Chapter 3

Processes

Contents

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocesses Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

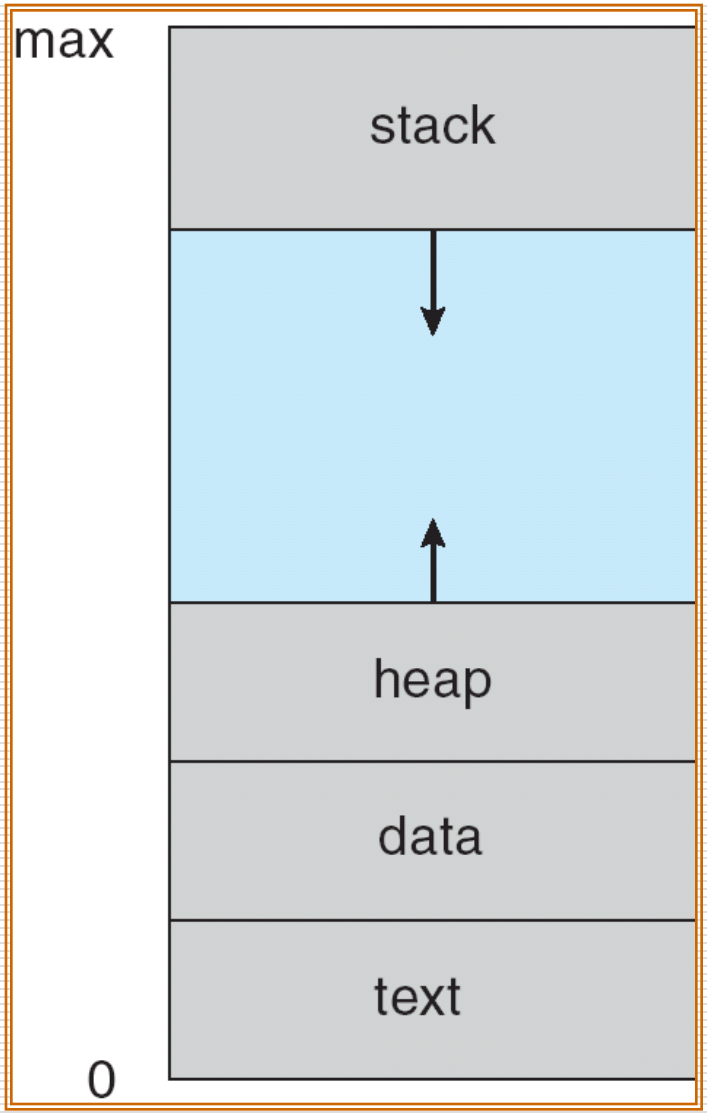
Objectives

- To introduce the notion of a process---a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication.
- To describe communications 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:
  - Text section
  - program counter
  - stack
  - data section

Process in Memory



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

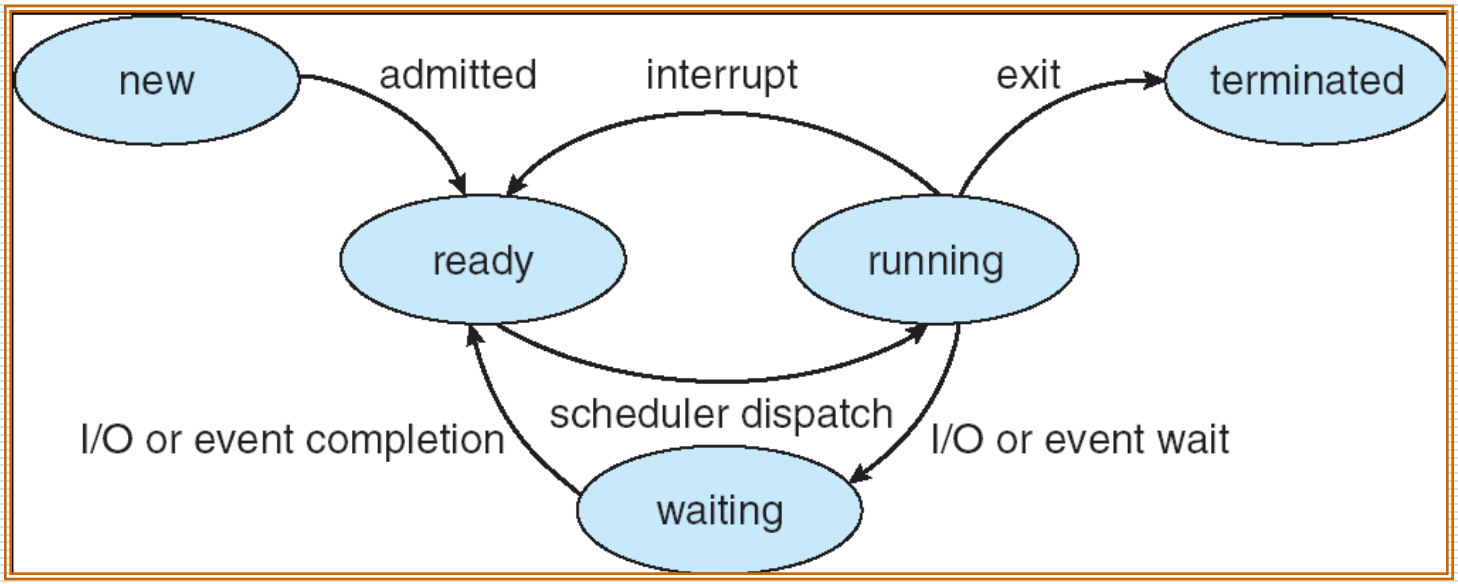
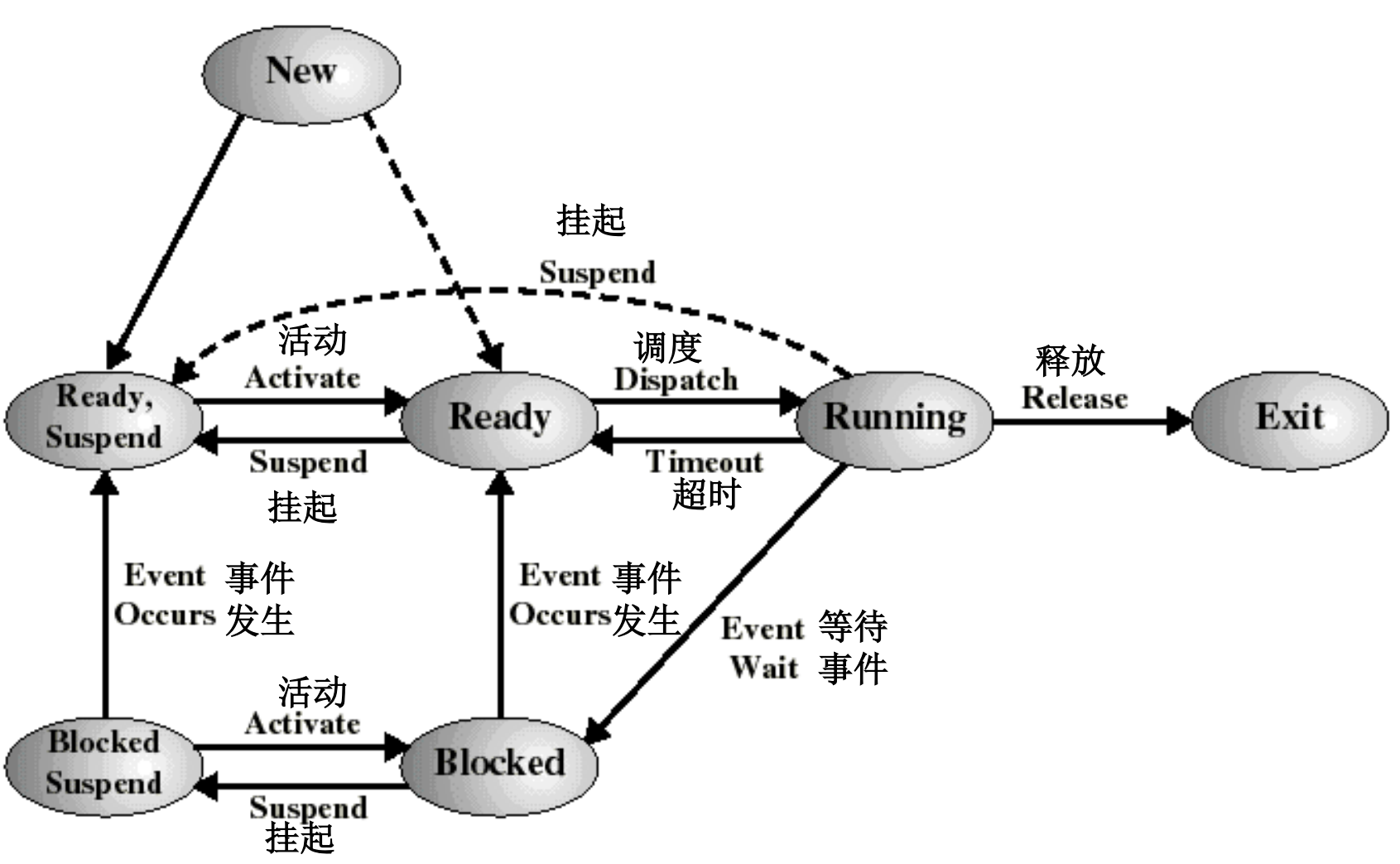


Diagram of Process State



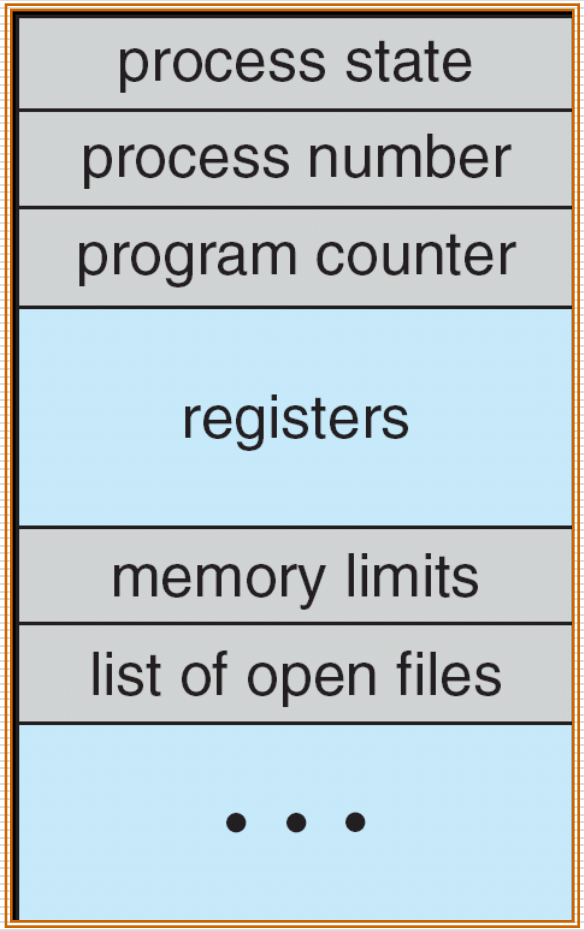
Process Control Block (PCB)

In order to manage processes, OS defines a data structure for each process to record the process's characteristics, and describe the change.

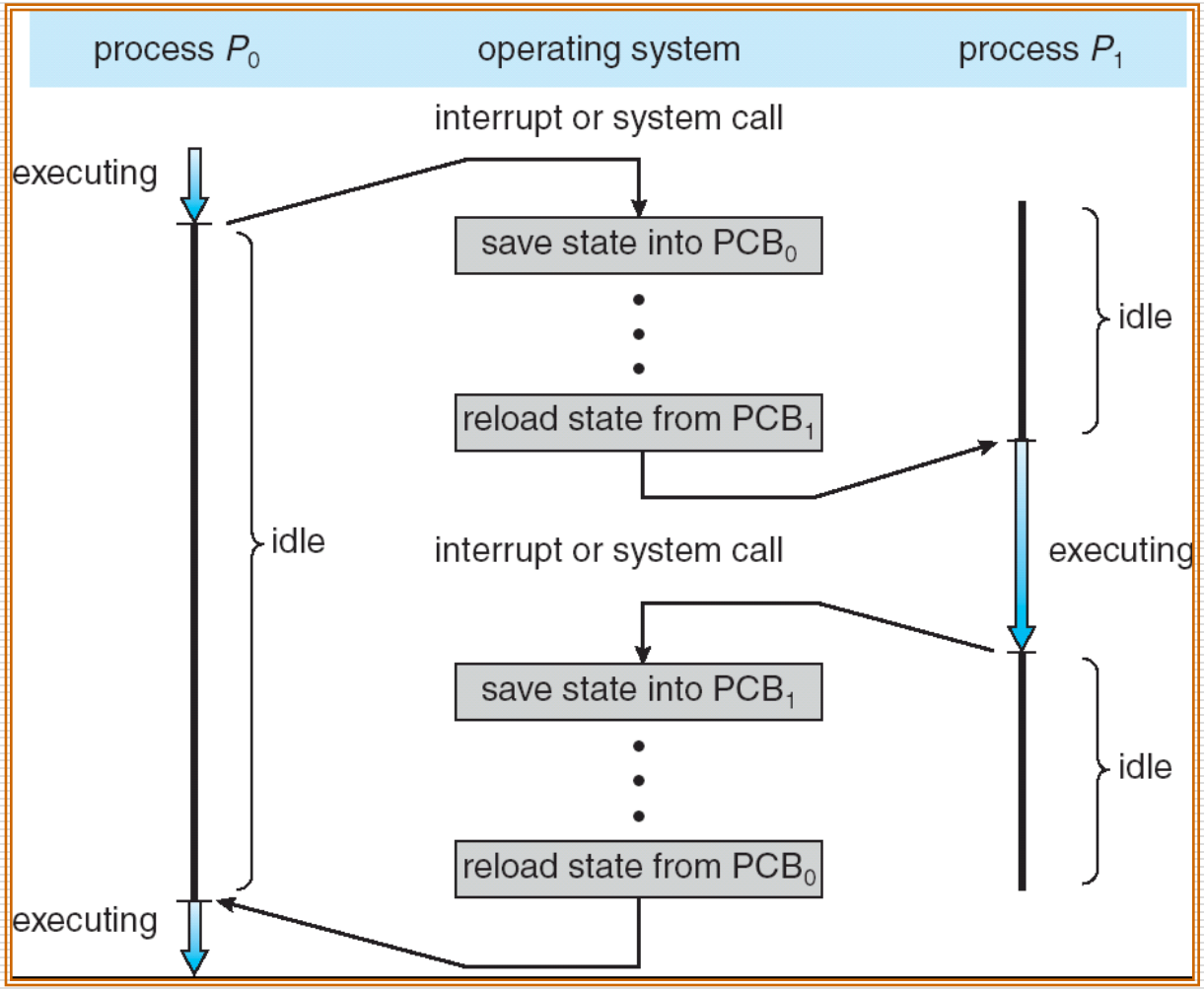
Information associated with each process

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

Process Control Block (PCB)



CPU Switch From Process to Process



PCB of Linux

The PCB in Linux is represented by the C structure task_struct.

This structure contains information for representing a process, including the state of the process, scheduling, and memory management information, list of open files, and pointers to the process's parent and any of its children.

Struct task_struct{

- /* task state */
- Volatile long state; //定义 task 运行的状态，以及信号
- struct linux_binfmt *binfmt;
- int exit_code, exit_signal;
- int pdeath_signal; /* The signal sent when the parent dies */
- /* 定义进程的用户号, 用户组以及进程组*/
- unsigned long personality;
- int dumpable:1;
- int did_exec:1;
- pid_t pid; //process identifier
- Long state; //state of the process
- pid_t pgrp;
- pid_t tty_old_pgrp;
- pid_t session;

PCB of Linux

- /* boolean value for session group leader */
- 是否为进程组的头
- int leader;
- /*
- * pointers to (original) parent process, youngest child, younger sibling,
- * older sibling, respectively. (p->father can be replaced with
- * p->p_pptr->pid)
- */
- 父子进程的一些指针
- struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
-
- /* PID hash table linkage. */
- 在调度中用的一些hash 表
- struct task_struct *pidhash_next;
- struct task_struct **pidhash_pprev;

PCB of Linux

- /* Pointer to task[] array linkage. */
- struct task_struct **tarray_ptr;
- struct wait_queue *wait_chldexit; /* for wait4() 等待队列 */
- struct semaphore *vfork_sem; /* for vfork() */
- unsigned long policy, rt_priority;
- unsigned long it_real_value, it_prof_value, it_virt_value;
- 进程的性质因为实时进程与普通进程的调度算法不一样所以应有变量区分
- 下面是进程的一些时间信息
- unsigned long it_real_incr, it_prof_incr, it_virt_incr;
- struct timer_list real_timer;
- struct tms times;
- unsigned long start_time;
- long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];//定义了时间片的大小
- /* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
- 内存信息
- unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
- int swappable:1;

PCB of Linux

- /* process credentials */
- uid_t uid,euid,suid,fsuid;
- gid_t gid,egid,sgid,fsgid;
- int ngroups;
- gid_t groups[NGROUPS];
- kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
- struct user_struct *user;
- /* limits */
- struct rlimit rlim[RLIM_NLIMITS];
- unsigned short used_math;
- char comm[16];
- /* file system info */
- int link_count;
- struct tty_struct *tty; /* NULL if no tty */
- /* ipc stuff */

PCB of Linux

- struct sem_undo *semundo;
- struct sem_queue *semsleeping;
- /* tss for this task */
- struct thread_struct tss;
- /* filesystem information */
- struct fs_struct *fs;
- /* open file information */
- struct files_struct *files; //list of open files
- /* memory management info */
- struct mm_struct *mm; //address space of this process
-
- /* signal handlers */
- spinlock_t sigmask_lock; /* Protects signal and blocked */
- struct signal_struct *sig;
- sigset_t signal, blocked;
- struct signal_queue *sigqueue, **sigqueue_tail;
- unsigned long sas_ss_sp;
- size_t sas_ss_size;
- };

Linux进程的状态

- TASK_RUNNING 可运行
- TASK_INTERRUPTIBLE 可中断的等待状态
- TASK_UNINTERRUPTIBLE 不可中断的等待状态
- TASK_ZOMBIE 僵死
- TASK_STOPPED 暂停
- TASK_TRACED
- TASK_DEAD

标识符

域名	含义
Pid	进程标识符
Uid、gid	用户标识符、组标识符
Euid、egid	有效用户标识符、有效组标识符
Suid、sgid	备份用户标识符、备份组标识符
Fsuid、fsgid	文件系统用户标识符、文件系统组标识符

进程通信

进程通信有关信息

域名	含义
Spinlock_t sigmask_lock	信号掩码的自旋锁
Long blocked	信号掩码
Struct signal *sig	信号处理函数
Struct sem_undo *semundo	为避免死锁而在信号量上设置的取消操作
Struct sem_queue *semsleeping	与信号量操作相关的等待队列

进程链接信息

名称	英文解释	中文解释［指向哪个进程］
p_opptr	Original parent	祖先
p_pptr	Parent	父进程
p_cptr	Child	子进程
p_ysptr	Younger sibling	弟进程
p_osptr	Older sibling	兄进程
Pidhash_next、Pidhash_pprev		进程在哈希表中的链接
Next_task、prev_task		进程在双向循环链表中的链接
Run_list		运行队列的链表

与时间有关的域

域名	含义
Start_time	进程创建时间
Per_cpu_utime	进程在某个CPU上运行时在用户态下耗费的时间
Per_cpu_stime	进程在某个CPU上运行时在系统态下耗费的时间
Counter	进程剩余的时间片

与文件系统相关的域和内存相关域

定义形式	解释
Sruct fs_struct *fs	进程的可执行映象所在的文件系统
Struct files_struct *files	进程打开的文件

定义形式	解释
Struct mm_struct *mm	描述进程的地址空间
Struct mm_struct *active_mm	内核线程所借用的地址空间

页面管理信息

定义形式	解释
Int swappable	进程占用的内存页面是否可换出
Unsigned long min_flat,maj_flt,nswap	进程累计的次（minor）缺页次数、主(major)次数及累计换出、换入页面数
Unsigned long cmin_flat,cmaj_flt,cnswap	本进程作为祖先进程，其所有层次子进程的累计的次（minor）缺页次数、主(major)次数及累计换出、换入页面数

Active processes in Linux

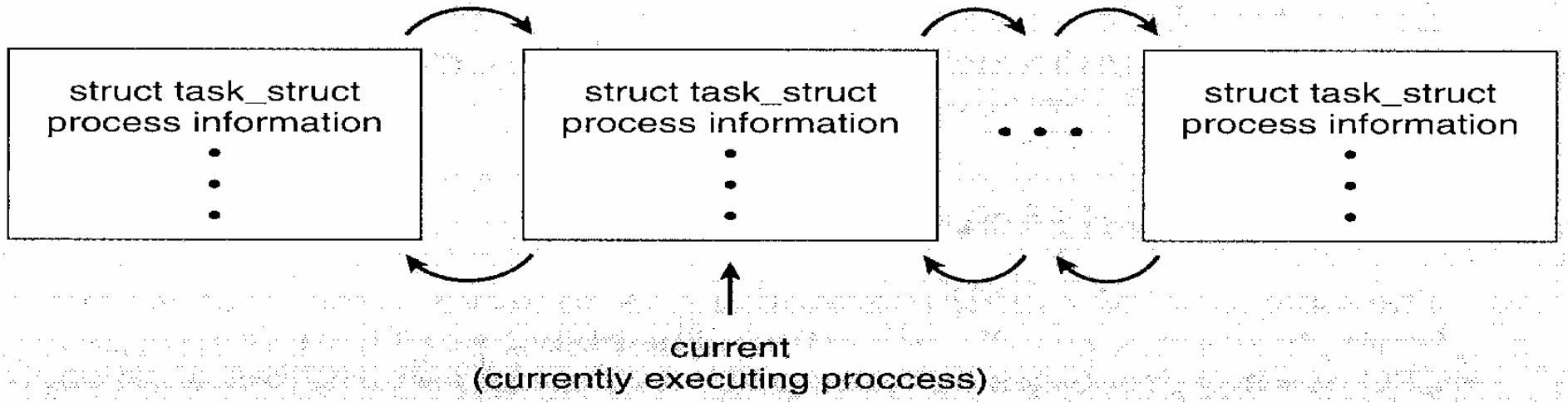


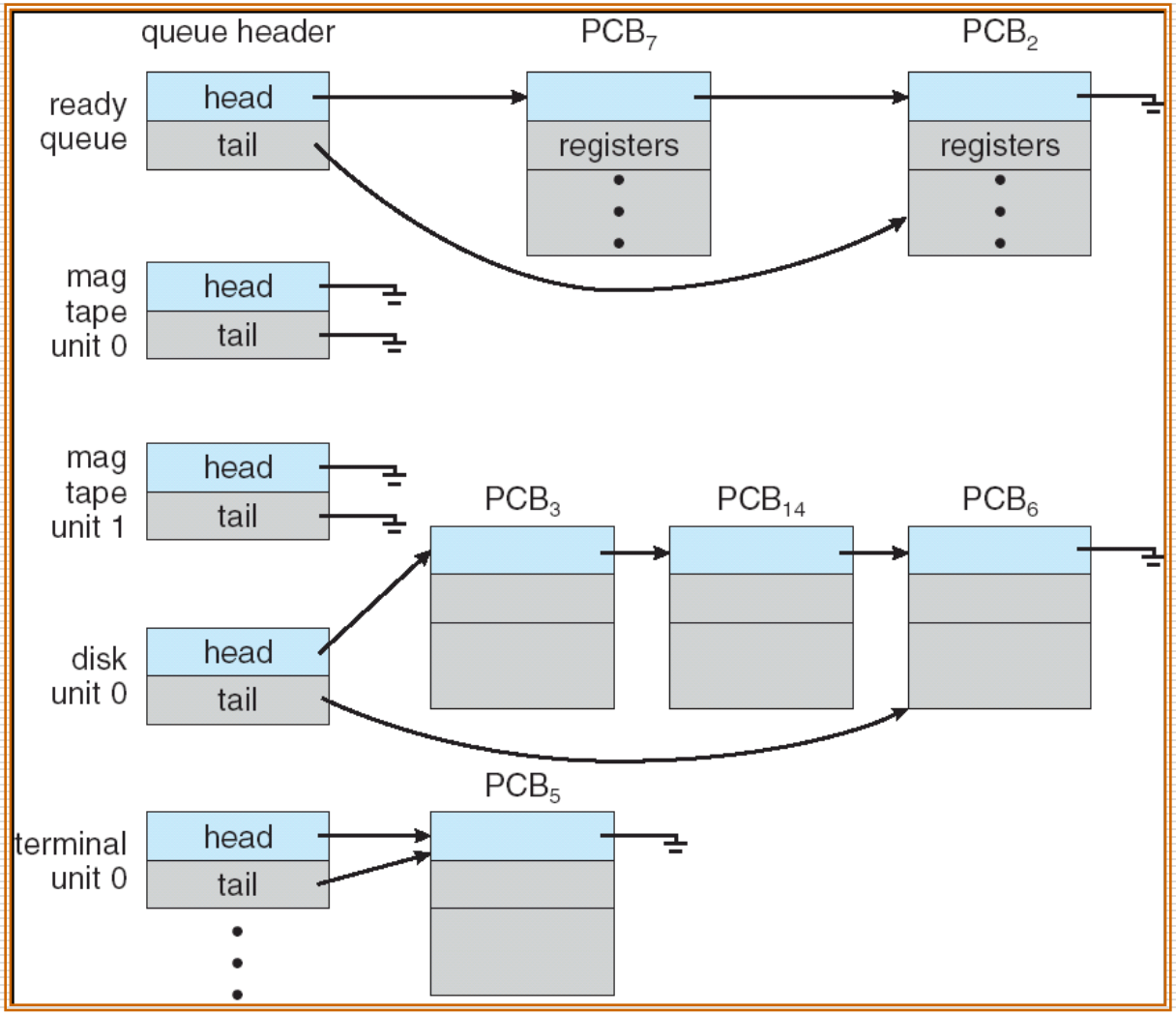
Figure 3.5 Active processes in Linux.

Process Scheduling Queues

In multiprogramming or time sharing systems, process scheduler is needed to select a process for running

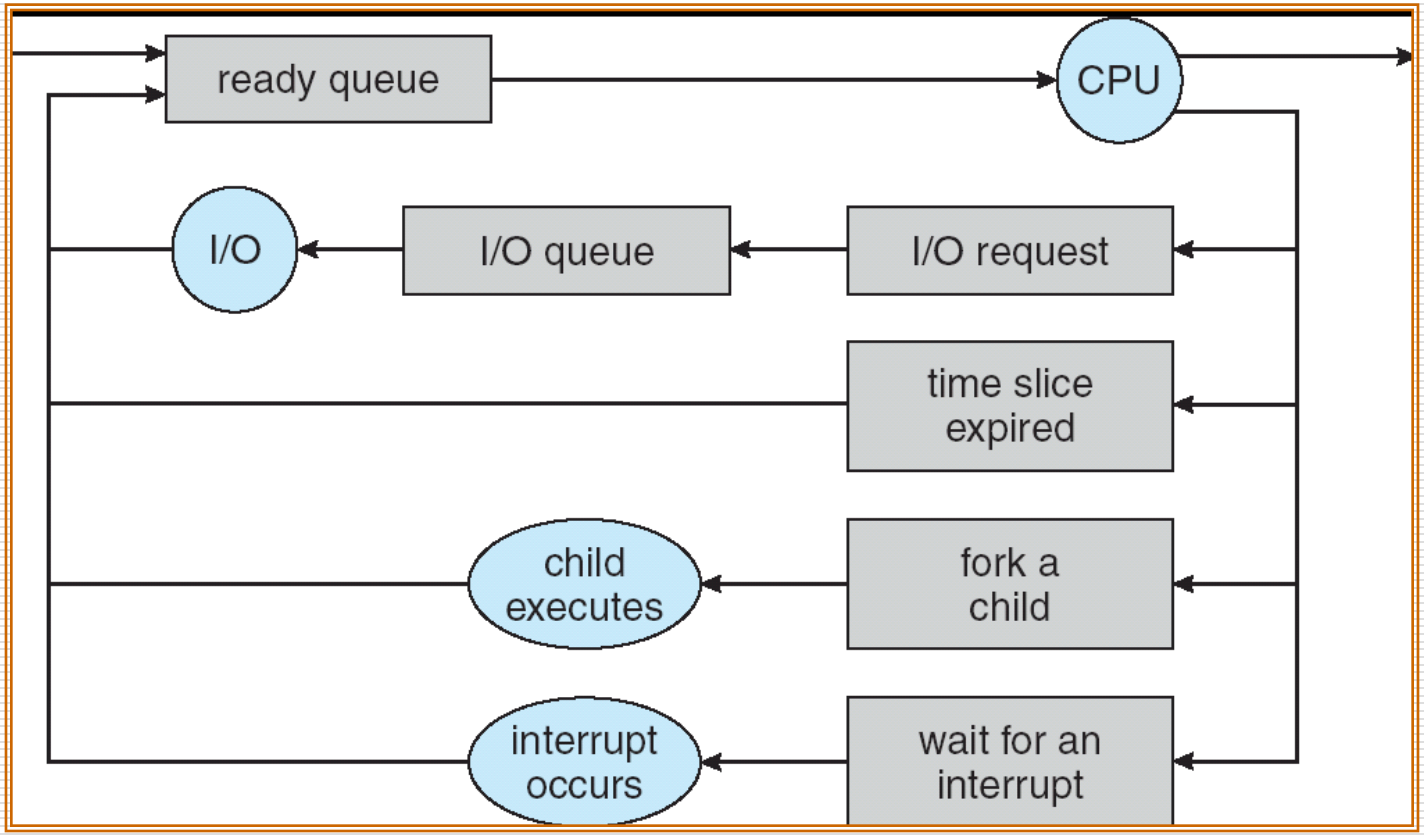
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- Once the process is allocated the CPU, one of the following events could occur:
  - Issue an I/O request
  - Create a new subprocess
  - Wait for an interrupt

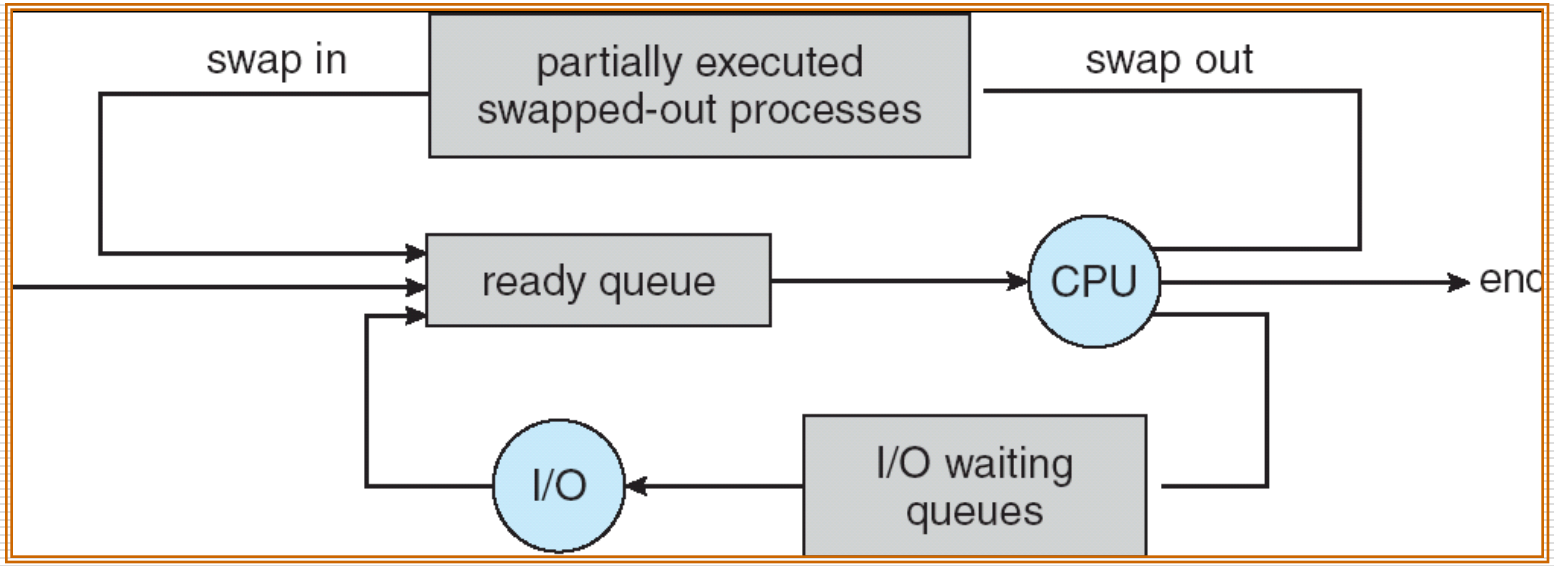


Schedulers

- **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

Addition of Medium Term Scheduling

- Swapping



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

Note: Long-term scheduler should make a careful selection between I/O bound process and CPU-bound process

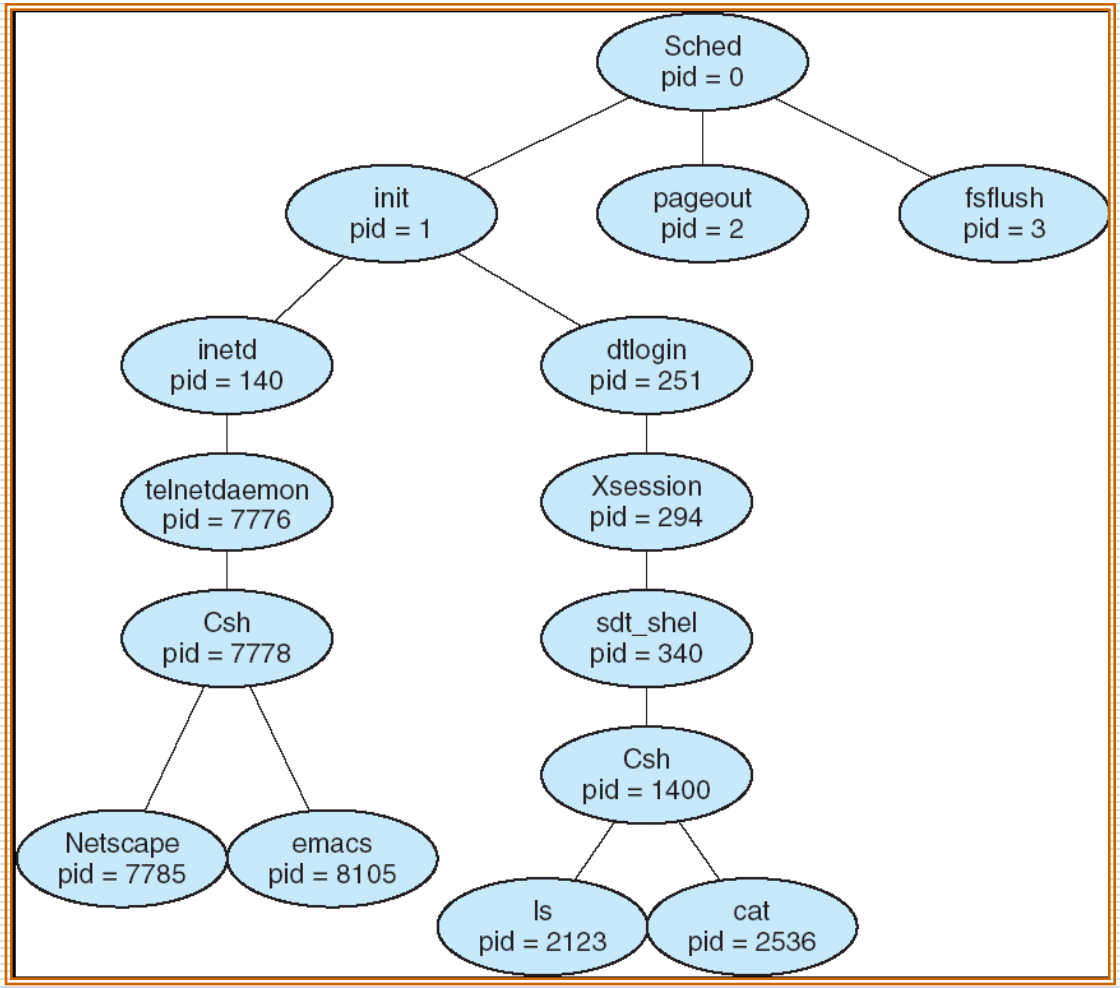
Context Switch

- 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
- PCB
- Context-switch time is overhead; the system does no useful work while switching—several milliseconds
- Time dependent on hardware support
  - Memory speed
  - The number of registers
  - The existence of special instructions
- Some special CPU—Sun UltraSPARC
  - Provide multiple sets of registers

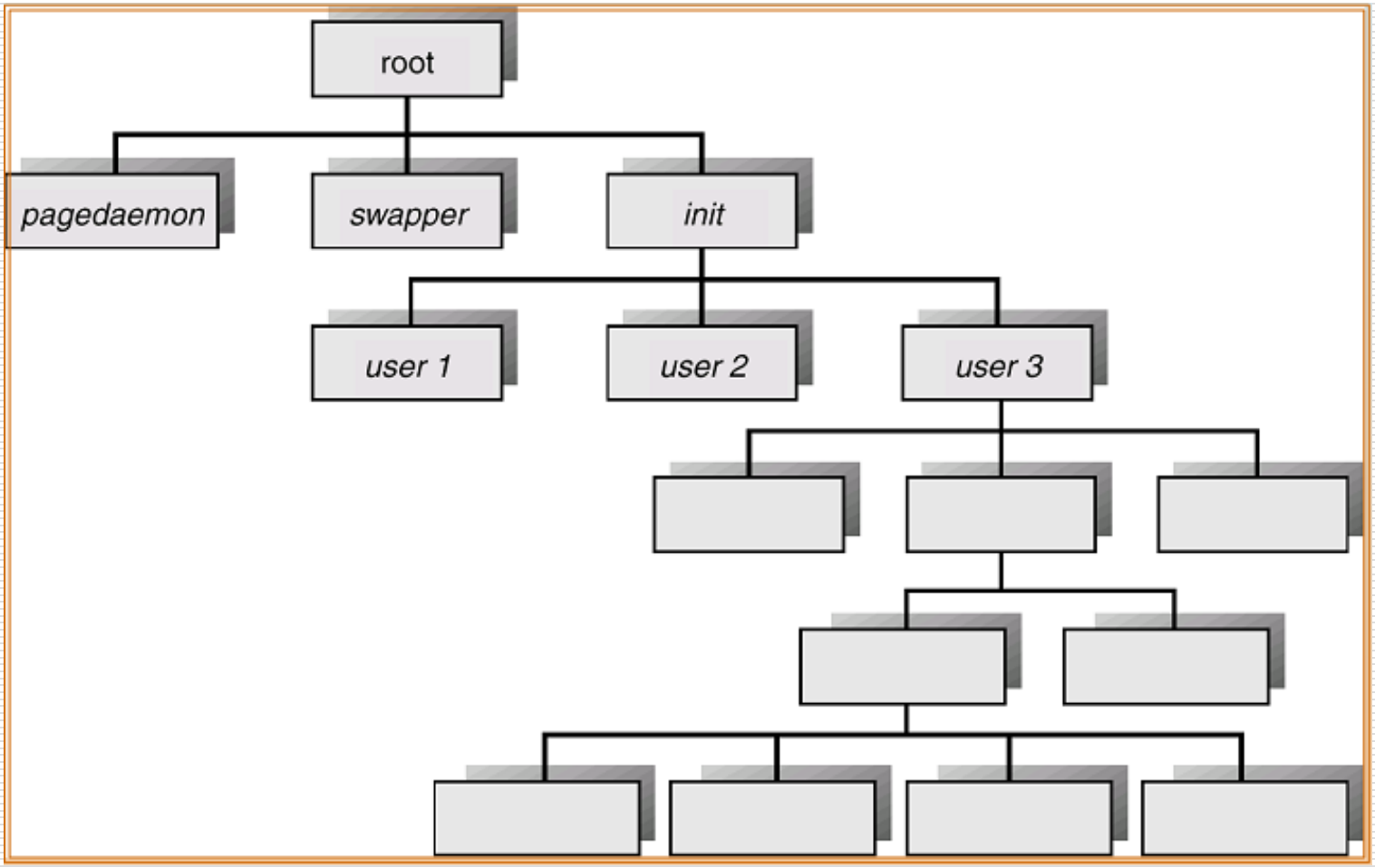
Process Creation

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

A tree of processes on a typical Solaris



a tree of processes in UNIX



Process Creation (Cont.)

- 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

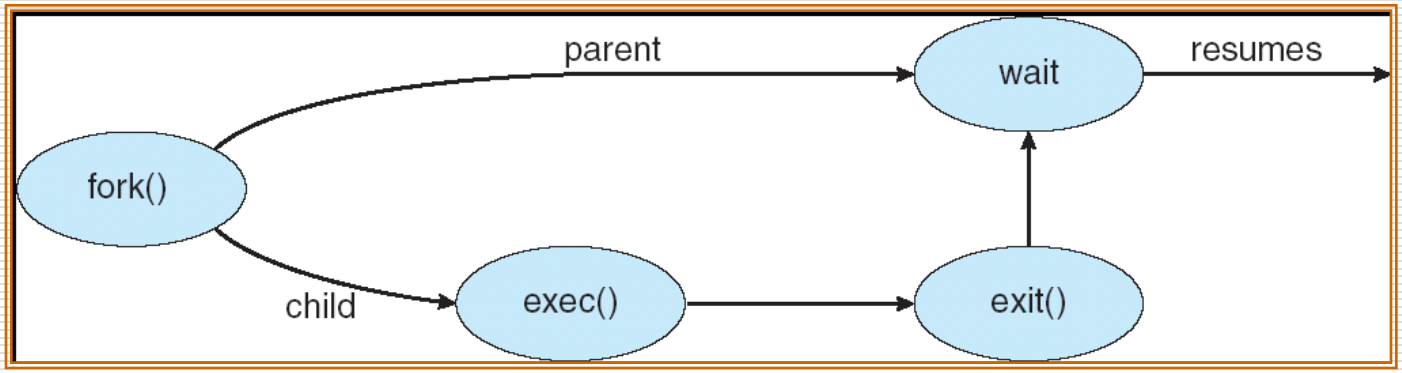
Process Creation (Cont.)

- 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's memory space with a new program

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);
        }
}
```

Process Creation



Another example

```
#include <sys/types.h>
#include<stdio.h>
#include<unistd.h>
int value = 5;
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0)
        value+=15;
    else if (pid>0)
    {
        wait(NULL);
        printf("parent: value = %d",value);
        exit(0);
    }
}
```

Another example

```
 1 #include<unistd.h>
 2 #include<stdio.h>
 3
 4 int main()
 5 {
 6     int pid;
 7     printf("Before the fork(),pid=%d\n",getpid());
 8     pid=fork();
 9     printf("After the fork(),pid=%d\n",getpid());
10     if(pid<0)
11         printf("error\n");
12     else if(pid==0)
13         printf("the chlid process! %d\n",getpid());
14     else if(pid>0)
15         printf("the parent process! %d\n",getpid());
16     return 1;
17 }
```

Process creation in windows

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
      "C:\\WINDOWS\\system32\\mspaint.exe", // command line
      NULL, // don't inherit process handle
      NULL, // don't inherit thread handle
      FALSE, // disable handle inheritance
      0, // no creation flags
      NULL, // use parent's environment block
      NULL, // use parent's existing directory
      &si,
      &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figure 3.12 Creating a separate process using the Win32 API.

Assignment

```
#include "stdio.h"
 #include "unistd.h"
 #include "sys/ types.h"

 main()
 {
     int pid;
     int i;
     char *flag[]={"child", "parent"};

     pid=fork();
     if(pid==0)
         i=0;
     else i=1;

     printf("%s:i=%d,&i=%d\n",flag[i],i,&i);
 }
```

运行结果如下：

```
child:i=0,&i=-1073751104
parent:i=1,&i=-1073751104
```

- 当父进程调用fork()创建子进程之后，下列哪些变量在子进程中修改之后，父进程里也会相应地作出改动？

A. 全局变量
B. 局部变量
C. 静态变量
D. 文件指针

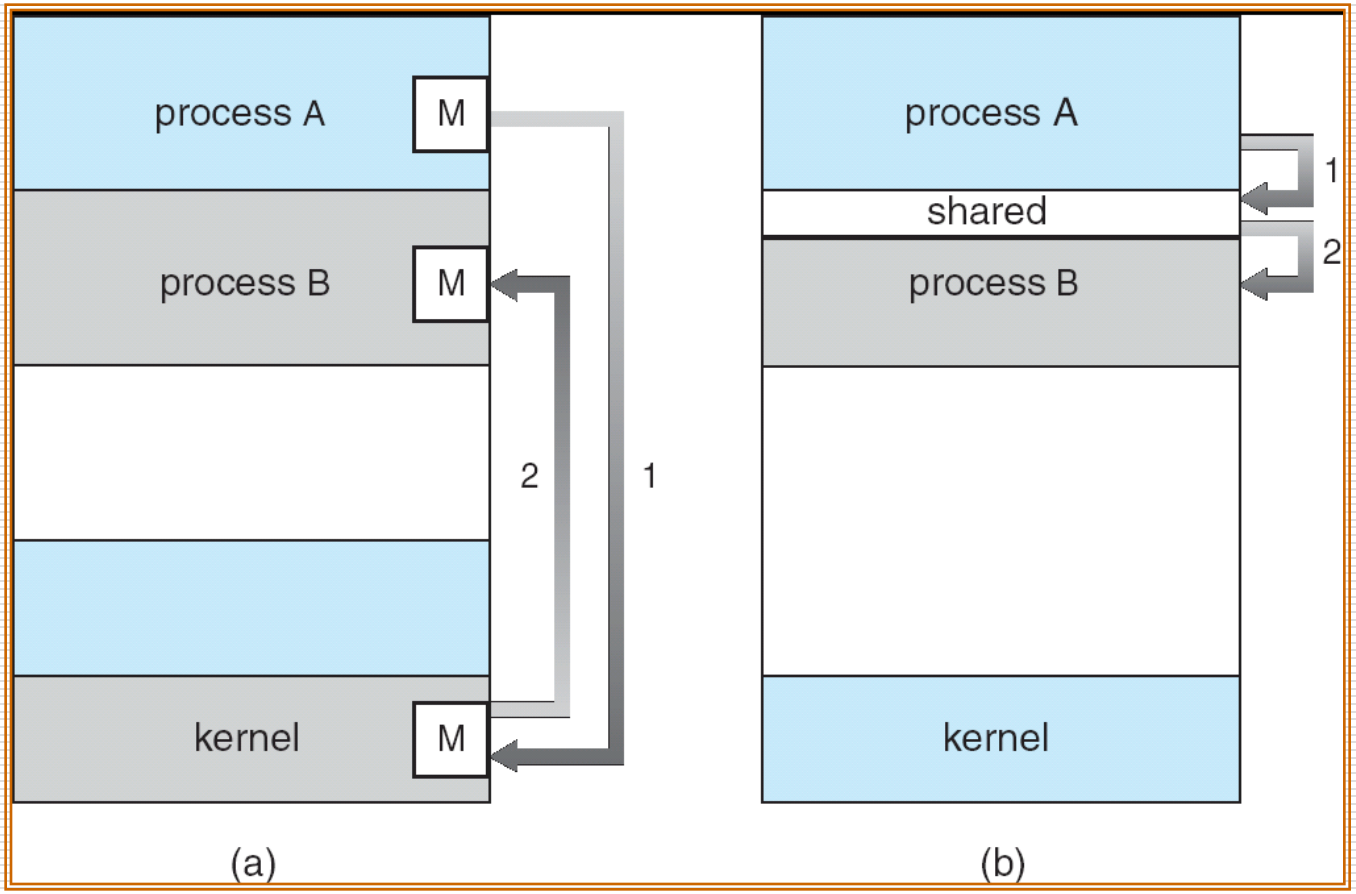
Process Termination

- 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*

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

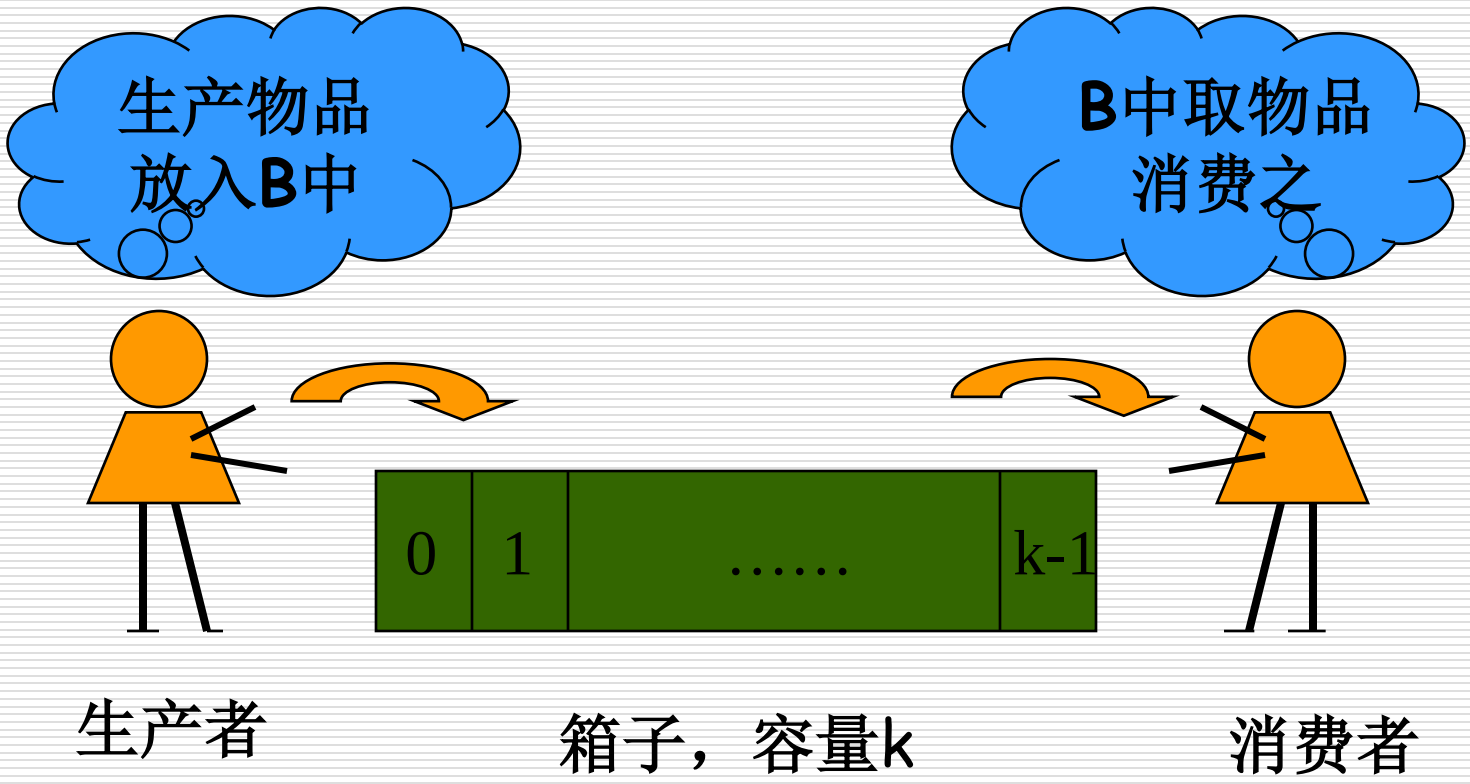
Communications Models



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

Producer-Consumer Problem



B:Array[0..k-1]Of item

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

缓冲区下一个空位

缓冲区第一个非空位

Bounded-Buffer – Insert() Method

```
while (true) {
   /* Produce an item */
   while (((in + 1) % BUFFER SIZE)  == 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;
}
```

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

Message-passing systems

- It provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space
- It is particularly useful in a distributed system.
- Such as the chat program

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, network)
  - logical (e.g., logical properties)

Classification of Message System

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

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 bi-directional

Direct Communication

- Symmetry addressing
- Asymmetry addressing
  - Send(p, message)
  - Receive(id, message)

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** makes the sender blocked until the message is received
  - **Blocking receive** makes the receiver blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send makes the sender to send the message and to continue
  - **Non-blocking** receive makes 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

Examples of IPC systems

- POSIX API
  - First to create a shared memory segment
    - Segment_id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR)
    - The first parameter tells the system to create a new shared memory segment.
    - The second one tells the system the size of the segment to be created.
    - The third one tells the system the rights that users can operate it.
  - Attach the shared memory to process's address space
    - Shared_memory = (char *) shmat(id, NULL, 0)
    - The first parameter is the integer identifier of shared memory segment
    - The second is a pointer location in memory indicating where the shared memory will be attached.
    - The third one defines the mode that the process can access
  - It is removed from the system
    - Shmctl()

P. 104

Examples of IPC systems

- Mach
  - In Mach, messages are sent to and received from mailboxes, called ports in Mach
  - Port_allocate() is used to create a mailbox.
  - For a single user, messages are queued in FIFO order.
  - Msg_send()
  - Msg_receive()
  - Port_status()
- If the mailbox is full, the sending thread has four options:
  - Wait indefinitely until there is room in the mailbox
  - Wait at most n milliseconds
  - Do not wait at all rather return immediately
  - Temporarily cache a message.

Examples of IPC systems

- Windows XP
  - The message-passing facility in XP is called the local procedure-call (LPC)
  - XP uses a port object to establish and maintain a connection between two processes
- The communication works as follows:
  - The client opens a handle to the subsystem's connection port object
  - The client sends a connection request
  - The server creates ports, and returns the handle to the client
  - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

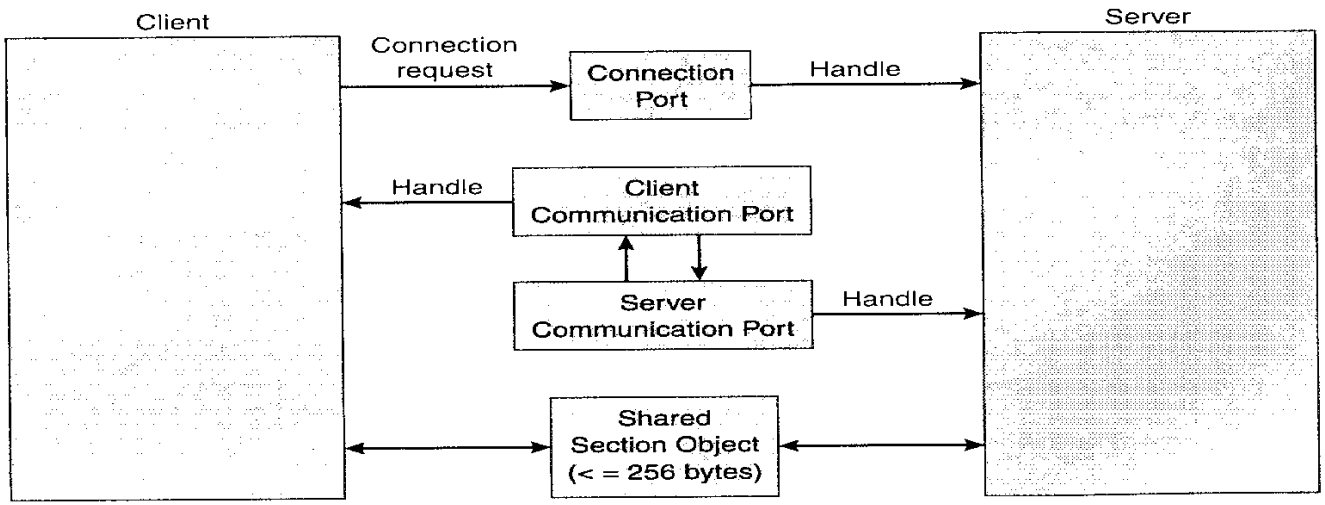
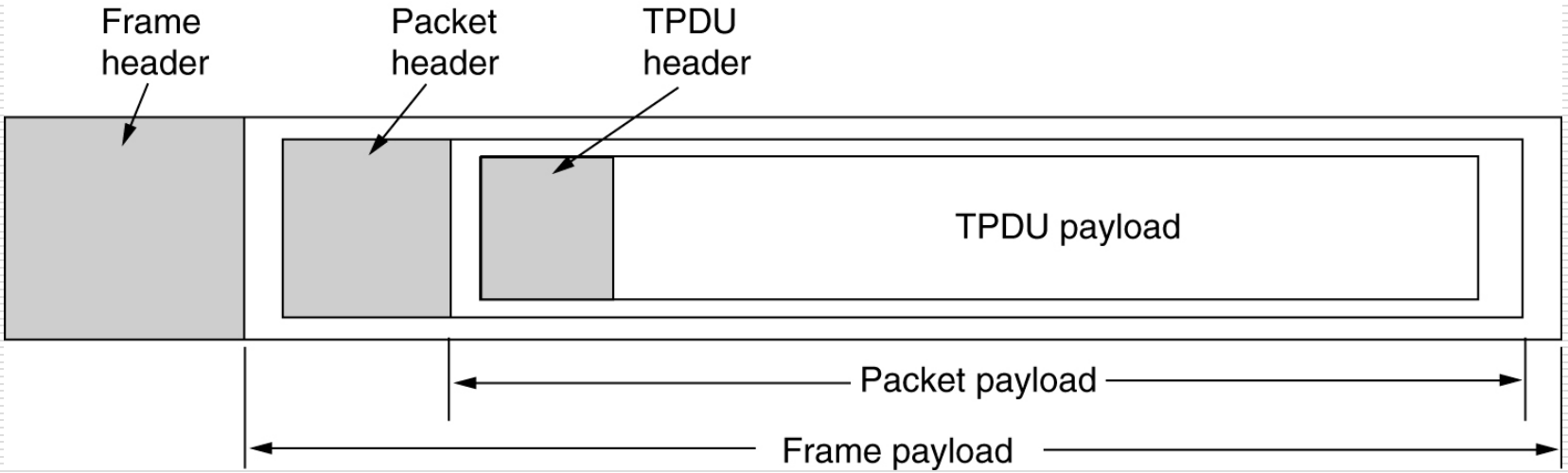


Figure 3.17 Local procedure calls in Windows XP.

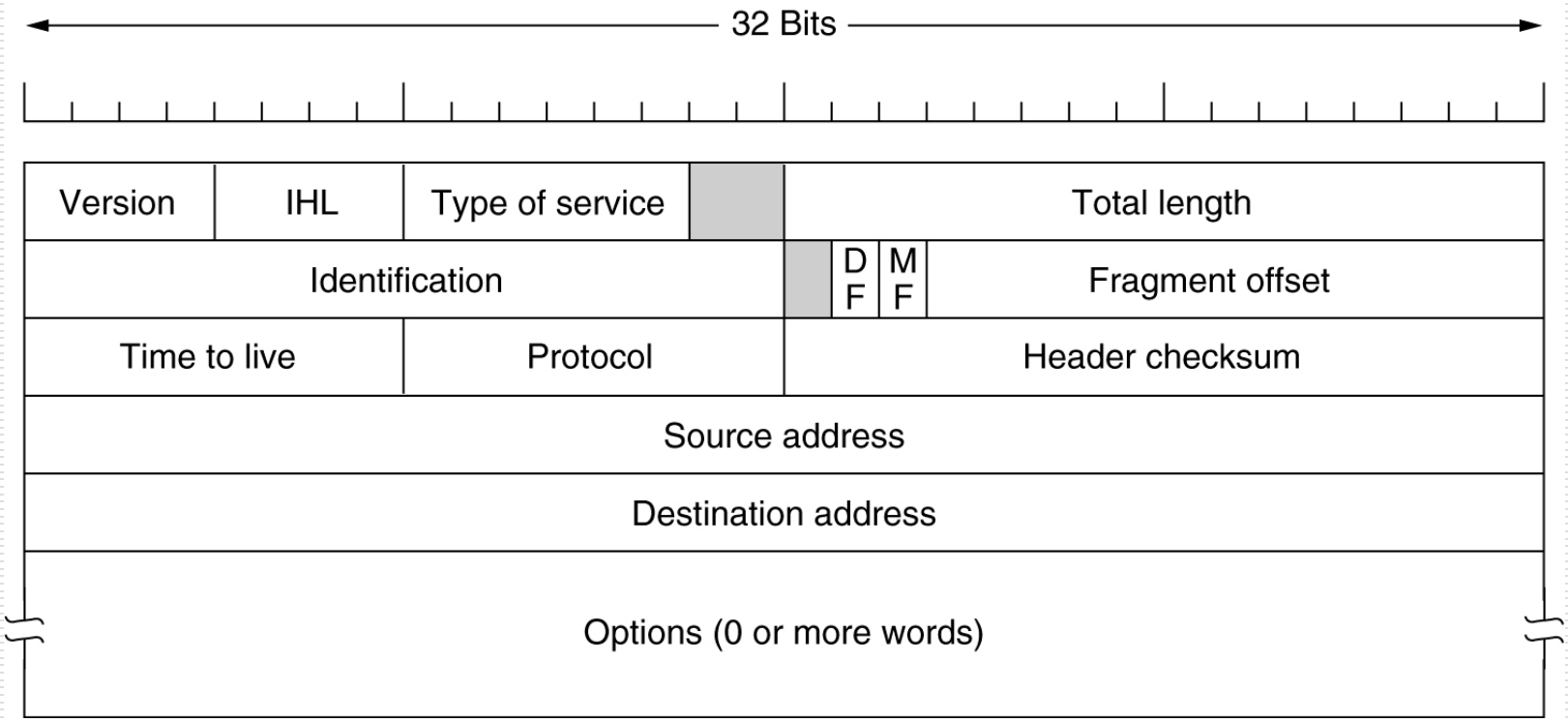
Client-Server Communication

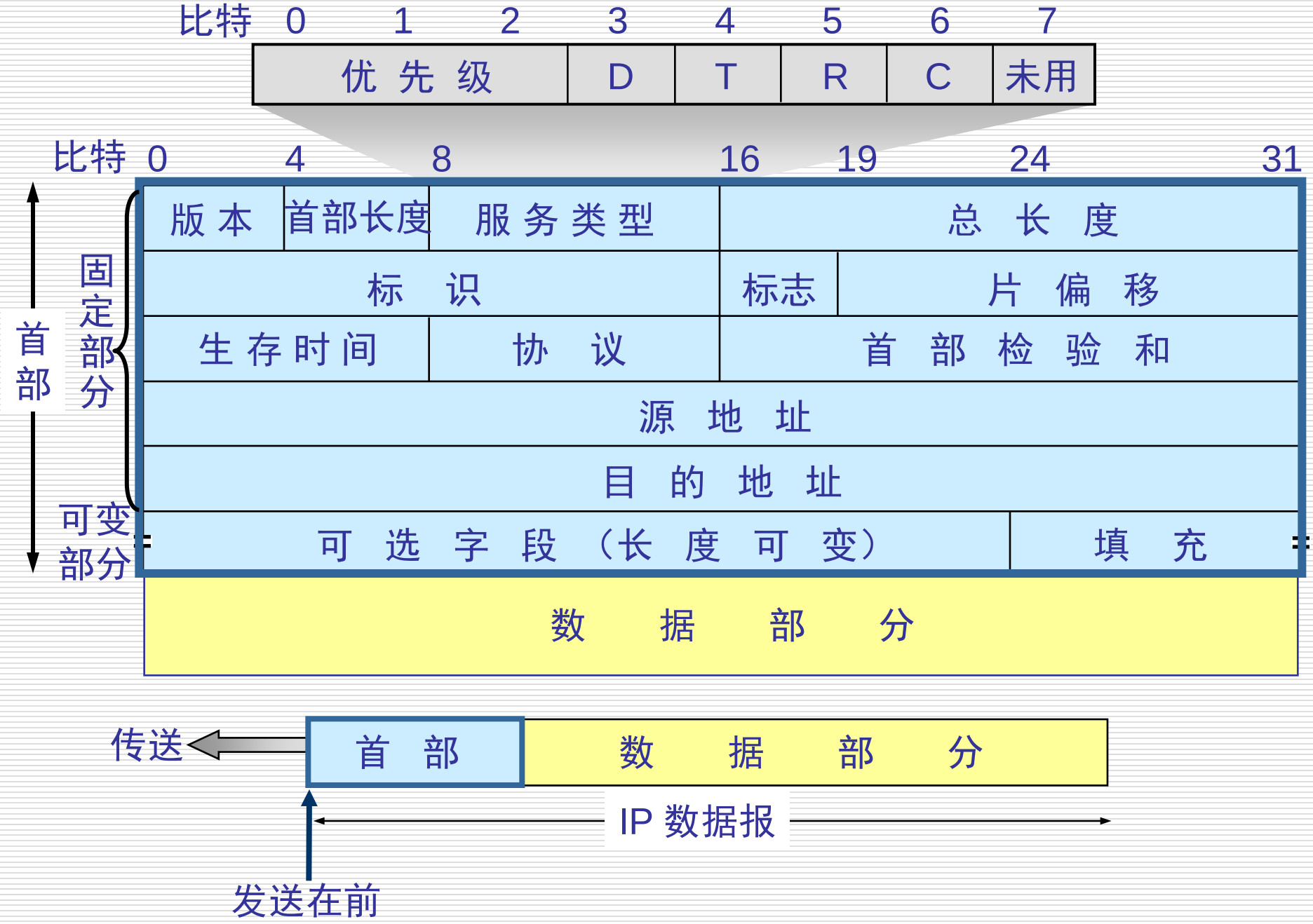
- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Data format on the Internet

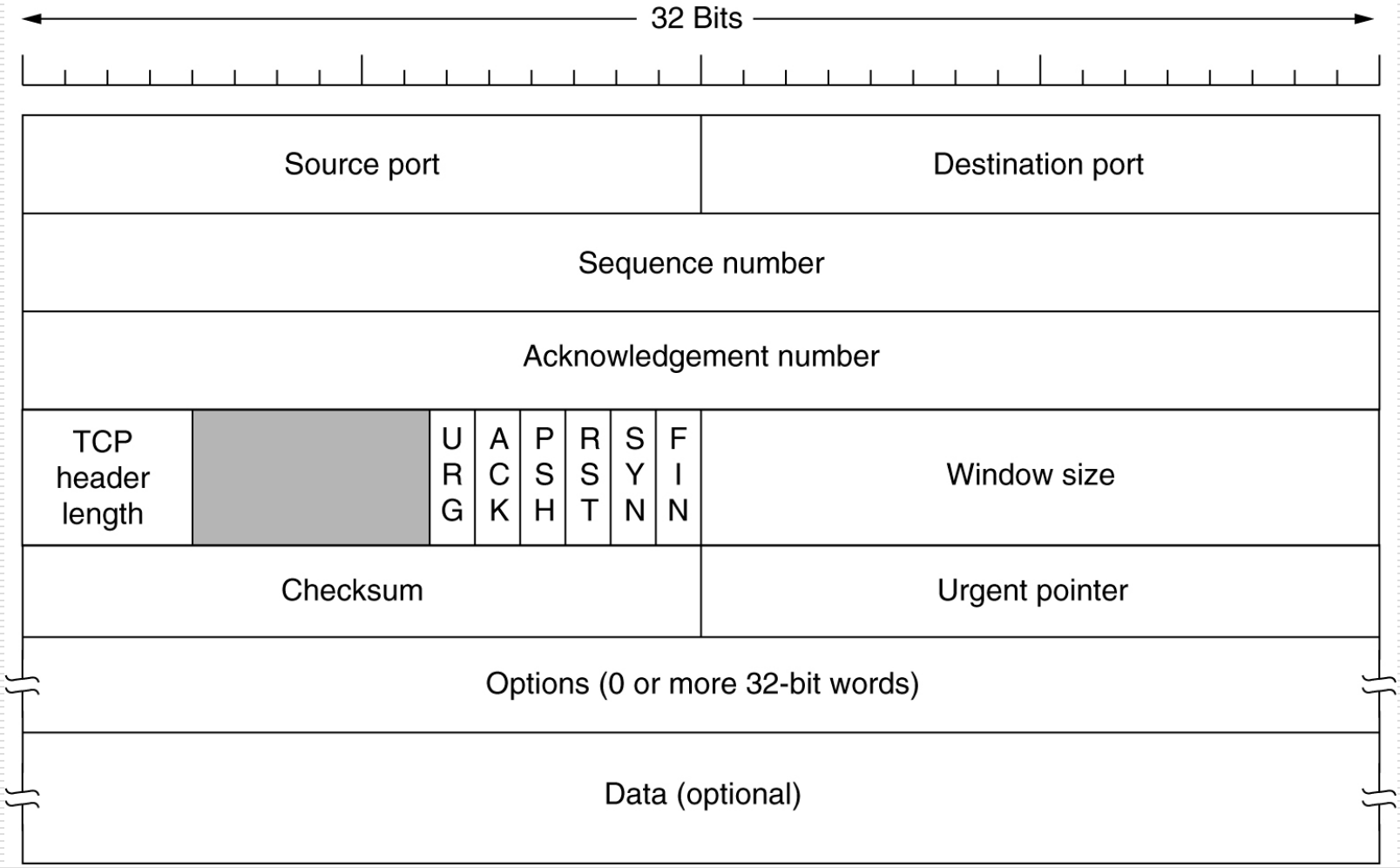


IP Header

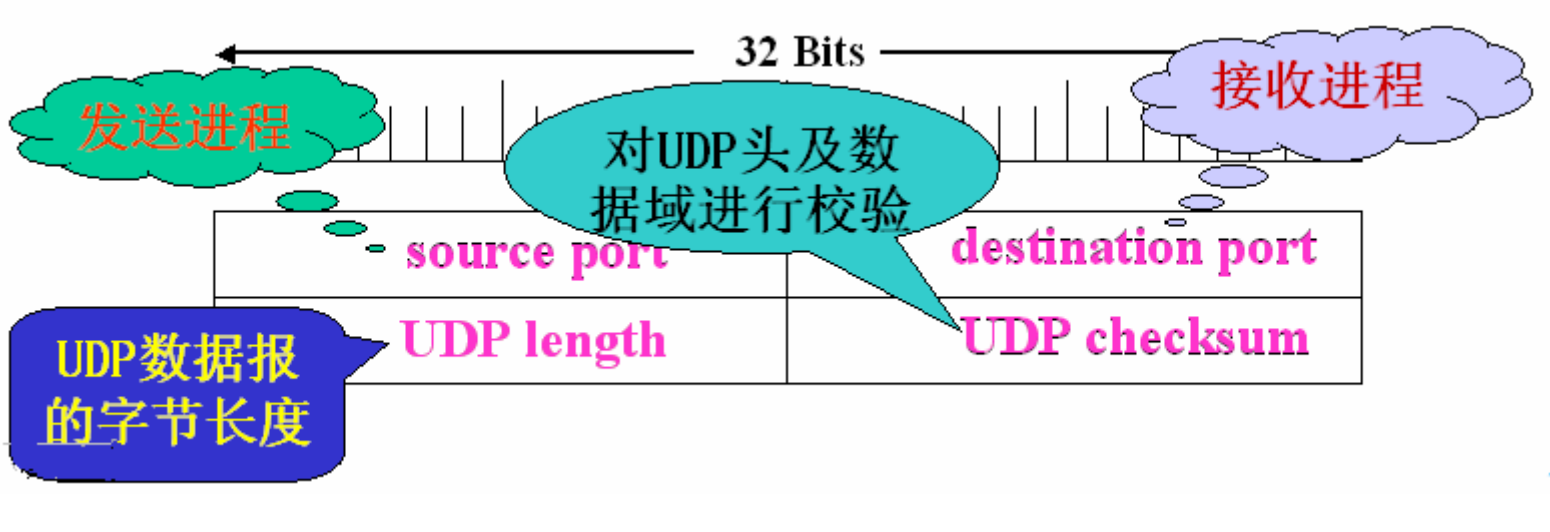

32 Bits
Version
IHL
Type of service
Total length
Identification
D F
M F
Fragment offset
Time to live
Protocol
Header checksum
Source address
Destination address
Options (0 or more words)


比特 0 1 2 3 4 5 6 7
优 先 级
D
T
R
C
未用
比特 0 4 8 16 19 24 31
版 本
首部长度
服 务 类 型
总 长 度
标 识
标志
片 偏 移
生 存 时 间
协 议
首 部 检 验 和
源 地 址
目 的 地 址
可 选 字 段 （长 度 可 变）
填 充
数 据 部 分
首部
固定部分
可变部分
传送
首 部
数 据 部 分
IP 数据报
发送在前

TCP Header


32 Bits
Source port
Destination port
Sequence number
Acknowledgement number
TCP header length
URG
ACK
PSH
RST
SYN
FIN
Window size
Checksum
Urgent pointer
Options (0 or more 32-bit words)
Data (optional)

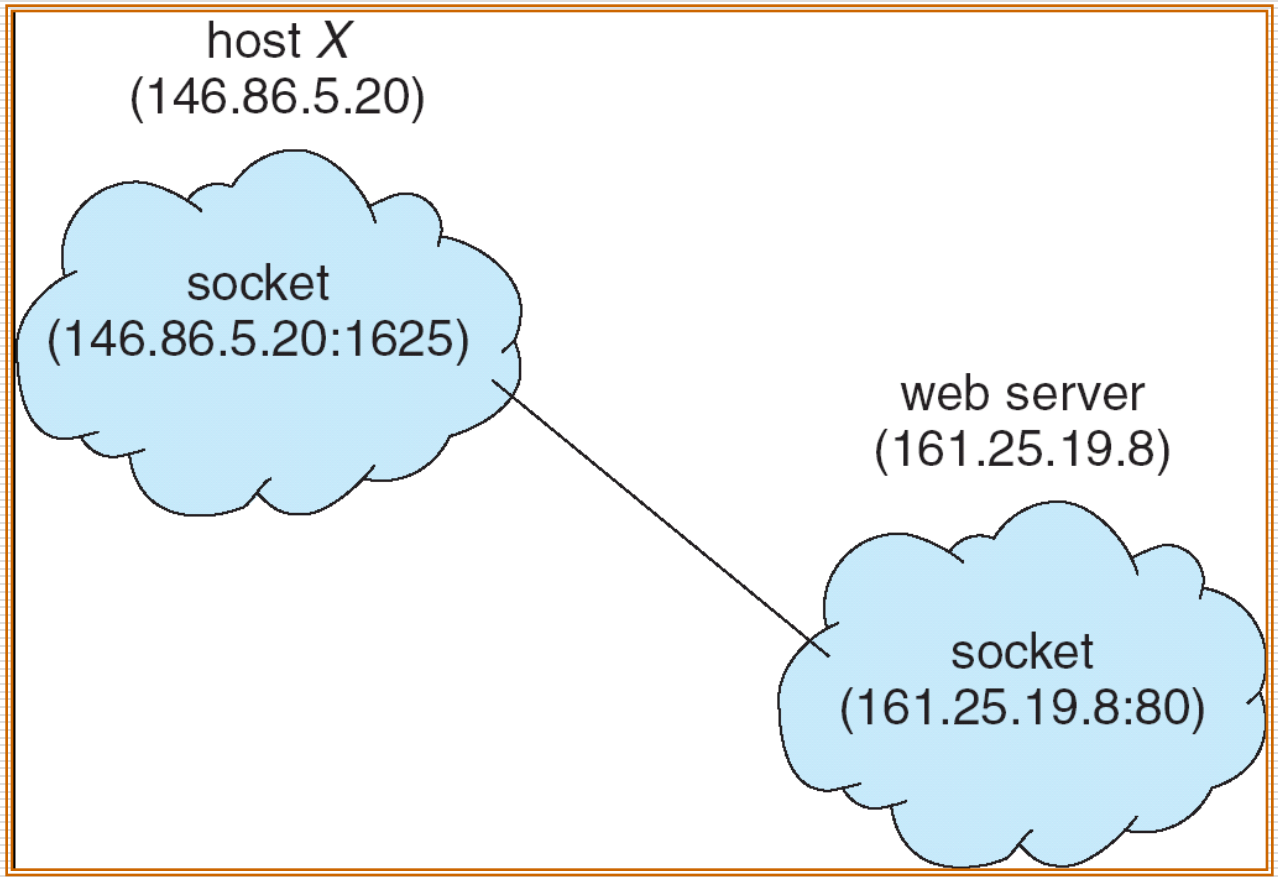
UDP Header


32 Bits
发送进程
对UDP头及数据域进行校验
接收进程
source port
destination port
UDP length
UDP checksum
UDP数据报的字节长度

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

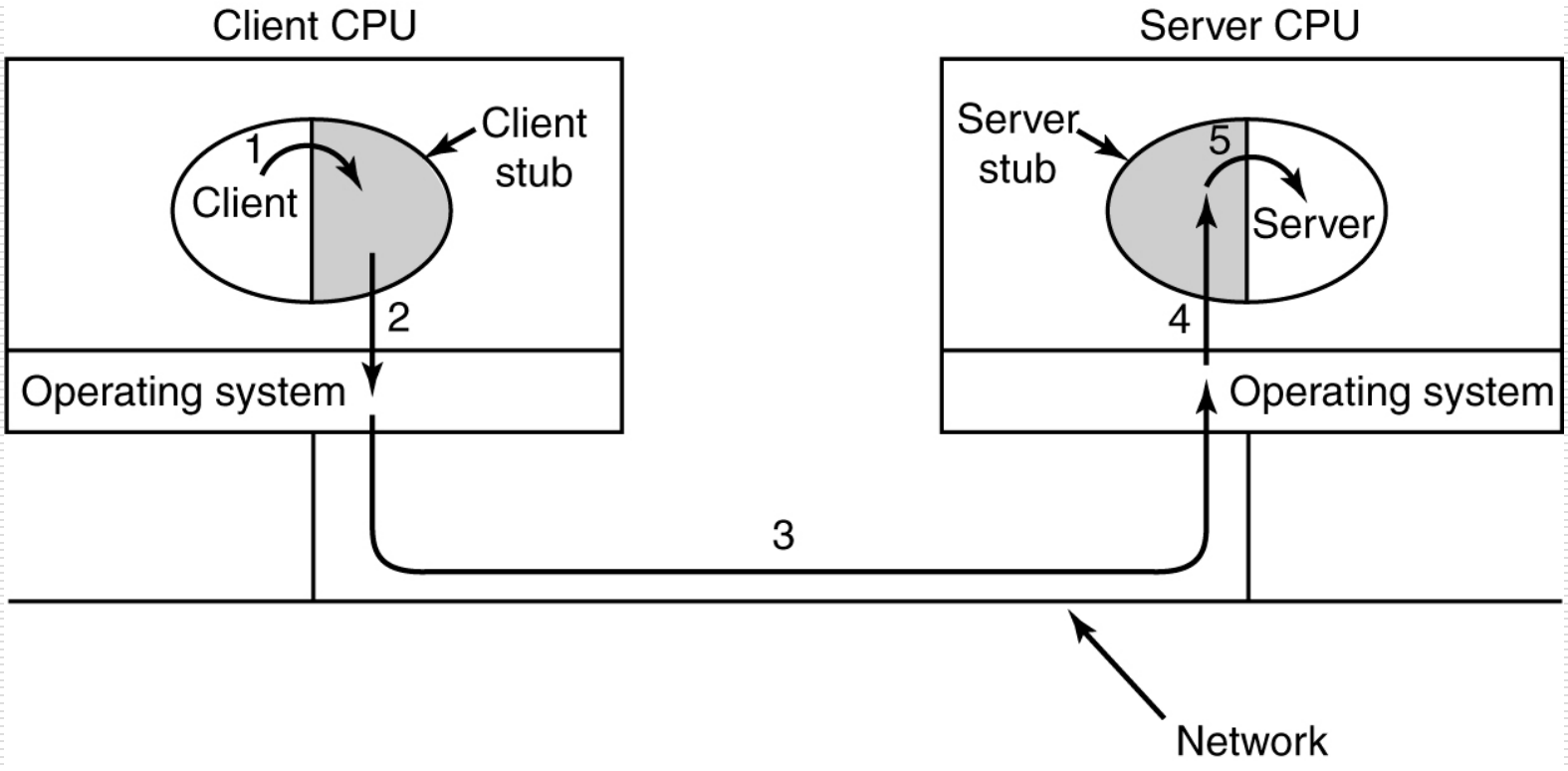


- P.109&110

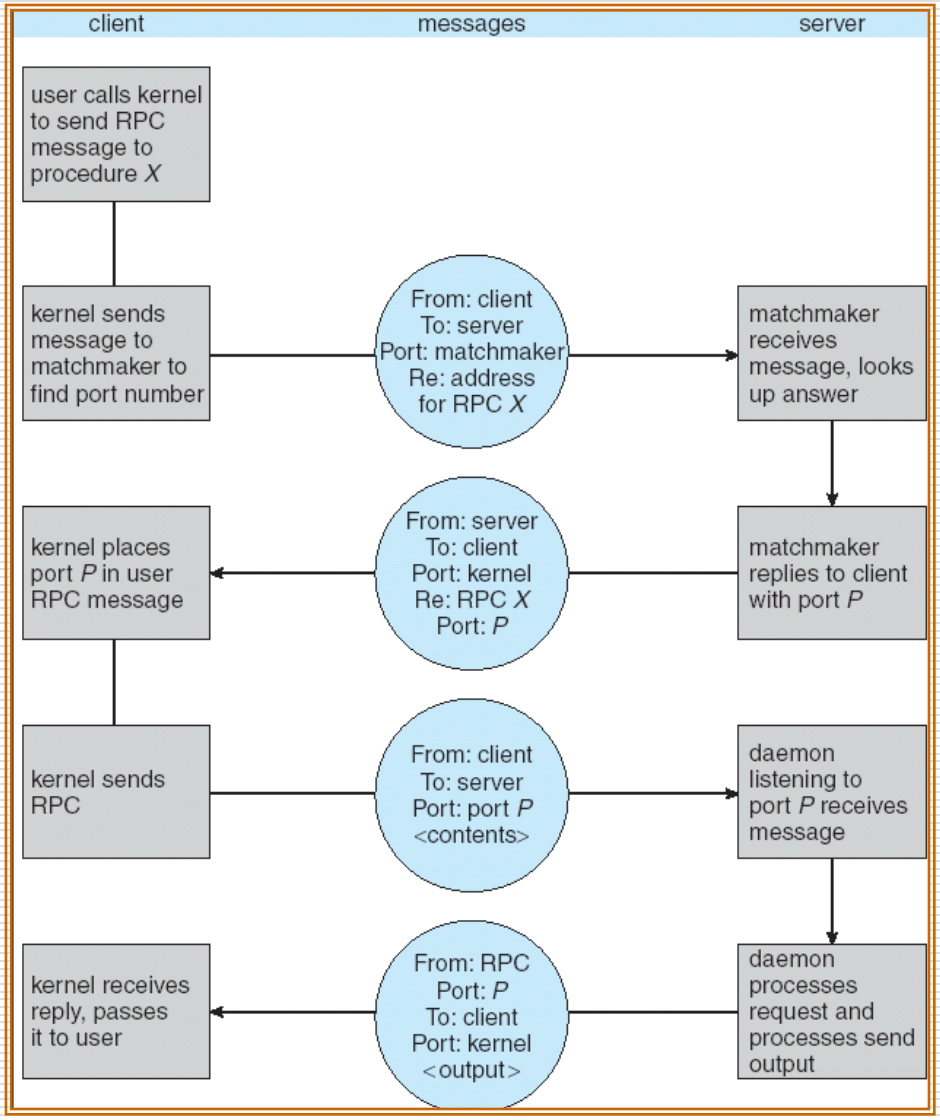
Remote Procedure Calls

- 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.

Remote Procedure Call

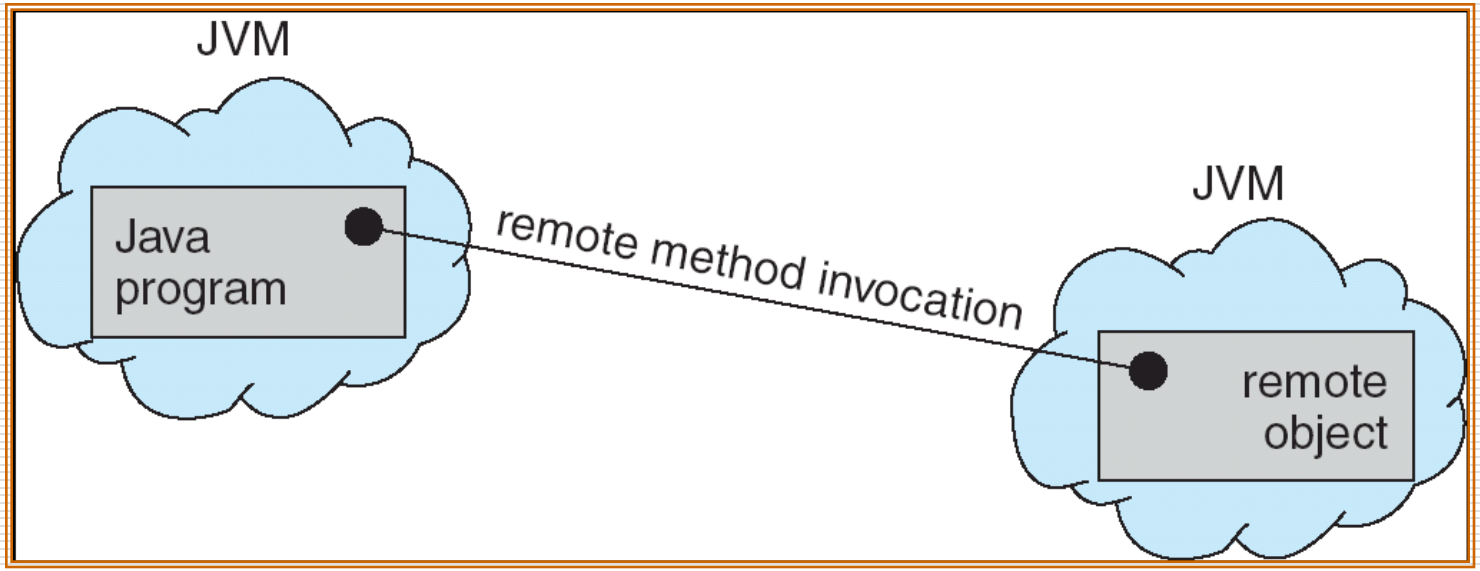


Execution of RPC

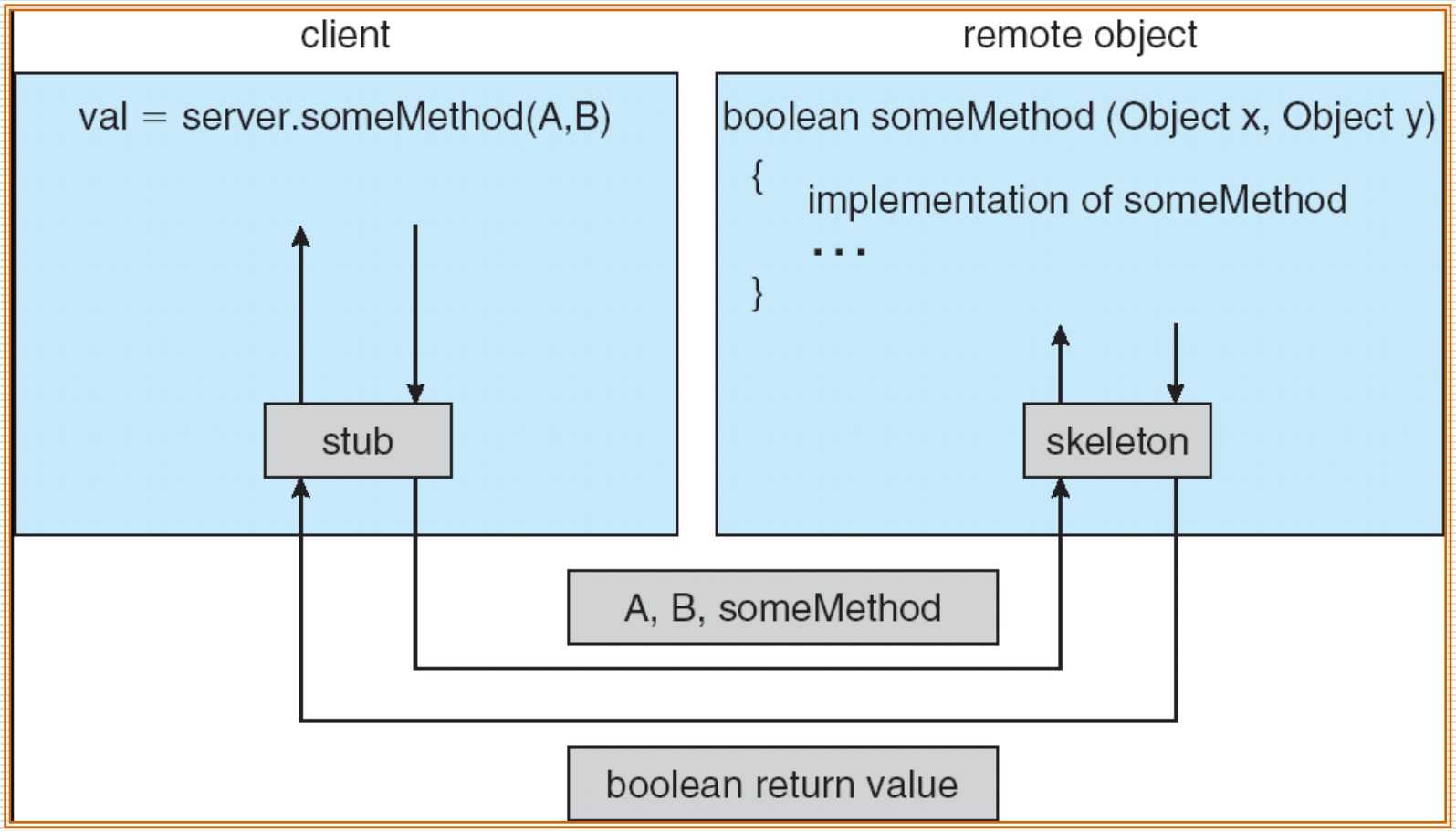


Remote Method Invocation

- 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



Assignment

- 3.2, 3.4,

End of Chapter 3

Any Question?