# Chapter 3　Processes

# Chapter 3   Processes

# 3.1 Process Concept

- **A process is a program in execution** (an active entity, i.e. it is a *running* program )

  - Basic unit of work on a computer, a job, a task.

  - A container of instructions with some resources:– e.g. CPU time (CPU carries out the instructions), memory, files, I/O devices to accomplish its task

  - Examples: compilation process, word processing process, scheduler (sched, swapper) process or daemon processes: ftpd, httpd

- jobs作业=user programs用户程序= tasks任务= *process* 进程

IBM

Multics

# Process Concept

- **进程是什么？**
  - 一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。
  - 正在执行中的程序 a program in execution

(1) 进程是可以并行执行的计算部分(S. E. Madnick, J. T. Donovan)；

(2) 进程是一个独立的可以调度的活动(E. Cohen, D. Jofferson)；

(3) 进程是一抽象实体，当它执行某个任务时，将要分配和释放各种资源(P. Denning)；

(4) 行为的规则叫程序，程序在处理机上执行时的活动称为进程(E. W. Dijkstra)；

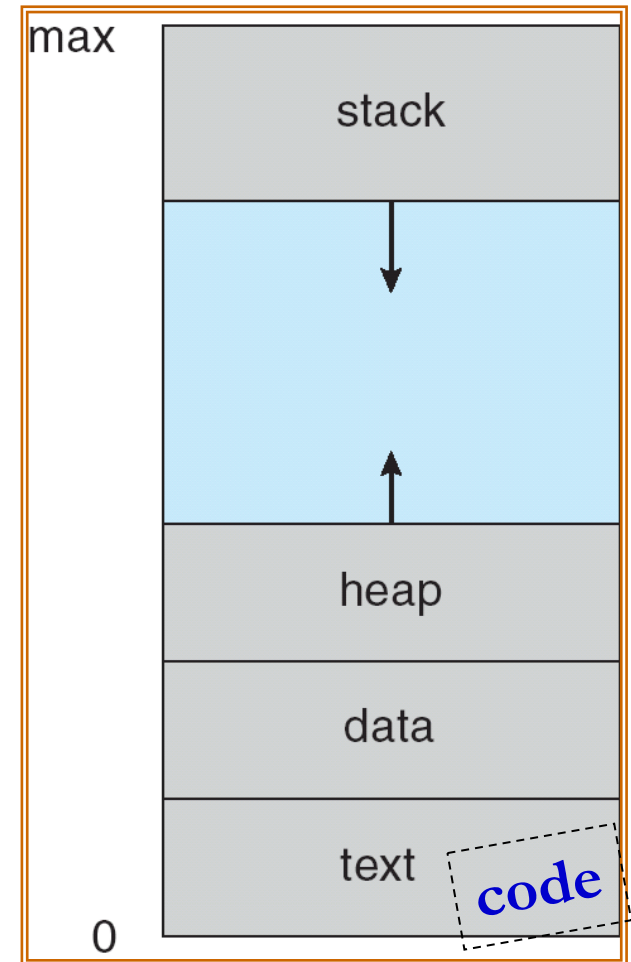(5) 一个进程是一系列逐一执行的操作，而操作的确切含义则有赖于以何种详尽程度来描述进程(Brinch Hansen)，等等。

张学尧，计算机操作系统教程，清华大学出版社

# Process in Memory

- A process includes:
  - Program counter (PC)
  - Registers
  - Data section (global data)
  - Stack (temporary data)
  - Heap
    (dynamically allocated memory)

max

stack

↓

↑

heap

data

text

*code*

0

# **Process State** 进程状态

■ As a process executes, it changes *state*

- New（新）: The process is being created.

- Running（运行、执行）: Instructions are being executed.

- Ready（就绪）: The process is waiting to be assigned to a processor (CPU).

- Waiting（等待、blocked阻塞）: The process is waiting for some event to occur.

- Terminated（终止）: The process has finished execution.

# State Transitions

- **A process may change state as a result:**
  - Program action (system call)
  - OS action (scheduling decision)
  - External action (interrupts)
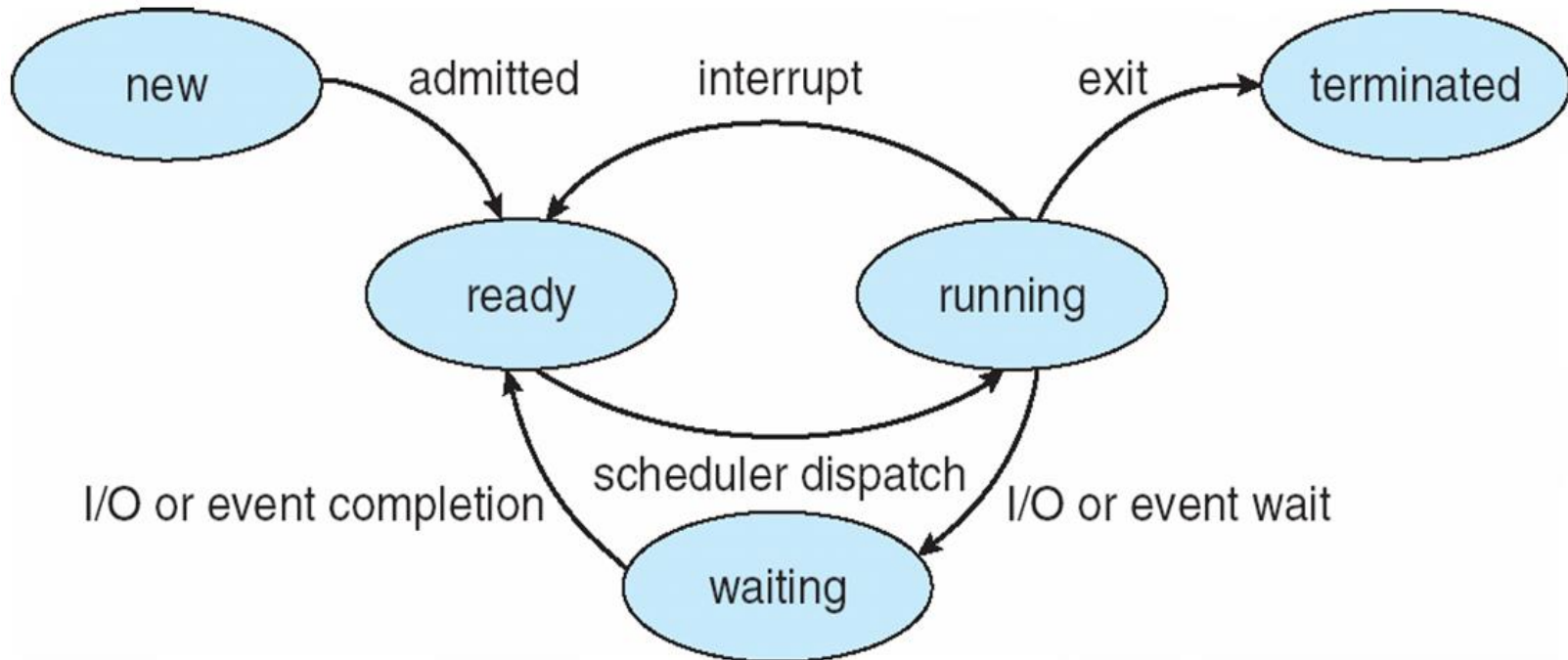
# 进程状态的转换

- 三个基本状态之间可能转换和转换原因如下：
  - 就绪→运行：当处理机空闲时，进程调度程序必将处理机分配给一个处于就绪状态的进程，该进程便由就绪状态转换为运行状态。
  - 运行→等待：处于运行状态的进程在运行过程中需要等待某一事件发生后（例如因I／O请求等待I／O完成后），才能继续运行，则该进程放弃处理机，从运行状态转换为等待状态。
  - 等待→就绪：处于等待状态的进程，若其等待的事件已经发生，于是进程由等待状态转换为就绪状态。
  - 运行→就绪：处于运行状态的进程在其运行过程中，因分给它的处理机时间片已用完，而不得不让出（被抢占）处理机，于是进程由运行态转换为就绪态。

- 等待→运行，就绪→等待这二种状态转换一般不可能发生。

# 进程状态与处理机

- **处于运行状态进程**:如系统有一个处理机，则在任何一时刻，最多只有一个进程处于运行状态。

- **处于就绪状态进程**:一般处于就绪状态的进程按照一定的算法（如先来的进程排在前面，或采用优先权高的进程排在前面）排成一个就绪队列。

- **处于等待状态进程**:处于等待状态的进程排在等待队列中。由于等待事件原因不同，等待队列也可以按事件分成几个队列。

# Process State

- Windows NT/2000 线程有**7种**状态；Linux进程有6种状态；Windows 2003 server线程有**9种**状态.

- include/linux/sched.h（2.6.15）

  ```
  124 #define TASK_RUNNING            0
  125 #define TASK_INTERRUPTIBLE      1
  126 #define TASK_UNINTERRUPTIBLE    2
  127 #define TASK_STOPPED            4
  128 #define TASK_TRACED             8
  129 /* in tsk->exit_state */
  130 #define EXIT_ZOMBIE             16
  131 #define EXIT_DEAD               32
  132 /* in tsk->state again */
  133 #define TASK_NONINTERACTIVE     64
  ```

# include/linux/sched.h（4.2）

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED          4
#define __TASK_TRACED8
/* in tsk->exit_state */
#define EXIT_DEAD               16
#define EXIT_ZOMBIE             32
#define EXIT_TRACE              (EXIT_ZOMBIE | EXIT_DEAD)
/* in tsk->state again */
#define TASK_DEAD               64
#define TASK_WAKEKILL 128
#define TASK_WAKING             256
#define TASK_PARKED             512
#define TASK_NOLOAD             1024
#define TASK_STATE_MAX          2048
#define TASK_STATE_TO_CHAR_STR "RSDTtXZxKWPN"
extern char ___assert_task_state[1 - 2*!!(
            sizeof(TASK_STATE_TO_CHAR_STR)-1 != ilog2(TASK_STATE_MAX)+1)];
/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE           (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED            (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED             (TASK_WAKEKILL | __TASK_TRACED)

……
```

# 进程与程序

■ **进程与程序的区别**

● 进程是动态的，程序是静态的：程序是有序代码的集合；进程是程序的执行。

● 进程是暂时的，程序的永久的：进程是一个状态变化的过程，程序可长久保存。

● 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）。

● 进程与程序的对应关系：通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序

# 实例

■ 例：一个只有一个处理机的系统中，OS的进程有运行、就绪、等待三个基本状态。假如某时刻该系统中有10个进程并发执行，在略去调度程序所占用时间情况下（在user mode下），请问：

- 这时刻系统中处于运行状态的进程数最多有几个？最少有几个？  1  0
- 这时刻系统中处于就绪状态的进程数最多有几个？最少有几个？  9  0
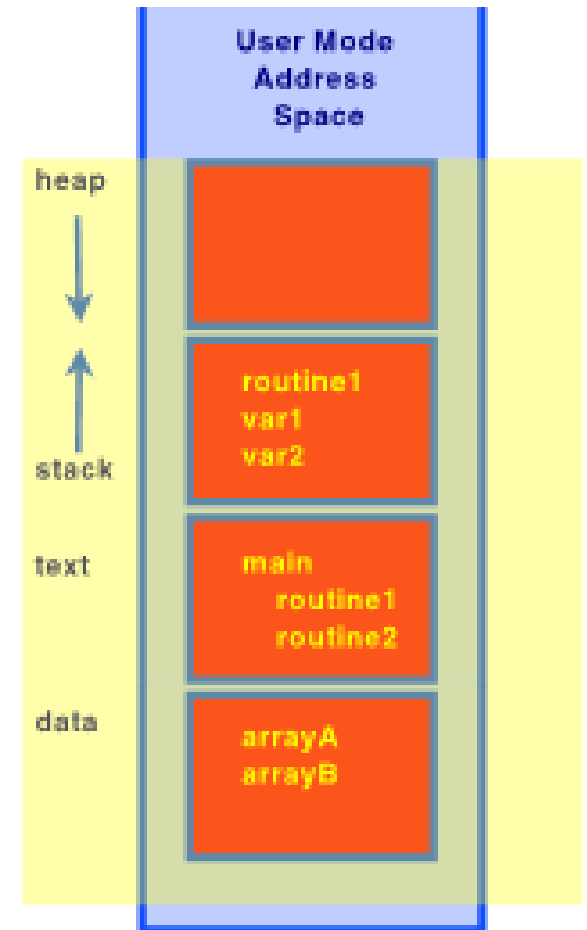- 这时刻系统中处于等待状态的进程数最多有几个？最少有几个？  10  0

# 3.1.3 What Makes up a Process?

User resources/OS Resources:

- **Program code (text)**

- **Data**
  - global variables
  - heap (dynamically allocated memory)

- **Process stack**
  - » function parameters
  - » return addresses
  - » local variables and functions

- **OS Resources, environment**
  - open files, sockets
  - Credential for security

- **Registers**
  - program counter, stack pointer

*PCB*



User Mode Address Space

| | |
|---|---|
| heap | |
| stack | routine1<br>var1<br>var2 |
| text | main<br>routine1<br>routine2 |
| data | arrayA<br>arrayB |

address space are the shared resources of a(ll) thread(s) in a program

# Process Control Block（PCB，进程控制块）

■ **What is needed to keep track of a Process?**

■ 每个进程在操作系统内用进程控制块来表示，它包含与特定进程相关的许多信息：

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

| process state |
| :---: |
| process number |
| program counter |
| registers |
| memory limits |
| list of open files |
| • • • |
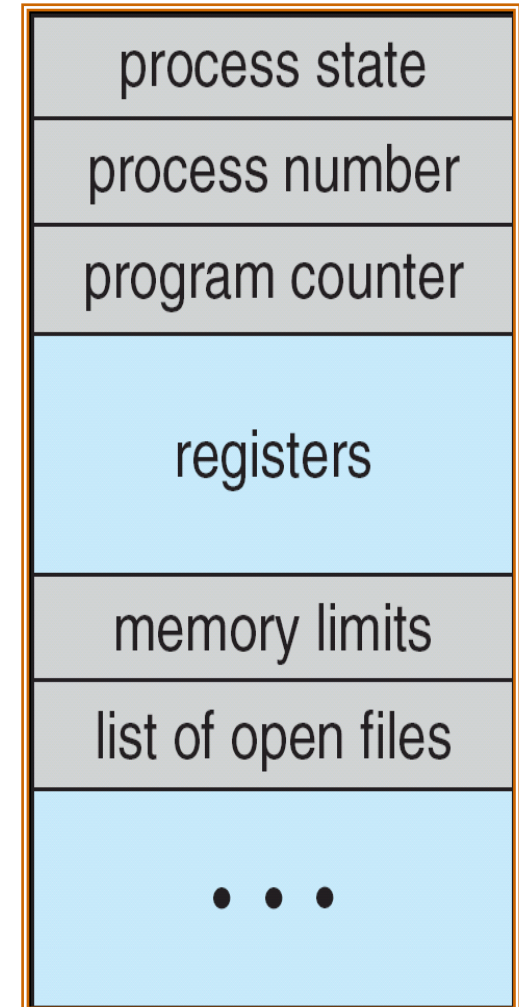
- In Linux a process' information is kept in a structure called <span style="color:red">struct task_struct</span> declared in

  #include/linux/sched.h

  struct task_struct

  pid_t pid; /* process identifier */

  long state; /* state for the process */

  unsigned int time_slice /* scheduling information */

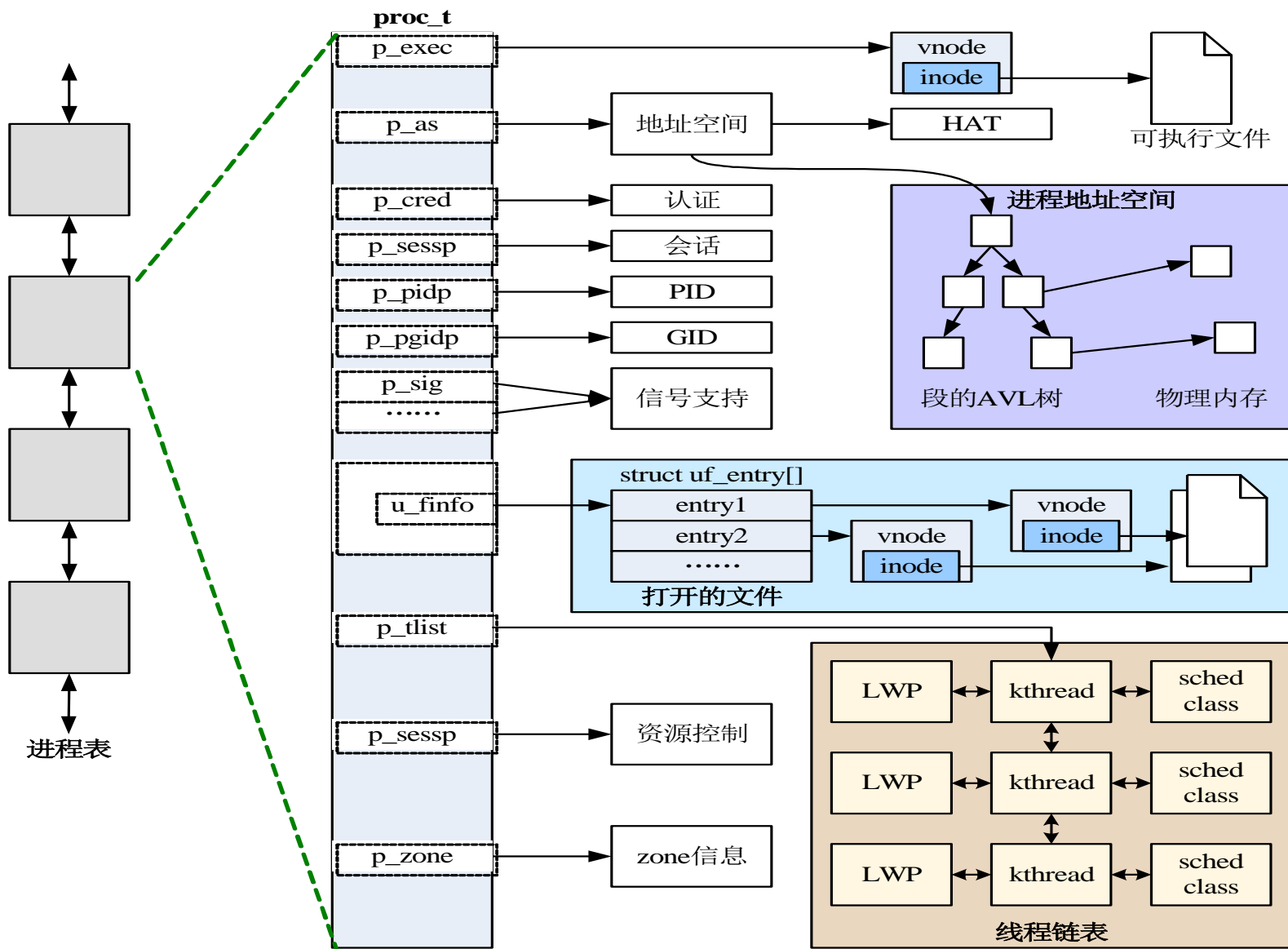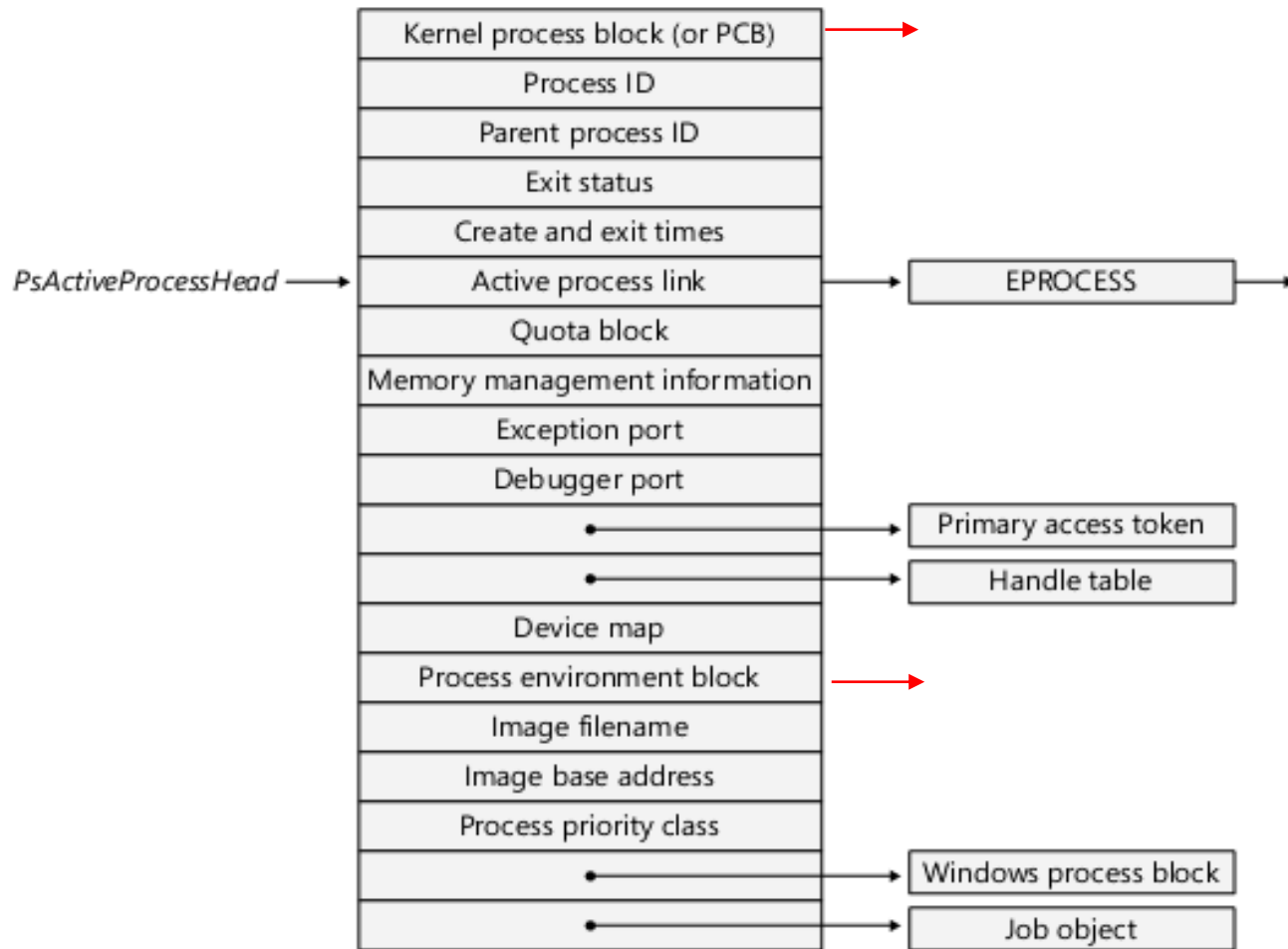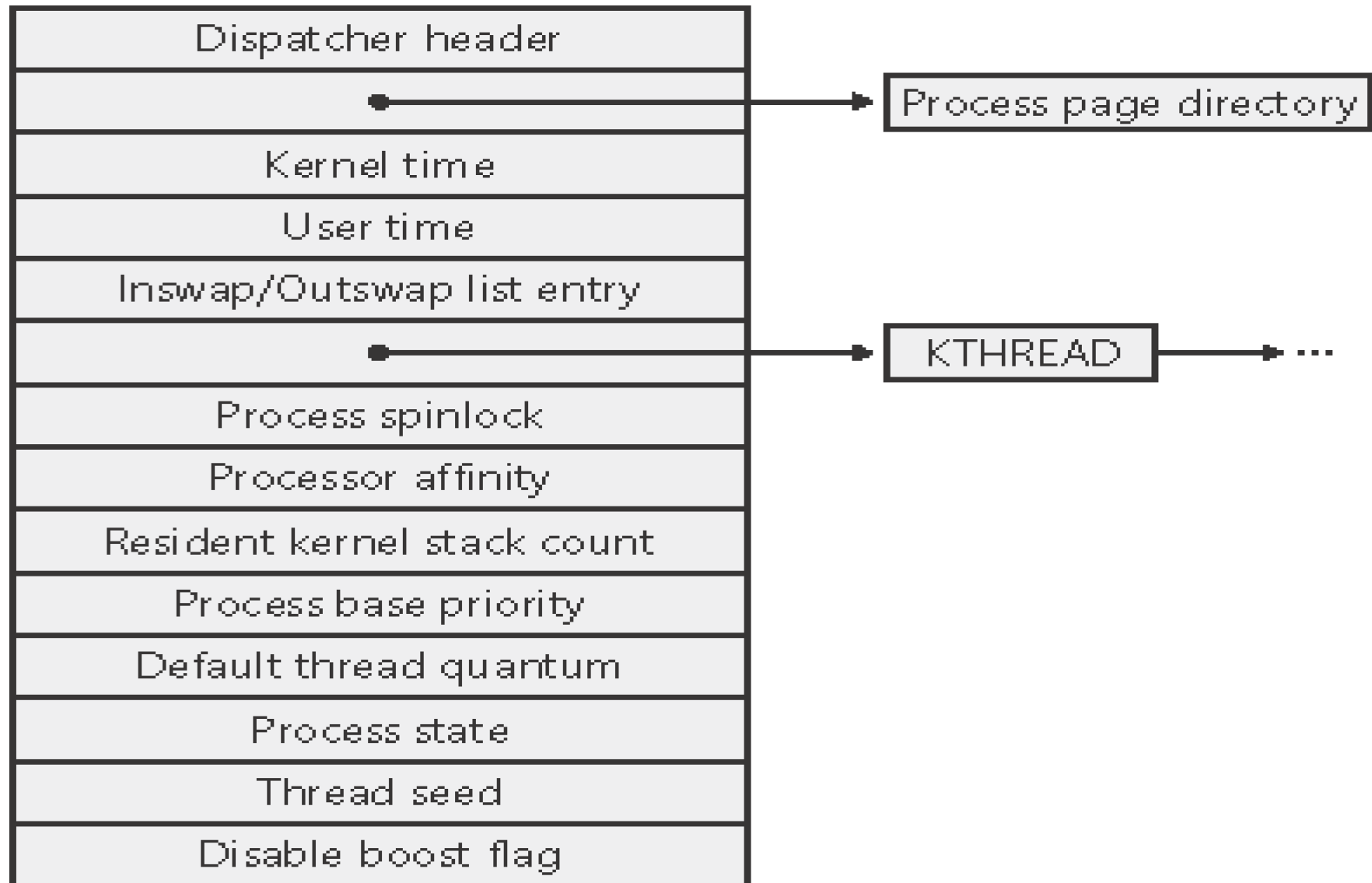  struct mm_struct *mm /* address space of this process */

  ……

# Open Solaris *proc*

# Windows *executive process block*

# Windows kernel process (KPROCESS) block

| |
|---|
| Dispatcher header |
| ● → Process page directory |
| Kernel time |
| User time |
| Inswap/Outswap list entry |
| ● → KTHREAD → ... |
| Process spinlock |
| Processor affinity |
| Resident kernel stack count |
| Process base priority |
| Default thread quantum |
| Process state |
| Thread seed |
| Disable boost flag |

| |
|---|
| Image base address |
| Module list |
| Thread-local storage data |
| Code page data |
| Critical section timeout |
| Number of heaps |
| Heap size information |
| ● |
| GDI shared handle table |
| Operating system version number information |
| Image version information |
| Image process affinity mask |

Process heap

# 3.2 Process Scheduling 进程调度
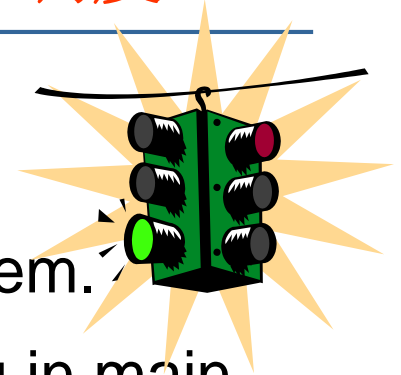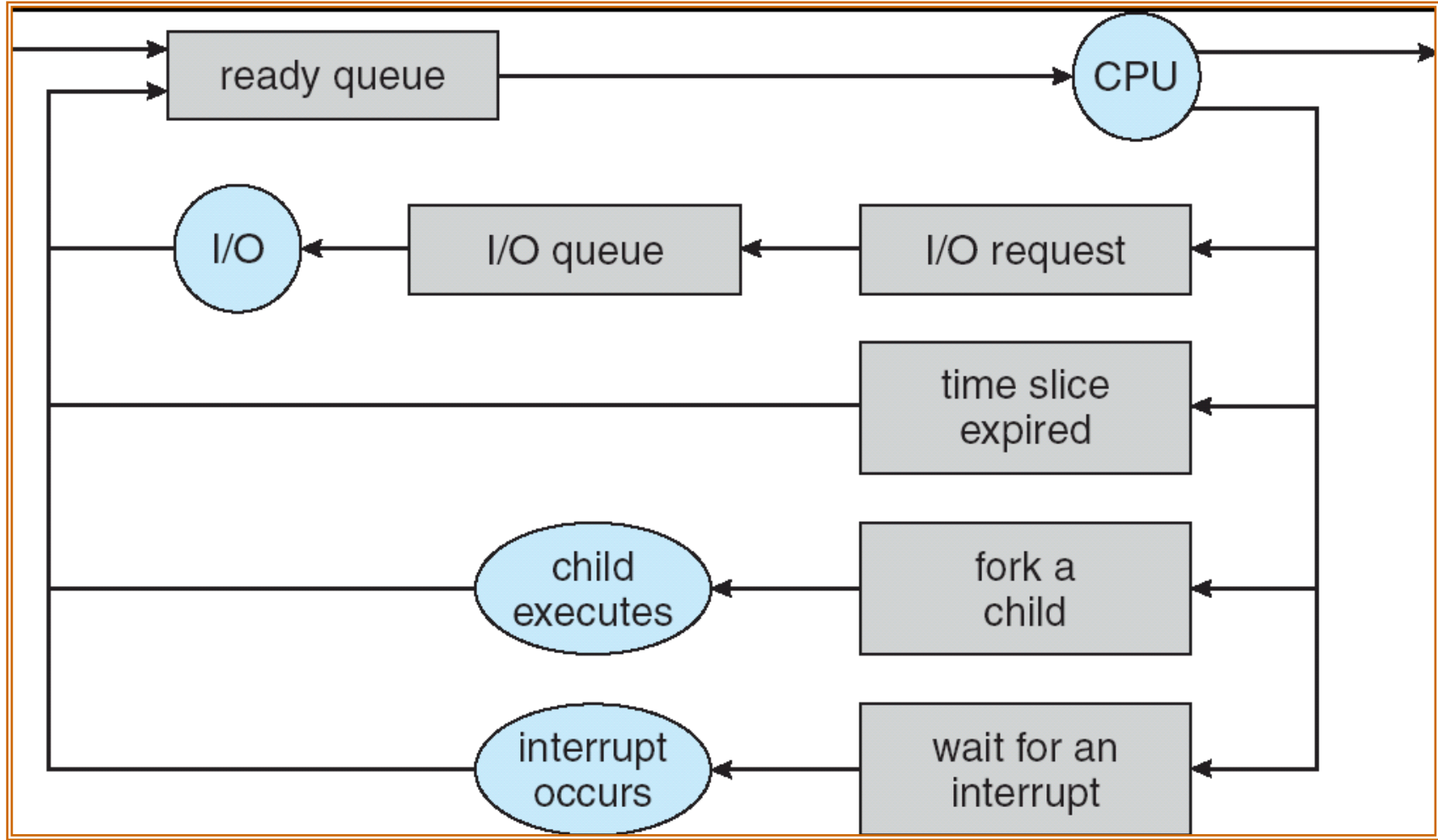
3.2.1 Scheduling Queues

- **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.

# Fig 3.4 Ready Queue And Various I/O Device Queues

# Fig 3.5 Representation of Process Scheduling

# 3.2.2 Schedulers

- Long-term scheduler (or job scheduler)长程调度（或作业调度）
  - selects which processes should be brought into the ready queue.
  - invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - controls the *degree of multiprogramming*多道程序的"道"
  - Most modern operating systems have no long-term scheduler (e.g. Windows, UNIX,Linux)

- Short-term scheduler (or CPU scheduler)短程调度（或CPU调度）
  - selects which process should be executed next and allocates CPU.
  - invoked very frequently (milliseconds) $\Rightarrow$ (must be fast).

- Medium-Term Scheduler 中程调度

# Fig 3.6 Addition of Medium Term Scheduling
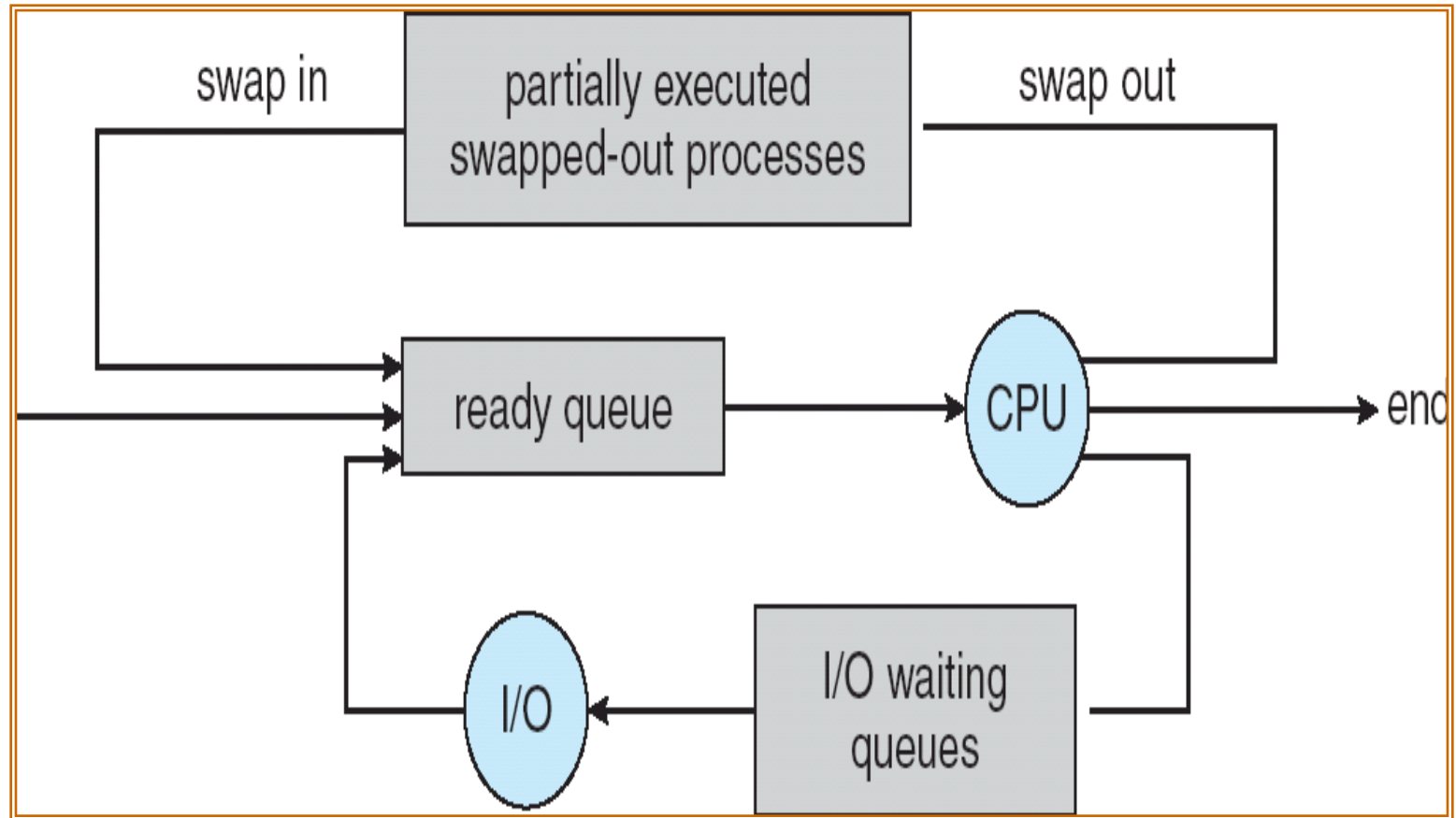
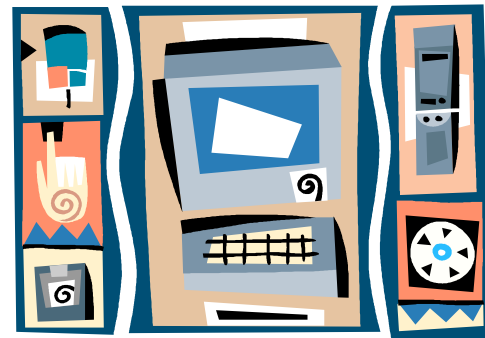- **Processes can be described as either:**
  - I/O-*bound process* （I/O型进程）
    - ▸ spends more time doing I/O than computations, many short CPU bursts.
  - *CPU-bound process* （CPU 型进程）
    - ▸ spends more time doing computations; few very long CPU bursts.

# 3.2.3 Context Switch（上下文切换）
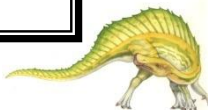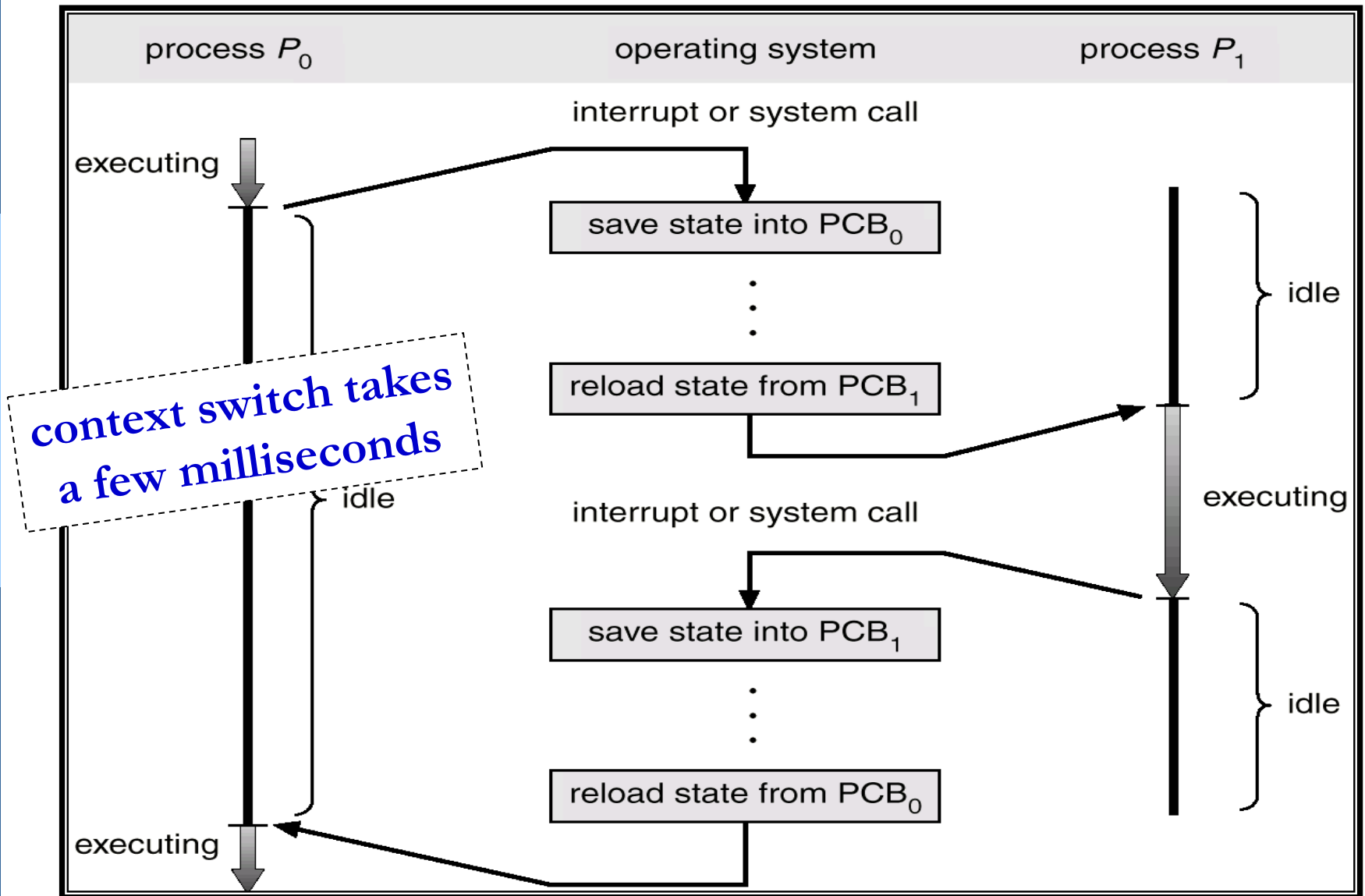
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch

- Context of a process represented in the PCB

- Context-switch time is overhead; the system does no useful work while switching

- Time dependent on hardware support

context switch takes a few milliseconds

# 3.3 Operating on Processes 进程操作

## 3.3.1 Process Creation进程创建

- 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：
  - 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.

■ Address space 地址空间

- Child duplicate of parent.
- Child has a program loaded into it.

# Process Creation (Cont.)

■ UNIX examples

- ● **fork** system call creates new process
  - ‣ int pid1 = fork();
  - ‣ 从系统调用 fork 中返回时，两个进程除了返回值 pid 1不同外，具有完全一样的用户级上下文。在子进程中，pid1 的值为0;父进程中，pid 1的值为子进程的进程号。

- ● **exec** system call used after a fork to replace the process' memory space with a new program.

# Process Creation
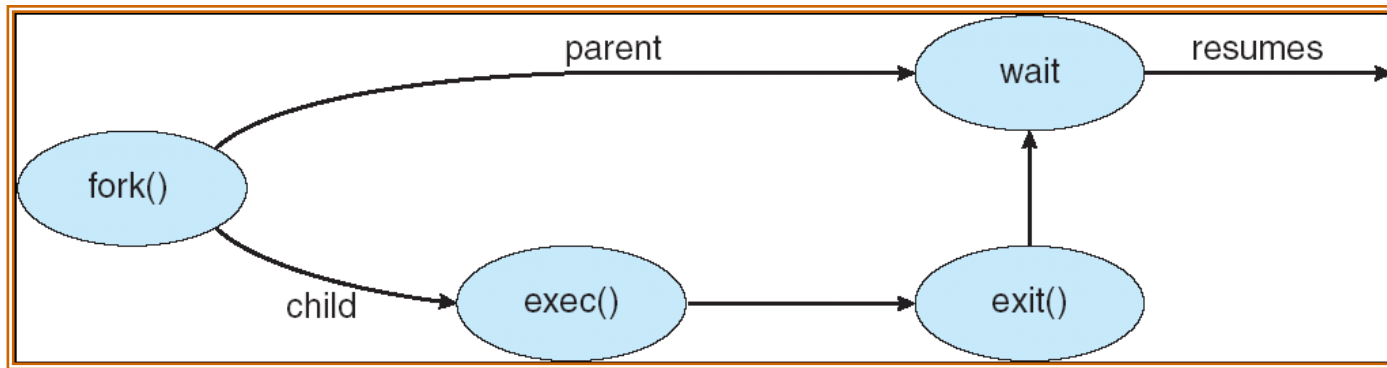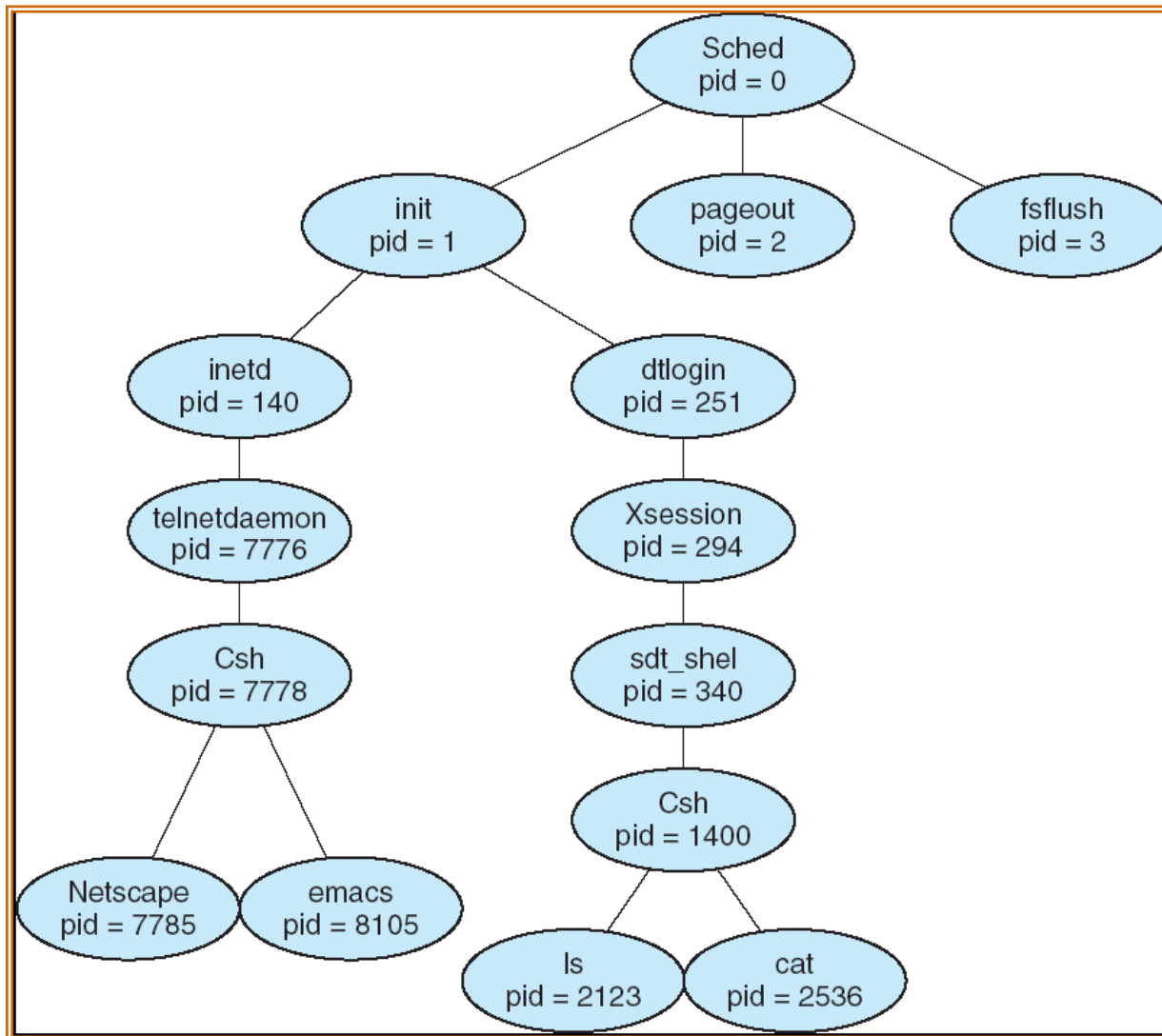
# C Program forking a separate process

```c
# include <stdio.h>
void main(int argc, char *argv[ ])
{ int pid1;
   pid1=fork();  /* fork another process */
   if (pid1<0){ fprintf(stderr, "Fork Failed");   exit(-1);   }
   else if (pid1==0) { execlp("/bin/ls","ls",NULL);
                      }  /* child process */
       else { wait(NULL);
              printf("child Complete");
              exit(0);
            } /*parent process */
}
```

- fork算法演示

- WINDOWS example：CreateProcess.cpp

- 引起进程终止的事件
  - 正常结束
  - 异常结束
  - 外界干预

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

# Process Termination (Cont.)

- *If a parent process is exiting, what happens to its child processes?*

  - Some operating systems do not allow child processes to continue running

  - Child processes may be terminated via cascading termination

  - Child processes may be inherited by a different parent process

# 3.4 Cooperating Processes 合作进程

■ *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

# Cooperating Processes (Cont.)

- Concurrent execution of cooperating processes requires mechanisms that allow processes to communicate with one another and synchronize their actions (chap 6).

- Common Paradigm for cooperating processes -- Producer-Consumer Problem （生产者-消费者问题）

- A *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.
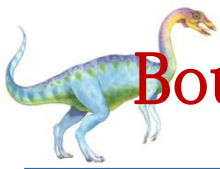
# Bounded-Buffer - Shared-Memory Solution

- The Buffer may either be provided by the operating system through the use of an interprocess-communication (IPC) facility , or by explicitly coded by the application programmer with the use of shared memory .

- Shared-Memory Solution to the Bounded-Buffer problem : Shared data

```
#define BUFFER_SIZE 10

Typedef struct {

    . . .

} item;

item buffer[BUFFER_SIZE];

int in = 0;      int out = 0;
```

Producer :

```
item nextProduced;
  while (1) {

    produce an item in nextProduced ;

      while (((in + 1) % BUFFER_SIZE) == out)
              ; /* do nothing */

      buffer[in] = nextProduced;

      in = (in + 1) % BUFFER_SIZE;

  }
```

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

Consumer :

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

- In Chap 6 we discuss how synchronization among cooperating processes .

# 3.5 Interprocess Communication (IPC)
## 进程间通信

- **IPC** provides a Mechanism for processes to communicate and to synchronize their actions without sharing the same address space .

- IPC is best provided by a message-passing system .

process Q

process P

communication channel
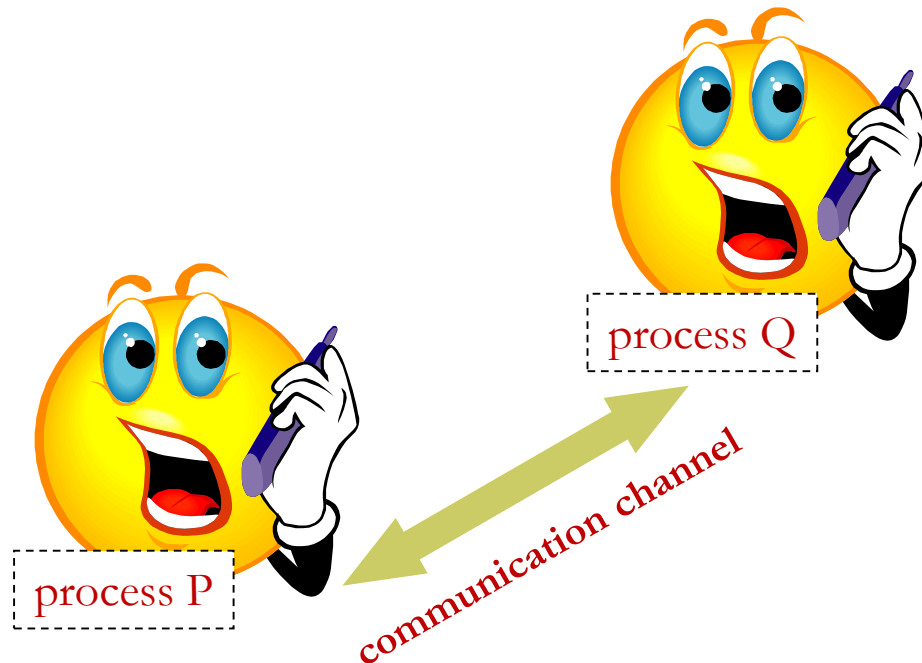
# IPC

- **直接通信**

- **间接通信**

- **常用通信机制：**
  - 信号(signal)
  - 共享存储区(shared memory)
  - 管道(pipe)
  - 消息(message)
  - 套接字(socket)

# Linux进程通信机制

■ Linux实现进程间通信(IPC Inter Process Communication)：

- System V IPC机制:
  - 信号量、消息队列、共享内存
- 管道（pipe）、命名管道
- 套接字（socket）
- 信号( signal )
- 文件锁(file lock)
- POSIX线程：
  - 互斥锁(互斥体、互斥量)（mutex）、条件变量(condition variables)
- POSIX：
  - 消息队列、信号量、共享内存

```
                                                                    ┌─ pipe
                                                    ┌─ byte ────────┼─ FIFO
                                                    │   stream      └─ stream socket
                                   ┌─ data ─────────┼─ pseudoterminal
                                   │   transfer      │
                                   │                 │              ┌─ System V message queue
                                   │                 └─ message ────┼─ POSIX message queue
                                   │                                └─ datagram socket
         communication ───────────┤
                                   │                 ┌─ System V shared memory
                                   │                 │
                                   └─ shared ────────┼─ POSIX shared memory
                                       memory        │                      ┌─ anonymous
                                                     └─ memory mapping ──────┤   mapping
                                                                            └─ mapped file

                                   ┌─ standard signal
         signal ───────────────────┤
                                   └─ realtime signal

                                   ┌─ semaphore ─────┬─ System V semaphore
                                   │                 │                      ┌─ named
                                   │                 └─ POSIX semaphore ─────┤
                                   │                                        └─ unnamed
                                   │                 ┌─ "record" lock (fcntl())
         synchronization ──────────┼─ file lock ─────┤
                                   │                 └─ file lock (flock())
                                   │
                                   ├─ mutex (threads)
                                   │
                                   └─ condition variable (threads)
```
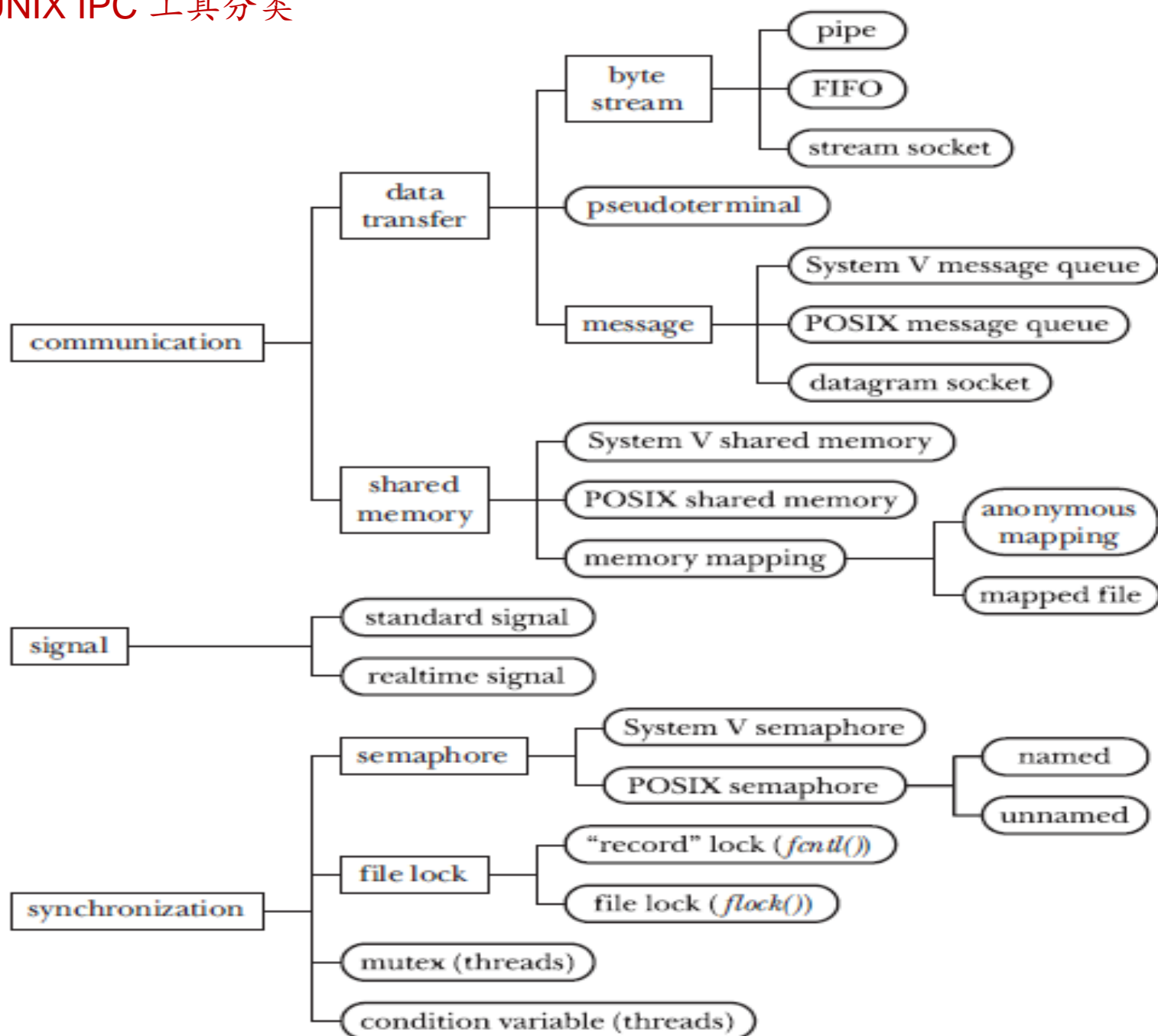
# Windows 2000/XP进程线程通信机制

- 基于文件映射的共享存储区
- 无名管道和命名管道

  server32pipe.c、client32pipe.c

  ▸ 启动多个client进程进行通信

- 邮件槽
- 套接字
- 剪帖板(Clipboard)
- 信号
- 其他同步机制

Windows Interprocess Communication

# 3.5.6 Naming

## 3.5.6.1 Direct Communication直接通信

■ Processes must name each other explicitly:

- send (*P, message*) – send a message to process P
- receive(*Q, message*) – receive a message from process Q
- A link may be associated with many processes.

- Each pair of processes may share several communication links.
- Link may be unidirectional or bi-directional.

    Operations

■ create a new mailbox

- send and receive messages through mailbox
- destroy a mailbox

# Indirect Communication 间接通信

- 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 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.
  - communication link

- **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 (Cont.)

- **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.

# 3.6 Client-Server Communication

- 3.6.1 Sockets

- 3.6.2 Remote Procedure Calls （RPC）

- 3.6.3 Remote Method Invocation (Java)

  (Java)

# 3.6.1 Sockets套接字

- 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.

# Fig 3.9 Communication using Socket

host *X*

(146.86.5.20)

socket

(146.86.5.2/1625)

web server

(161.25.19.8)

socket

(161.25.19.8/80)
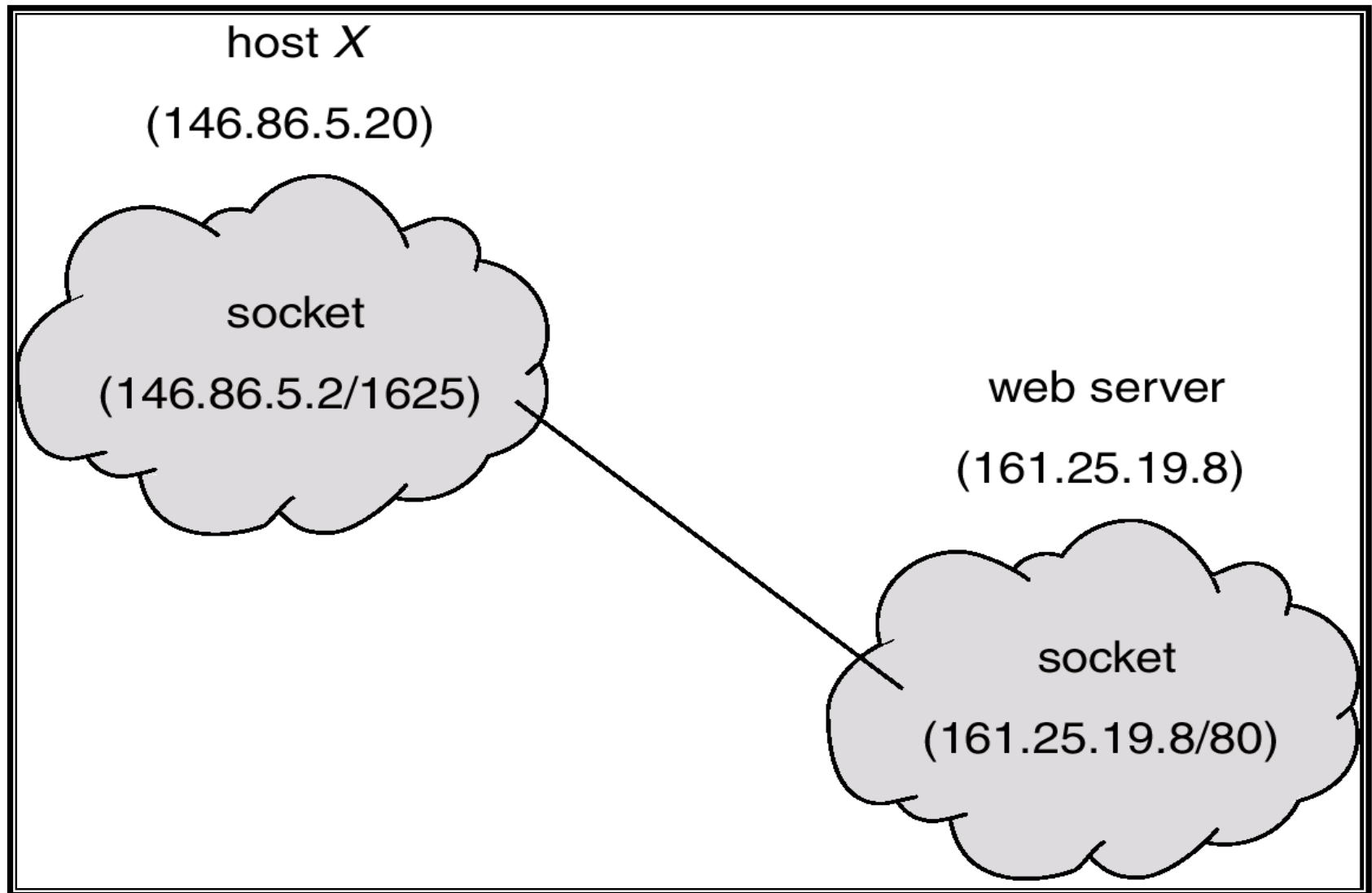
# 3.6.2 Remote Procedure Calls

- 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.
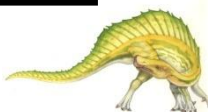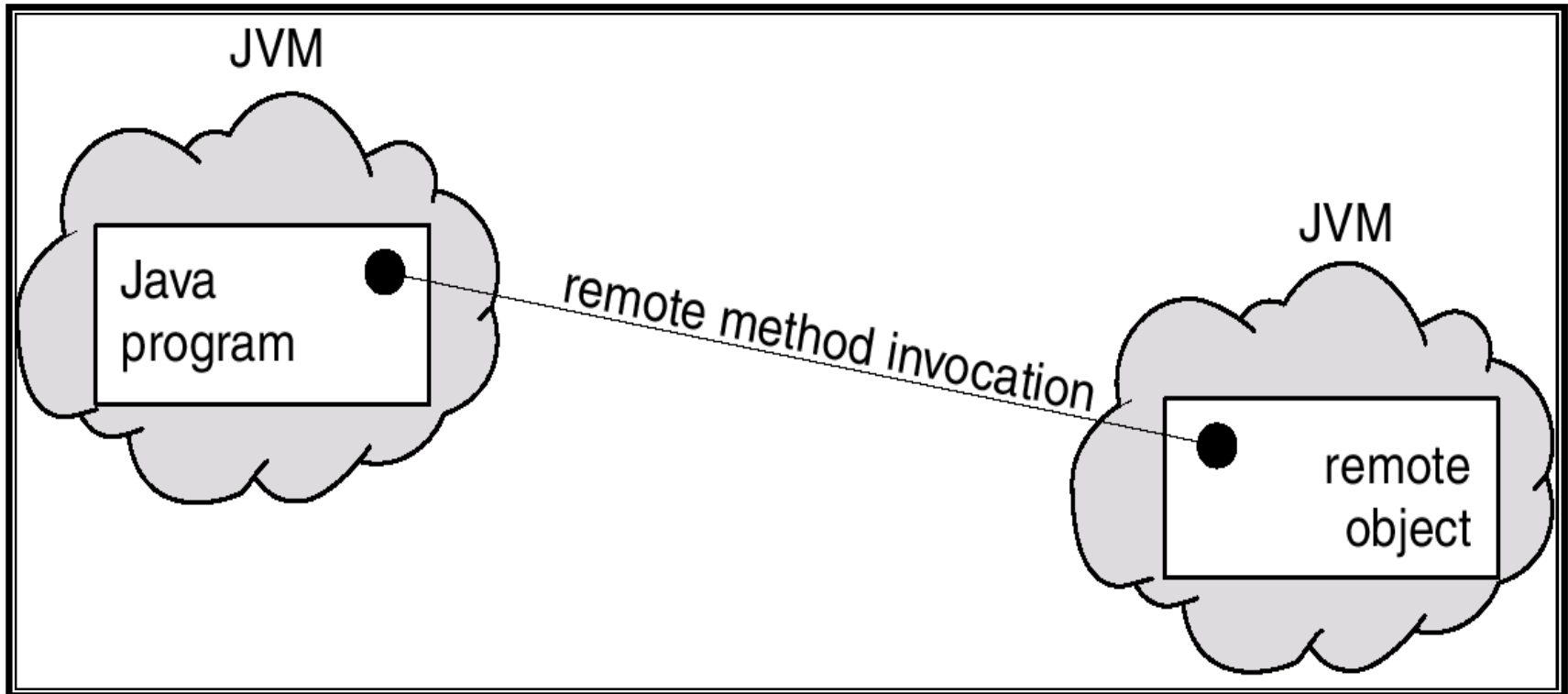
# Fig 3.12 Execution of RPC
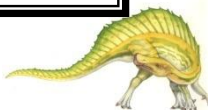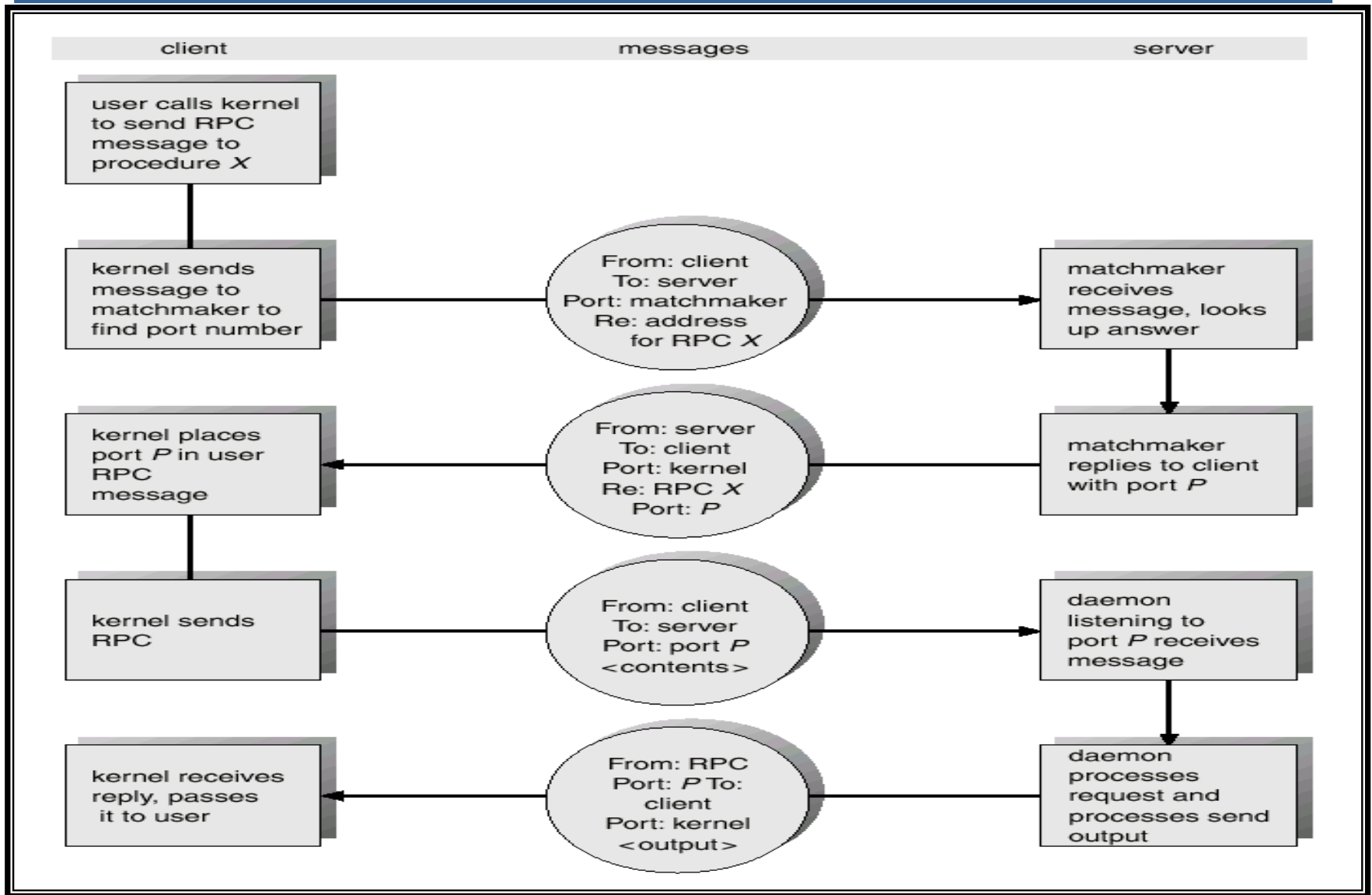
- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.

  RMI allows a Java program on one machine to invoke a method on a remote object.
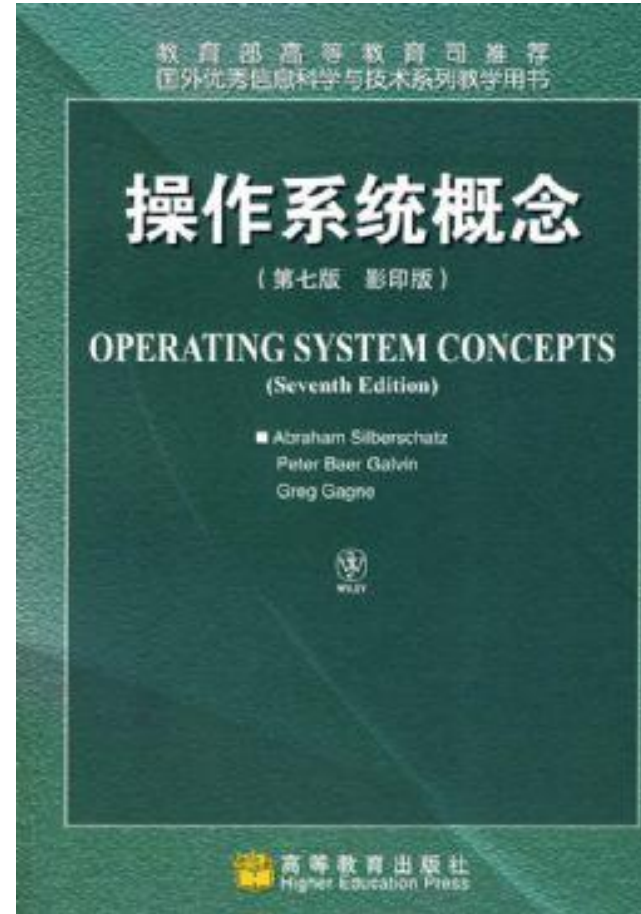
# Homework

■ 布置在"作业系统"中，请按时完成

   3.2、3.4

■ lab exercise

     3.6、3.11

# Reading Assignments

- **Read for this week:**
  - Chapters 3
    of the text book:

- **Read for next week:**
  - Chapters 4
    of the text book:

# End of Chapter 3