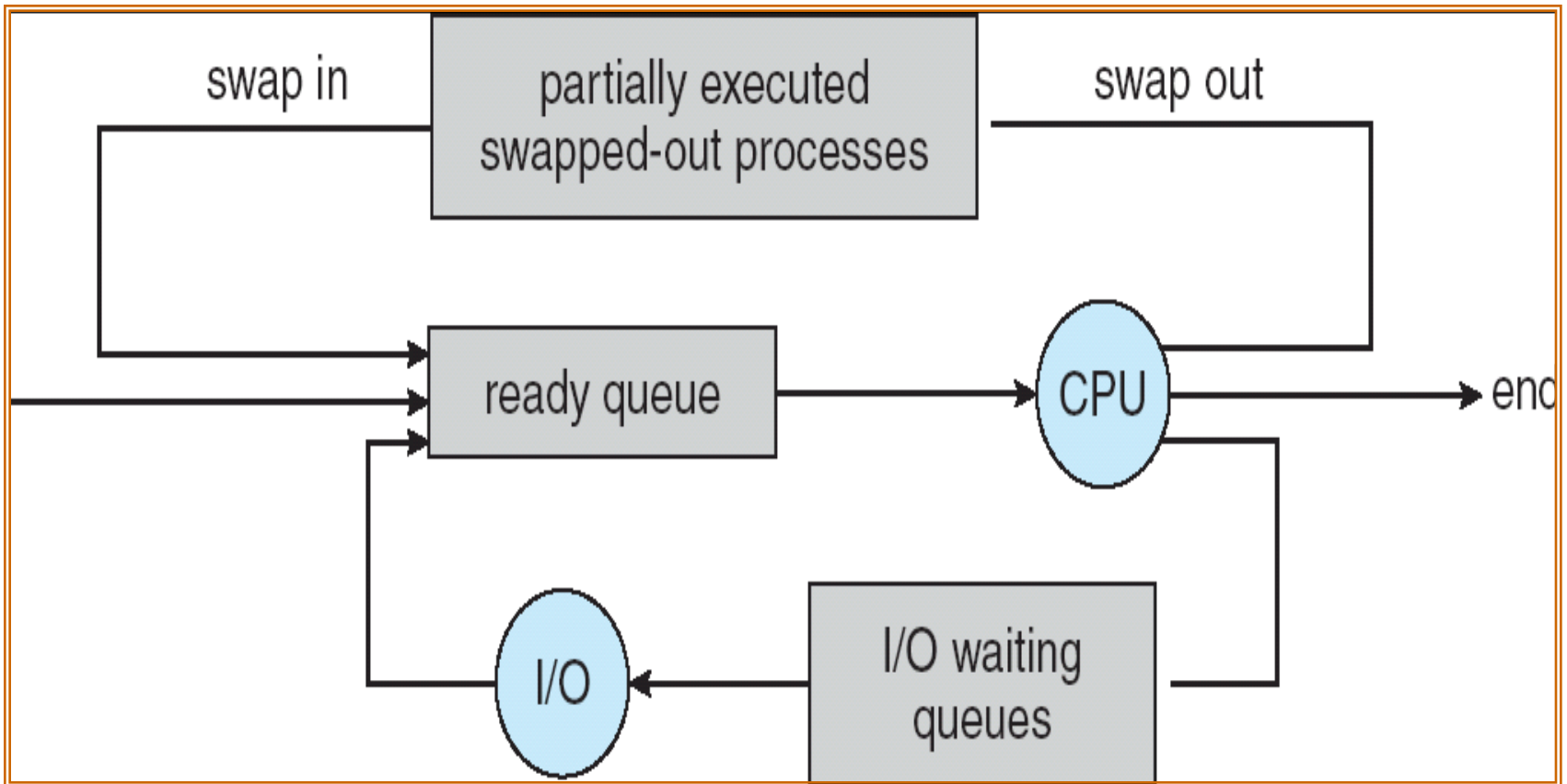
Schedulers

- **Processes can be described as either**
  - *I/O-bound process* **– spends more time doing I/O than doing computations.**
  - *CPU-bound process* **– spends more time doing computations than doing I/O.**
- **The long-term scheduler select a good process mix of I/O-bound and CPU-bound processes.**

Context Switch

- **What is a process context?**

  - **The *context* of a process is represented in the PCB of the process and includes the values of CPU registers, the process state, the program counter, and other memory/file management information.**

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.**

- **Context-switch time dependent on hardware support.**

CPU Switch From Process to Process
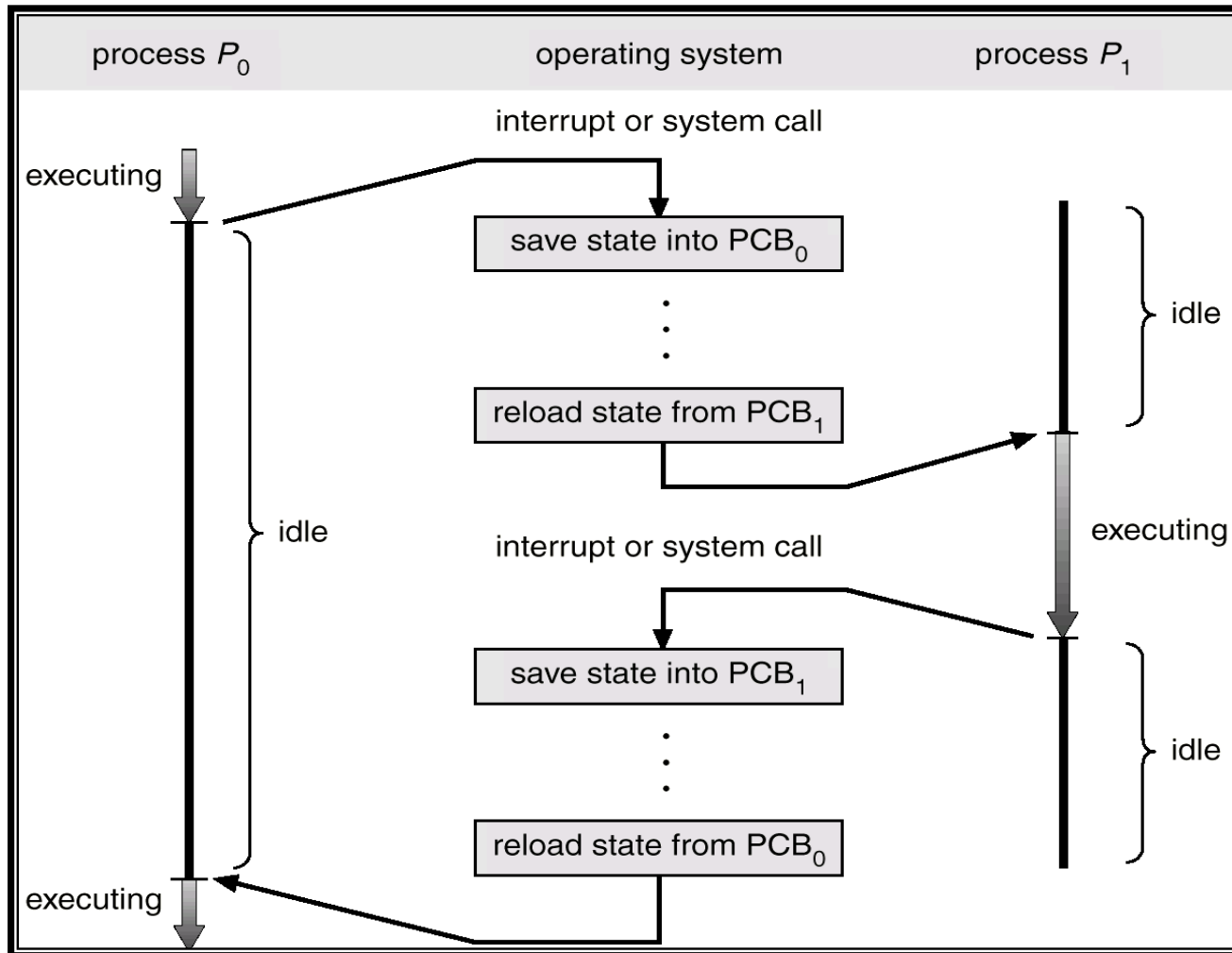
Exercise

- 下列哪一种情况不会引起进程之间的切换？
    - A. 进程调用本程序中定义的函数进行计算
    - B. 进程处理I/O请求
    - C. 进程创建子进程并等待子进程结束
    - D. 产生中断

Outline

- **Process Concept**
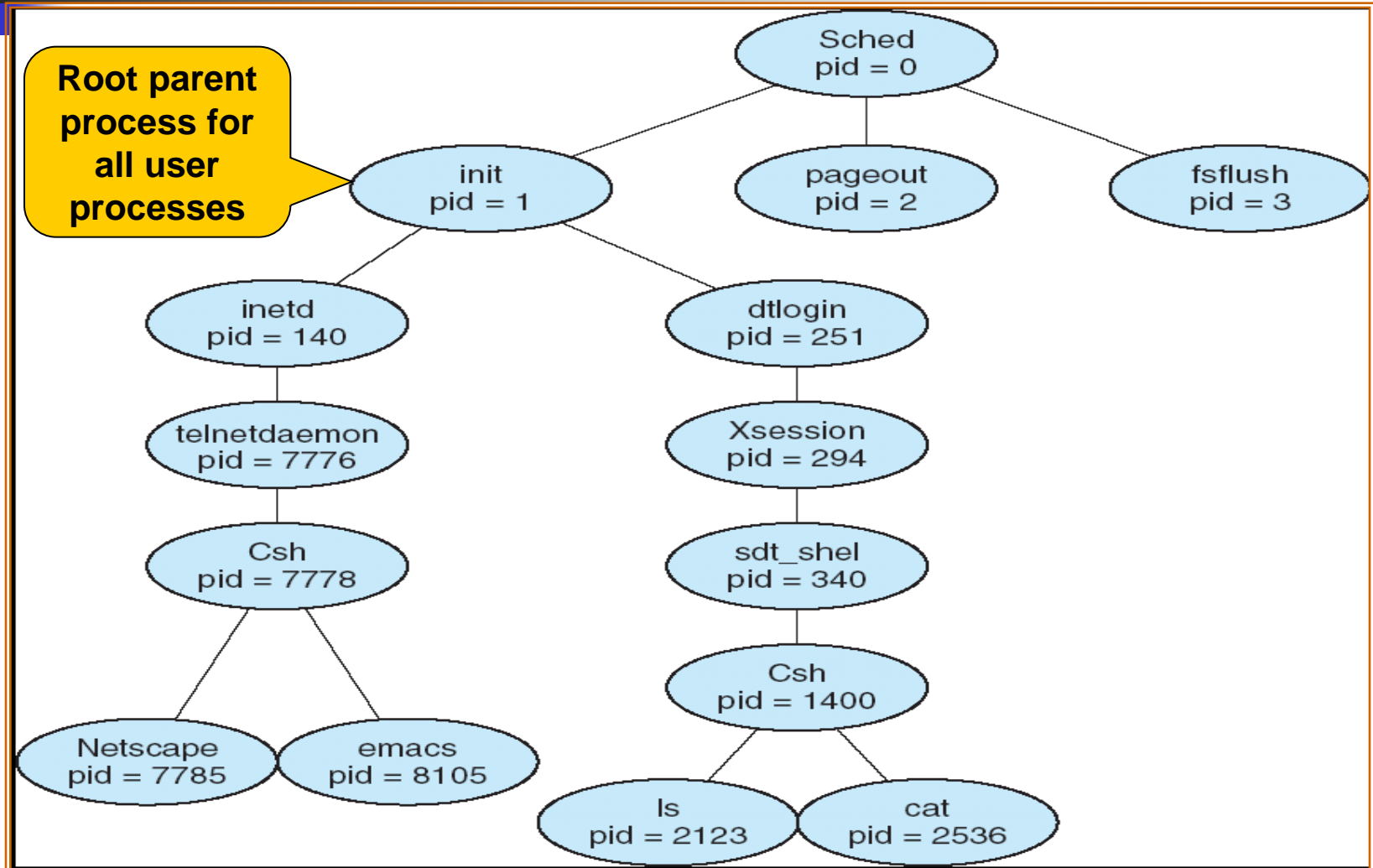
- **Process Scheduling**

- **Operations on Processes**

- **Interprocess Communication**

- **Communication in Client-Server Systems**

Process Creation

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

- **A unique process identifier (an integer number) is used to identify process.**

Processes Tree on Solaris

Process Creation

- **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

- **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's memory space with a new program.

Process Creation (UNIX)
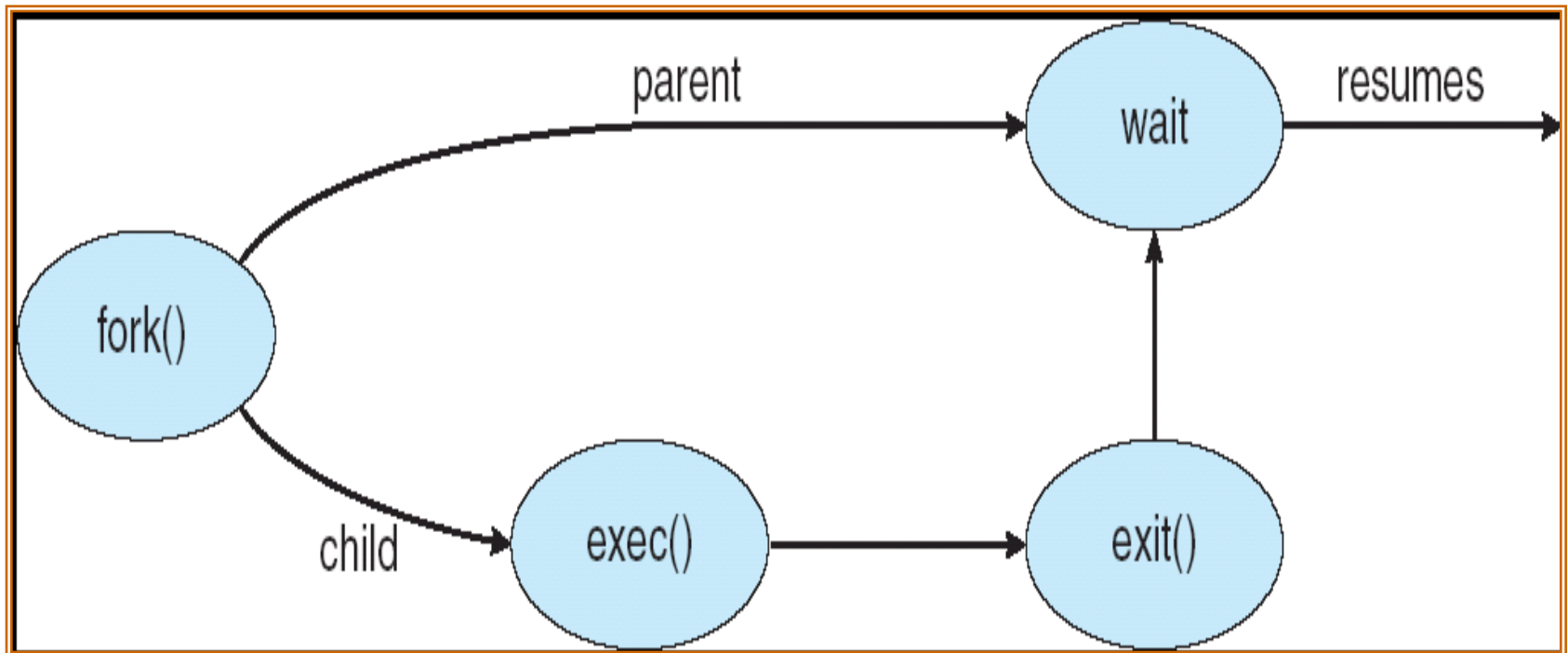
```c
pid = fork();
if (pid < 0){/*error occured*/
    fprintf(stderr, "Fork failed");
    exit(-1);}
else if(pid == 0){/*child process*/
    execlp("/bin/ls","ls",NULL);
}
else {/*parent process*/
    wait(NULL);
    printf("Child Complete");
    exit(0);
}
```

作业3

- 下列程序的输出是什么？

```
#include <sys/wait.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int value = 5;
int main()
{  pid_t pid;
   pid = fork();
   if (pid == 0) {  printf("child process, value1 : %d\n", value);
                    value += 15;
                    printf("child process, value2 : %d\n", value);}

   else if (pid > 0) { printf("parent process, value3 : %d\n", value);
                       wait(NULL);
                       printf("parent process, value4 : %d\n", value);
                       exit(0);   }
}
```

Process Termination

- **Process executes last statement and asks the operating system to delete it (exit).**
  - Output data from child to parent (via wait).
  - Process's resources are deallocated by OS.

Process Termination

- **Parent may terminate execution of children processes (abort).**
  - **Child has exceeded allocated resources.**
  - **Task assigned to child is no longer required.**
  - **Parent is exiting.（并非所有OS都如此，UNIX）**
    - **Operating system does not allow child to continue if its parent terminates.**
    - **Cascading termination (级联终止).**

Outline

- **Process Concept**

- **Process Scheduling**

- **Operations on Processes**

- **Interprocess Communication**

- **Communication in Client-Server Systems**

Interprocess Relationship

- *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

InterProcess Communication (IPC)

- **Communication may take place using either shared memory or message passing.**
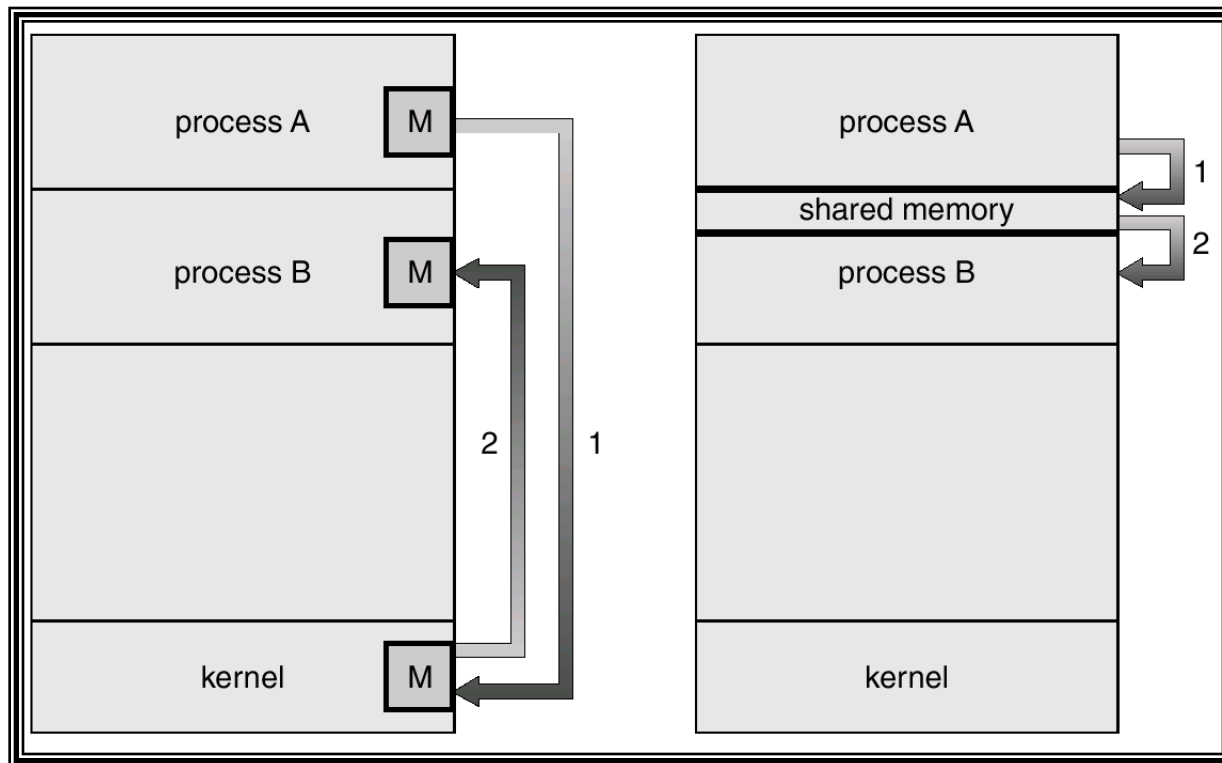
Shared-Memory Systems

- **Requiring communicating processes to establish a region of shared memory.**

- **A shared-memory region resides in the address space of the process creating the shared-memory segment.**

- **The processes are responsible for ensuring that they are not writing to the same location simultaneously.**

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;
```

Bounded-Buffer – Producer Process

```
item nextProduced;

while (1) {
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer – Consumer Process

```
item nextConsumed;

while (1) {
        while (in == out)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

Discussion

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

Why?

- *如何实现可使用缓冲的最大空间数为 BUFFER_SIZE?*

Message-Passing Systems

- **MPS 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?**

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 bidirectional.**

Indirect Communication

- **Messages are sent and received from mailboxes (also referred to as ports).**
  - **Each mailbox has a unique id.**
  - **Two 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 bidirectional.**

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**
- **Non-blocking is considered asynchronous**
- **send and receive primitives may be either blocking or non-blocking.**

Buffering

- **Queue of messages attached to the link implemented in one of three ways.**

  - **Zero capacity** – 0 messages.
    Sender must wait for receiver.

  - **Bounded capacity** – finite length of n messages.
    Sender must wait if link full.

  - **Unbounded capacity** – infinite length.
    Sender never blocks.

Outline

- **Process Concept**

- **Process Scheduling**

- **Operations on Processes**

- **Interprocess Communication**

- **Communication in Client-Server Systems**

Client-Server Communication

- **Sockets**

- **Remote Procedure Calls**

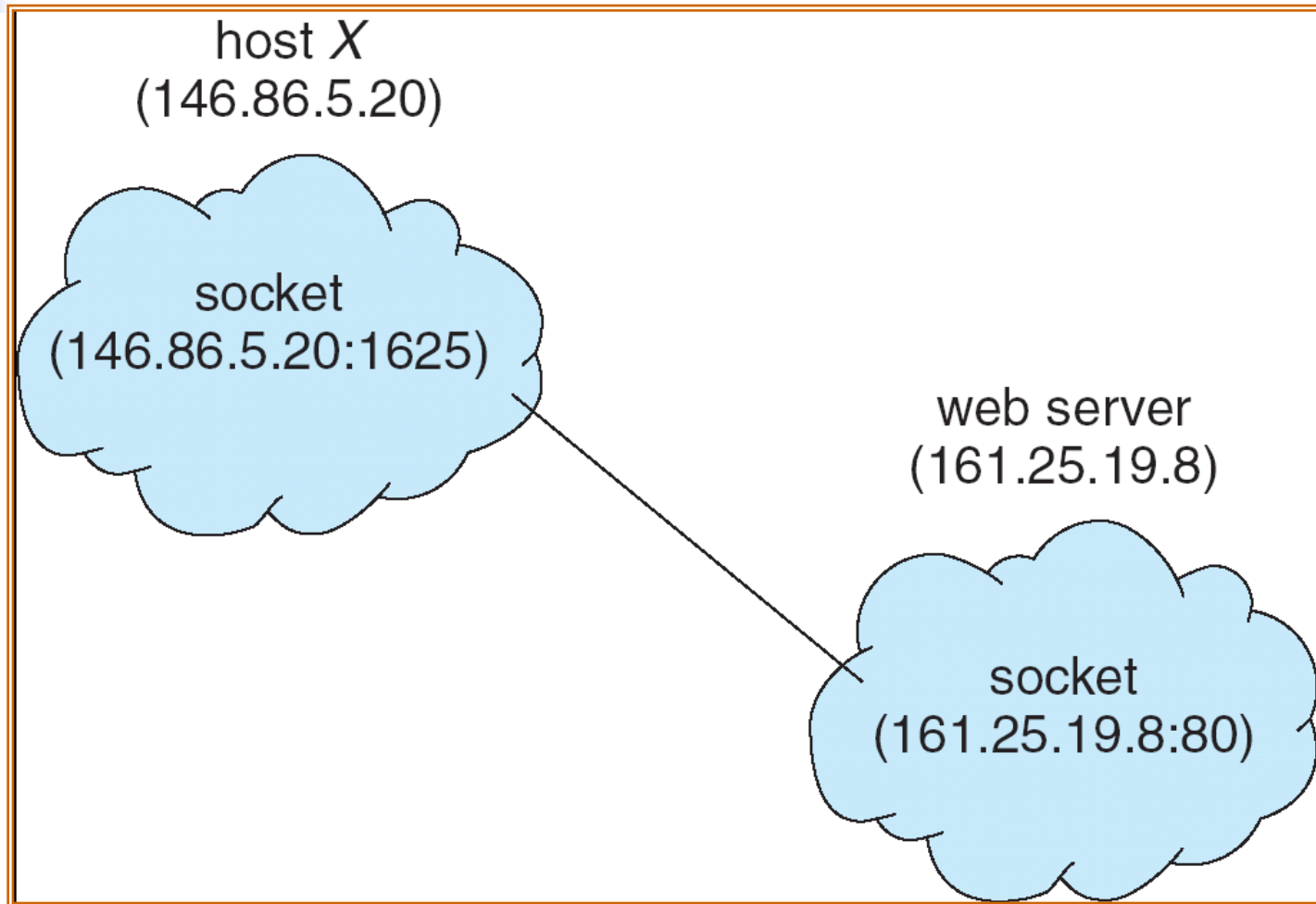- **Remote Method Invocation (Java)**

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

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 Communication

- **All connections must be unique.**

- **Common and Efficient**

- **A low-level form of communication between distributed processes**

- **Sockets allow only an unstructured stream of bytes to be exchanged between the communicating processes.**

Remote Procedure Calls

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

- **RPC allow a client to invoke a procedure on a remote host as it would invoke a procedure locally.**

- **Stubs – client-side proxy for the actual procedure on the server.**

- **Each message exchanged contains an identifier of the function and the parameters to pass to it.**

Remote Procedure Calls

- **The client-side stub** locates the port on the server and *marshals* the parameters.

- **The server-side stub** receives this message, unpacks the marshaled parameters, and performs the function on the server.

- Any output is send back to the requester in a separate message.

- Messages exchanged in RPC communication are **well structured.**
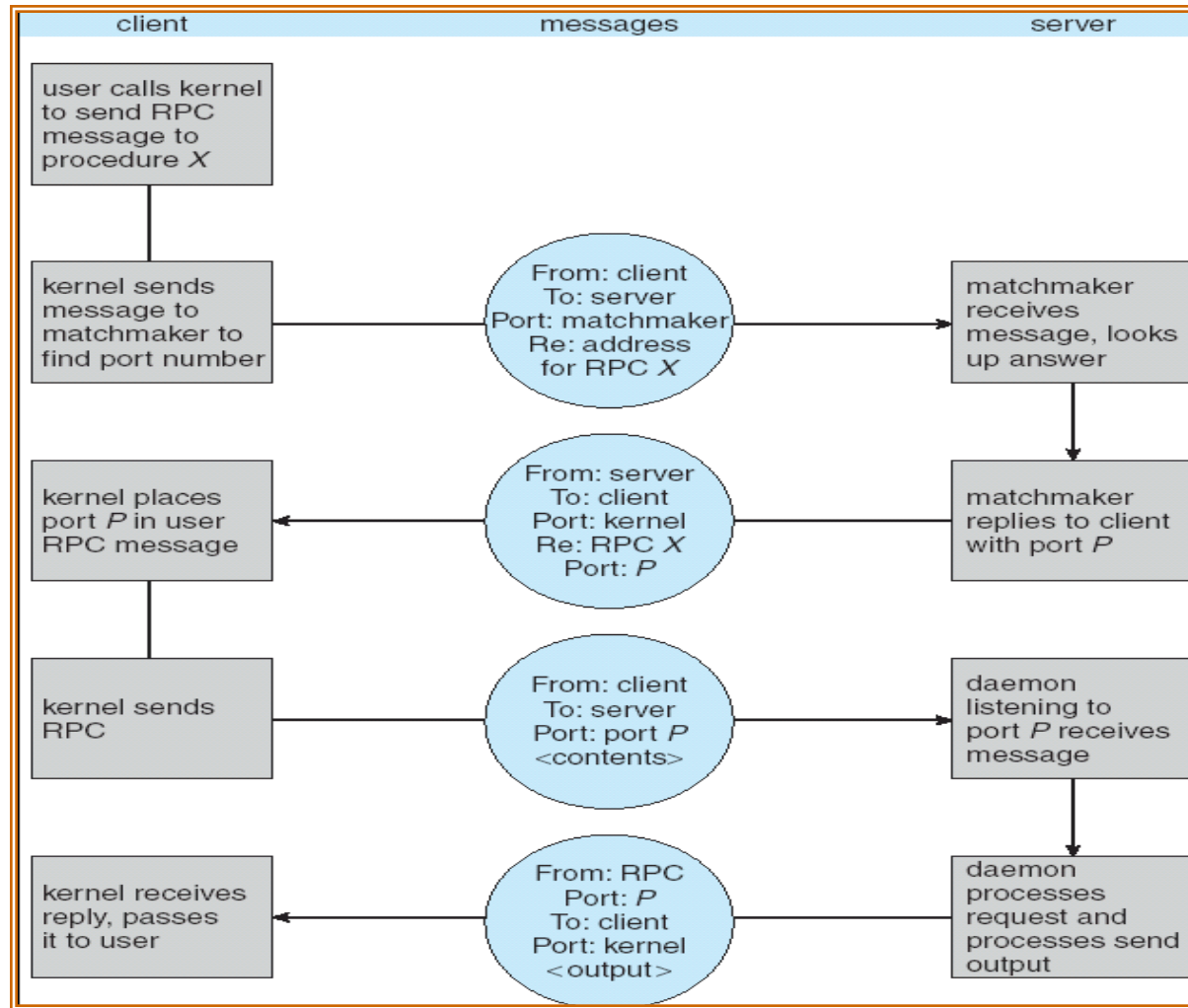
Remote Procedure Calls

- **Some Issues**
  - **The differences in data representation on the client and server machines.**
    - **eXternal Data Representation (XDR)**
  - **The semantics of a call**
    - **Exactly once（lost？）: ACK Messages**
    - **At most once（repeated?）: attaching a timestamp to each message.**

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.
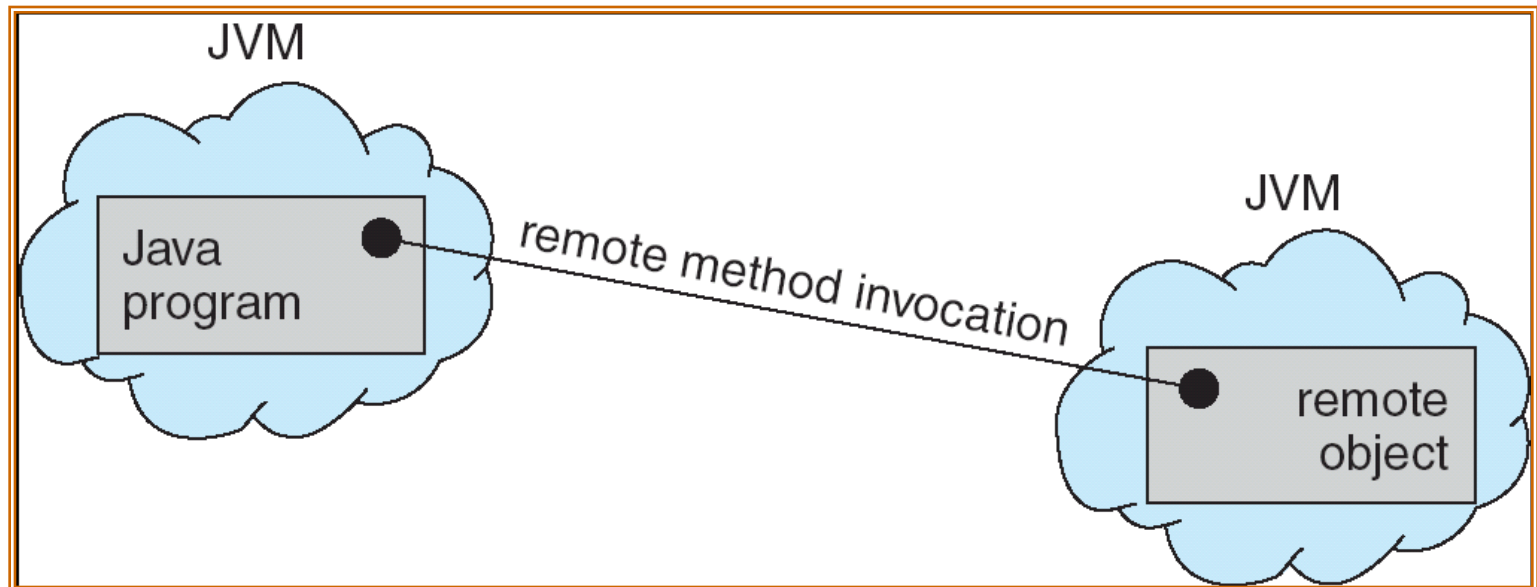
Remote Method Invocation

- **Differences between RPC and RMI**
  - **RPC only calls remote procedures or functions, and RMI supports invocation of methods on remote objects**
  - **The parameters to remote procedures are ordinary data structures in RPC, and that are objects in RMI**

Remote Method Invocation

- **RMI implements the remote object using stubs and skeletons.**
  - **Stub is a proxy for the remote object and resides with the client.**
  - **Skeleton resides with the server, receives the messages from stub and invokes the desired method on the server.**

Marshalling Parameters