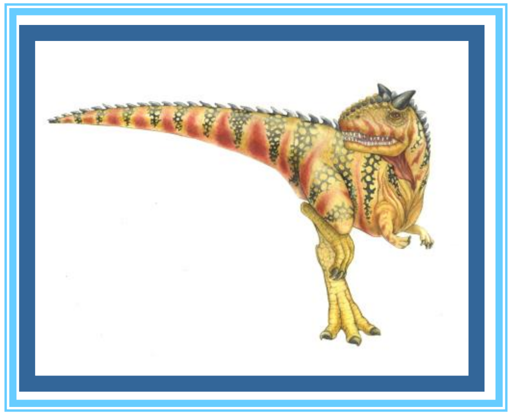
Chapter 3: Processes



Chapter 3: Processes

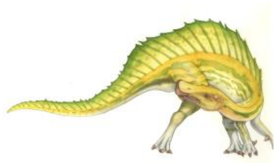
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication(IPC)
- Examples of IPC Systems
- Communication in Client-Server Systems

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 communication in client-server systems

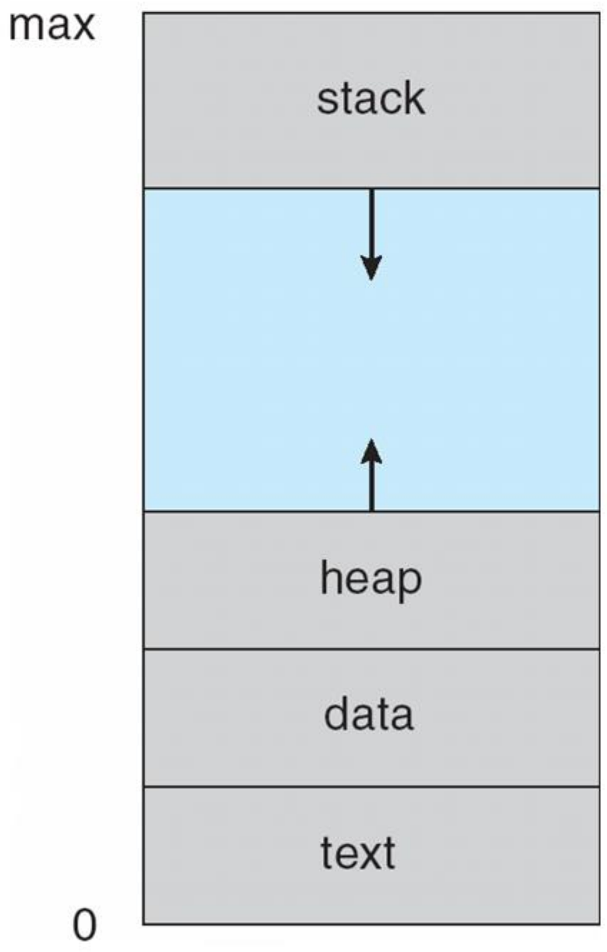
Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Allow multiple programs to loaded into memory and to be executed concurrently.（Textbook uses the terms “job” or “task” and process almost interchangeably）
- Process – a program in execution; process execution must progress in sequential fashion
- A program is a passive entity; a process is an active entity. Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program



Process Concept

- Two types process
  - System process
  - User process
- A process includes(空间布局):
  - program counter
  - stack
  - data section



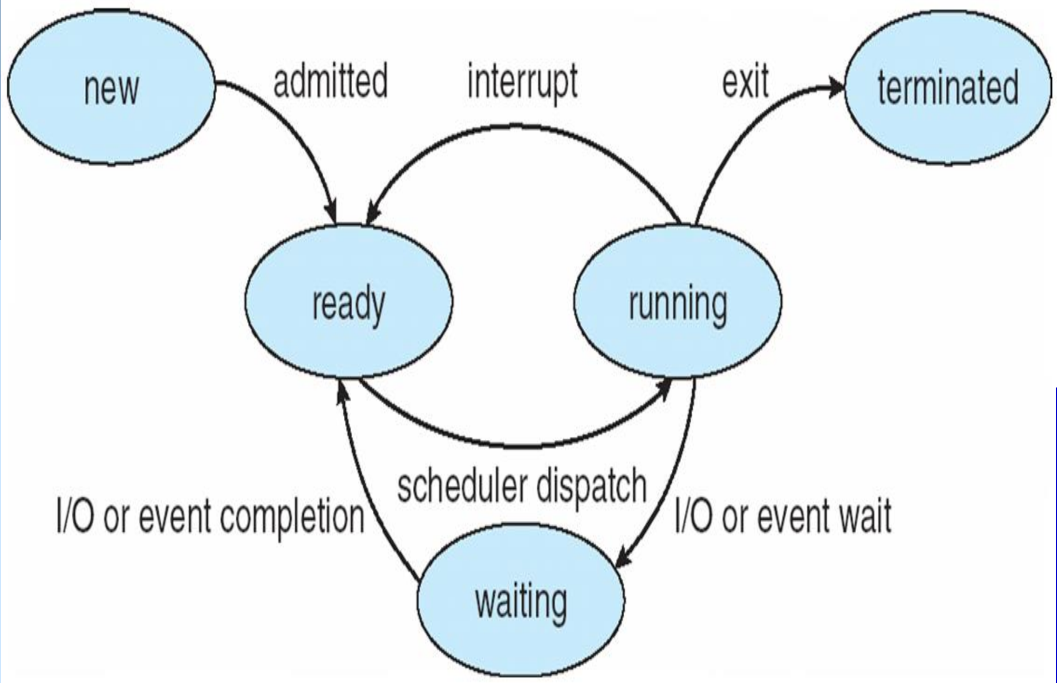
Process Concept

- A process is related to:
  - PCB(Process Control Block)
  - A executable file
  - Some state
  - A queue (ready, run, waiting)
  - Some resources (memory, file, CPU, device)

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



运行状态 $\longrightarrow$ 等待状态

进程阻塞

等待状态 $\longrightarrow$ 就绪状态

进程唤醒

新建进程置为就绪状态

进程创建

进程终止（消亡）

进程撤消

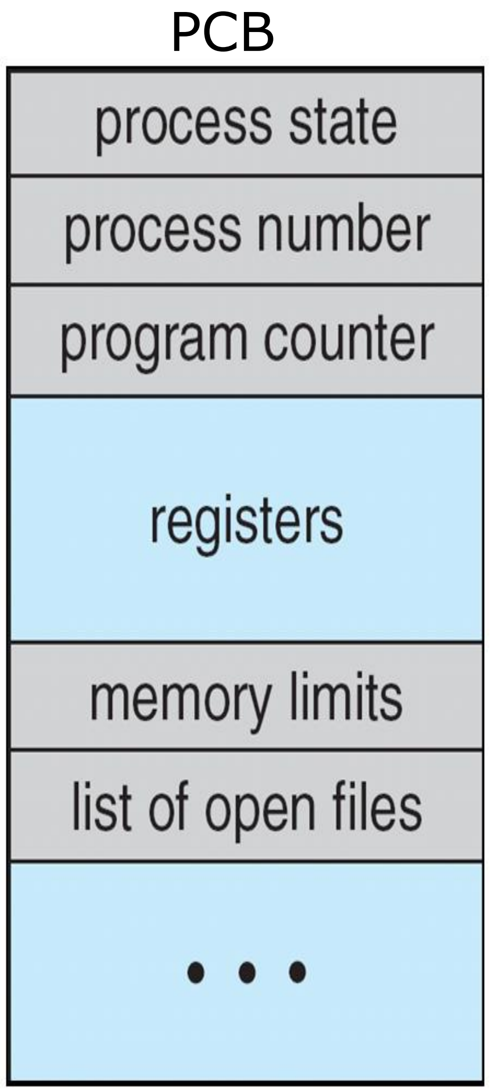
就绪状态 $\longrightarrow$ 运行状态

进程调度

Process Control Block (PCB)

Information associated with each process

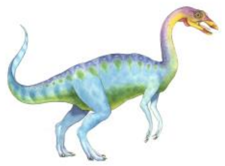
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



PCB in Linux

- task_struct: PCB of a process
- Task array: (1)Saves all pointers of task_struct

  (2)default size is 512（default process num.）
- Information in task_struct of Linux
  - Id: process id, user id, group id
  - State:
  - Scheduling information:
  - Family information: father, son
  - Communication information
  - Timer
  - file opened
  - Context



PCB in Linux

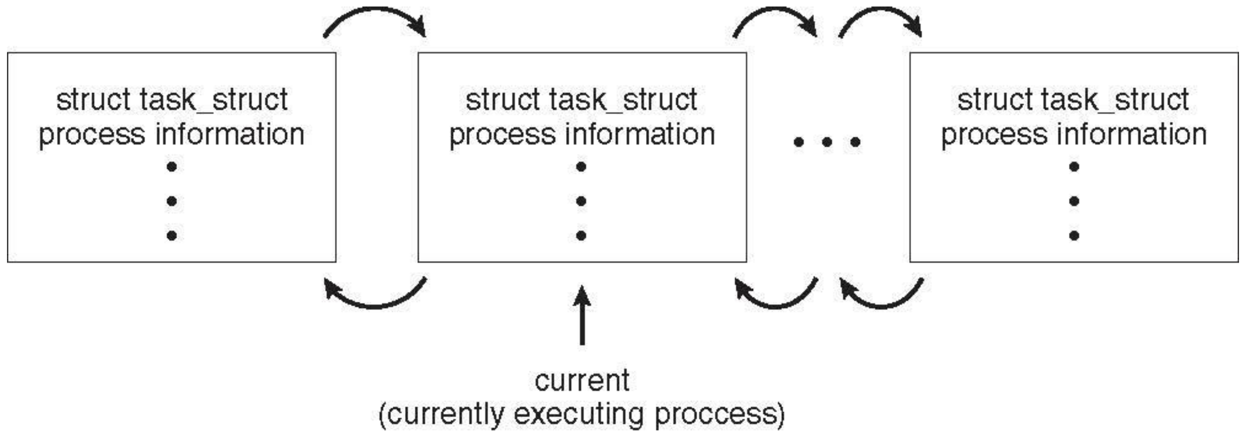
- Process Scheduling Information in task_struct of Linux
  - Policy: realtime or normal
  - Priority : the whole time a process can execution

    [variesing in different Linux versions]
  - Rt_priority : the relative priority of a realtime process
  - Counter: time a process can execution

Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



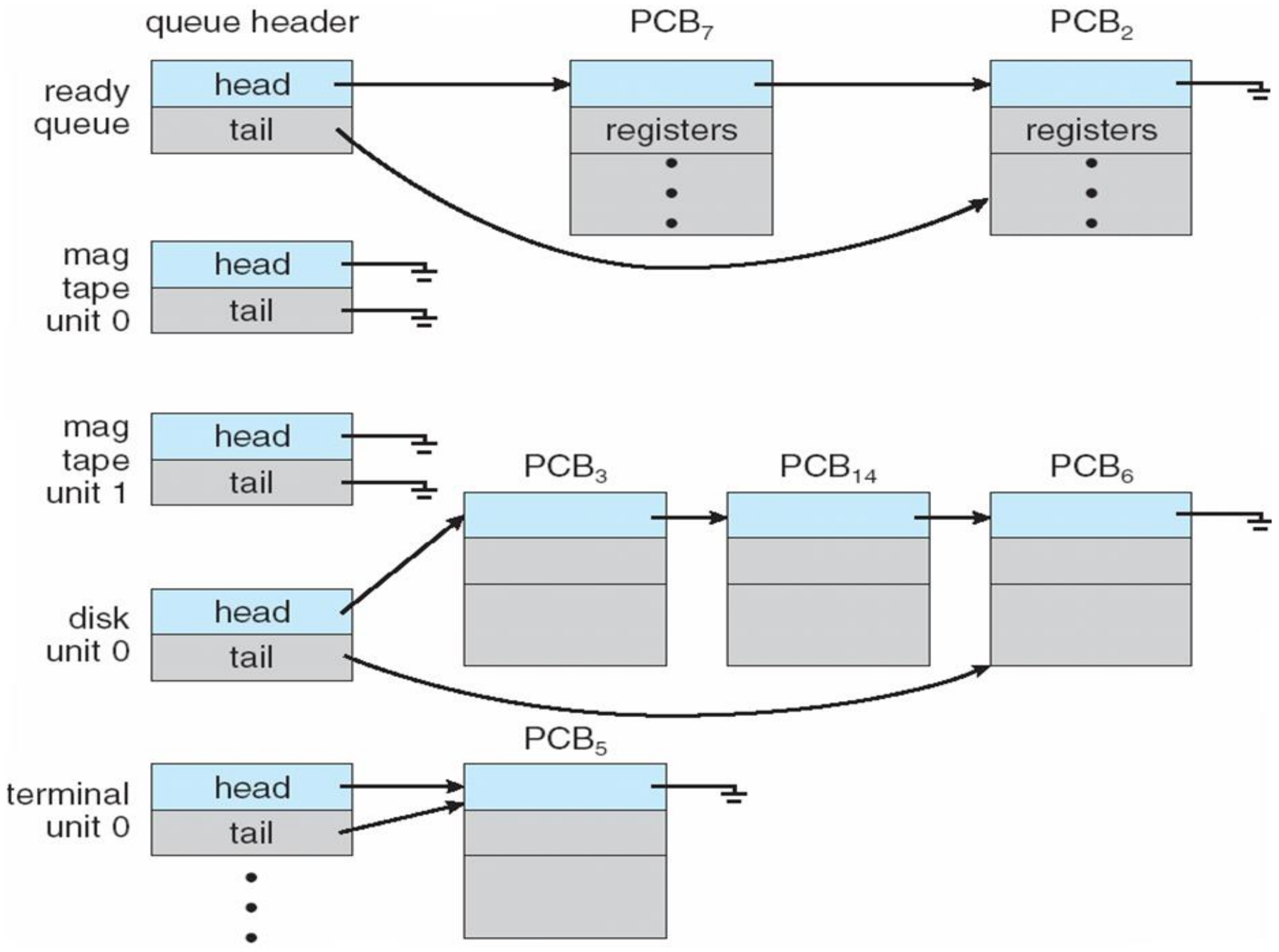
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

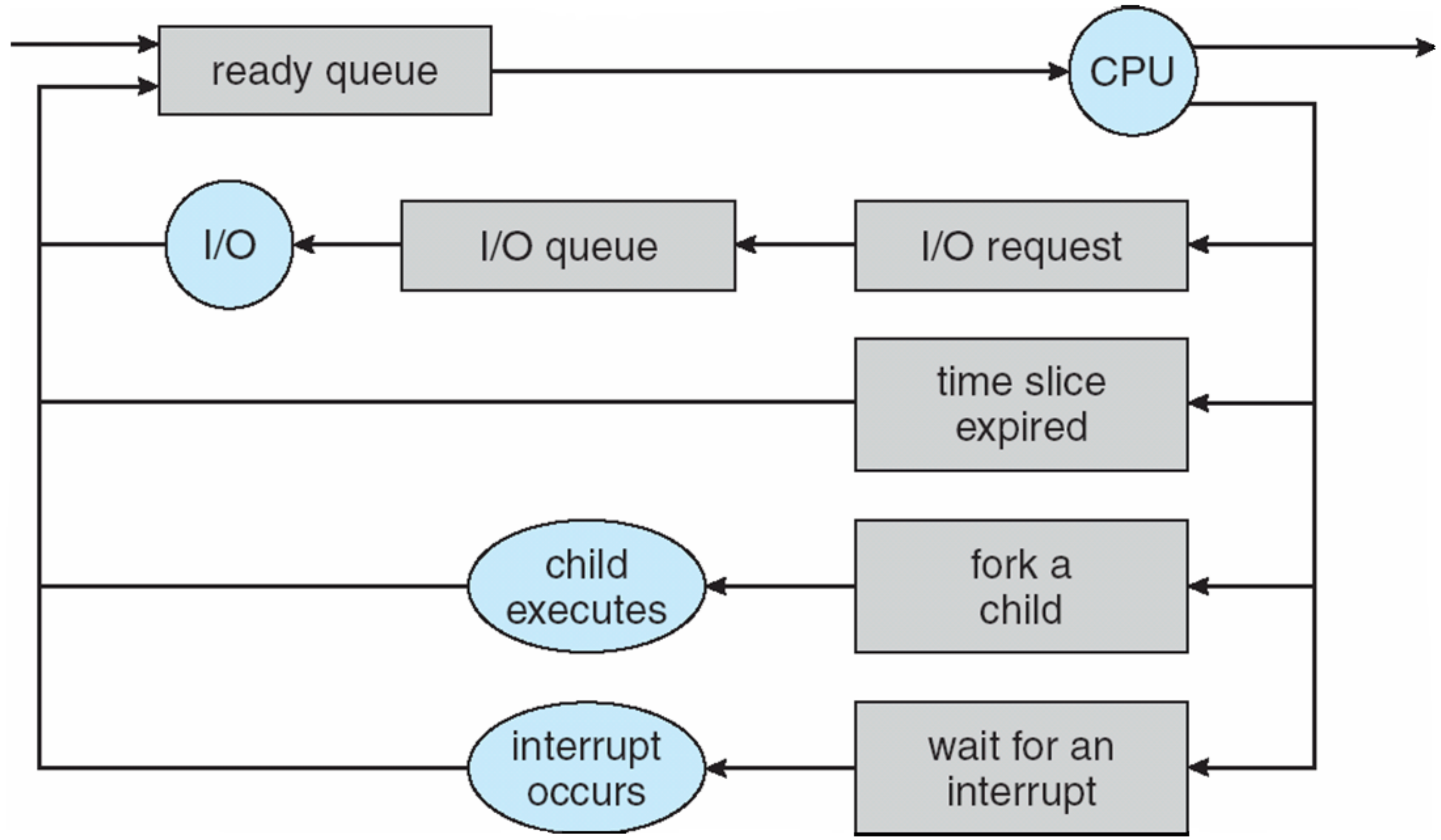
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **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

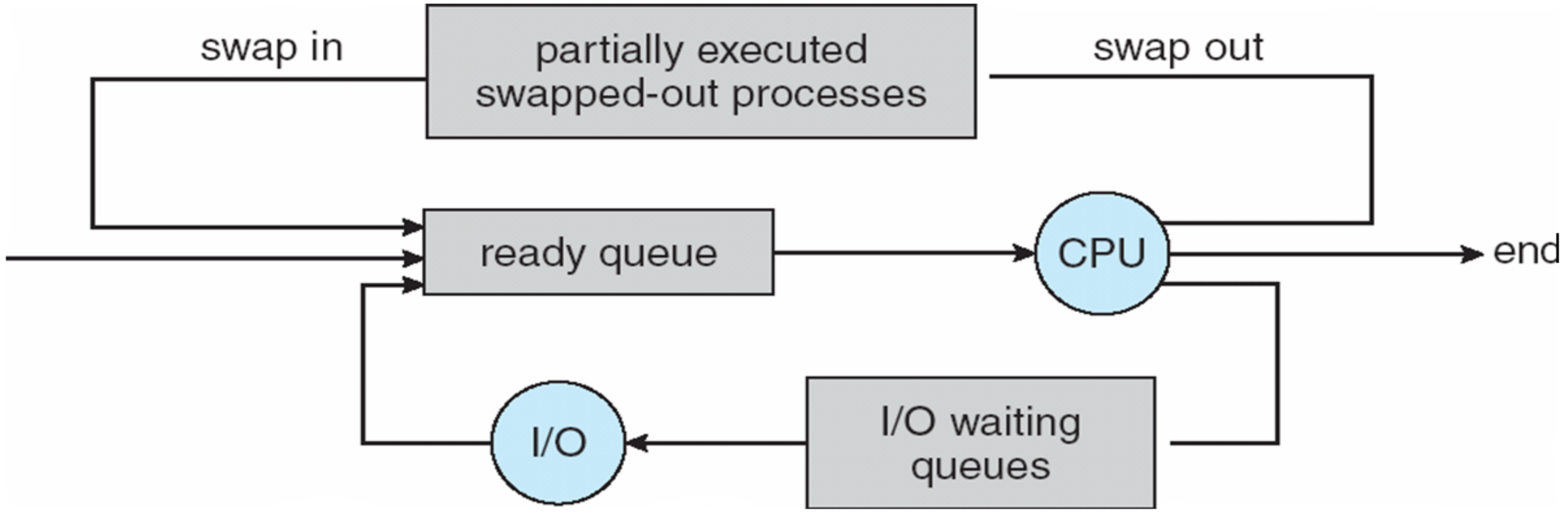


Representation of Process Scheduling



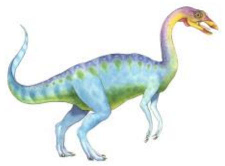
Schedulers

- **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；
- Medium-term scheduling（交换式调度）；



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*

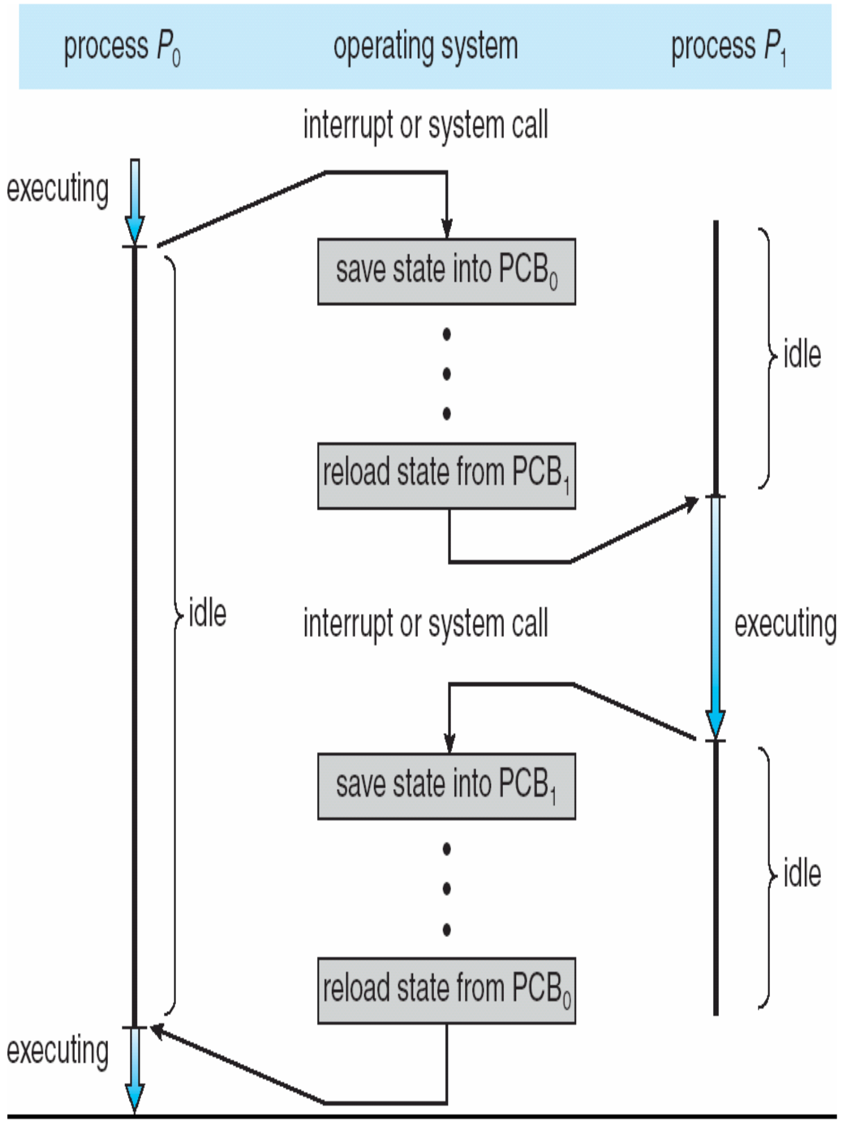


Schedulers (Cont)

- 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
  - Others?
- Long-term scheduler controls the process mix of I/O-bound process and CPU-bound process.

CPU Switch From Process to Process

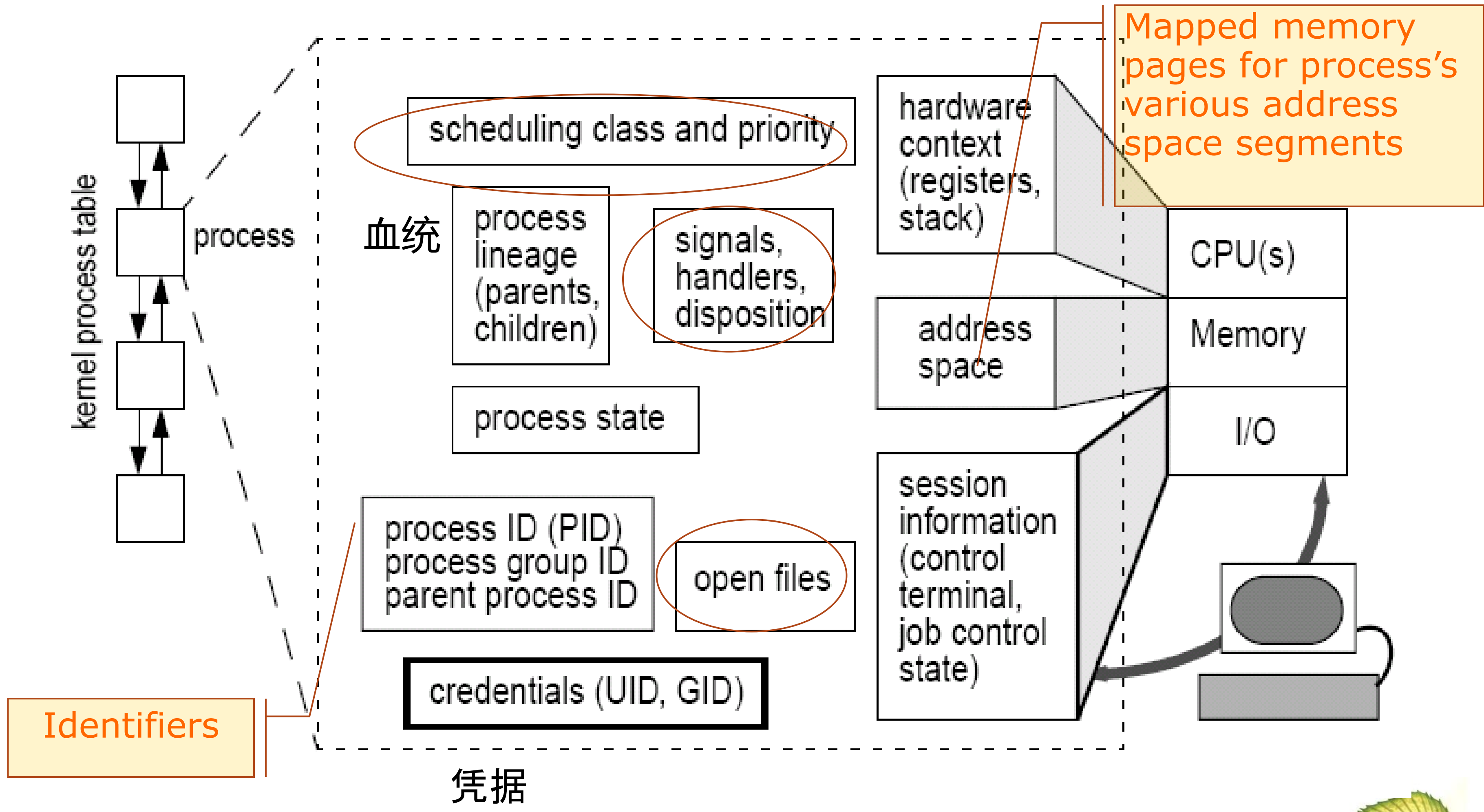
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB:
  - The values of CPU registers
  - The process state
  - Memory-management information



Context Switch

- Process context switch in Linux:
  - Saving user data:
    - Code, data, stack, shared memory
  - Saving Register data
    - PC\PSW\SP(Stack Pointer)\PCBP(PCB指针)\ISP (中断栈指针)
  - Saving memory management information
    - Virtual memory management data, e.g. Page table

Solaris: Process Execution Environment


kernel process table
process
血统
scheduling class and priority
process lineage (parents, children)
signals, handlers, disposition
process state
process ID (PID) process group ID parent process ID
open files
credentials (UID, GID)
hardware context (registers, stack)
address space
session information (control terminal, job control state)
CPU(s)
Memory
I/O
Mapped memory pages for process's various address space segments
Identifiers
凭据

Process control

- 进程有生命周期:产生、运行、暂停、终止。
- 实现这些状态的操作叫进程控制——CPU管理的部分（其他还有进程同步、通信和调度）。进程控制包括：
  - 进程创建→创建原语
  - 进程撤消→撤销原语
  - 进程阻塞→阻塞原语
  - 进程唤醒→唤醒原语

> 原语：是由若干条机器指令构成的用以完成特定功能的一段程序。执行期间不允许中断。
>
> 实现原子性:1.关中断；2.固化为机器指令
>
> 为什么要原语?

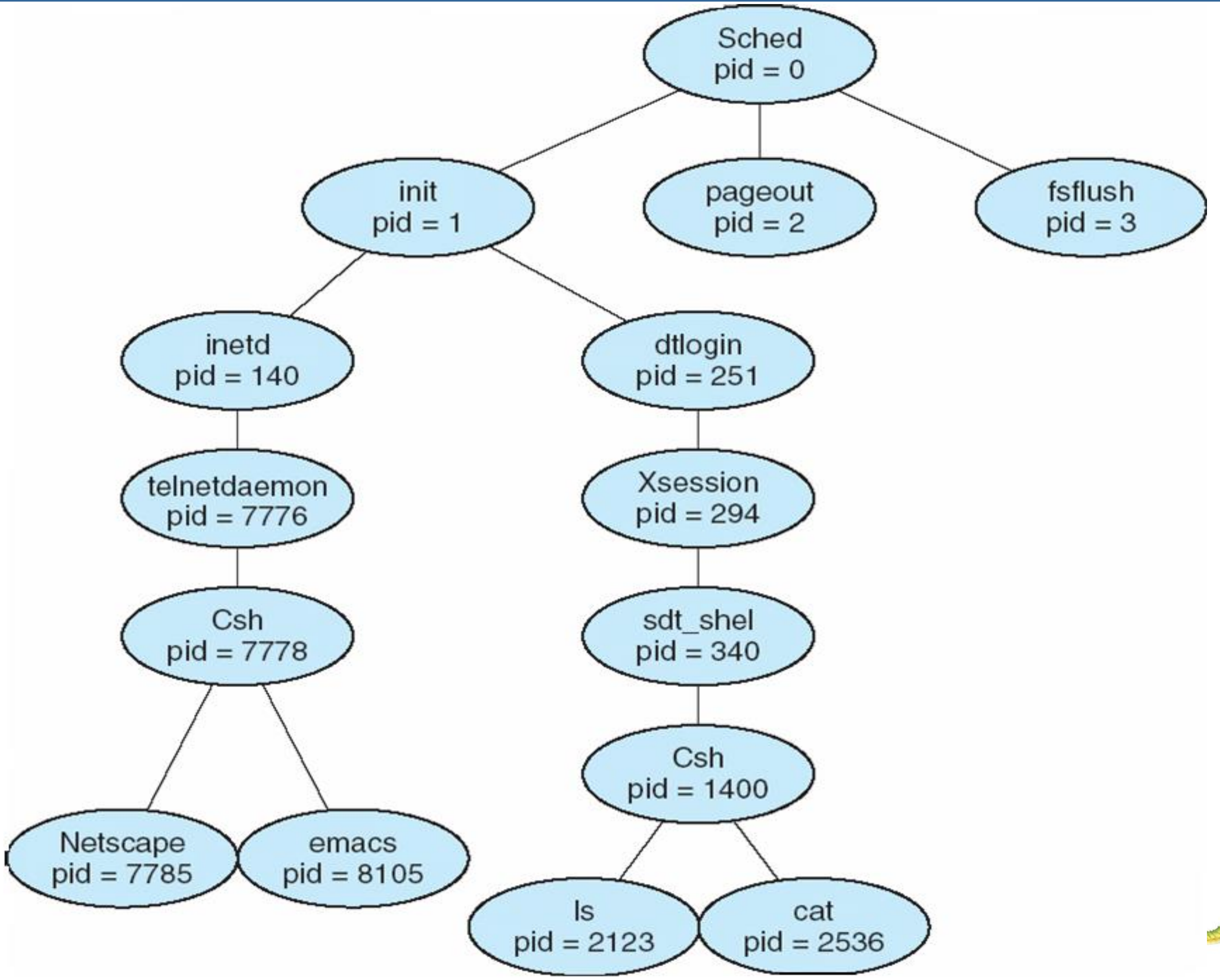
Operations on processes

- Process creation
- UNIX examples
  - 在UNIX系统中用户键入一个命令（如date，ps,ls)，shell就创建一个进程。
  - 用户程序可使用fork()系统调用创建多个进程，每个进程执行一个程序段。

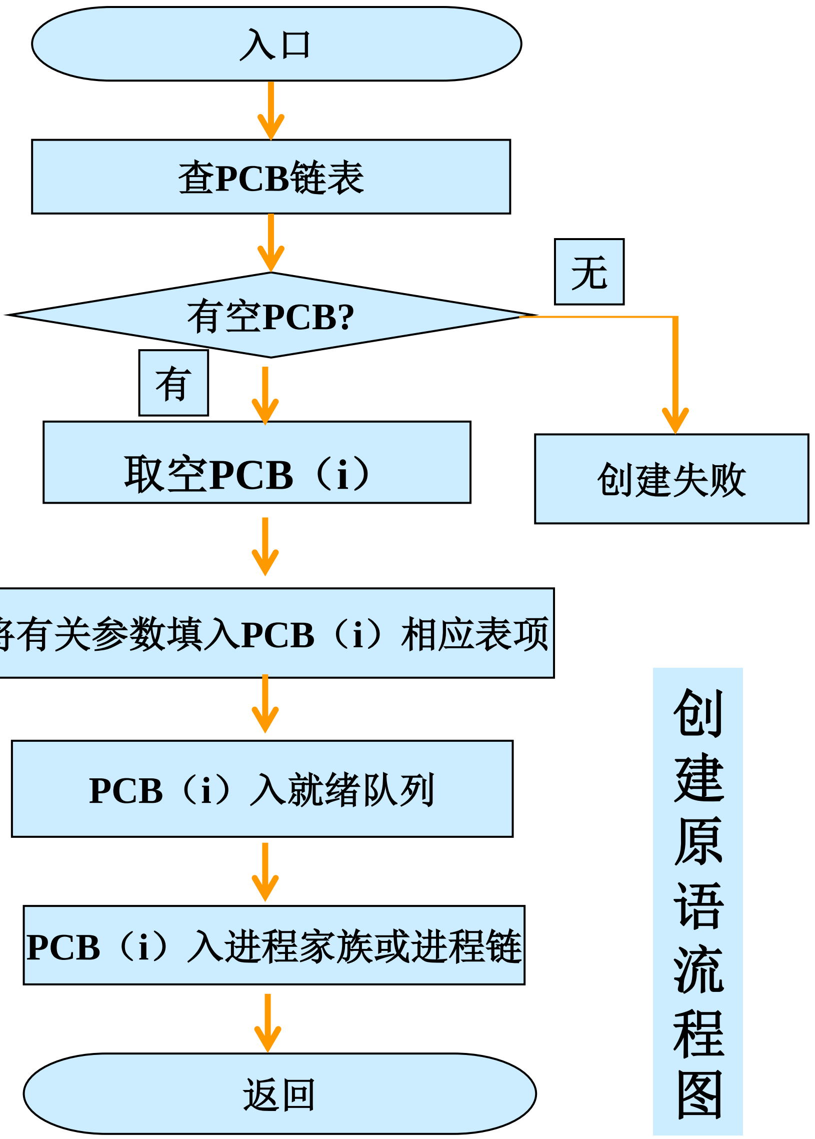
Process Creation

- **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

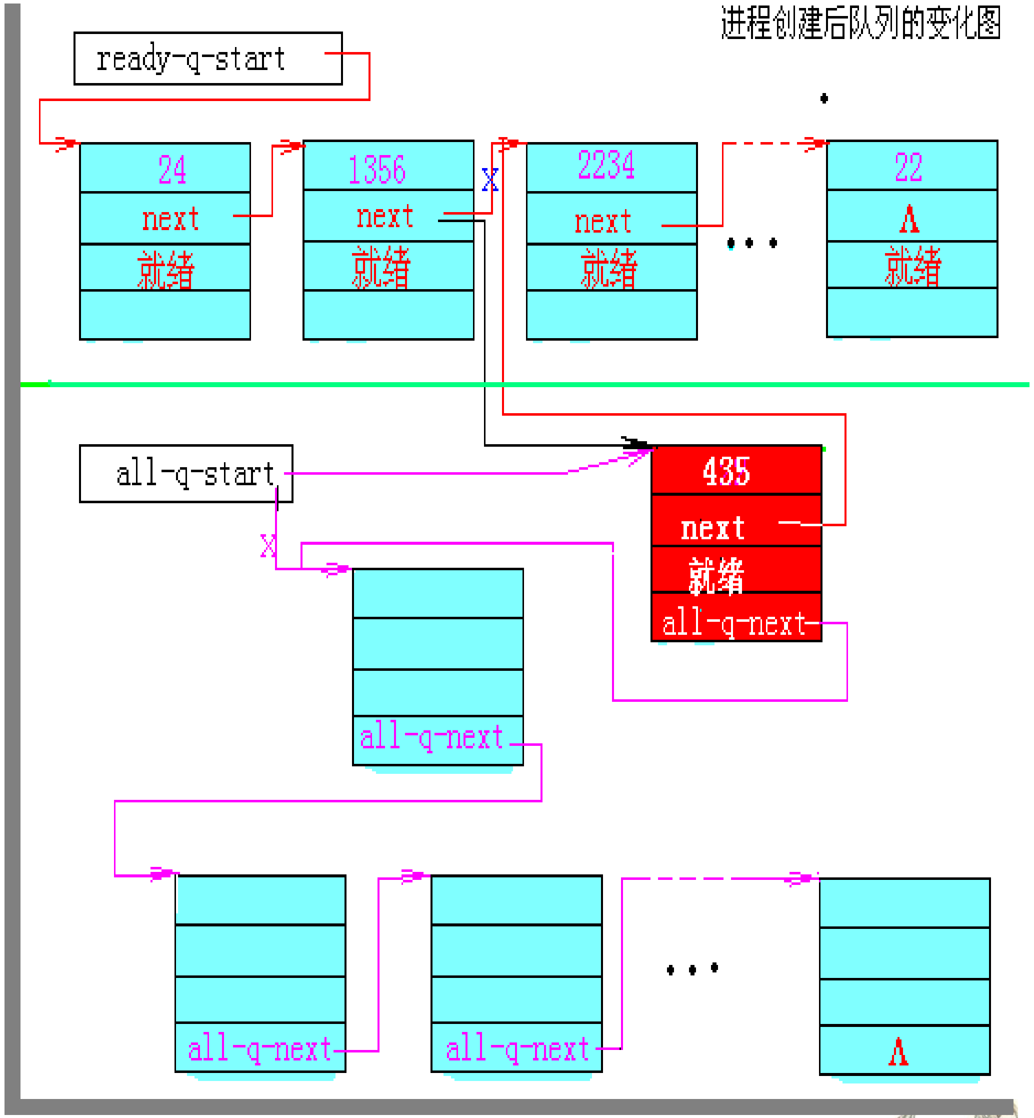
A tree of processes on a typical Solaris



进程创建过程

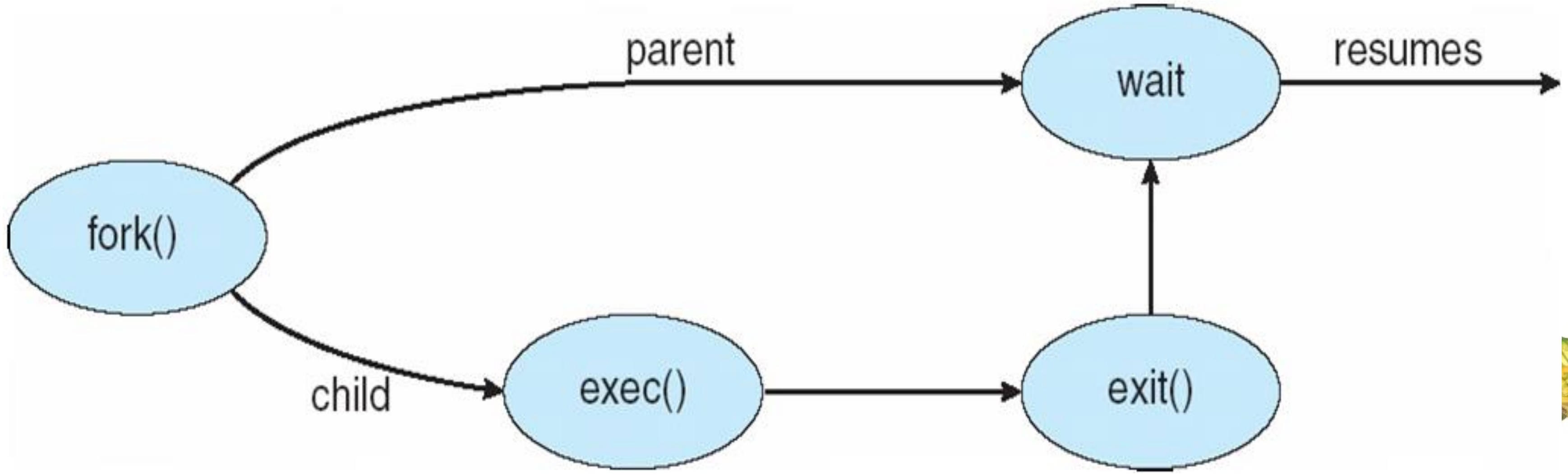


创建原语流程图



Process Creation (Cont)

- Address space
  - Child duplicate of parent;
  - Child has a program loaded into it;
- UNIX examples
  - **fork** system call creates new process;
  - **exec** system call used after a **fork** to replace the process' memory space with a new program;



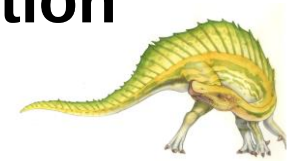
C Program Forking Separate Process

```
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);
     }
}
```

- pid=fork(); 返回值如果：pid>0，CPU在父进程中; pid=0，在子进程中; pid<0,创建失败;

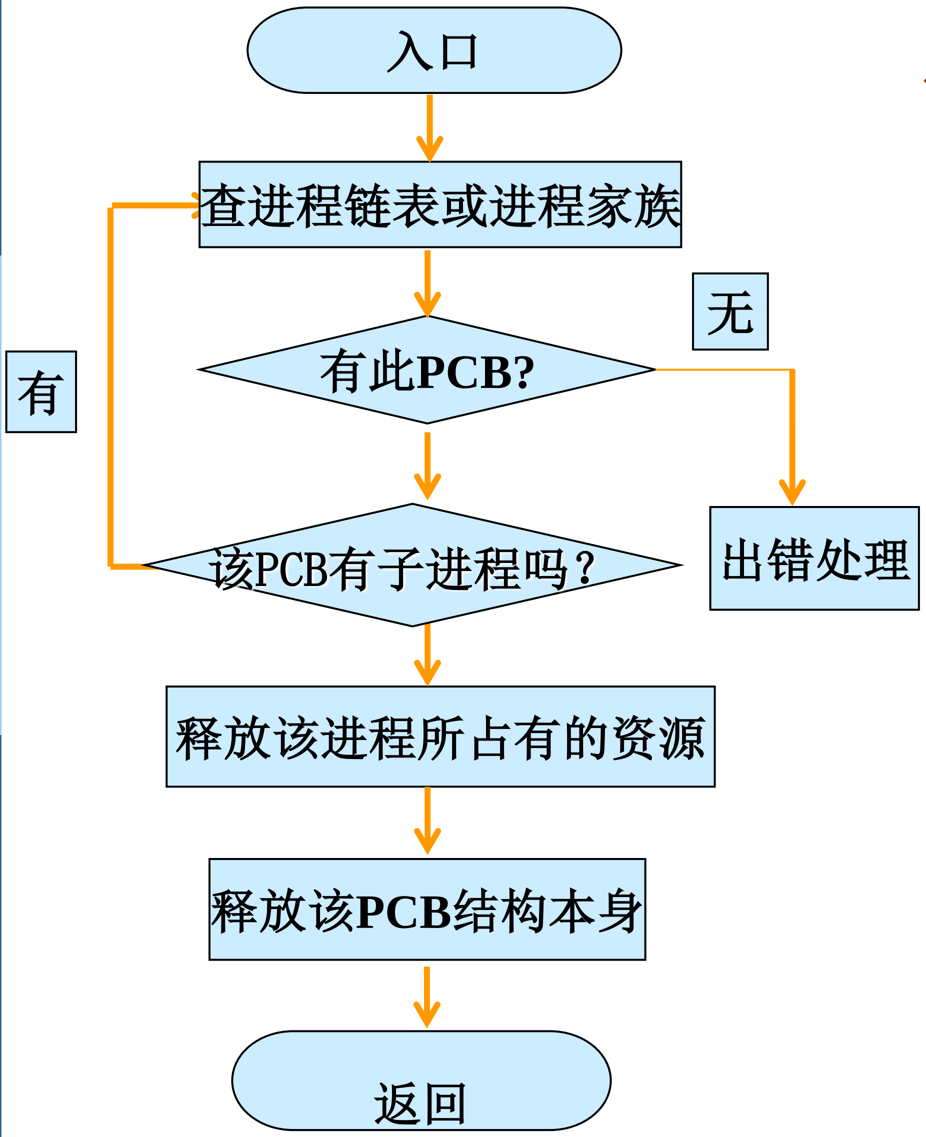
Process Termination

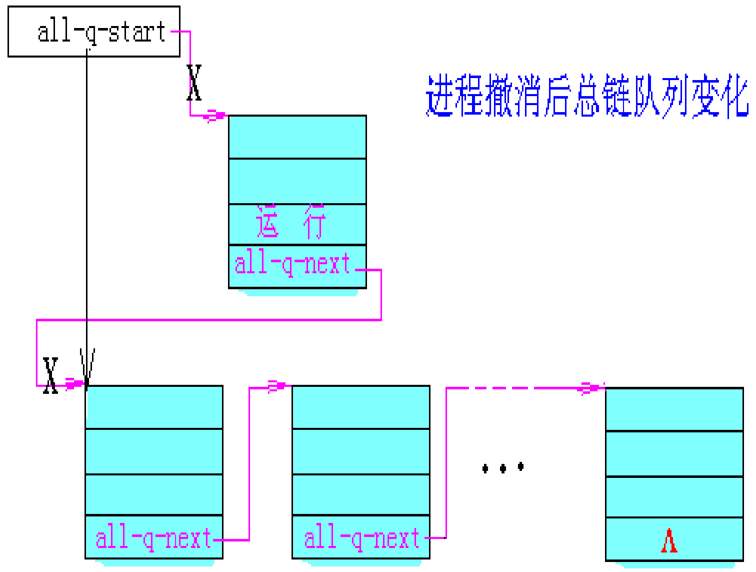
- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  1. Child has exceeded allocated resources
  2. Task assigned to child is no longer required
  3. If parent is exiting
     - Some operating system do not allow child to continue if its parent terminates
       - All children terminated - **cascading termination**



Process Termination

- UNIX系统exit()：进程自我终止
- 释放占用的所有资源：
  - 释放内外存空间；
  - 关闭所有打开文件；
  - 释放当前目录；
  - 释放共享内存段和各种锁lock；





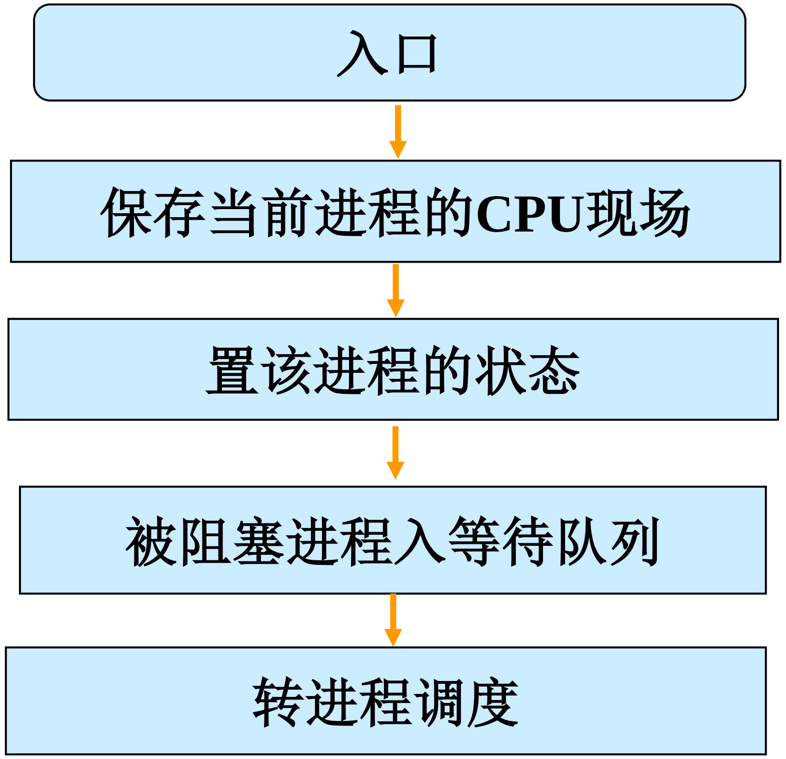
注：进程撤消是在进程处于运行状态下进行的。

Process Blocking

- 引起进程阻塞的事件：

  运行状态进程因等待某个事件的发生（比如等待打印机、同步事件等）而不能继续运行时，调用阻塞原语，把进程置为阻塞状态，并转进程调度程序（让出处理机）。

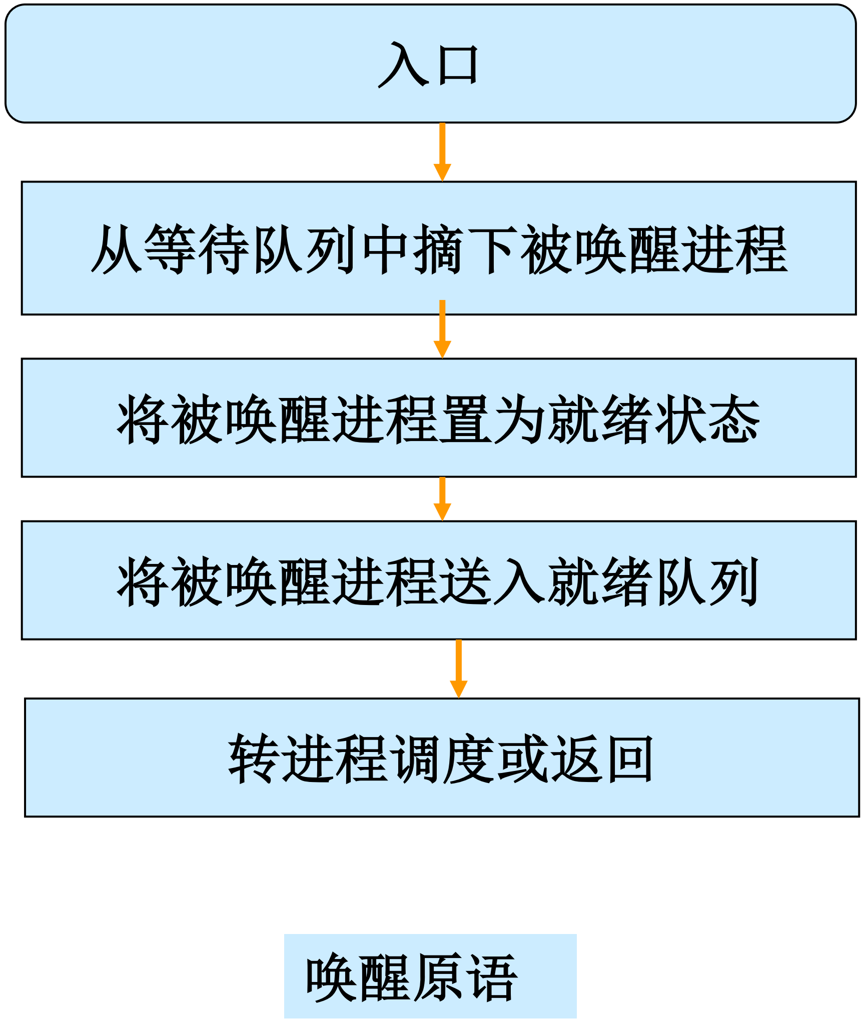
- 引起队列的改变。



阻塞原语：转进程调度是不让处理机空转

Process wakeup

- 当进程等待的事件发生时，该进程将被唤醒(由唤醒操作完成)。
- 唤醒进程有两种方法:
  - 由系统进程唤醒;
  - 事件发生时唤醒;



UNIX相关系统调用介绍

1. fork(): 创建一个子进程.

2. exec(): 装入并执行相应文件.

3. wait(): 父进程处于阻塞状态, 等待子进程终止, 其返回值为所等待子进程的进程号.

4. exit(): 子进程自我终止, 释放所占资源, 通知父进程可以删除自己 。

5. getpid():获得进程号.

6. lockf(files, function, size):用于锁定文件的某些段或整个文件。

7. kill(pid, sig): 一个进程向同一用户的其他进程pid发送一中断信号

8. signal(sig, function): 捕捉中断信号sig后执行function规定的操作。

值	名字	说明
01	SIGHUP	挂起
02	SIGINT	中断，当用户从键盘键入 “del”键时
03	SIGQUIT	退出，当用户从键盘键入 “quit”键时
04	SIGILL	非法指令
05	SIGTRAP	断点或跟踪指令
06	SIGIOT	IOT指令
07	SIGEMT	EMT指令
08	SIGFPE	浮点运算溢出
09	SIGKILL	要求终止进程
10	SIGBUS	总线错误
11	SIGSEGV	段违例，即进程试图去访问其地址空间以外的地址
12	SIGSYS	系统调用错
13	SIGPIPE	向无读者的管道中写数据
14	SIGALARM	闹钟
15	SIGTERM	软件终止
16	SIGUSR1	用户自定义信号
17	SIGUSR2	用户自定义信号
18	SIGCLD	子进程死
19	SIGPWR	电源故障

3.7 UNIX相关系统调用介绍

9. pipe(fd);int fd[2];

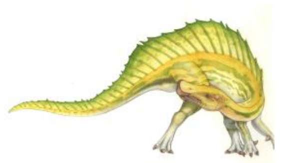
其中fd[1]是写端，fd[0]是读端；一方向管道中写，另一方从管道中读出，从而实现进程间通信。

10. 暂停一段时间sleep；在指定的时间seconds内挂起本进程。

用格式为：“unsigned sleep(unsigned seconds);”；返回值为实际的挂起时间。

11. 暂停并等待信号pause；调用格式为“int pause(void);”

调用pause挂起本进程以等待信号，接收到信号后恢复执行。当接收到中止进程信号时，该调用不再返回。



实例

- 使用fork()创建两个子进程。让每个进程在屏幕上显示一个字符：父进程显示字符“a”；子进程分别显示字符“b”和字符“c”。

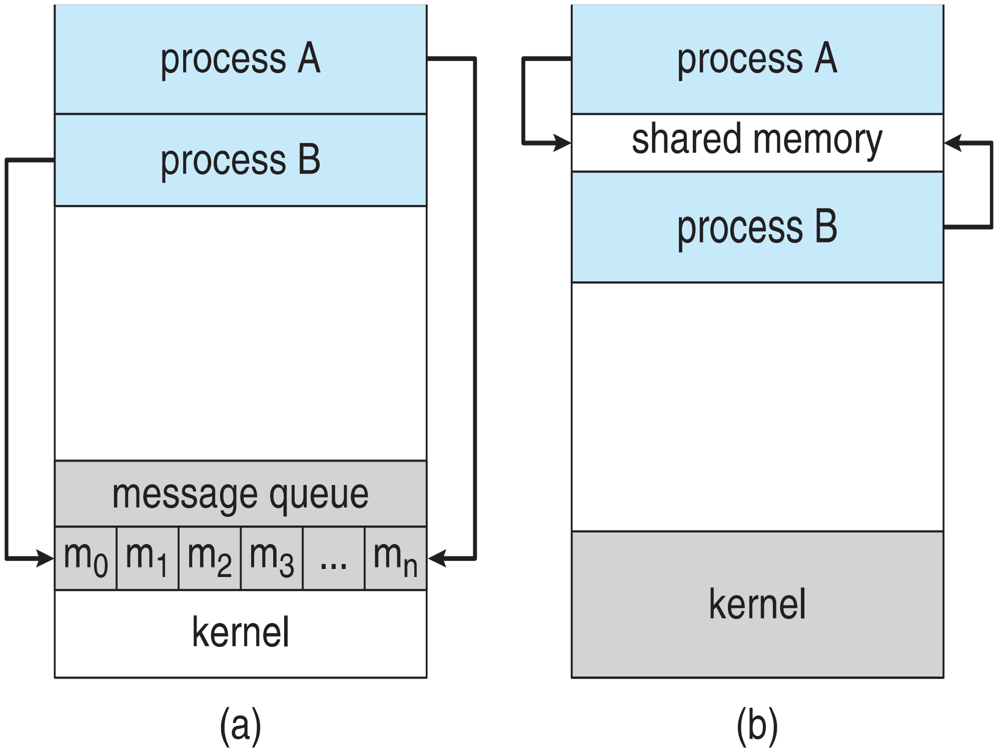
```
#include <stdio.h>
main()
{   int p1,p2;
    while((p1 = fork()) == -1);
    if(p1==0)------------创建成功一子进程
        putchar('b');
    else
    { while((p2 = fork()) == -1);
        if(p2 == 0)
            putchar(‘c’); -------创建成功另一子进程
        else
            putchar(‘a’);--------父进程
    }
}
```

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing;
  - Computation speedup;
  - Modularity;
  - Convenience;

Interprocess Communication

- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Message passing(a)可适用与网络间通信
  - Shared memory(b)



- Interprocess communication using Shared memory requires communicating processes to establish a shared memory;
- The form of the exchanged data and location are determined by the communicating processes and are not under the OS's control;

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;

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 (10个单元) items in the buffer at the same time.

Bounded-Buffer – Shared-Memory Solution

Bounded-Buffer : Producer/Consumer

```
/* Produce an item */
while (true) {
        while (((in = (in + 1) % BUFFER SIZE count)  == out) ;
              /* do nothing -- no free buffers */
              buffer[in] = item;
              in = (in + 1) % BUFFER SIZE;
 }

/* Consume an item */
while (true) {
      while (in == out) ; // do nothing -- nothing to consume
         item = buffer[out]; // remove an item from the buffer
          out = (out + 1) % BUFFER SIZE;
          return item;
 }
```

Interprocess Communication – Message Passing

- 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 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?（一对通信进程间可以有多少个链接）
- 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

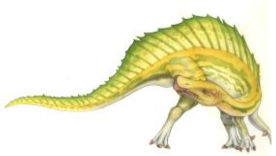
- Direct Communication
- Indirect Communication

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
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional



Indirect Communication

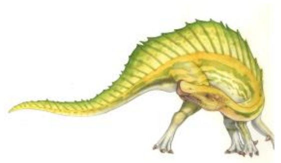
- Operations
  - create a new mailbox;
  - send and receive messages through mailbox;
  - destroy a mailbox;
- Primitives are defined as:

  send(*A*, *message*) – send a message to mailbox A

  receive(*A*, *message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  ① Allow a link to be associated with at most two processes
  ② Allow only one process at a time to execute a receive operation
  ③ Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.



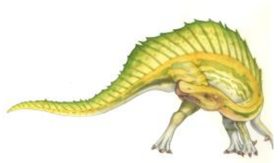
Message-passing----Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Buffering

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

1. Zero capacity – 0 messages
   Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full
3. Unbounded capacity – infinite length
   Sender never waits



Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

```
segment id = shmget(IPC PRIVATE, size, S IRUSR | S
```

MODE:READ,WRITE ,OR BOTH

Pointer indicating the share memory address
NULL: selecting by OS

  - Process wanting access to that sha memory must attach it

1:read only
0:R/W

```
shared memory = (char *) shmat(id, NULL, 0);
```

  - Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

  - When done a process can detach(分离) the shared memory from its address space

```
shmdt(shared memory);
```

Examples of IPC Systems - Mach

- **Mach communication is message based**
  - Even system calls are messages
  - Each task gets two mailboxes(ports) at creation
  - Only three system calls needed for message transfer

    `msg_send(), msg_receive(), msg_rpc()`
  - **Mailboxes needed for commuication, created via**

    `port_allocate()`

Local Procedure Calls in Windows XP

Message-passing centric via local procedure call (LPC) facility

- Only works between processes on the same system
- LPC是一种称为“端口(Port)”的进程间通信机制，端口分“连接端口(connection port)”和“通信端口(communication port)”两种；

服务器端

客户端

1、NtConnectPort()创建客户方的无名通信端口，向上述命名的连接端口发出连接请求；

2、线程被唤醒、返回所创建的无名通信端口handle;

3、NtRequestWaitReplyPort()向服务方发送报文，请求得到LPC服务，并因等待而被阻塞。

1、NtCreatePort()创建一个命名的连接端口Port.

2、NtListenPort()等待客户线程的连接请求

3、收到连接请求后，通过NtAcceptConnectPort()创建服务方无名通信端口并返回该端口handle。通过NtCompleteConnectPort()唤醒客户线程。

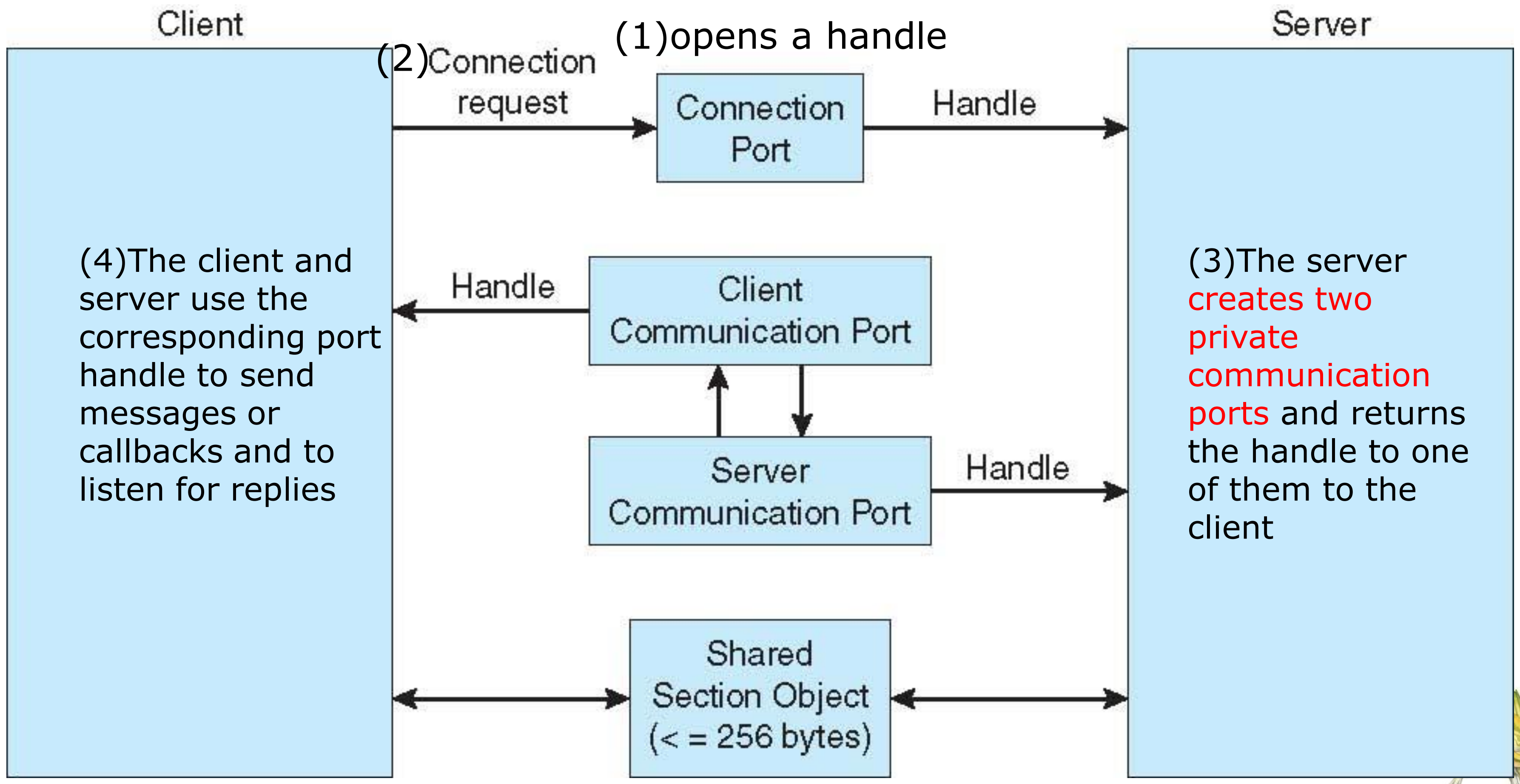
4、创建新线程负责为客户线程提供服务。

5、服务线程因接收到报文而被唤醒，并根据报文内容提供相应的LPC服务。

6、服务线程通过NtReplyPort()向客户方发送回答报文。客户线程解除阻塞。

Local Procedure Calls in Windows XP

- Uses ports (like mailboxes) to establish and maintain communication channels, Communication works as follows:



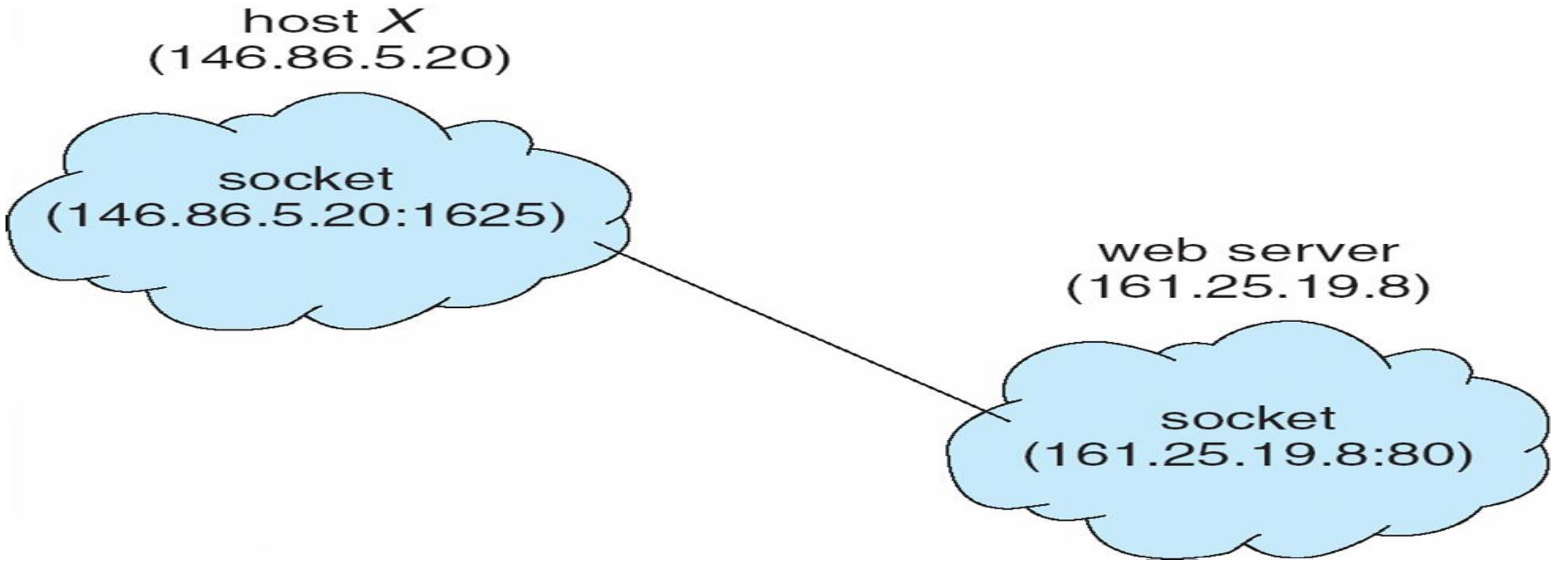
Client
Server
(1)opens a handle
(2)Connection request
Connection Port
Handle
(3)The server creates two private communication ports and returns the handle to one of them to the client
(4)The client and server use the corresponding port handle to send messages or callbacks and to listen for replies
Handle
Client Communication Port
Server Communication Port
Handle
Shared Section Object (< = 256 bytes)

Communications in Client-Server Systems

- 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

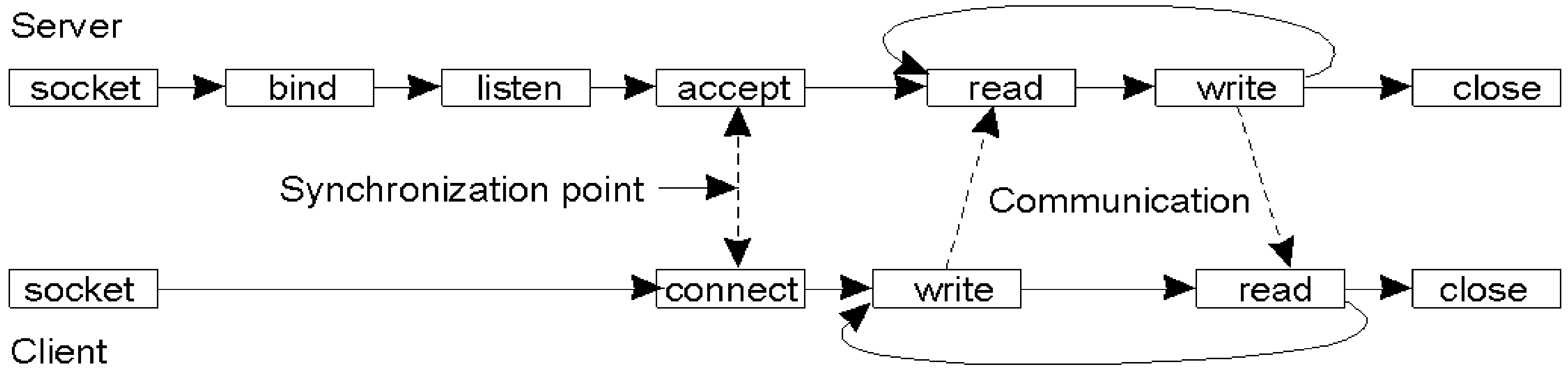


Message-Oriented Transient Communication Berkeley Sockets

Socket primitives for TCP/IP.

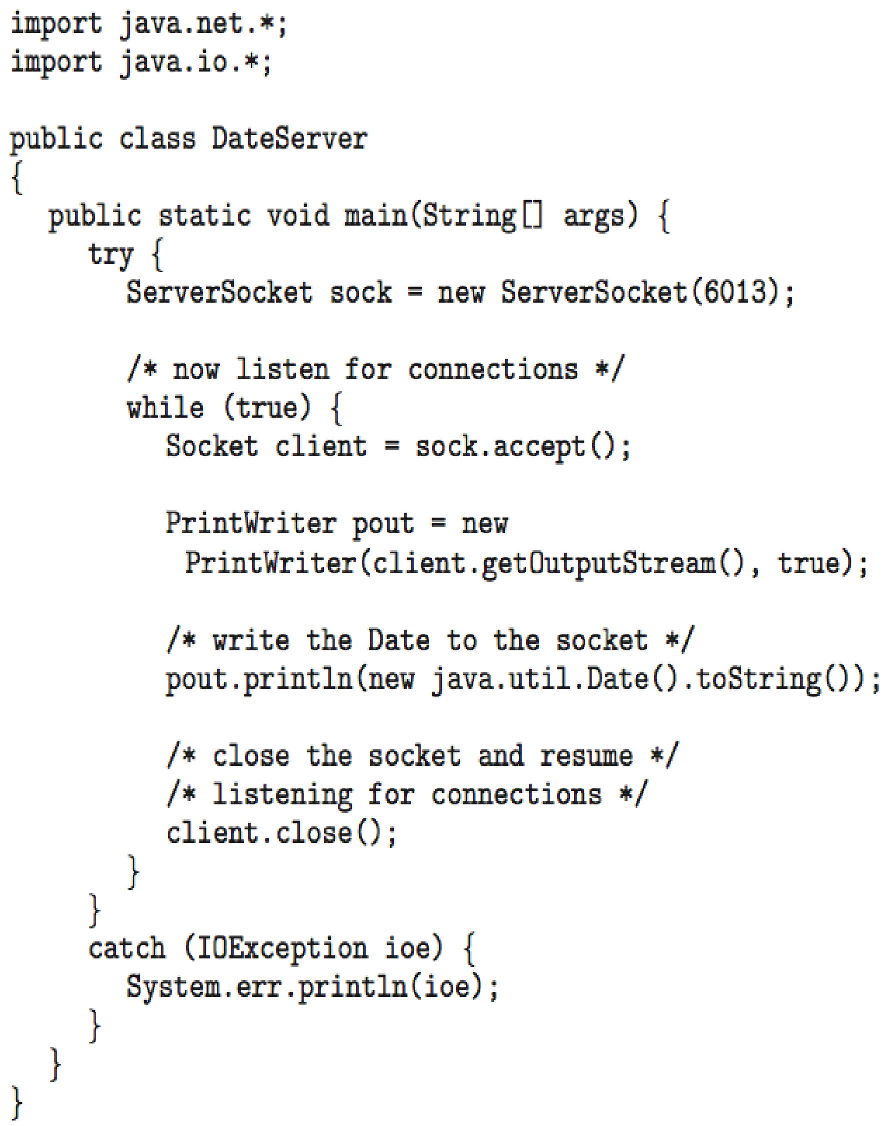
Primitive	Meaning
Socket	**Create a new communication endpoint**
Bind	**Attach a local address to a socket**
Listen	**Announce willingness to accept connections**
Accept	**Block caller until a connection request arrives**
Connect	**Actively attempt to establish a connection**
Send	**Send some data over the connection**
Receive	**Receive some data over the connection**
Close	**Release the connection**

服务器端



Sockets in Java

- Three types of sockets
  - **Connection-oriented** (**TCP**)(右图)
  - **Connectionless** (**UDP**)
  - **`MulticastSocket`** class– data can be sent to multiple recipients
- Consider this “Date” server:



```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

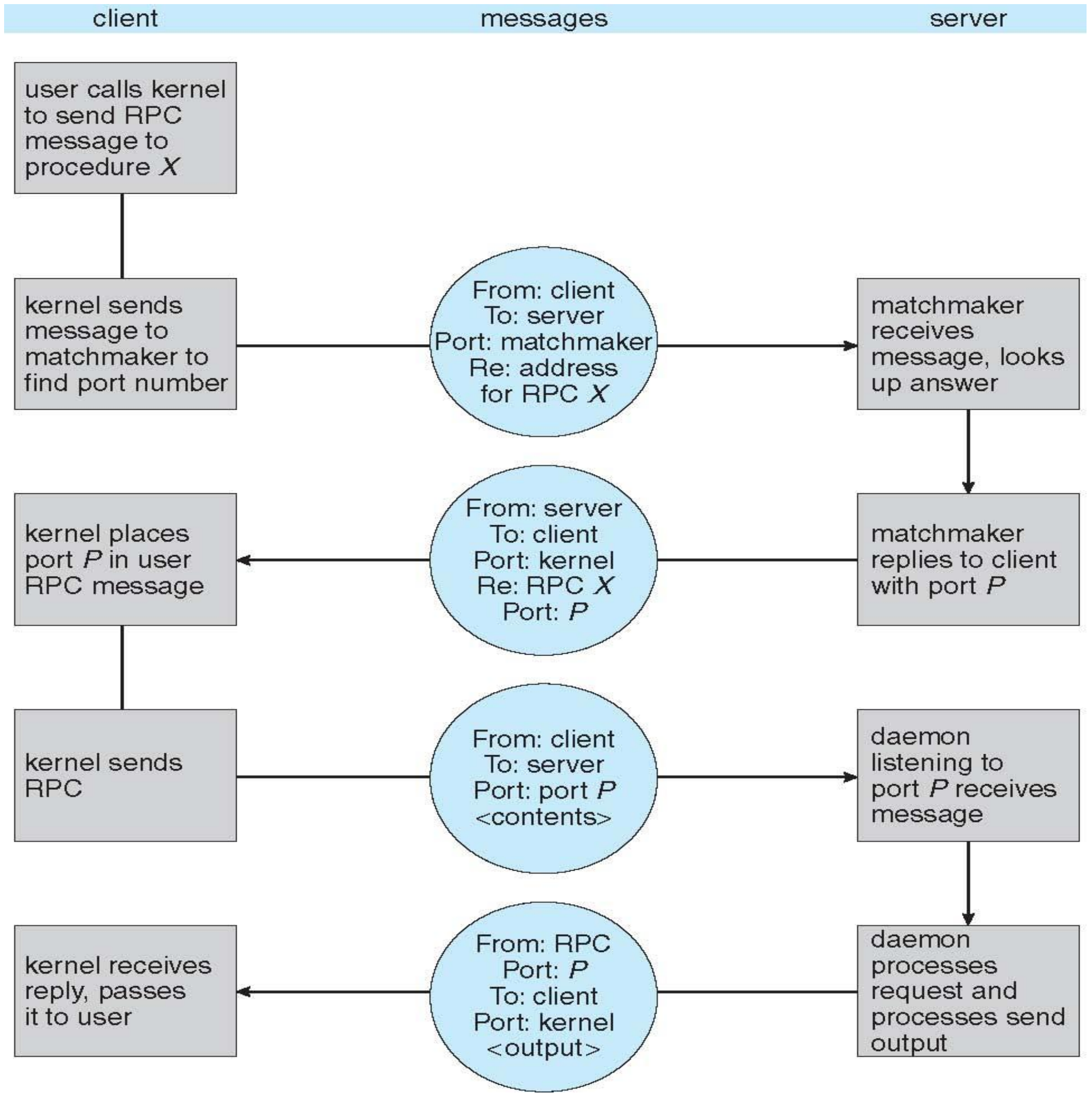
Remote Procedure Calls

- 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 peforms the procedure on the server

Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian（左边低位和右边低位）**
- 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（类似于目录服务器）

Execution of RPC

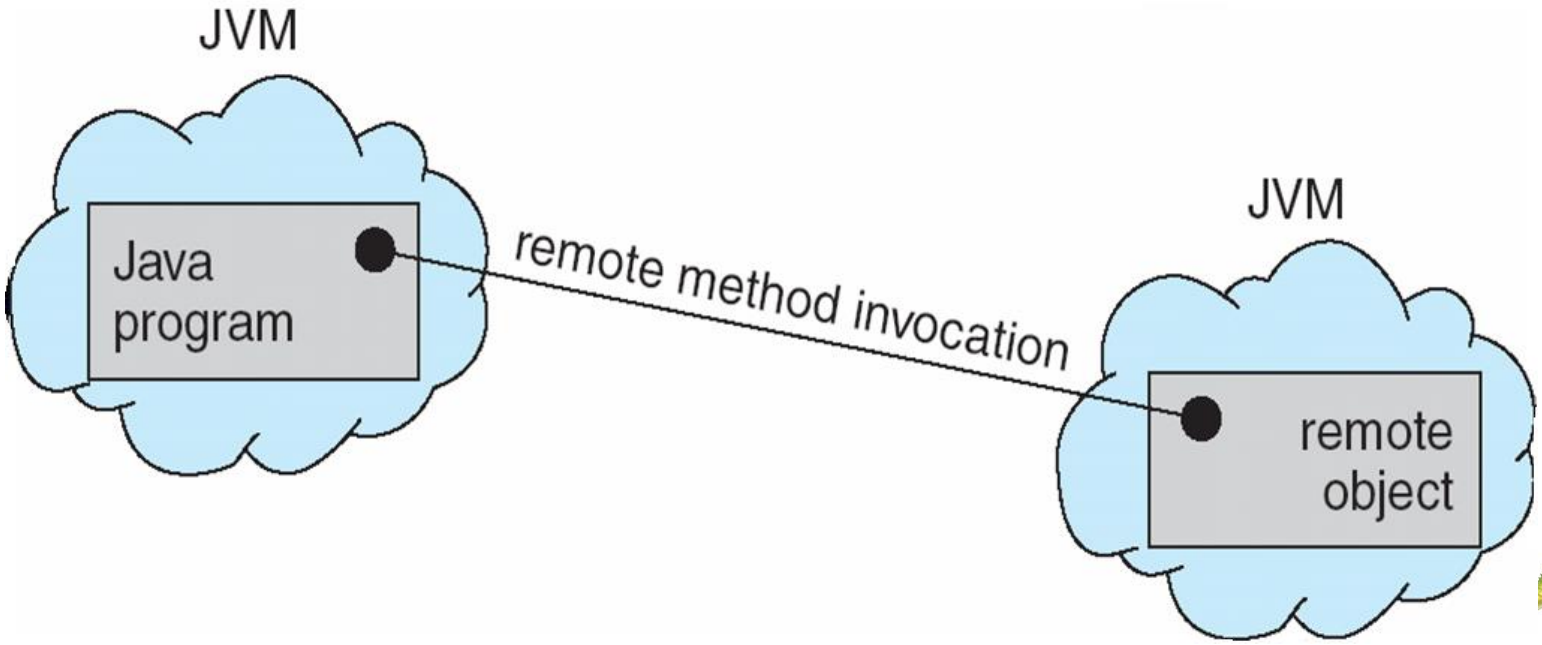

client
messages
server
user calls kernel to send RPC message to procedure *X*
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC *X*
matchmaker receives message, looks up answer
matchmaker replies to client with port *P*
From: server To: client Port: kernel Re: RPC *X* Port: *P*
kernel places port *P* in user RPC message
kernel sends RPC
From: client To: server Port: port *P* <contents>
daemon listening to port *P* receives message
daemon processes request and processes send output
From: RPC Port: *P* To: client Port: kernel <output>
kernel receives reply, passes it to user

Execution of RPC

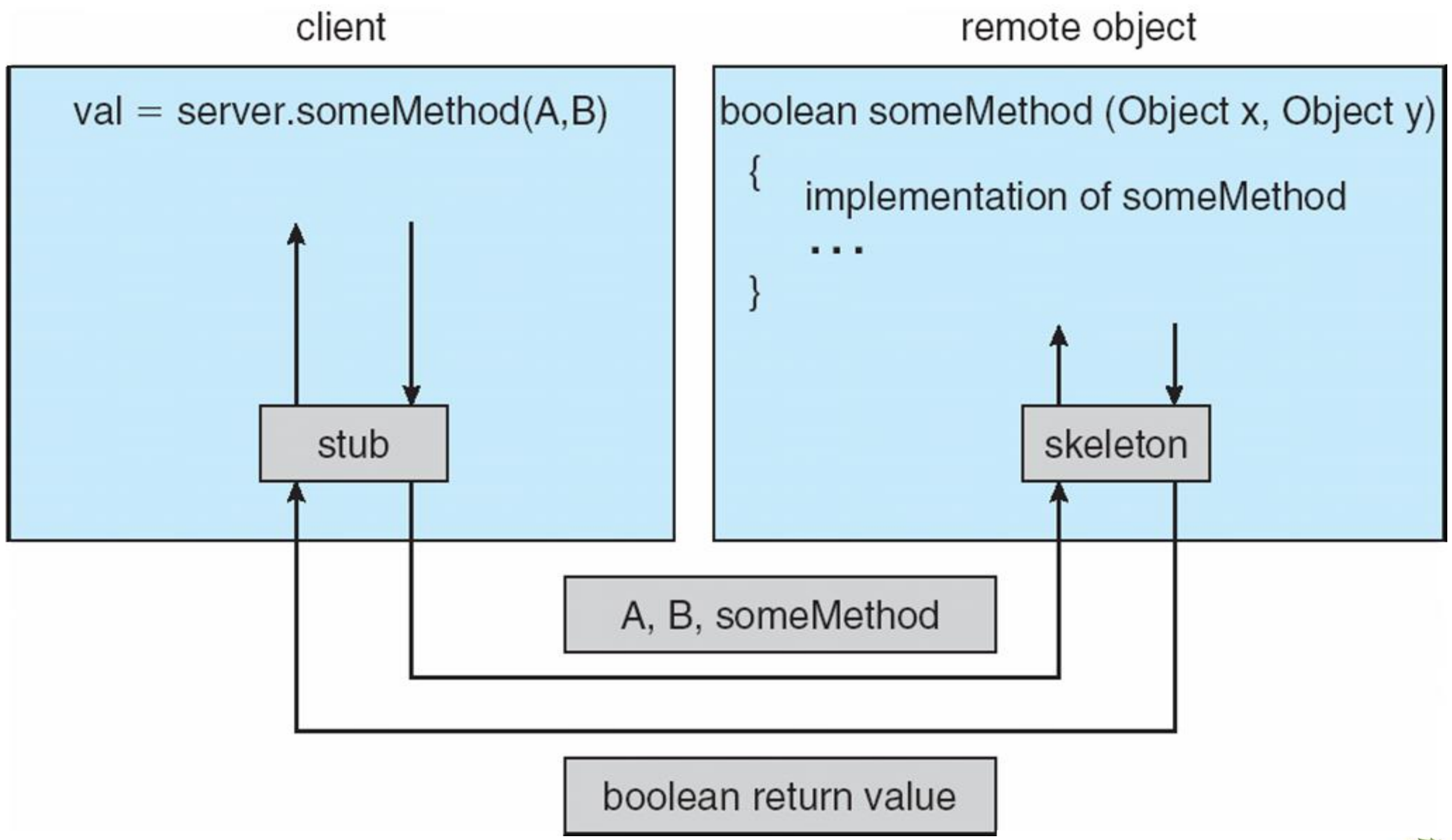
1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

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


client
remote object
val = server.someMethod(A,B)
boolean someMethod (Object x, Object y)
{
implementation of someMethod
. . .
}
stub
skeleton
A, B, someMethod
boolean return value

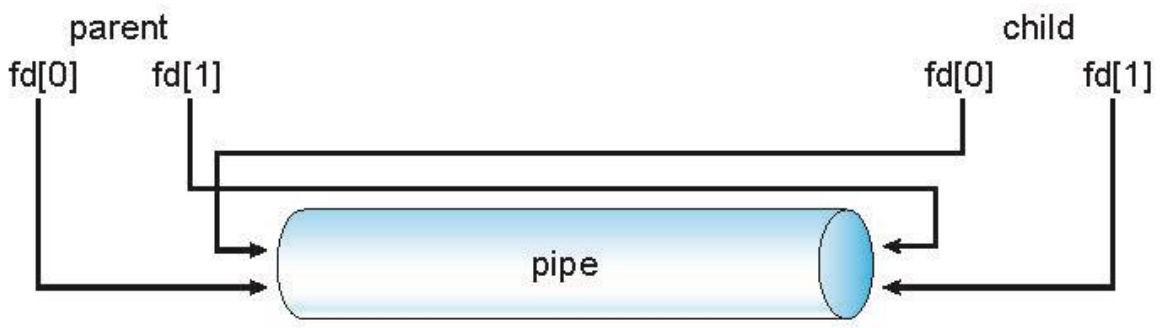
序列化与反序列化

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

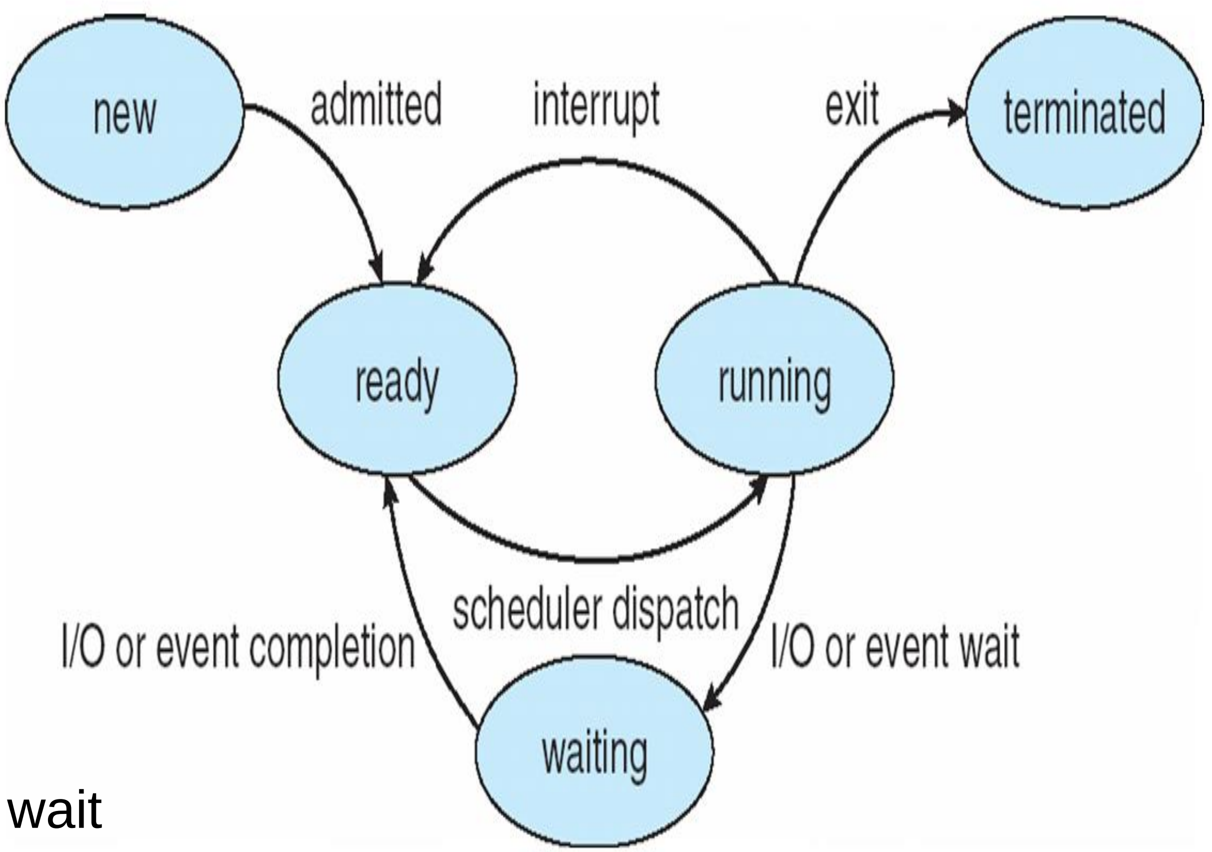


- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook

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

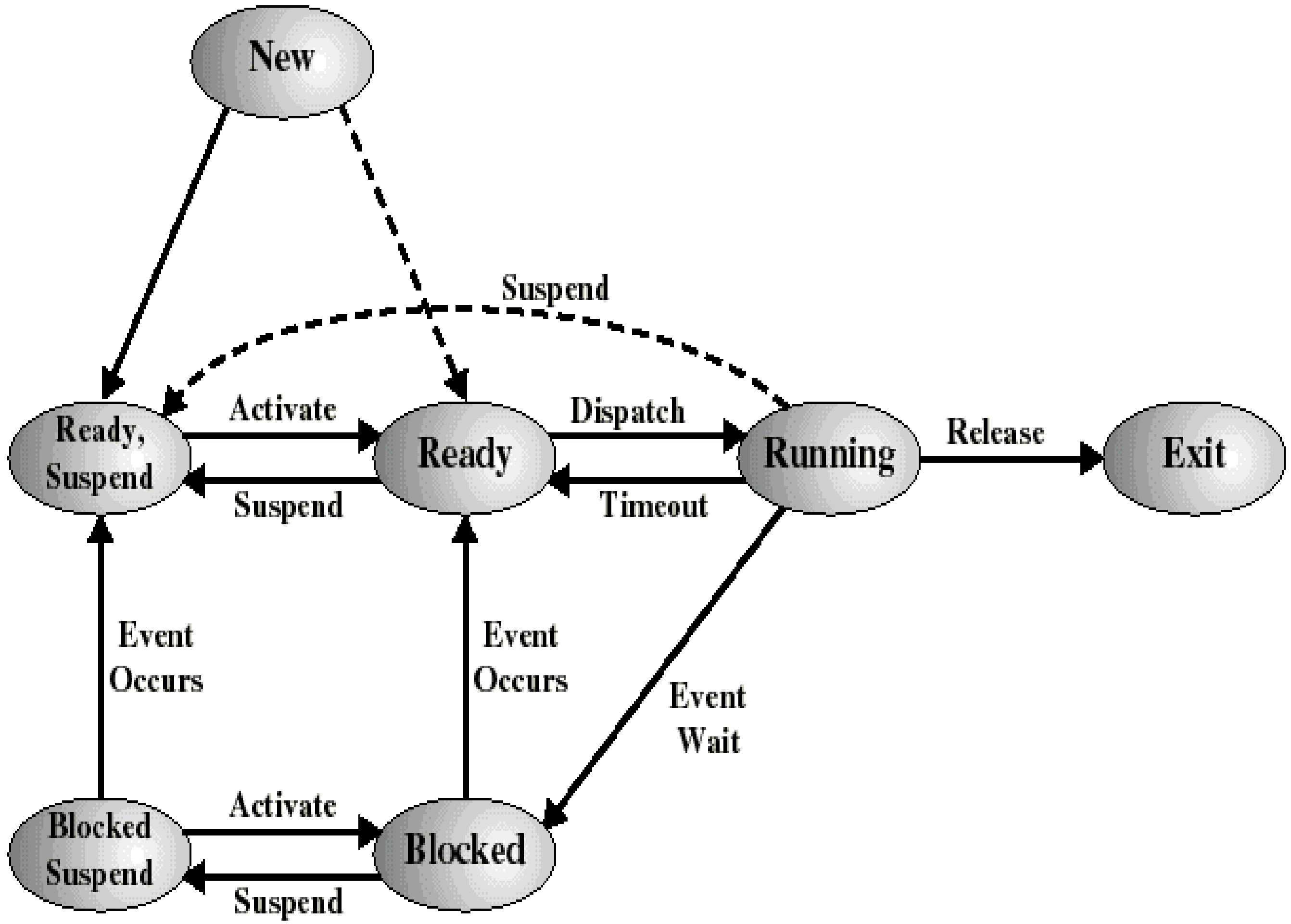
Process Control Issues



Process Control?

- Create()---- Creation
- Exit()----Termination
- Wait()----I/O or event wait
- Wakeup()----- I/O or event completion
- Scheduler()
- Interrupt()/Yield()

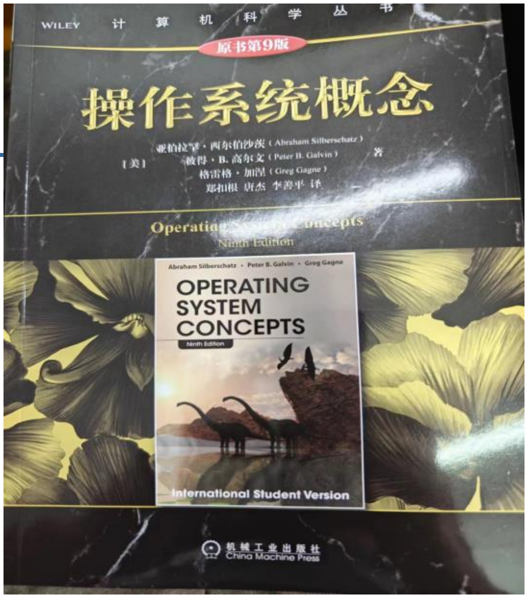
Process Control Issues



Process Control?

- ……
- Suspend()
- Activate()

Exercises


WILEY 计算机科学丛书
原书第9版
操作系统概念
[美] 亚伯拉罕·西尔伯沙茨 (Abraham Silberschatz) 彼得·B. 高尔文 (Peter B. Galvin) 格雷格·加涅 (Greg Gagne) 著
郑扣根 唐杰 李善平 译
Operating System Concepts Ninth Edition
OPERATING SYSTEM CONCEPTS
International Student Version
机械工业出版社 China Machine Press

- 3.1
- 3.2
- 给出Scheduler()和Interrupt()原语的处理流程；
- 编程题：通过系统调用fork()创建子进程，通过exec()系列中的某个执行另一个独立的程序，场景自设。
- 有一台应用服务器以Socket编程方式向客户端提供英汉互译词典的服务。请完成服务器端程序编码和一个客户端应用，并分别部署到两台计算机上进行测试。**说明：**不考虑服务器端词典的容量，用数据库或使用Map在内存中保存少量的英汉词汇对应表。不要局限于此，尽量通过google、soso、百度发现更多、更好的方式，并对其优劣进行对比。

End of Chapter 3