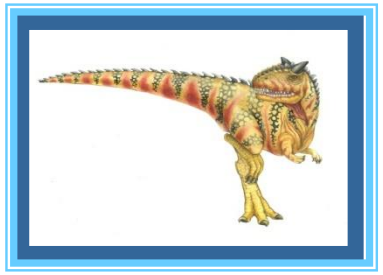
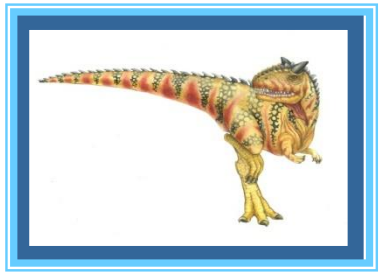
Chapter 3 Processes



Chapter 3 Processes

- **Process Concept**
- **Process Scheduling**
- **Operations on Processes**
- **Interprocess Communication**
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

3.1 Process Concept



Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

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. Jefferson)；

(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)；

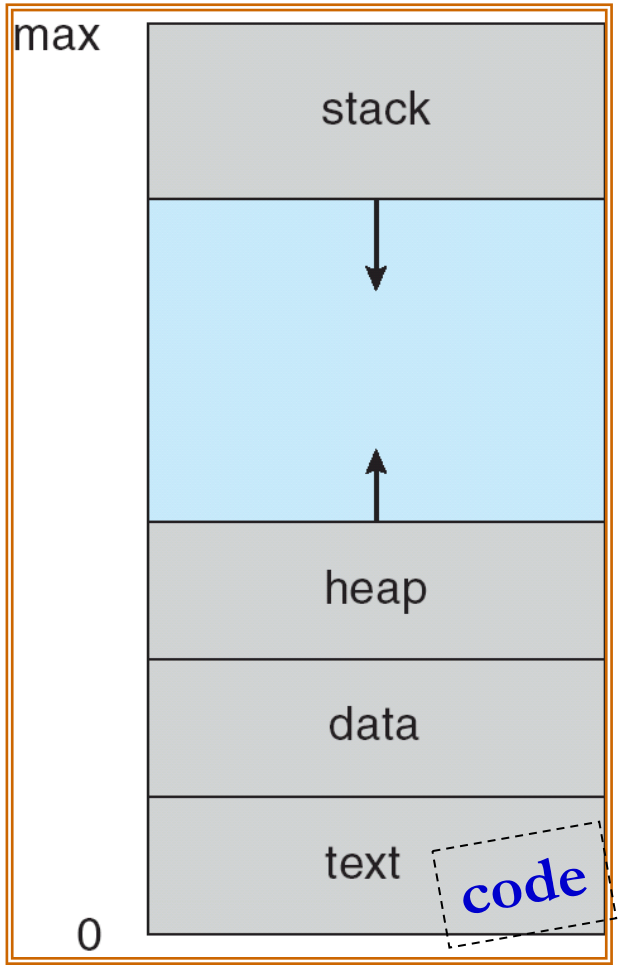
(4) 行为的规则叫程序，程序在处理机上执行时的活动称为进程(E. W. Dijkstra)；

(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。

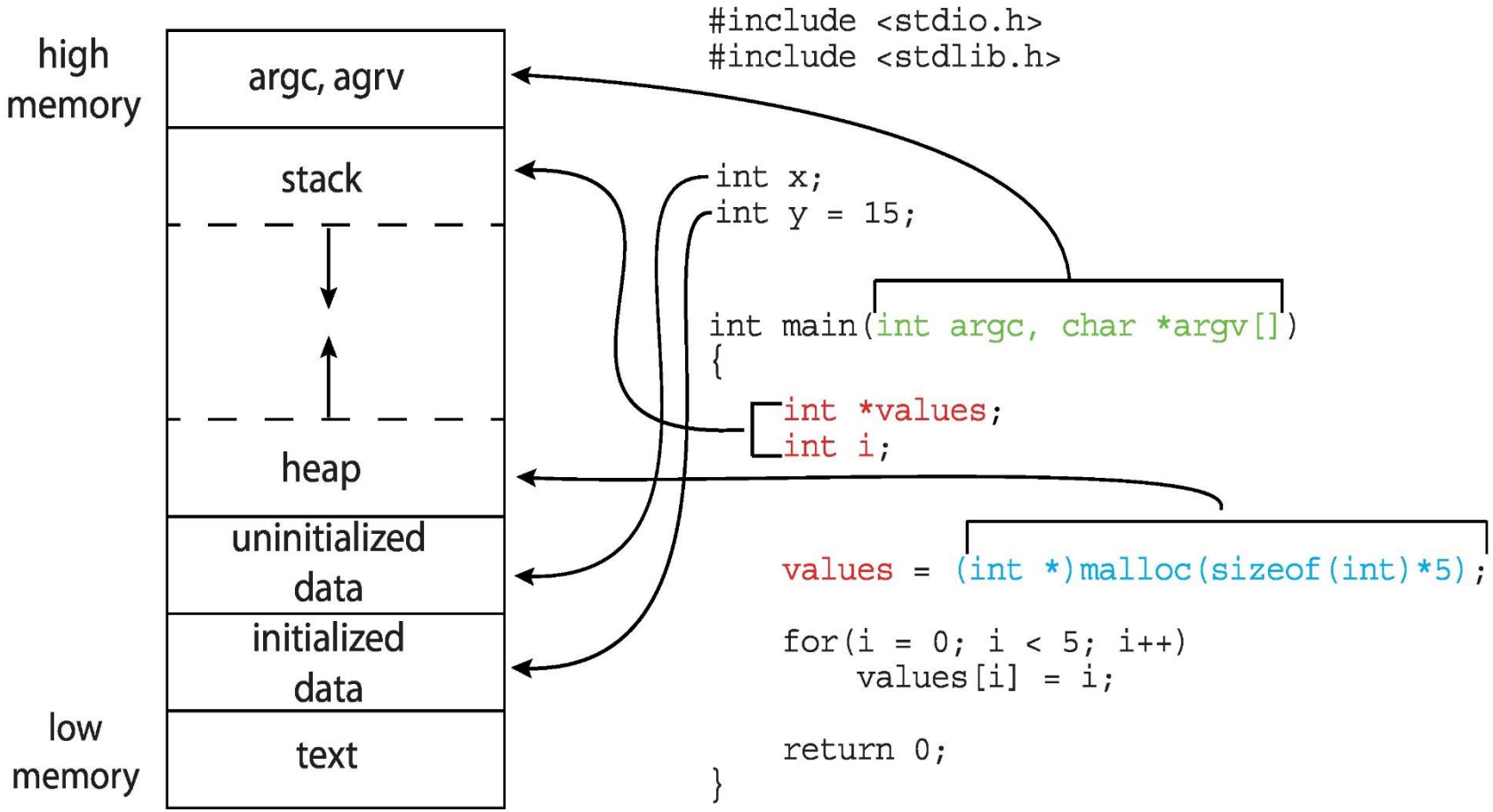
张学尧，计算机操作系统教程，清华大学出版社

Process in Memory

- A process includes:
  - The program code, also called text section
  - Program counter (PC)
  - Registers
  - Data section (global data)
  - Stack (temporary data)
  - Heap
    (dynamically allocated memory)



Memory Layout of a C Program

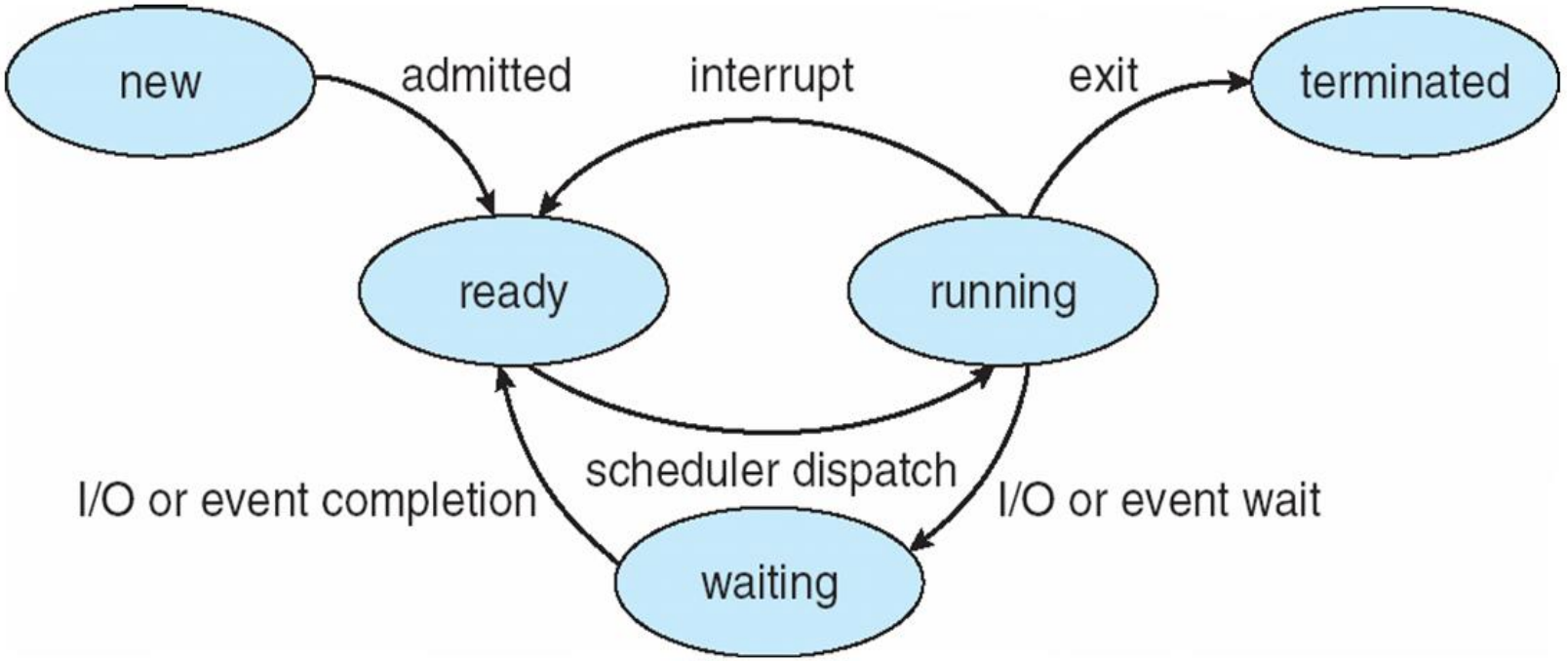


Process State 进程状态

- 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

```
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

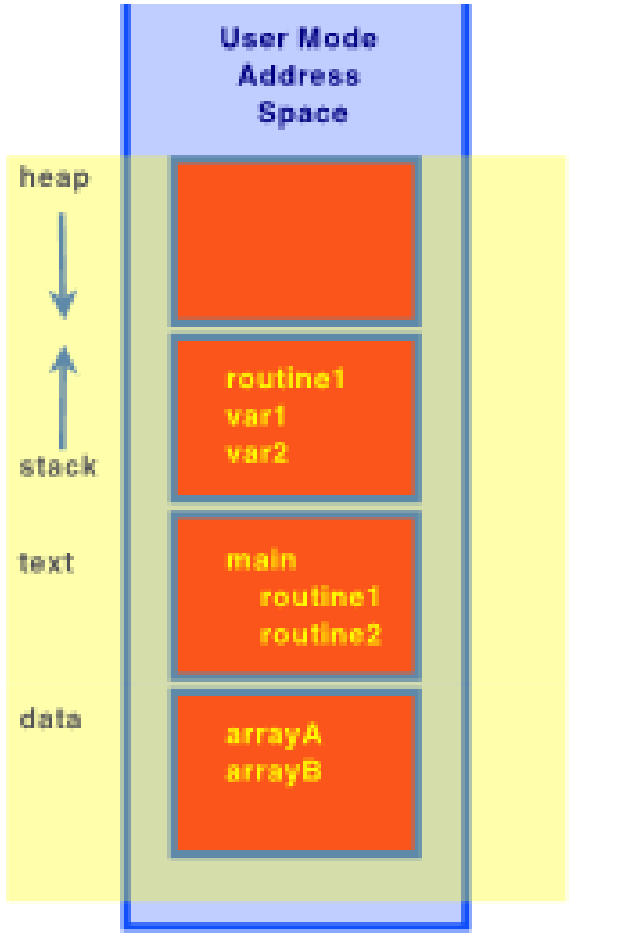
- 例：一个只有一个处理机的系统中，OS的进程有运行、就绪、等待三个基本状态。假如某时刻该系统中有10个进程并发执行，在略去调度程序所占用时间情况下（在user mode下），请问：
  - 这时刻系统中处于运行状态的进程数最多有几个？最少有几个？ 1 0
  - 这时刻系统中处于就绪状态的进程数最多有几个？最少有几个？ 9 0
  - 这时刻系统中处于等待状态的进程数最多有几个？最少有几个？ 10 0

有进程处于就绪状态，则必有进程处于运行状态，不一定有进程处于等待状态。

What Makes up a Process?

User resources/OS Resources:

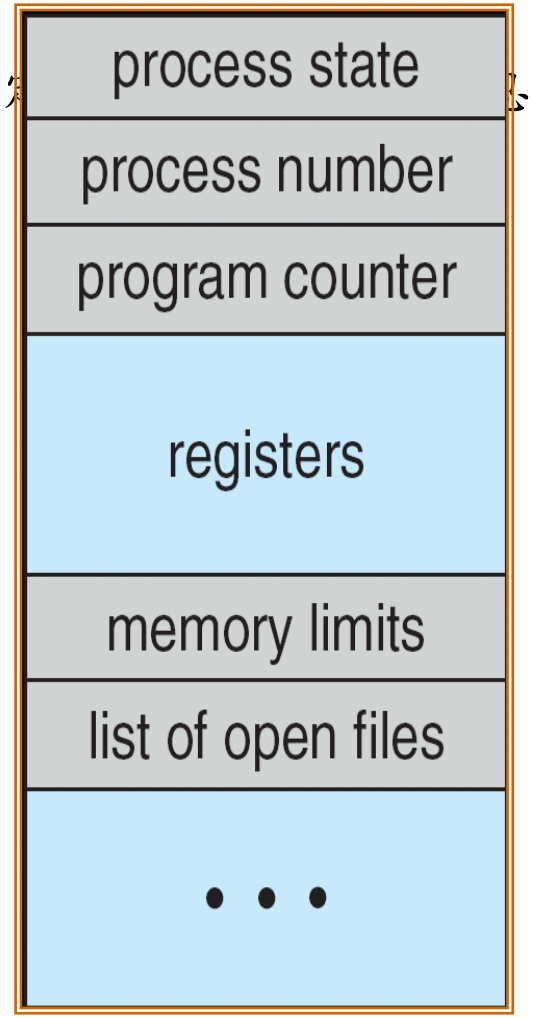
- Program code (text)
- Data
  - global variables
  - heap (dynamically allocated memory)
- Process stack
  - » function parameters
  - » return addresses
  - » local variables and functions
- OS Resources, environment
  - open files, sockets
  - Credential for security
- Registers
  - program counter, stack pointer



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[illegible]息：
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - File management
  - I/O status information



LINUX PCB

- In Linux a process' information is kept in a structure called struct task_struct declared in

#include/linux/sched.h

struct task_struct

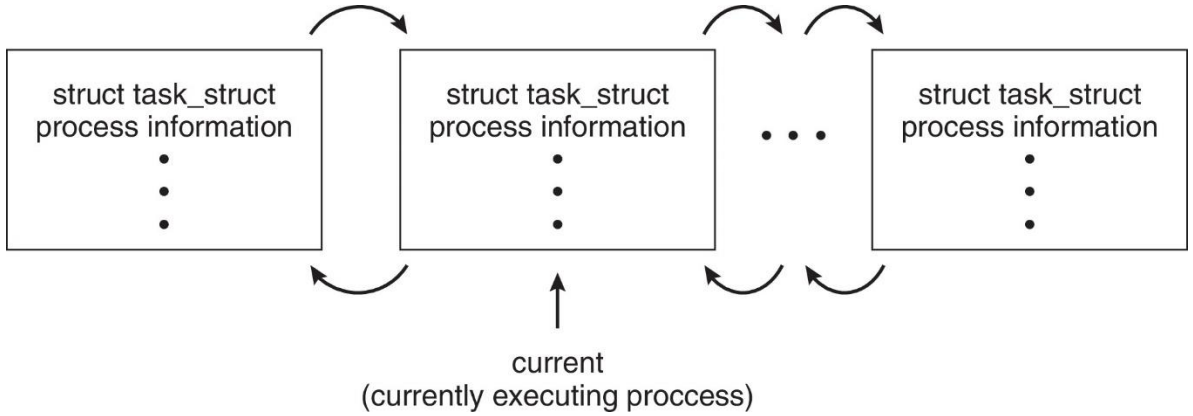
pid_t pid; /* process identifier */

long state; /* state for the process */

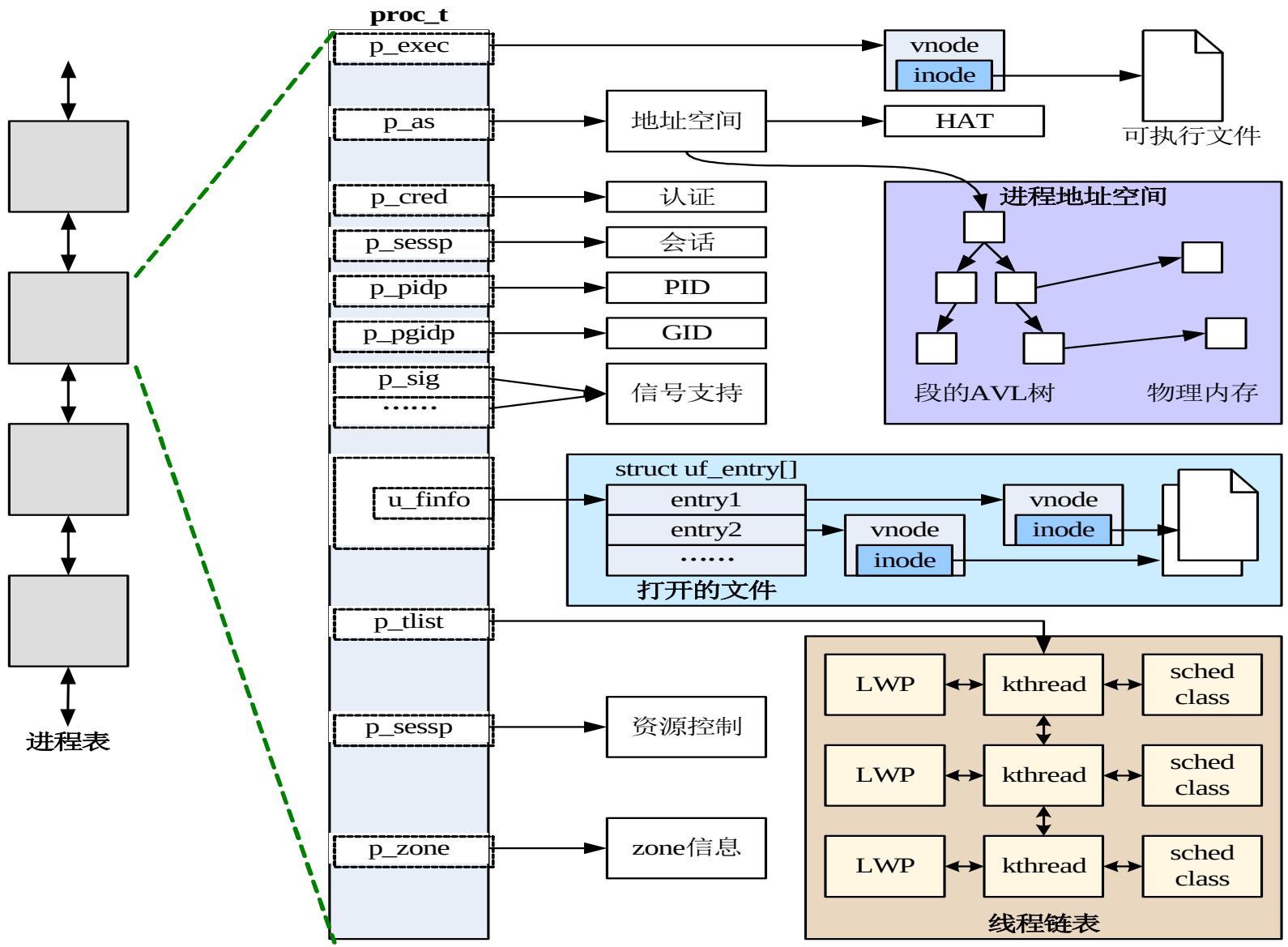
unsigned int time_slice /* scheduling information */

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

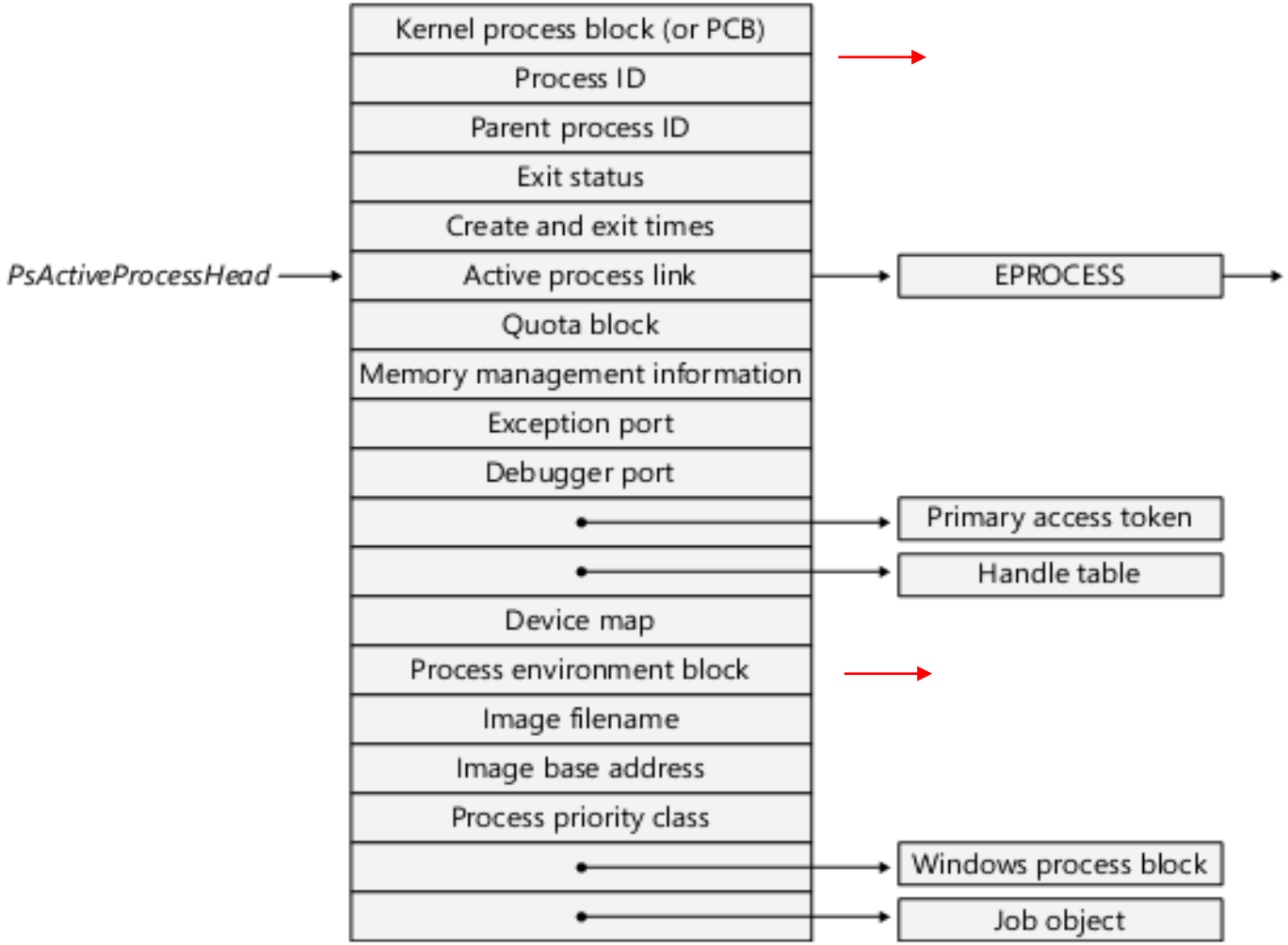
……



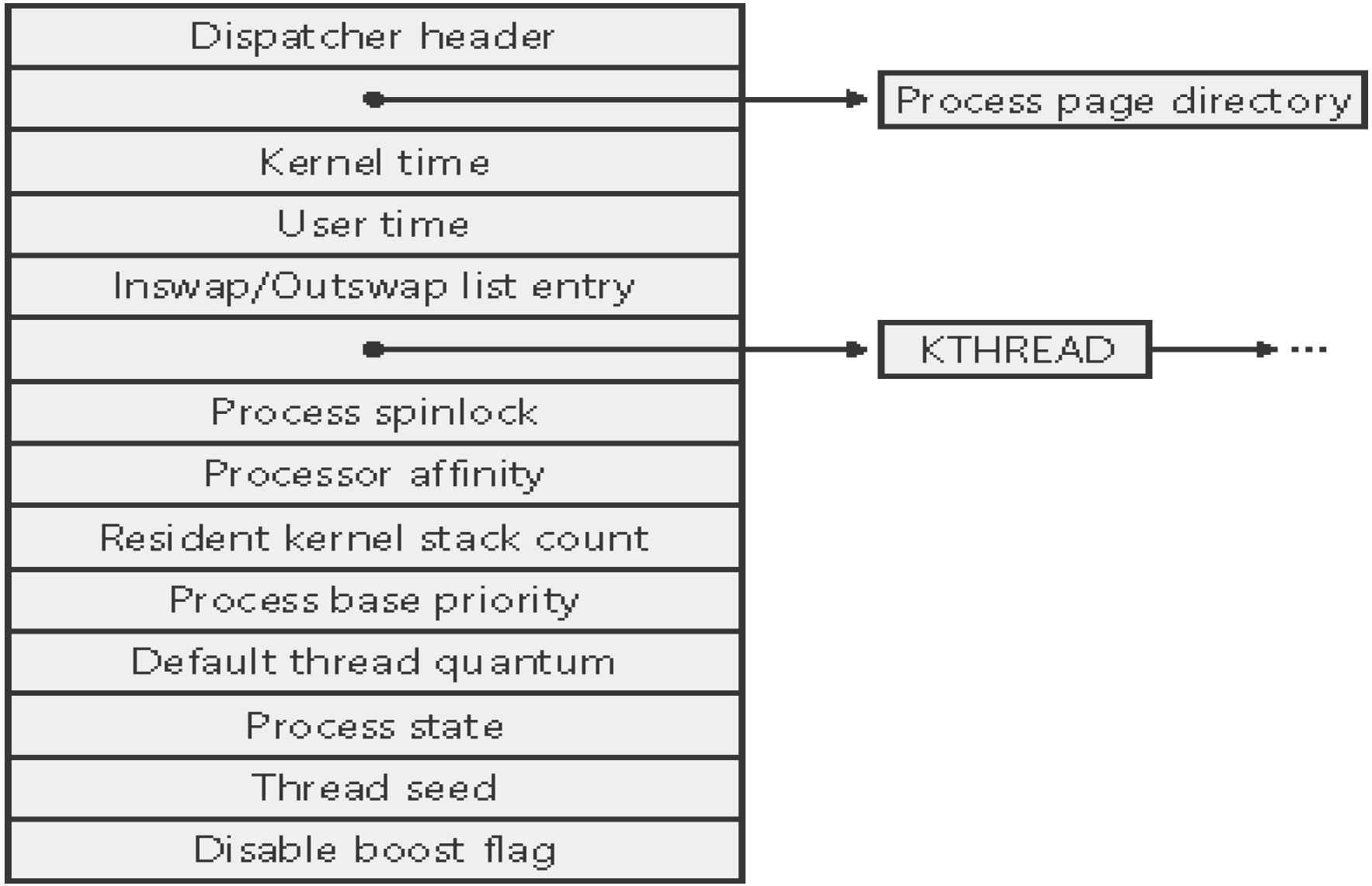
Open Solaris *proc*



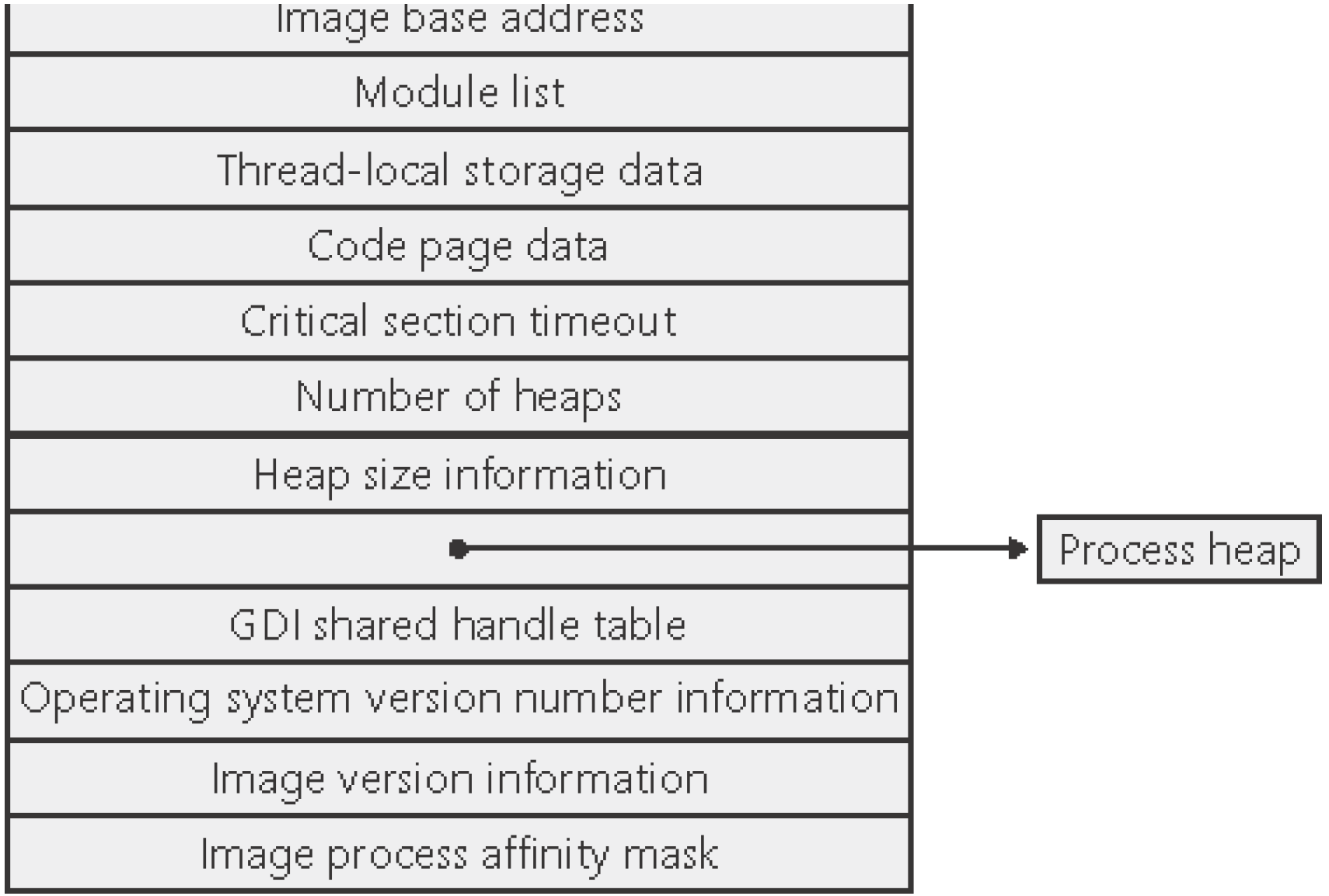
Windows *executive process block*



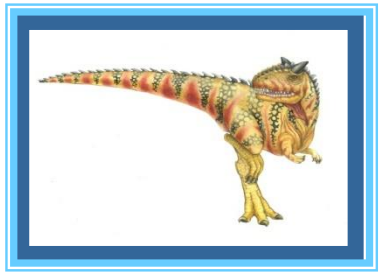
Windows kernel process (KPROCESS) block



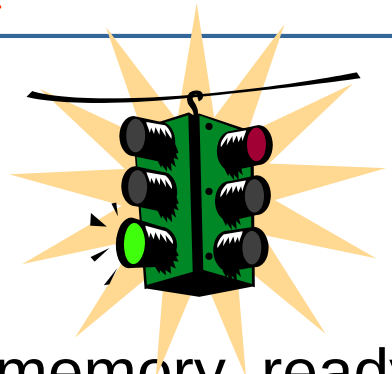
Windows process environment block（PEB）



3.2 Process Scheduling



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

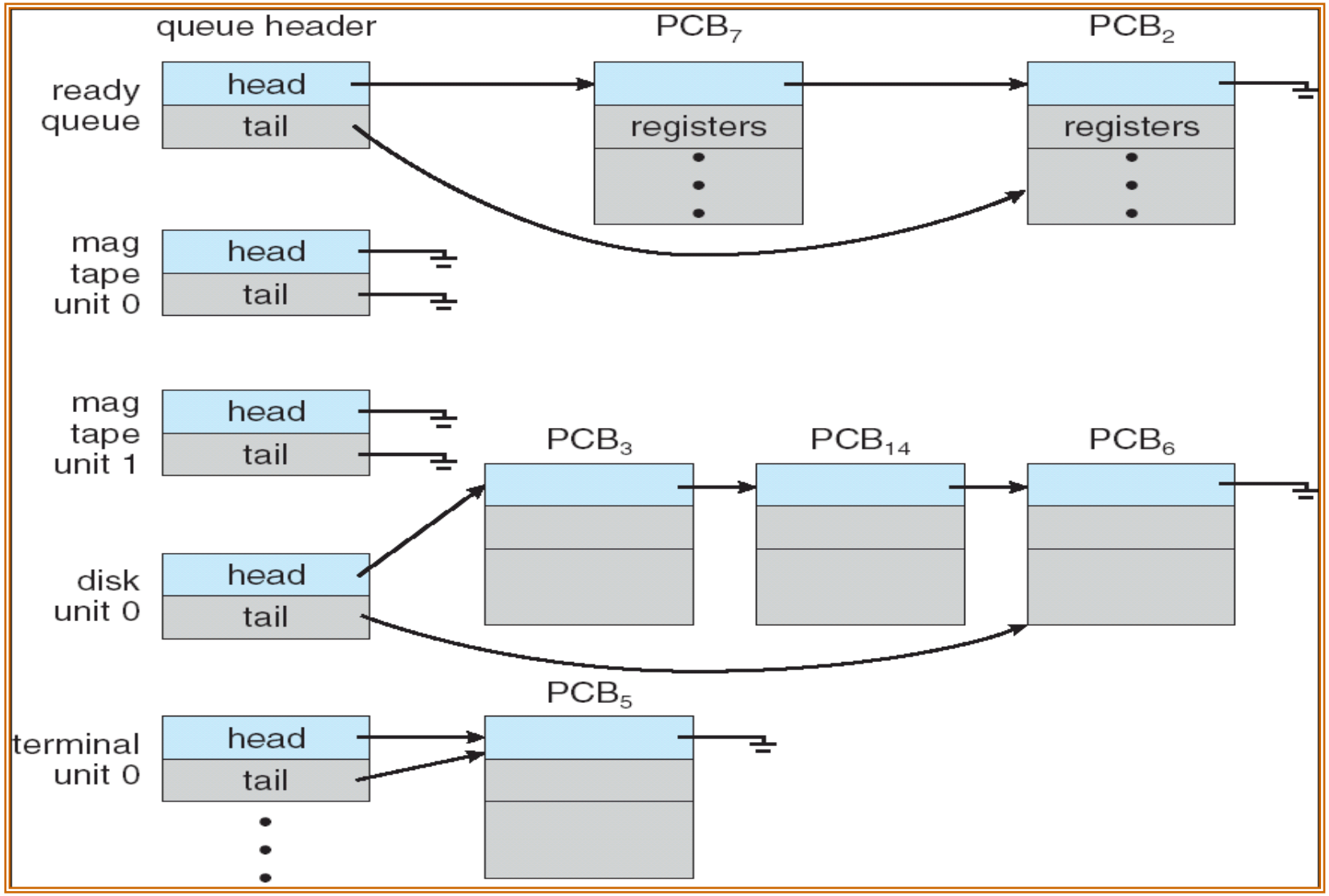
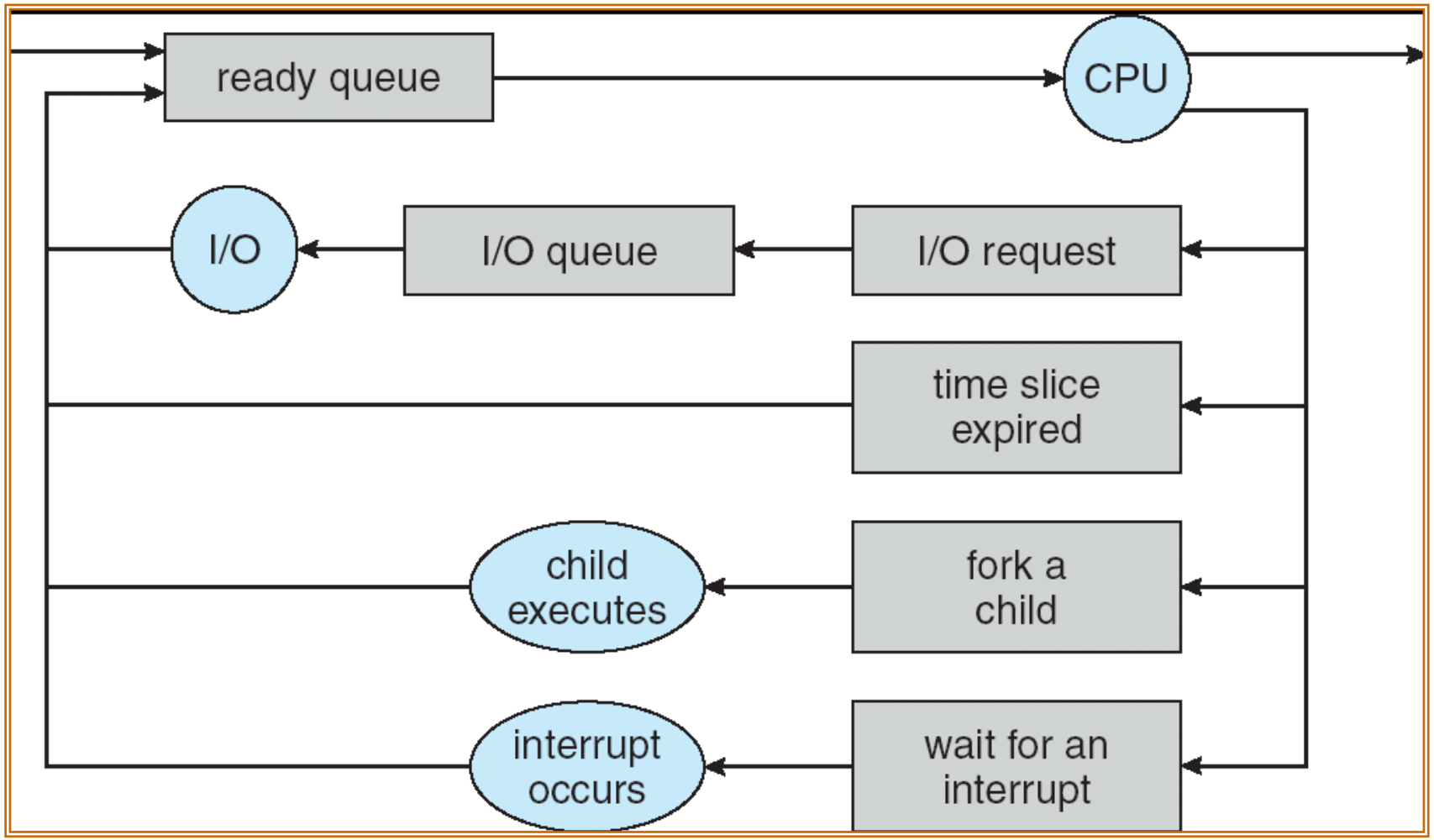


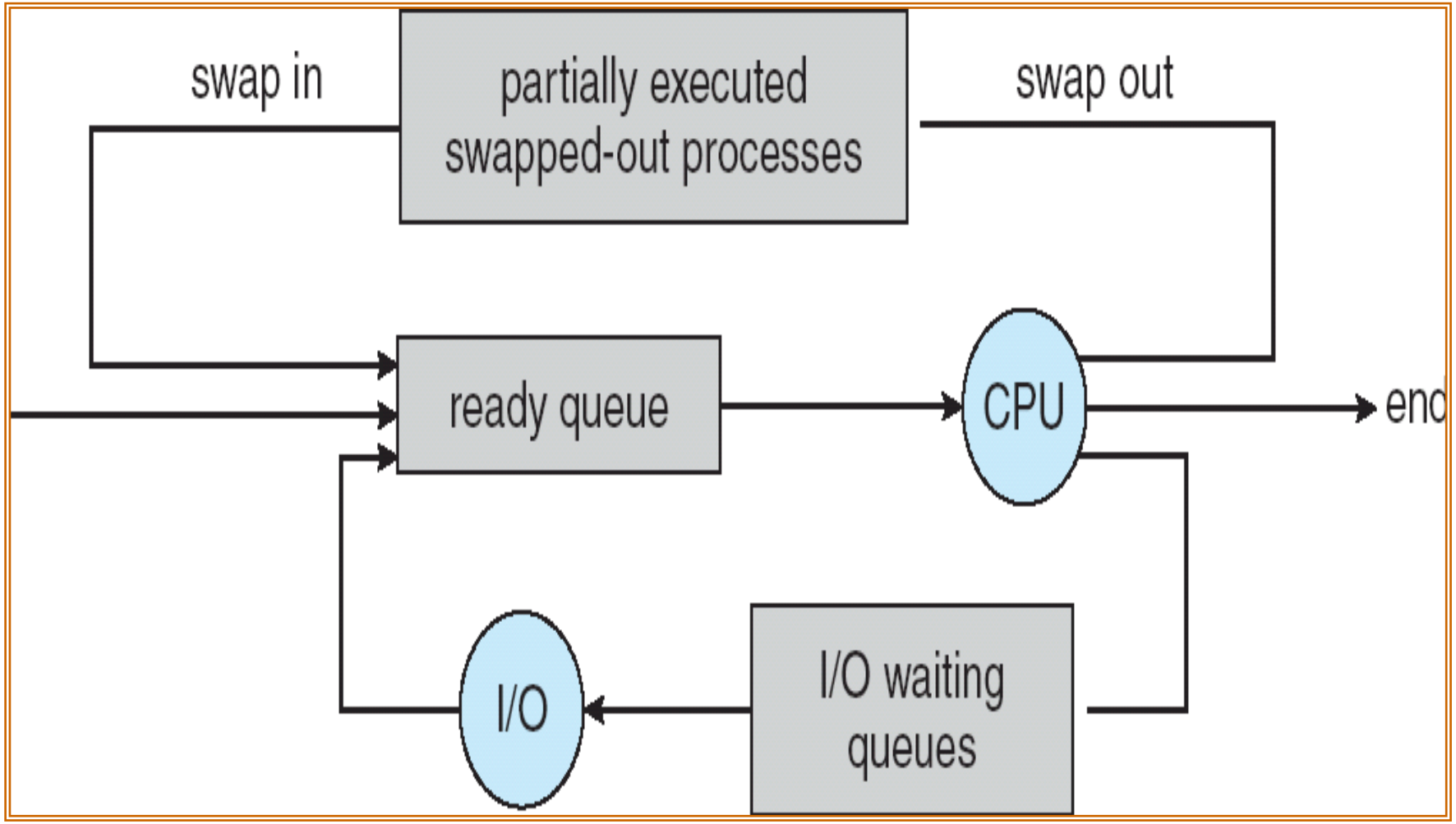
Fig 3.5 Representation of Process Scheduling



3.2.2 Schedulers

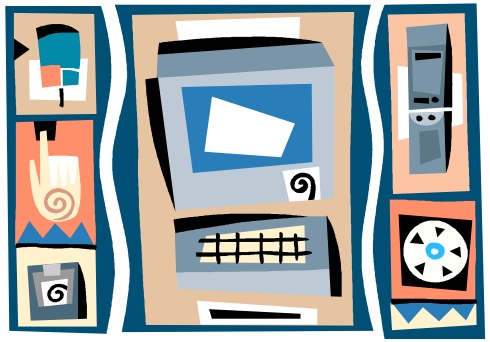
- Long-term scheduler (or job scheduler)长程调度（或作业调度）
  - selects which processes should be brought into the ready queue.
  - invoked very infrequently (seconds, minutes) ⇒ (may be slow)
  - controls the *degree of multiprogramming*多道程序的“道”
  - Most modern operating systems have no long-term scheduler (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) ⇒ (must be fast).
- Medium-Term Scheduler 中程调度

Fig 3.6 Addition of Medium Term Scheduling



Schedulers (Cont.)

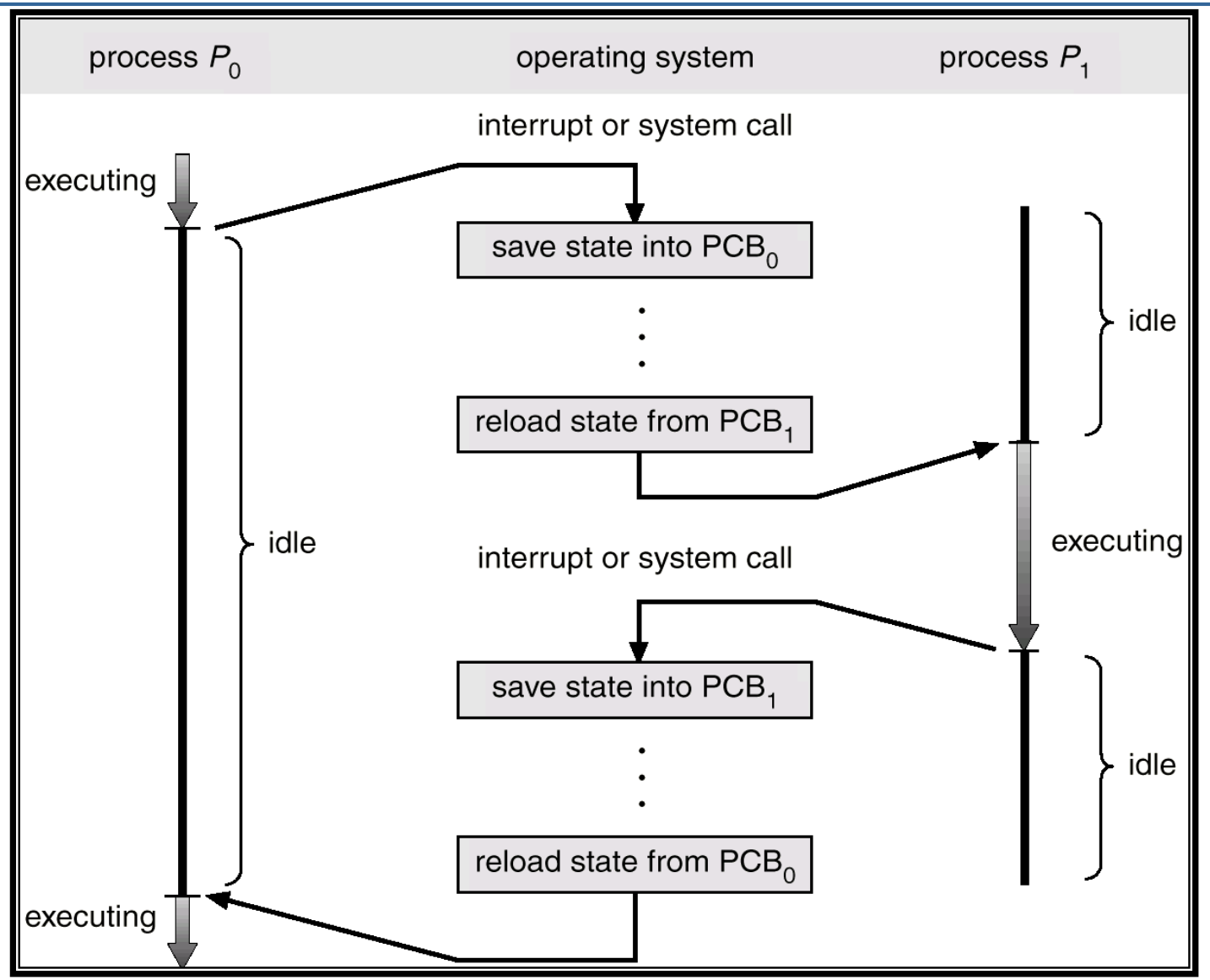
- Processes can be described as either:
  - *I/O-bound process* （I/O型进程）
    - spends more time doing I/O than computations, many short CPU bursts.
  - *CPU-bound process* （CPU 型进程）
    - spends more time doing computations; few very long CPU bursts.



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

CPU Switch From Process to Process



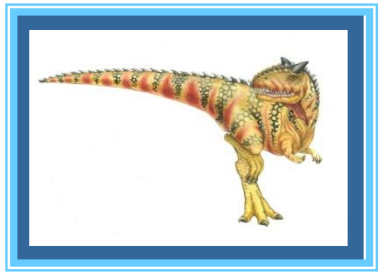
A **context switch** occurs when the CPU switches from one process to another.

context switch takes a few milliseconds

Multitasking in Mobile Systems

- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

3.3 Operations on Processes



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.

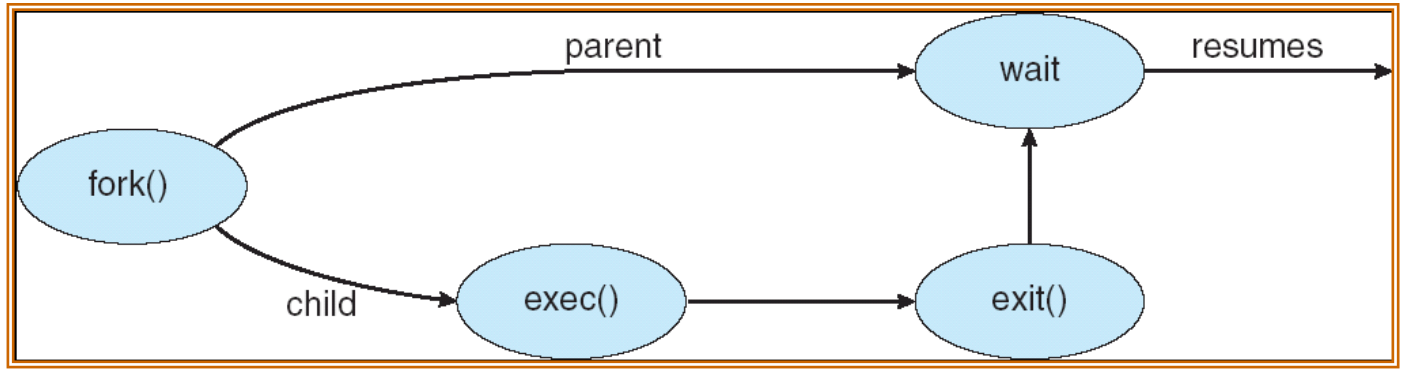
Operating on Processes (Cont.)

- 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.

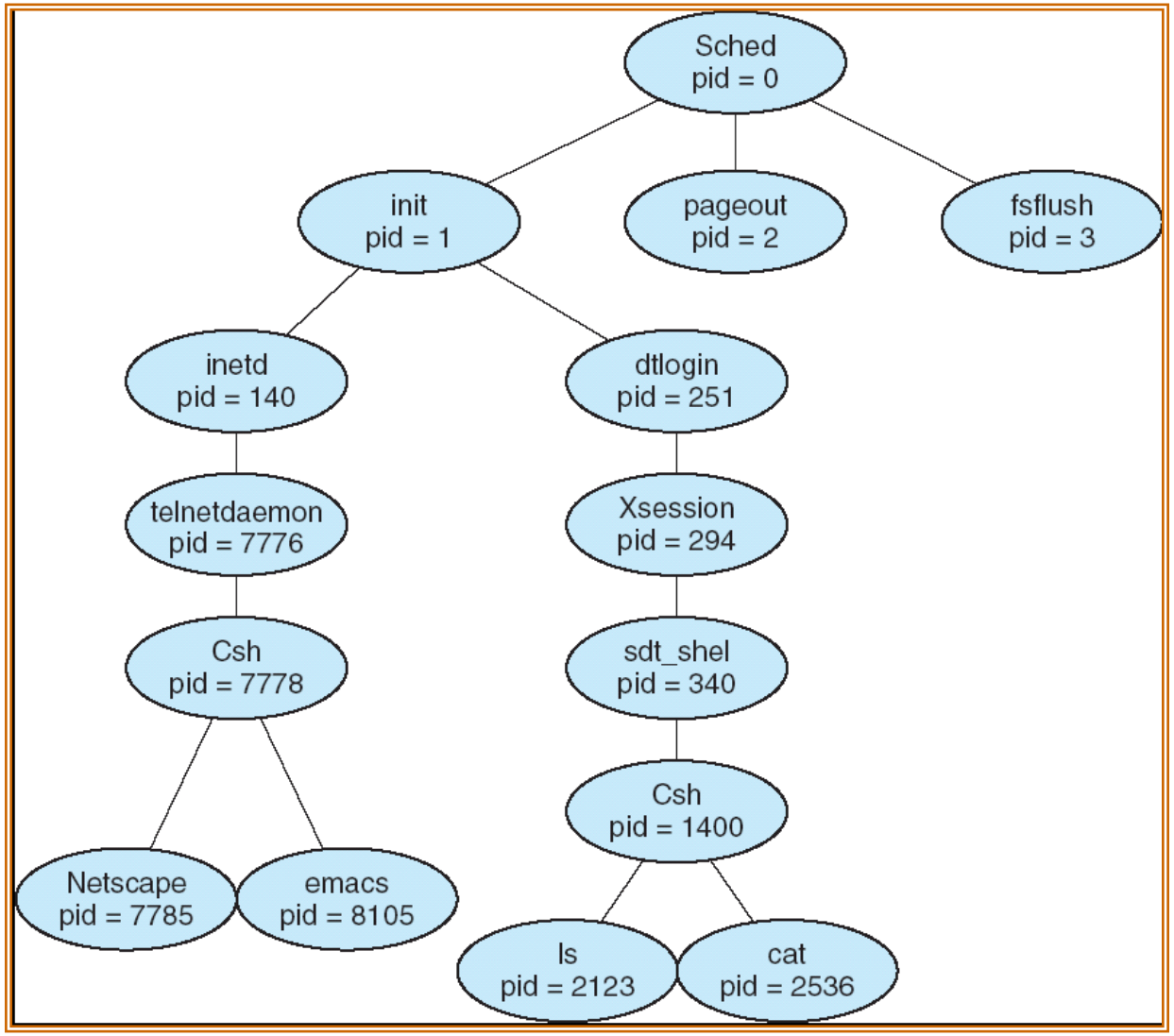
Process Creation (Cont.)

- 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



A tree of processes on a typical Solaris



C Program forking a separate process

```
 # 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

3.3.2 Process Termination 进程终止

- 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.

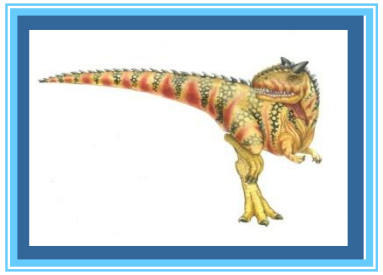
Process Termination (Cont.)

- *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

Android Process Importance Hierarchy

- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- Android will begin terminating processes that are least important.

3.4 Interprocess Communication

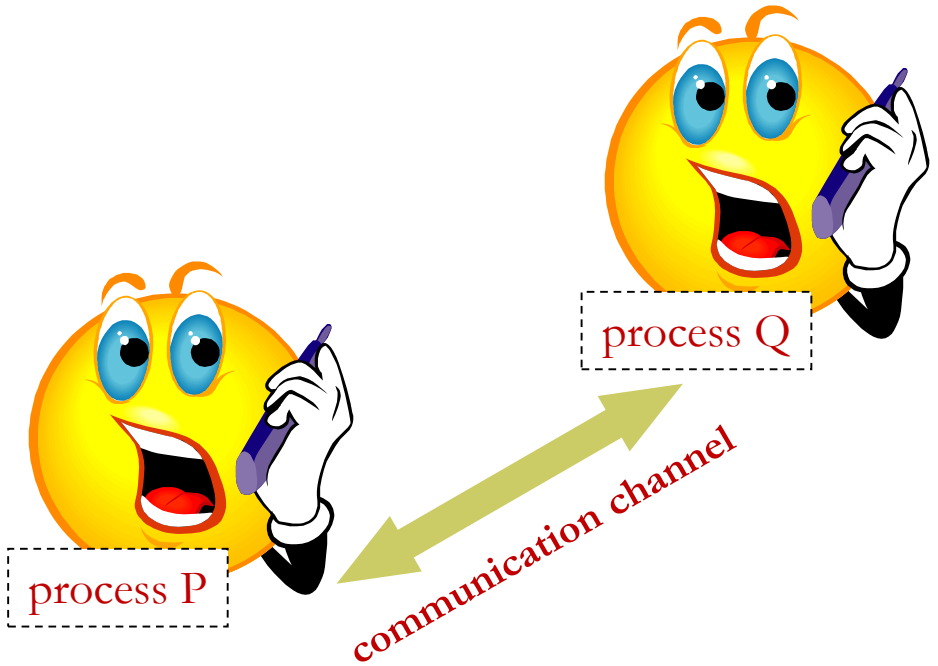


Interprocess Communication 进程通信

- *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 need **interprocess communication** (**IPC**)

Interprocess Communication (IPC) 进程间通信

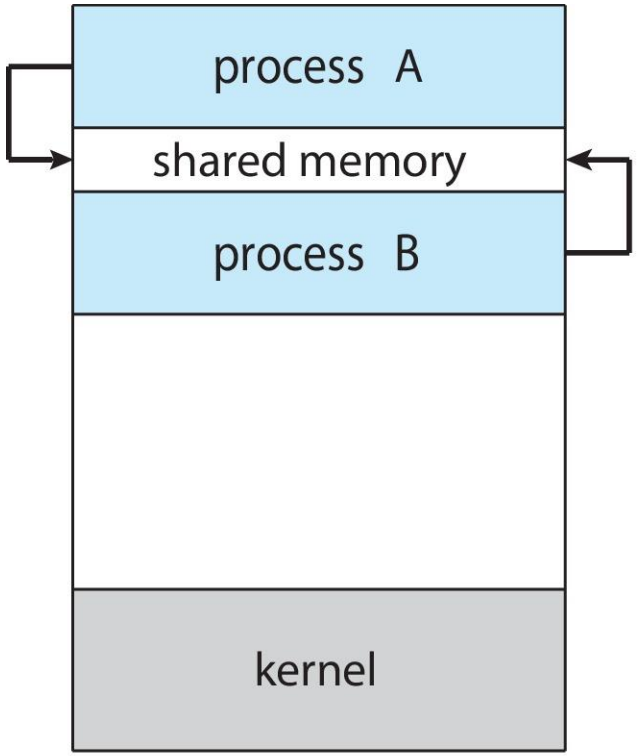
- IPC provides a Mechanism for processes to communicate and to synchronize their actions without sharing the same address space .



Communications Models

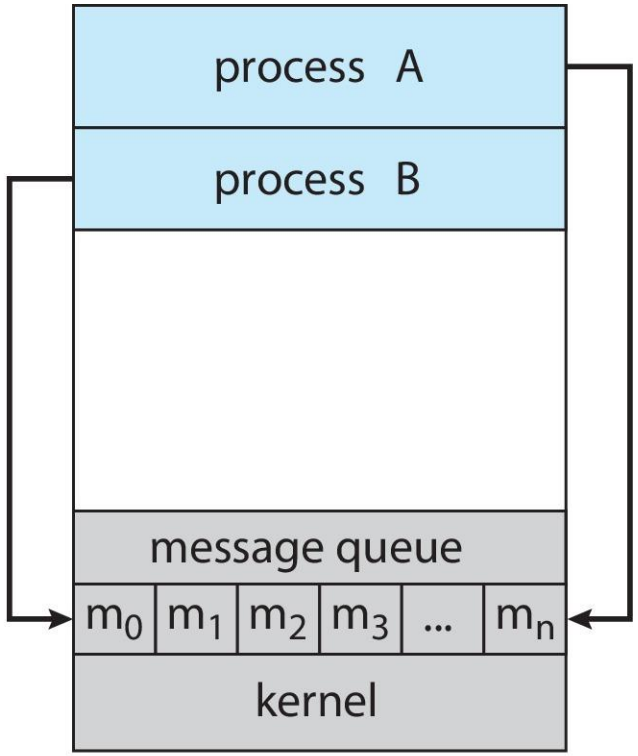
- Two models of IPC
  - **Shared memory**
  - **Message passing**

(a) Shared memory.



(a)

(b) Message passing.



(b)

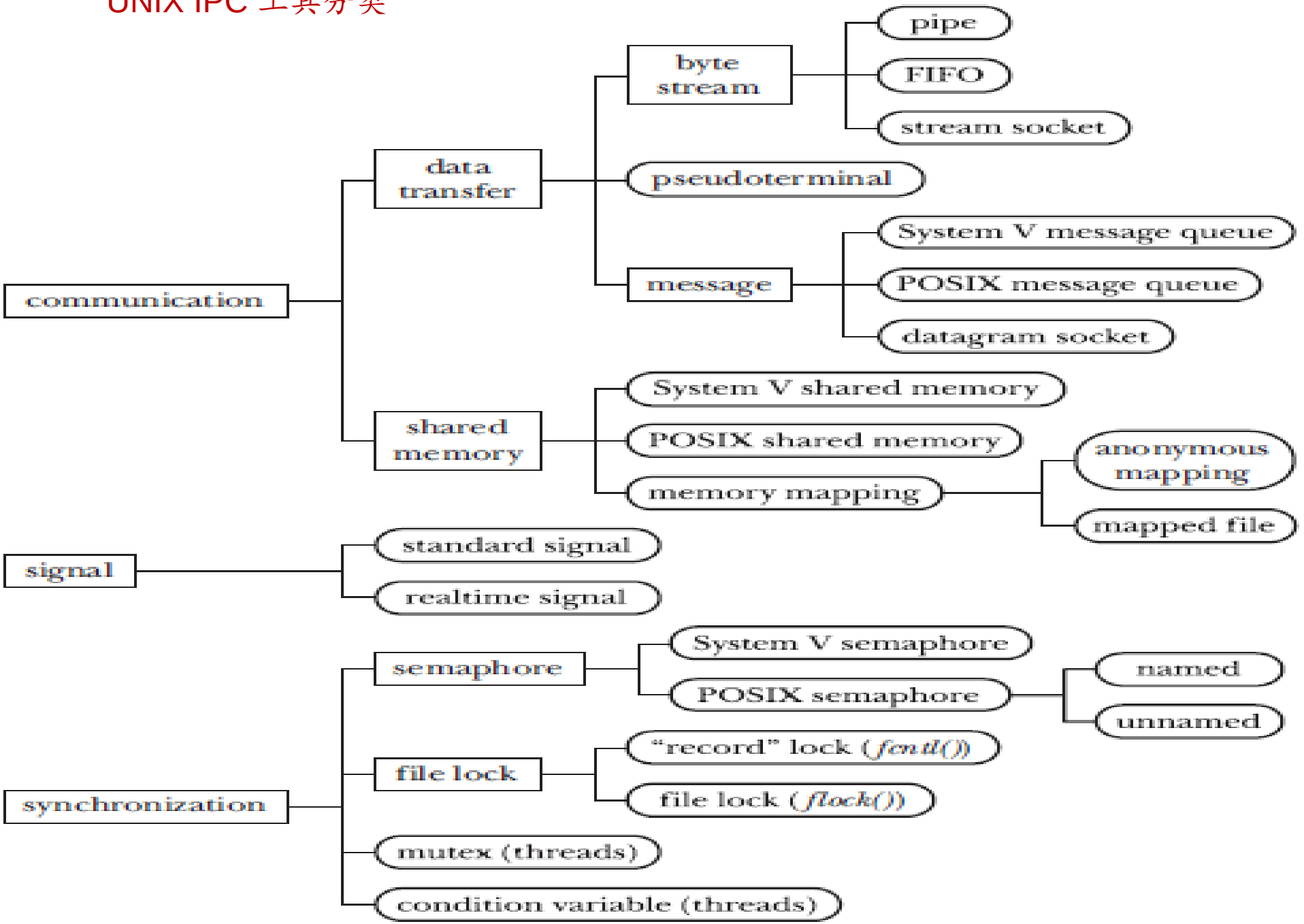
IPC

- 通信类型：
  - 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

UNIX IPC 工具分类



Windows 进程线程通信机制

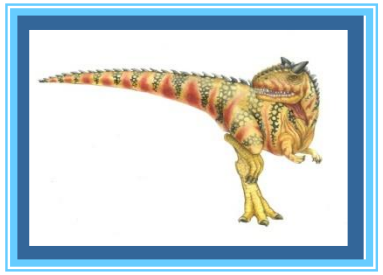
- 基于文件映射的共享存储区
- 无名管道和命名管道

  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 IPC in Shared-Memory Systems



Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.

Bounded-Buffer – Shared-Memory Solution

- 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*（无限缓冲区） places no practical limit on the size of the buffer.
  - *bounded-buffer*（有限缓冲区） assumes that there is a fixed buffer size.

Bounded-Buffer – Shared-Memory Solution

- 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;
```

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

Bounded-Buffer – Producer Process

Producer :

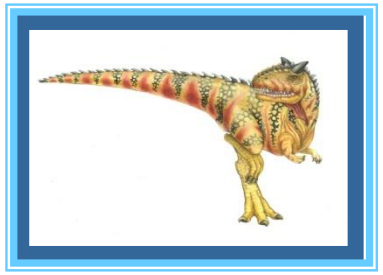
```
item nextProduced;
  while (1) {
     produce an item in nextProduced ;
      while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer – Consumer Process

Consumer :

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

3.6 IPC in Message-Passing Systems



Interprocess Communication – Message Passing

- 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*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - 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?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

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
- 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（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 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 (port)
  - 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（阻塞）** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking （非阻塞）** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Producer – Message Passing

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

        send(next_produced);
}
```

Consumer– Message Passing

```
message next_consumed;

while (true) {
     receive(next_consumed)

     /* consume the item in next_consumed */
 }
```

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     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

实例

- 例：设计一个程序，要求用函数msgget创建消息队列，从键盘输入的字符串添加到消息队列。创建一个进程，使用函数msgrcv读取队列中的消息并在计算机屏幕上输出。
- 分析：程序先调用msgget函数创建、打开消息队列，接着调用msgsnd函数，把输入的字符串添加到消息队列中。子进程调用msgrcv函数，读取消息队列中的消息并打印输出，最后调用msgctl函数，删除系统内核中的消息队列。

实例

```
//msgfork.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <unistd.h>
struct msgmbuf  {                    /*结构体，定义消息的结构*
    long msg_type;                   /*消息类型*/
    char msg_text[512]; };           /*消息内容*/
int main(){
int qid,len;
key_t key;
struct msgmbuf msg;
```

实例

```
if((key=ftok(".",'a'))==-1) {  /*调用ftok函数，产生标准的key*/
     perror("产生标准key出错");
    exit(1);}

/*调用msgget函数，创建、打开消息队列*/
if((qid=msgget(key,IPC_CREAT|0666))==-1) {
    perror("创建消息队列出错");
    exit(1);}
printf("创建、打开的队列号是：%d\n",qid); /*打印输出队列号*/
```

实例

```
int pid=fork();
if (pid>0){
    printf(" 我是父进程PID=：%d 发送消息\n",getpid())
     puts("请输入要加入队列的消息：");
     /*键盘输入的消息存入变量msg_text*/
     if((fgets((&msg)->msg_text,512,stdin))==NULL) {
            puts("没有消息");
            exit(1);}
     msg.msg_type=getpid();
     len=strlen(msg.msg_text);
     /*调用msgsnd函数，添加消息到消息队列*/
     if((msgsnd(qid,&msg,len,0))<0) {
            perror("添加消息出错");
            exit(1);}
```

实例

```
else{
    printf("我是子进程PID=：%d 接收消息\n",getpid())
            /*调用msgrcv函数，从消息队列读取消息*/
     if((msgrcv(qid,&msg,512,0,0))<0) {
            perror("读取消息出错");
            exit(1);}
            /*打印输出消息内容*/
     printf("读取的消息是：%s\n",(&msg)->msg_text);
            /*调用msgctl函数，删除系统中的消息队列*/
     if((msgctl(qid,IPC_RMID,NULL))<0){
            perror("删除消息队列出错");
            exit(1);}
     }
exit (0);}
```

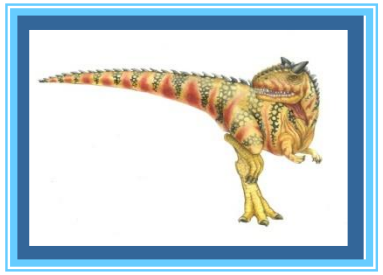
实例

- 分成两个独立的程序：msgsnd.c,msgrcv.c。分别编译和运行

```
[root@localhost syspro]# ./msgsnd
创建、打开的队列号是：131072
 我发送消息进程PID=：3784 发送消息
请输入要加入队列的消息：
sajhsdaj
[root@localhost syspro]#
```

```
[root@localhost syspro]# ./msgrcv
创建、打开的队列号是：131072
我是接收消息进程PID=：3798 接收消息
读取的消息是：sajhsdaj
```

3.7 Examples of IPC Systems



Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    `shm_fd = shm_open(name, O CREAT | O RDWR, 0666);`
  - Also used to open an existing segment
  - Set the size of the object

```
ftruncate(shm_fd, 4096);
```

  - Use `mmap()` to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by `mmap()`.

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main()
{
/* the size (in bytes) of shared memory
object */
const int SIZE = 4096;
 /* name of the shared memory object */
const char *name = "OS";
 /* strings written to shared memory */
const char *message 0 = "Hello";
const char *message 1 = "World!";
  /* shared memory file descriptor */
int fd;
  /* pointer to shared memory obect */
char *ptr;
  /* create the shared memory object */
fd = shm open(name,O CREAT | O RDWR,0666);
/* configure the size of the shared memory object */
ftruncate(fd, SIZE);
   /* memory map the shared memory object */
ptr = (char *)mmap(0, SIZE, PROT READ | PROT
WRITE, MAP SHARED, fd, 0);
   /* write to the shared memory object */
sprintf(ptr,"%s",message 0);
ptr += strlen(message 0);
sprintf(ptr,"%s",message 1);
ptr += strlen(message 1);
return 0;
}
```

IPC POSIX Consumer

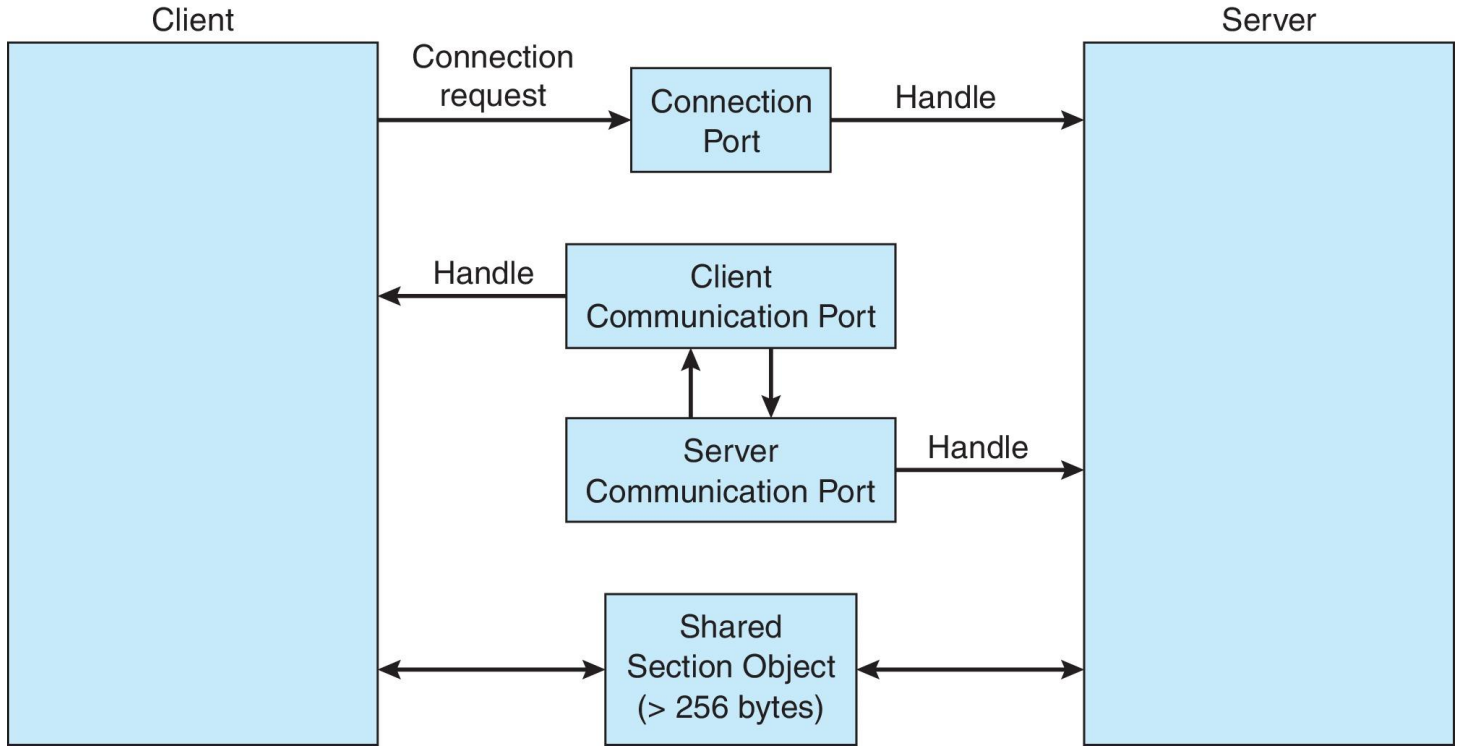
```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;
```

```
/* open the shared memory object */
fd = shm_open(name, O_RDONLY, 0666);
/* memory map the shared memory object */
ptr = (char *)
mmap(0, SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
/* read from the shared memory object */
printf("%s",(char *)ptr);
/* remove the shared memory object */
shm_unlink(name);
return 0;
}
```

Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - 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 two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows

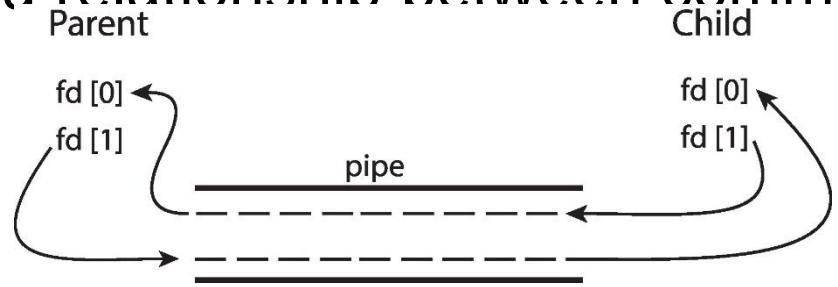


Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- **Named pipes** – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes

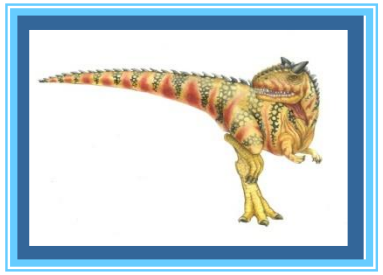


- Windows calls these **anonymous pipes**

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

3.8 Communication in Client–Server Systems



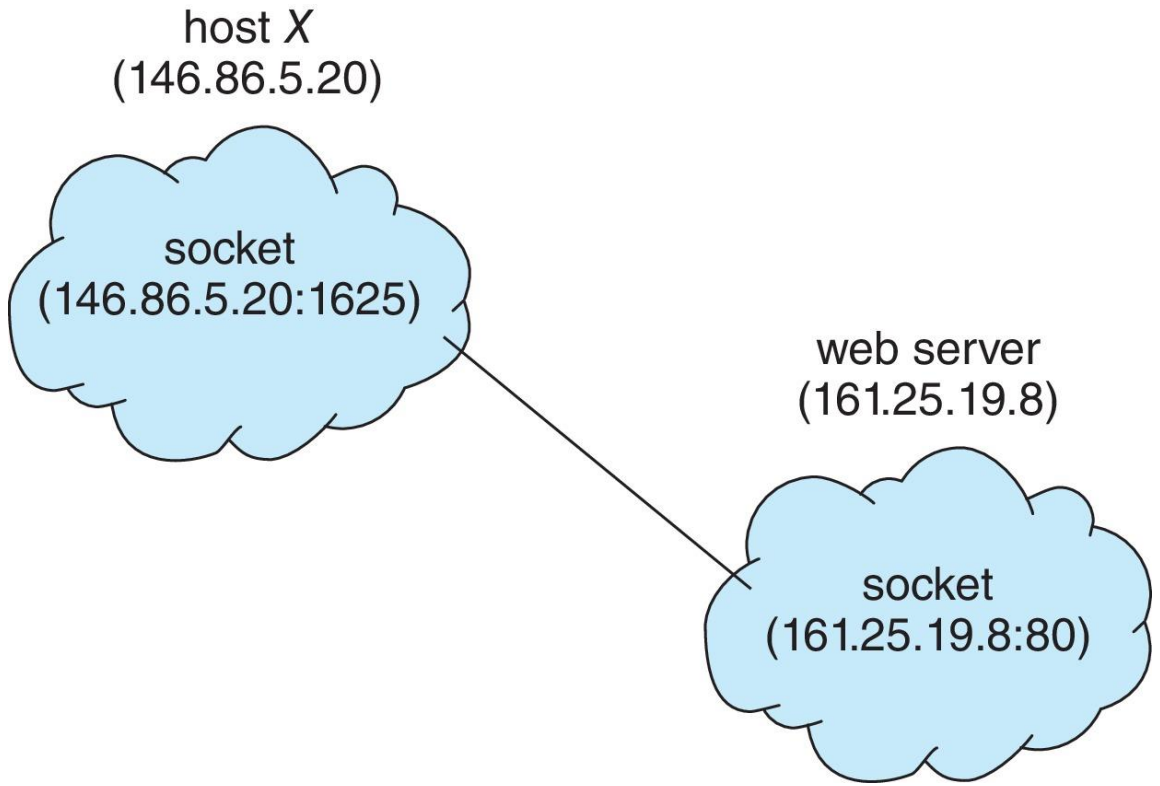
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication



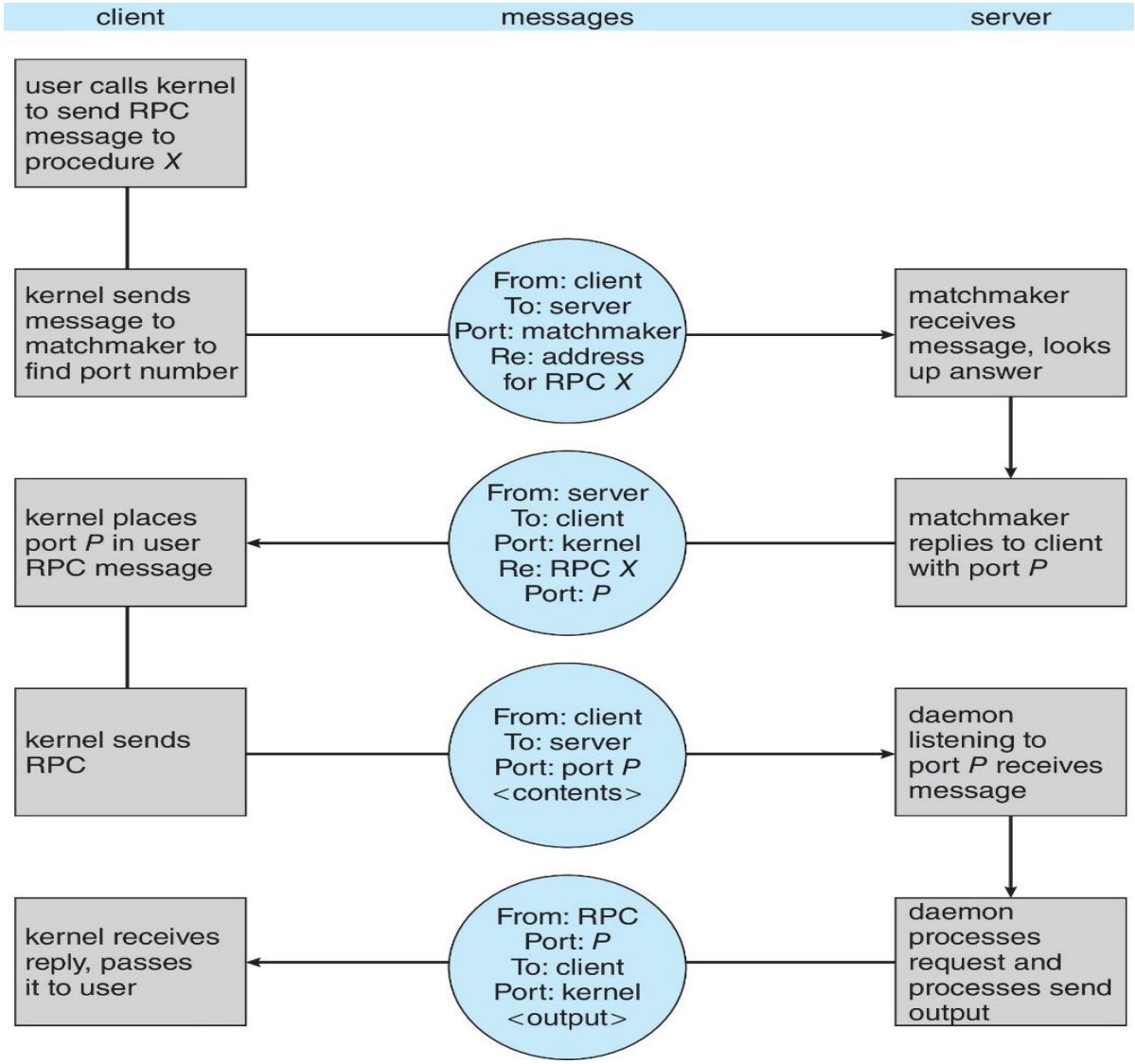
Remote Procedure Calls（远过程调用）

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **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
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Execution of Android RPC

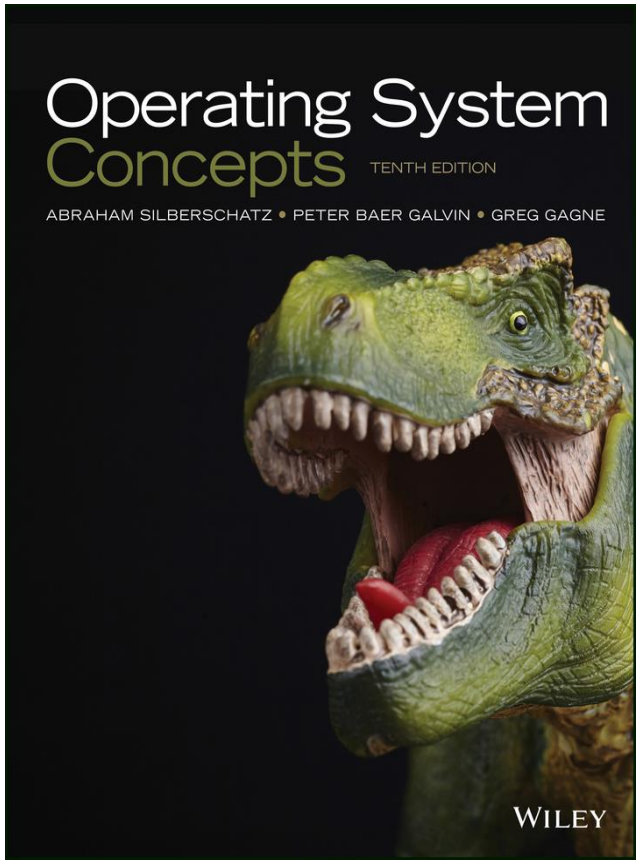


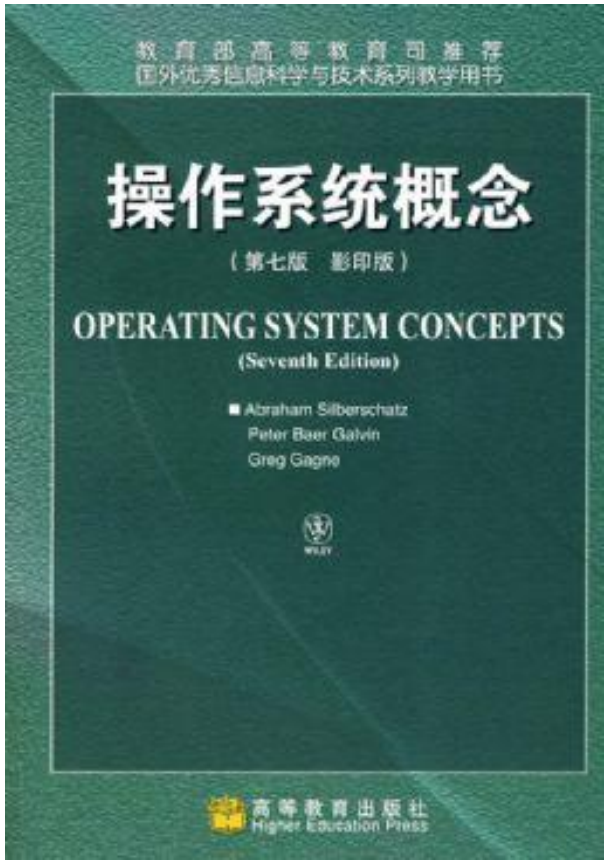
Homework

- 布置在“学在浙大”中，请按时完成

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