第三章　进程

1. 进程的概念；
2. 进程调度；
3. 进程操作；
4. 进程间的通信(C-S通信、Pipe通信)

Note : These lecture materials are based on the lecture notes prepared by the authors of the book titled Operating System Concepts.

第一节、进程的概念 (Process Concept)

1.1 进程概念

1.2 进程控制块

1.3 进程的状态

定义：执行中的程序

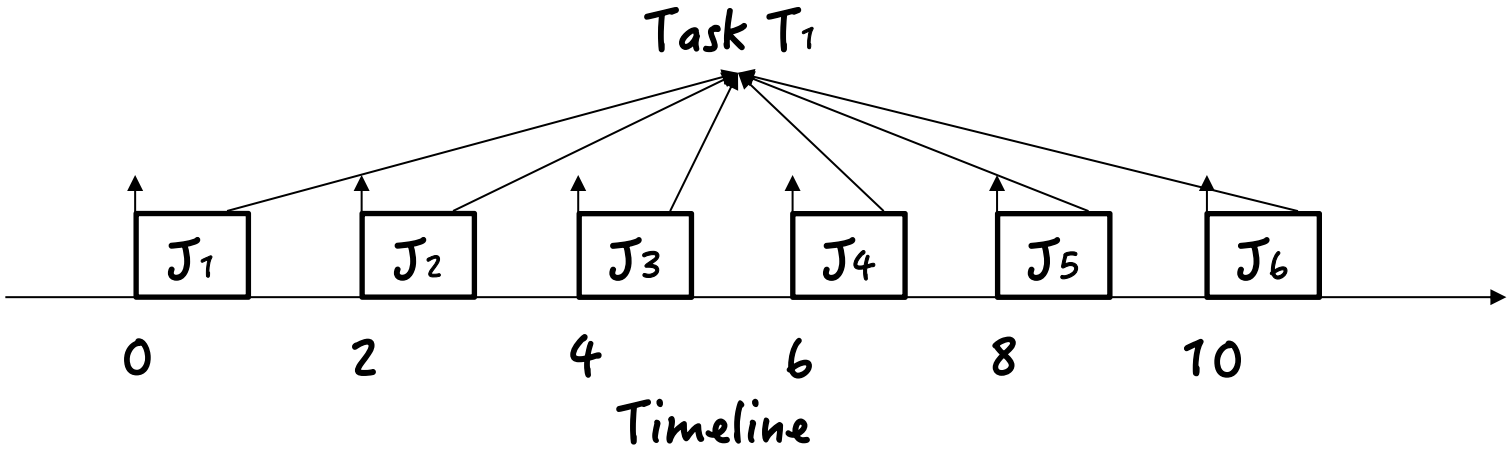
问： 执行的含义?

操作系统会执行程序、进程、任务或作业

- Terminology

任务（task） vs. 作业（job）

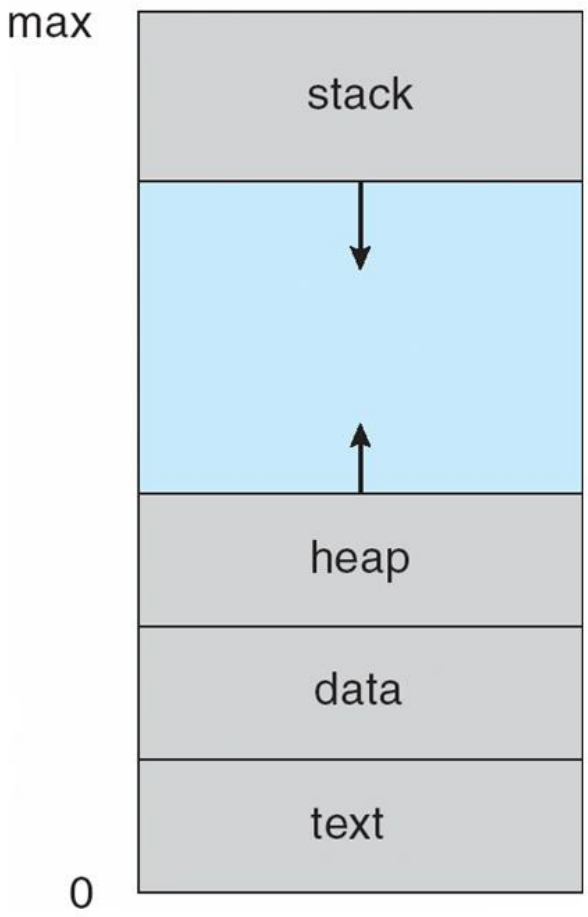
作业是任务的一个实例



程序 vs. 进程

- 程序是被动的实体，如存储在磁盘上的包含一系列指令的文件内容，进程是活动的实体。
- 当一个程序被载入内存时，这个程序就会变成进程。

进程在内存中的表现形式



函数参数
返回值
局部变量

← 运行中动态分配的内存

← 全局变量

← 程序代码

一个进程一般由以下内容，即进程在内存中的结构形式。

1. 代码段（code section）

   又称文本段（text section），通过程序计数器和处理器寄存器的内容来表示当前活动（current activity）

2. 栈（stack）:

   包含临时数据，如函数参数，返回地址，局部变量

3. 数据段（data section）

   包含全局变量

4. 堆（heap）

   在进程运行期间动态分配的内存

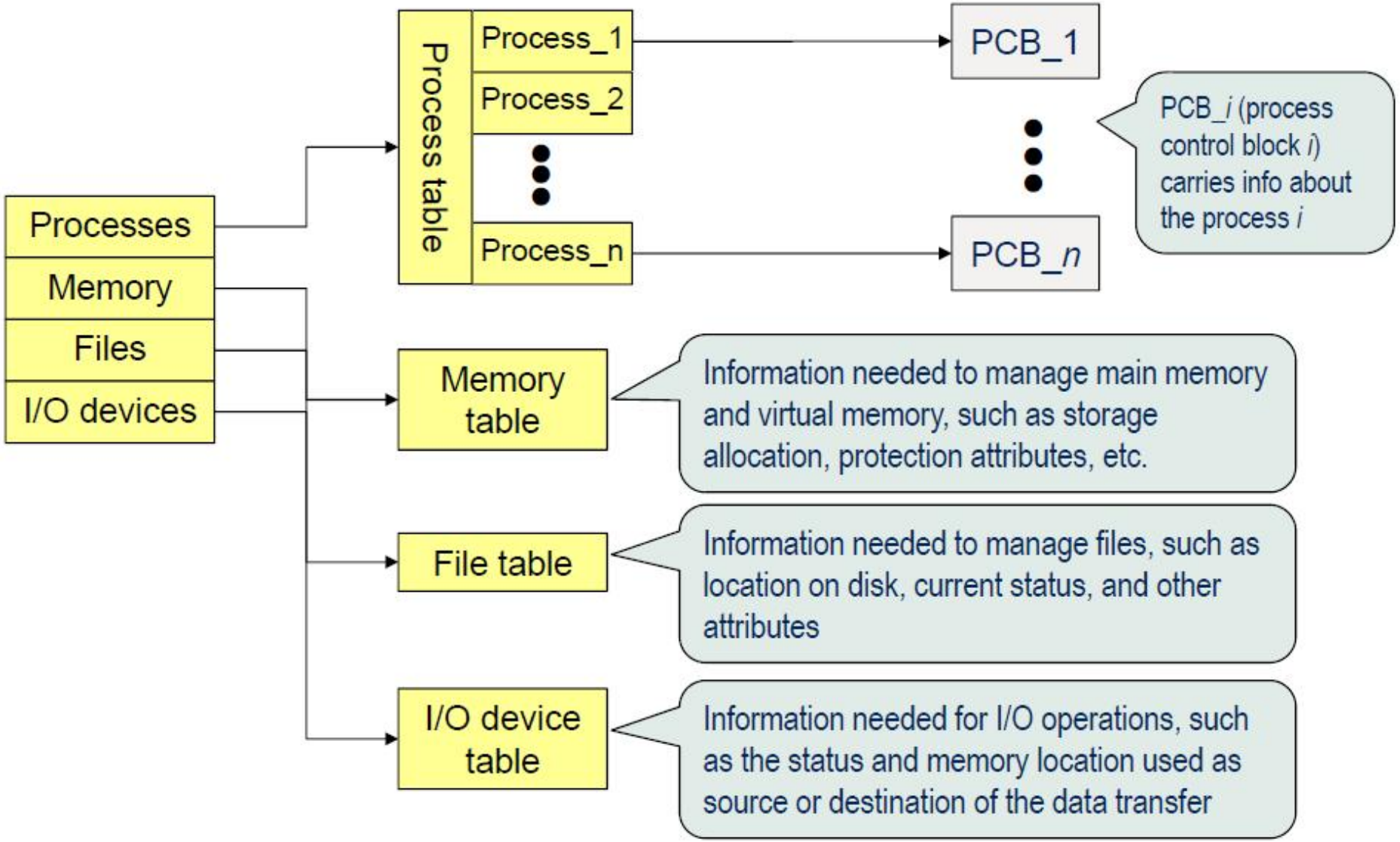
操作系统管理物理和逻辑资源

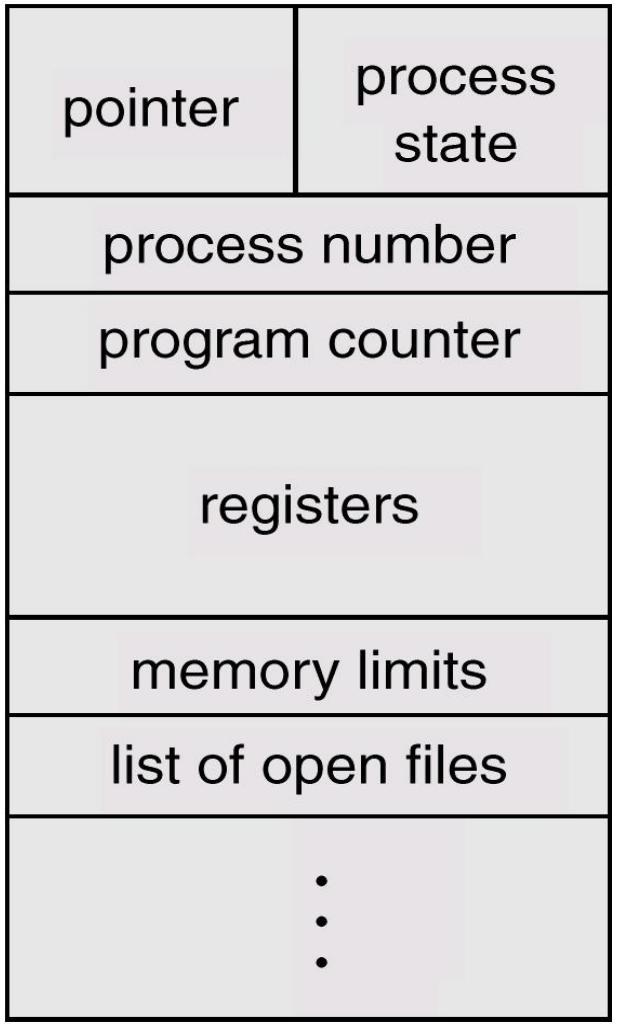
1. 物理资源： 处理器，内存，I/O设备等
2. 软件资源→逻辑资： 进程，虚拟内存，数据结构等

操作系统是通过“控制表”对系统中的每个资源进行管理

例如，进程表、内存表、I/O 设备表，文件表等

系统中的控制表


Processes
Memory
Files
I/O devices
Process table
Process_1
Process_2
Process_n
PCB_1
PCB_n
PCB_i (process control block i) carries info about the process i
Memory table
Information needed to manage main memory and virtual memory, such as storage allocation, protection attributes, etc.
File table
Information needed to manage files, such as location on disk, current status, and other attributes
I/O device table
Information needed for I/O operations, such as the status and memory location used as source or destination of the data transfer



进程控制块

每个进程在操作系统内用进程控制块（Process Control Block：PCB）来表示，

又称任务控制块（Task Control Block：TCB）即进程在操作系统中体现的数据结构。

它包含许多与一个特定进程相关的信息。

1. 进程ID：PID- 大部分操作系统是通过唯一的进程标识符来识别系统的进程
2. 进程状态：新的、就绪、运行、等待、停止等
3. 进程计数器：表示进程要执行的下一个指令的地址
4. CPU 寄存器：包括累加器、索引寄存器、堆栈指针、通用寄存器和其他条件码信息寄存器
5. CPU 调度信息：进程的优先级、调度队列的指针和其他调度参数
6. 内存管理信息：包括基地址和界限地址、页表或段表等
7. 记账信息：CPU 使用时间，时间界限等
8. I/O 状态信息：I/O 设备进程分配状态，打开文件表等

进程的数据结构是

task_struct

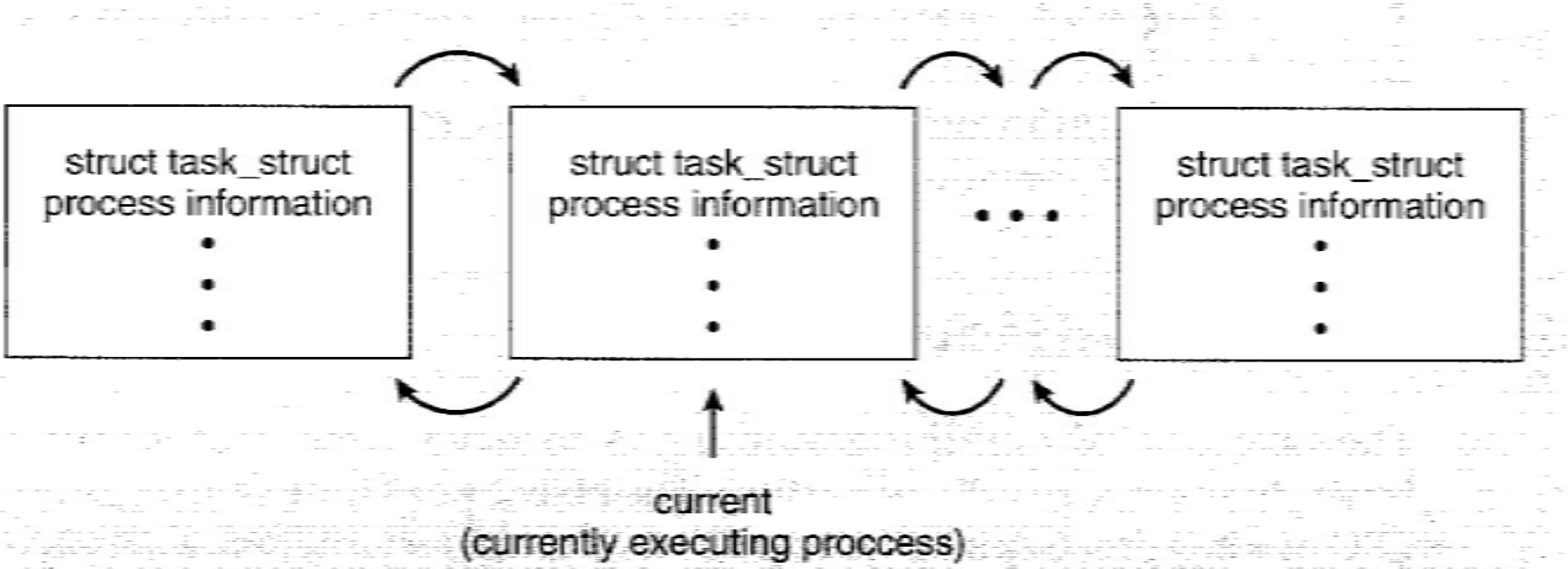
```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm struct *mm; /* address space of this process */
```

https://lxr.missinglinkelectronics.com

Active Process in Linux

$/usr/src/linux-source-3.13.0/linux-source-3.13.0/include/linux/sched.h

Linux中的进程表示



Active Process in Linux

https://lxr.missinglinkelectronics.com

task_struct 的数据结构实现代码在“sched.h”文件中

文件路径（linux kernel 3.13 版本为例）：

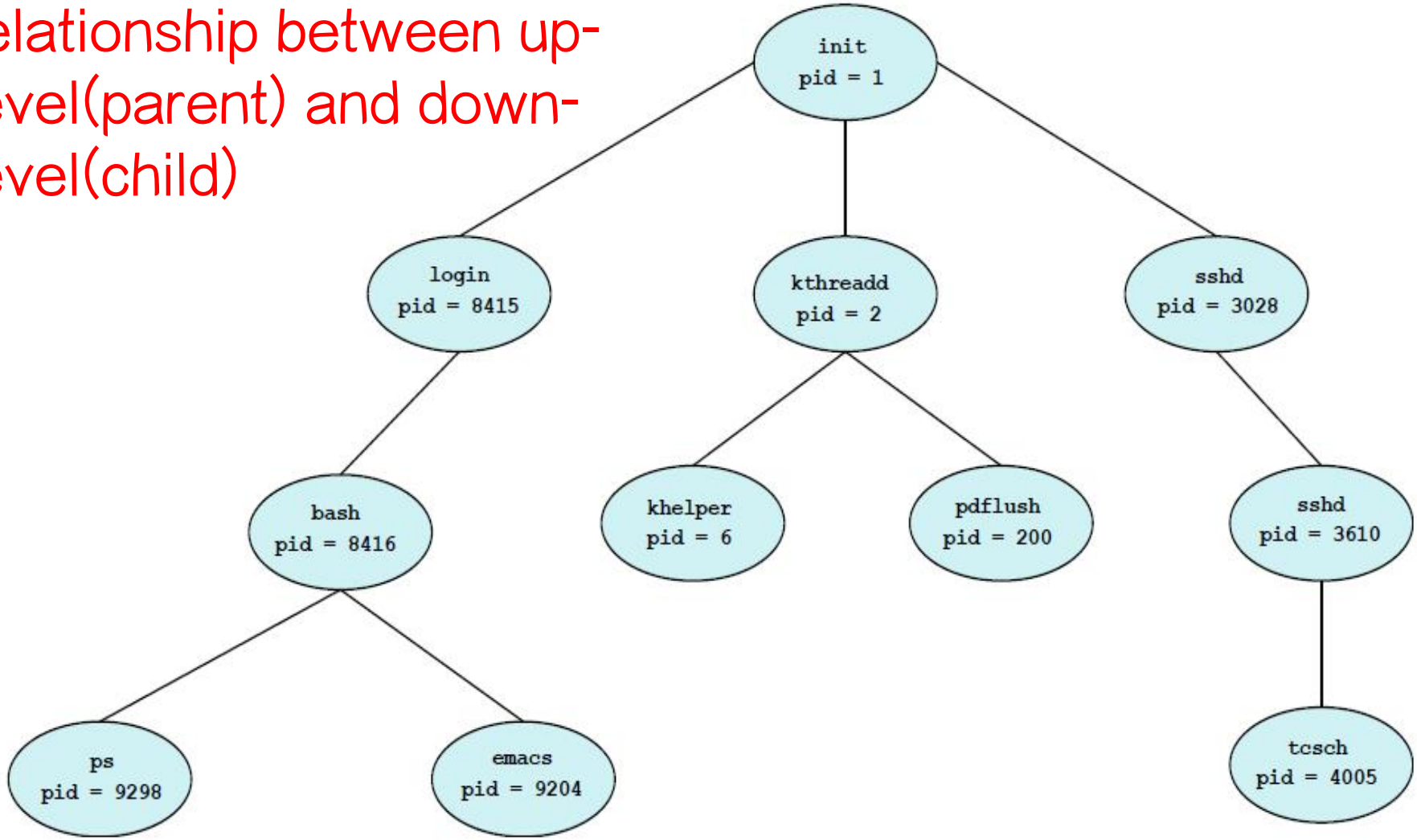
$/usr/src/linux-source-3.13.0/linux-source-3.13.0/include/linux/sched.h

Linux 常用命令

1. $pmap –d PID： 进程内存使用量的查询
2. $top：系统中的进程内存使用量查询
3. $ps：查看进程， pstree -p
4. $grep： 是一种问题搜索工具，它能使用正规表达式搜索文体，并把匹配的行打印出来

进程之间是树形结构

Please remember the relationship between up-level(parent) and down-level(child)

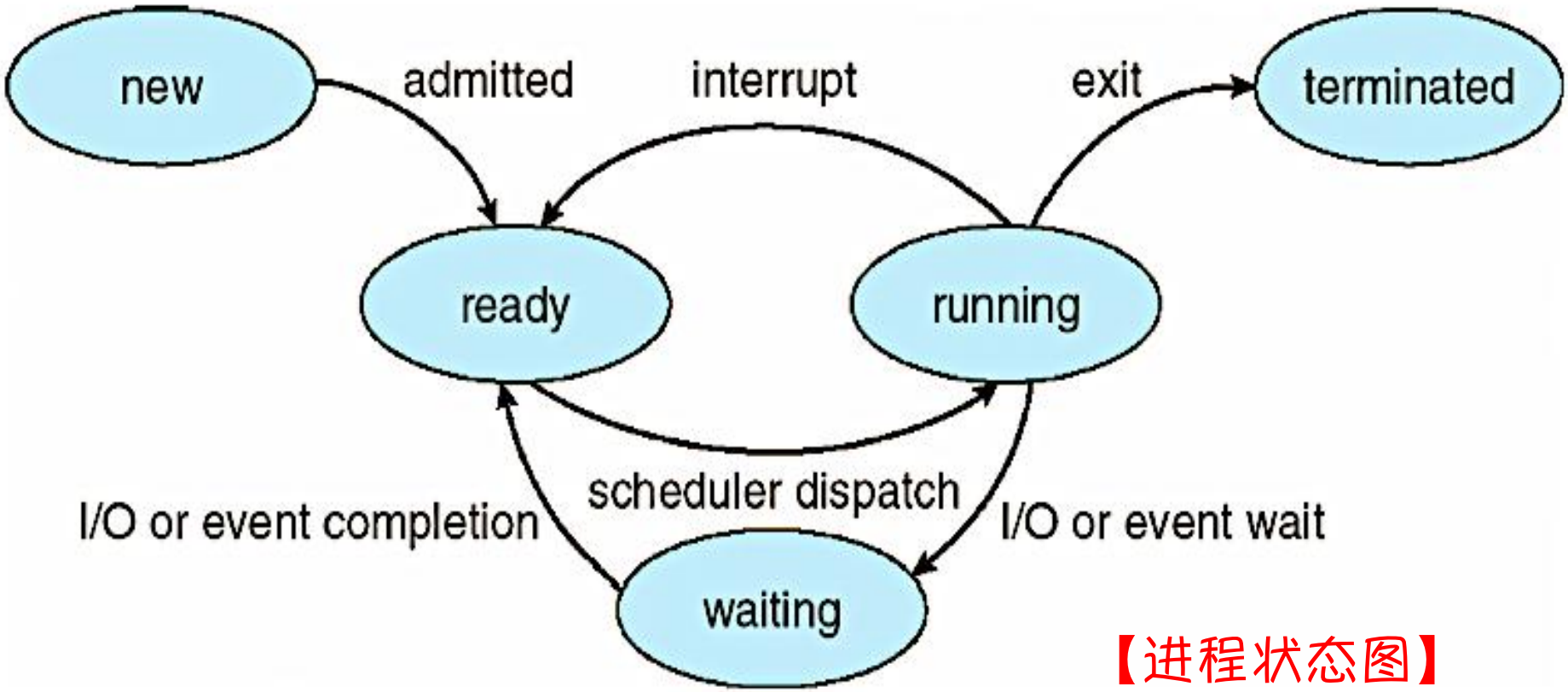


Tips: Linux 根目录下 /proc 文件夹的使用

N 为进程ID	
/proc/N/cmdline	进程启动命令
/proc/N/cwd	链接到进程当前工作目录
/proc/N/environ	进程环境变量列表
/proc/N/exe	链接到进程的执行命令文件
/proc/N/fd	包含进程相关的所有的文件描述符
/proc/N/maps	与进程相关的内存映射信息
/proc/N/mem	指代进程持有的内存,不可读
/proc/N/root	链接到进程的根目录
/proc/N/stat	进程的状态
/proc/N/statm	进程使用的内存的状态
/proc/N/status	进程状态信息,比stat/statm更具可读性
/proc/self	链接到当前正在运行的进程

进程在运行时，自身的状态会发生变化

1. 新的(new): 进程正在被创建
2. 就绪(ready): 进程等待分配处理器
3. 运行(running): 指令正在被执行
4. 等待(阻塞)waiting(blocking): 进程等待某个事件的发生
5. 终止(terminated): 进程完成执行



【进程状态图】

dispatch: 分派

CPU在进程间的切换

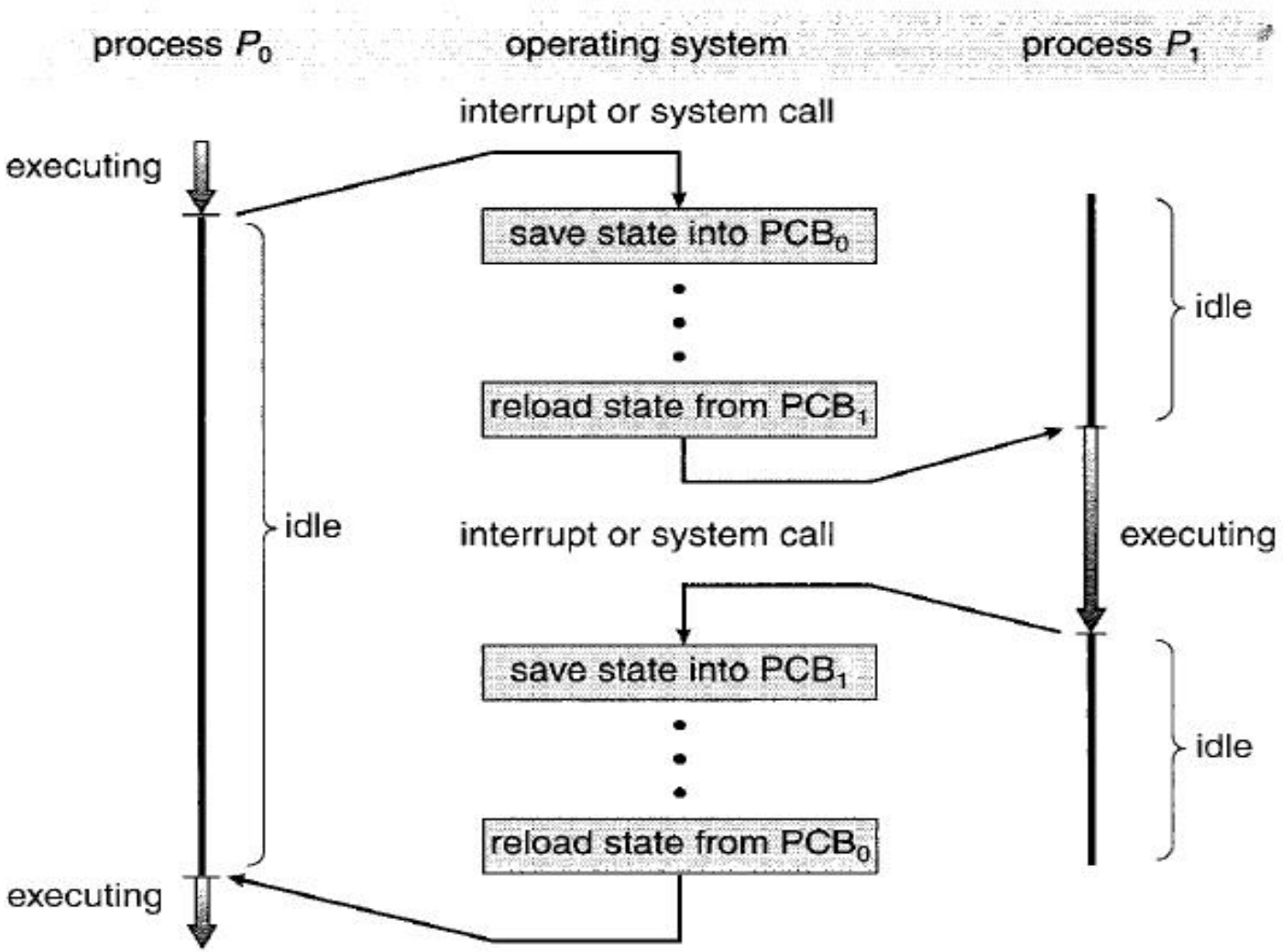
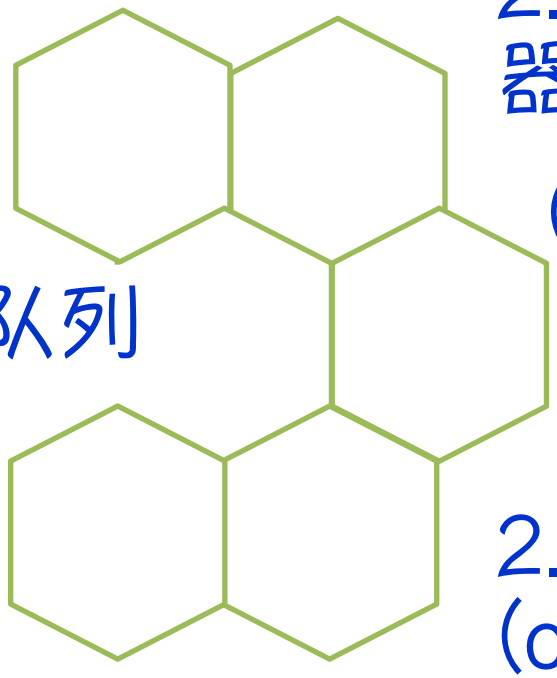


Figure 3.4 Diagram showing CPU switch from process to process.

第二节、进程调度
(Process Scheduling)

2.2 调度程序/调度器（scheduler）



2.1 进程调度队列

2.3 上下文切换（context switch）

- 多道程序(multiprogramming)的目的是无论何时都有进程在运行，从而使CPU的利用率达到最大。为此，CPU需要在多个可用进程之间进行快速切换，调度程序从多个可用进程中选择一个进程运行

- 操作系统持有**就绪队列**和**一组设备队列**，进程可以在多个调度队列之间移动

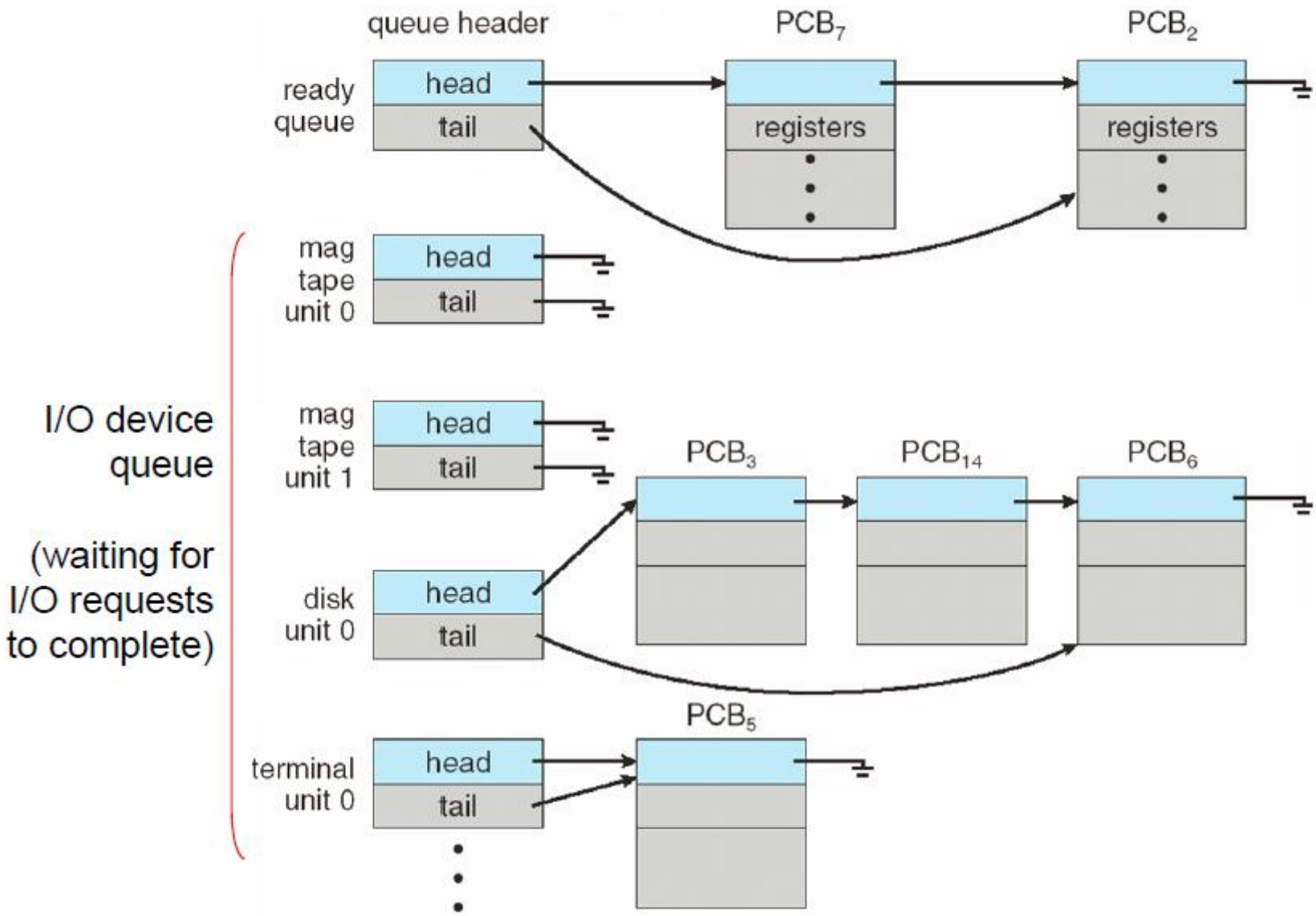
  1. 就绪队列：

  驻留在内存中的就绪并等待运行的进程(如进程创建时，被放到该队列)，即等待分配CPU的进程，一般用链表来实现

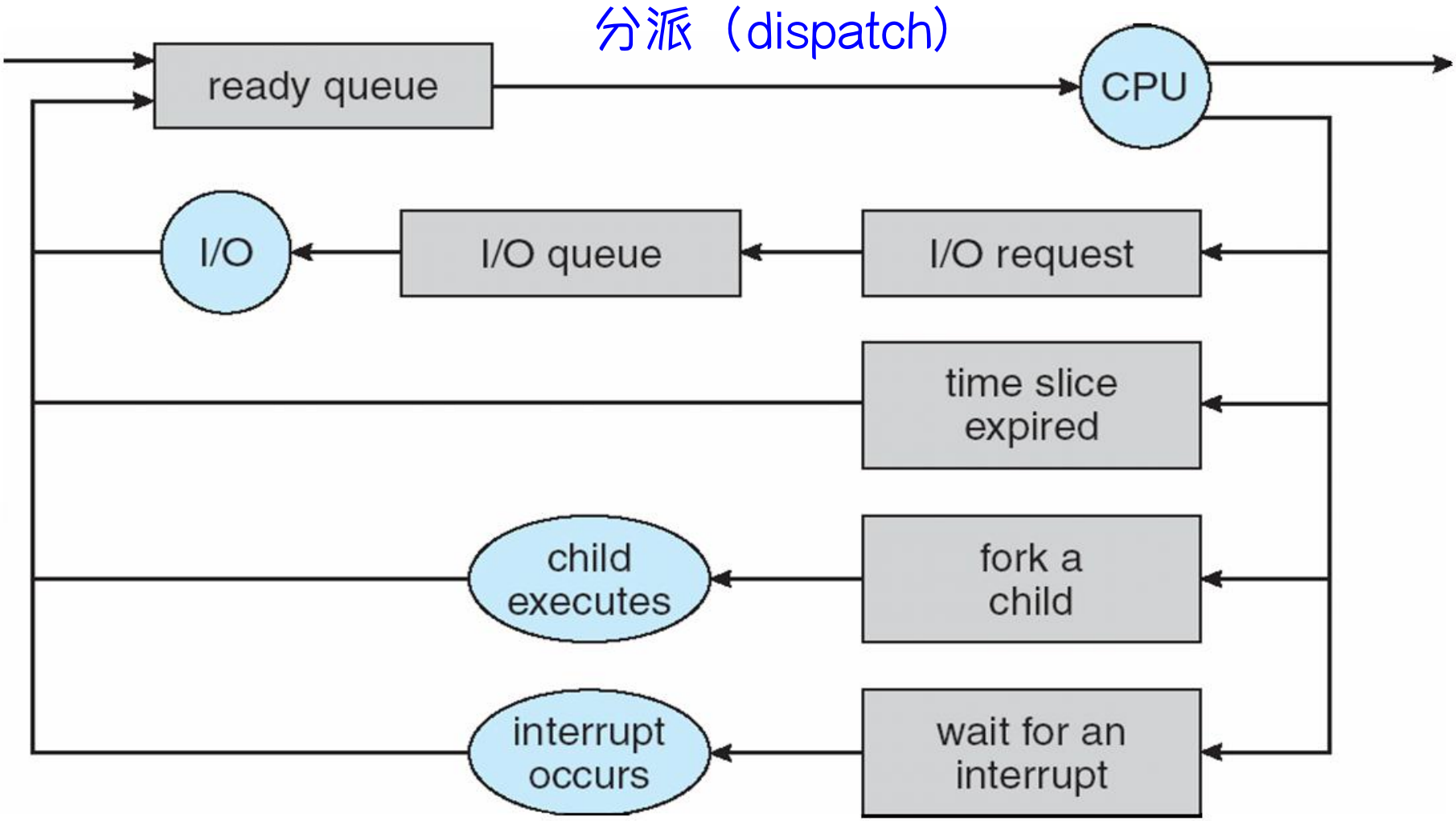
  2. 设备队列

  等待特定I/O设备的进程队列

就绪队列和I/O设备队列



表示进程调度队列图


分派（dispatch）
ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

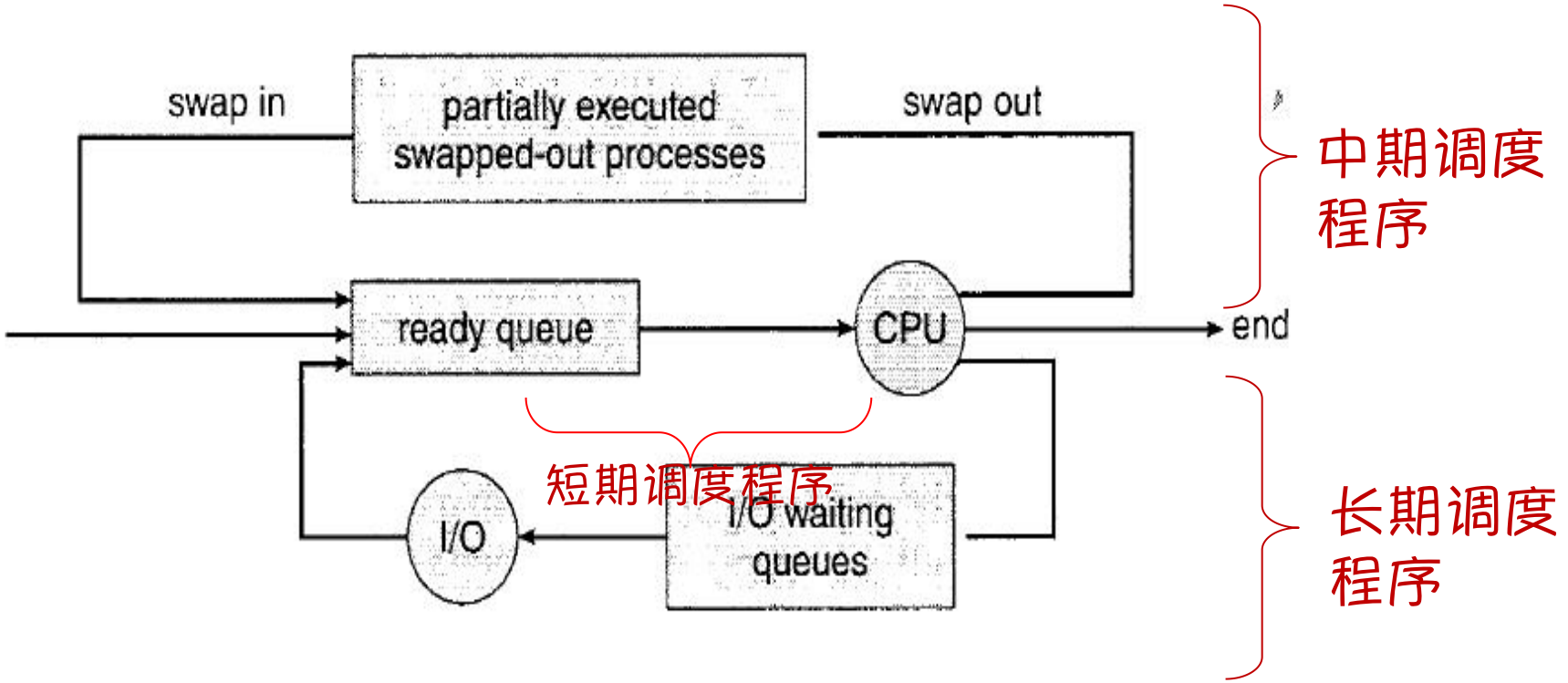
当进程分配到CPU并运行时（运行状态），也就是进程运行过程中，可能发生下面事件中的一种并进入到就绪状态：

1. 进程可能发出一个 I/O 请求，并被放到 I/O 队列
2. 进程可能创建一个新的子进程，并等待该子进程结束
3. 进程可能会由于等待中断而强制释放 CPU，并被放回到就绪队列
4. 用完时间片（time slice / time quantum），并被放到就绪队列

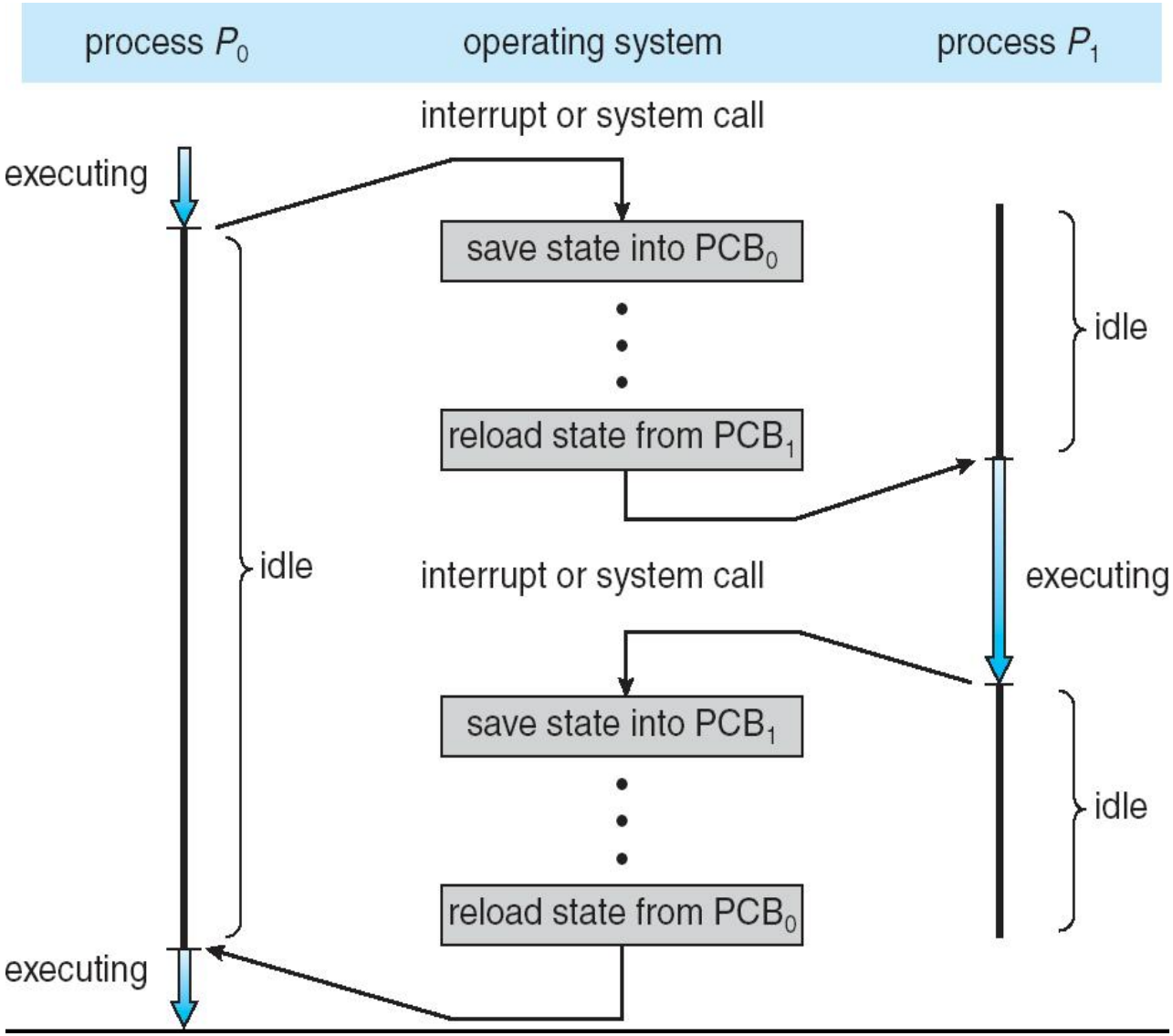
主要区别是执行的频率

1. **长期调度程序（作业调度程序）：**
   从存储设备的缓冲池中选择进程，并装入就绪队列中等待执行（I/O调度）
2. **短期调度程序（CPU调度程序）**
   从准备执行队列中选择一个进程，并为之分配CPU（CPU调度）
3. **中期调度程序：分时系统**
   中期调度程序的核心思想是能将进程从内存中移出（从CPU竞争中移出），从而降低多道程序设计的程度（Swap In and Swap Out）

分时系统，增加了中期调度程序



当CPU从当前进程切换到另一个进程时，系统必须保存当前进程的相关信息，以备被切换的进程恢复运行



- **上下文切换的定义**

  将CPU切换到另一个进程需要保存当前进程的状态，并恢复另一个进程状态，这一任务称为上下文切换

- **上下文的切换依赖于**
  - 硬件支持，如内存速度、寄存器数量、指令运行时间、载入/保存时间等

- 上下文切换时间是额外的开销，因为上下文切换时系统不能做什么有用的工作

- 为了提高上下文切换速度，
  - 有的处理器（如 Sun UltraSPARC）提供多组寄存器集合，上下文切换只需简单地改变当前寄存器组的指针

第三节、进程的操作（Process Operation）

3.1 进程创建

3.2 进程终止

Operating system must provide mechanisms for process management

系统中的进程结构是树形结构，进程和进程之间存在着父子关系，即父进程与子进程

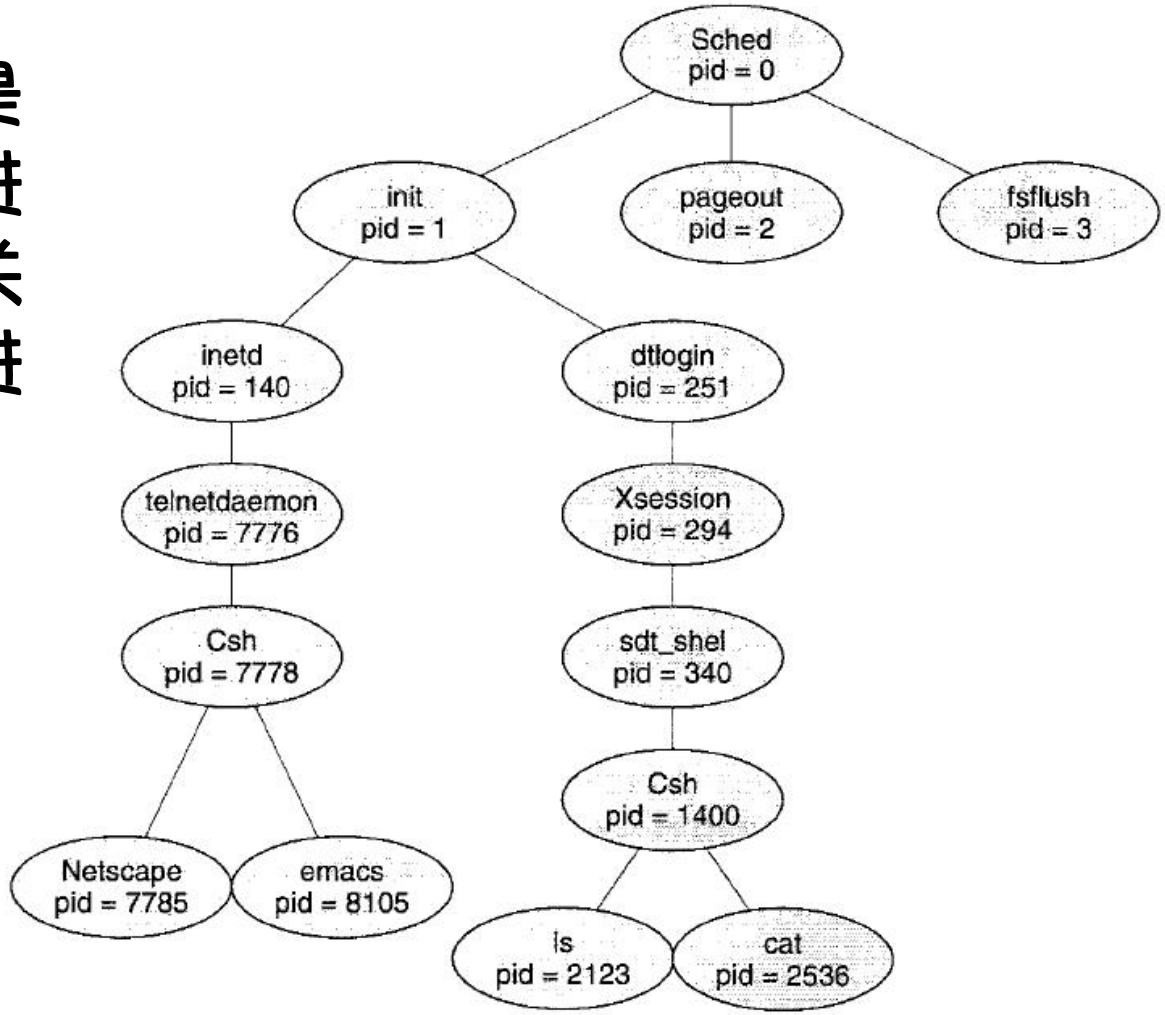


Figure 3.9 A tree of processes on a typical Solaris system.

操作系统是根据一个唯一的进程标识符（PID）来识别系统中的进程。

一个进程可以创建另一个新进程，那么，创建的进程称为父进程，被创建的新进程称为子进程（child process）

进程创建选项

1. 资源共享选项

① 父进程和子进程共享所有资源
② 子进程共享父进程的部分资源
③ 父进程和子进程不共享资源

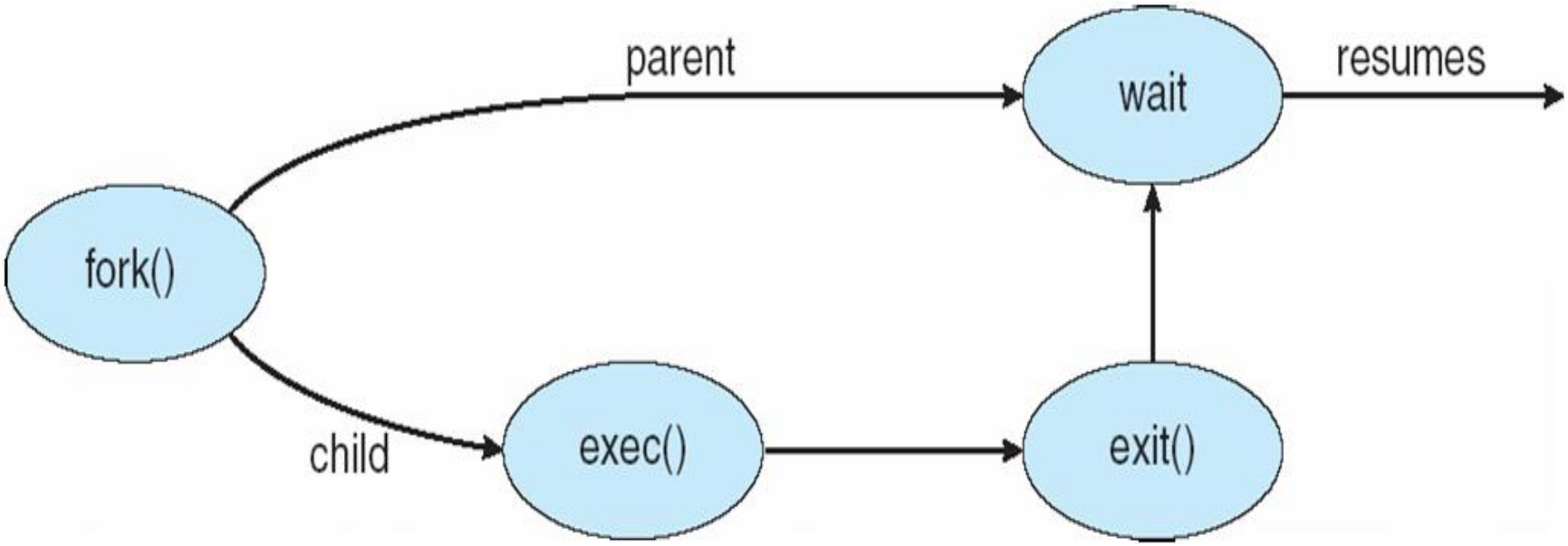
2. 执行选项

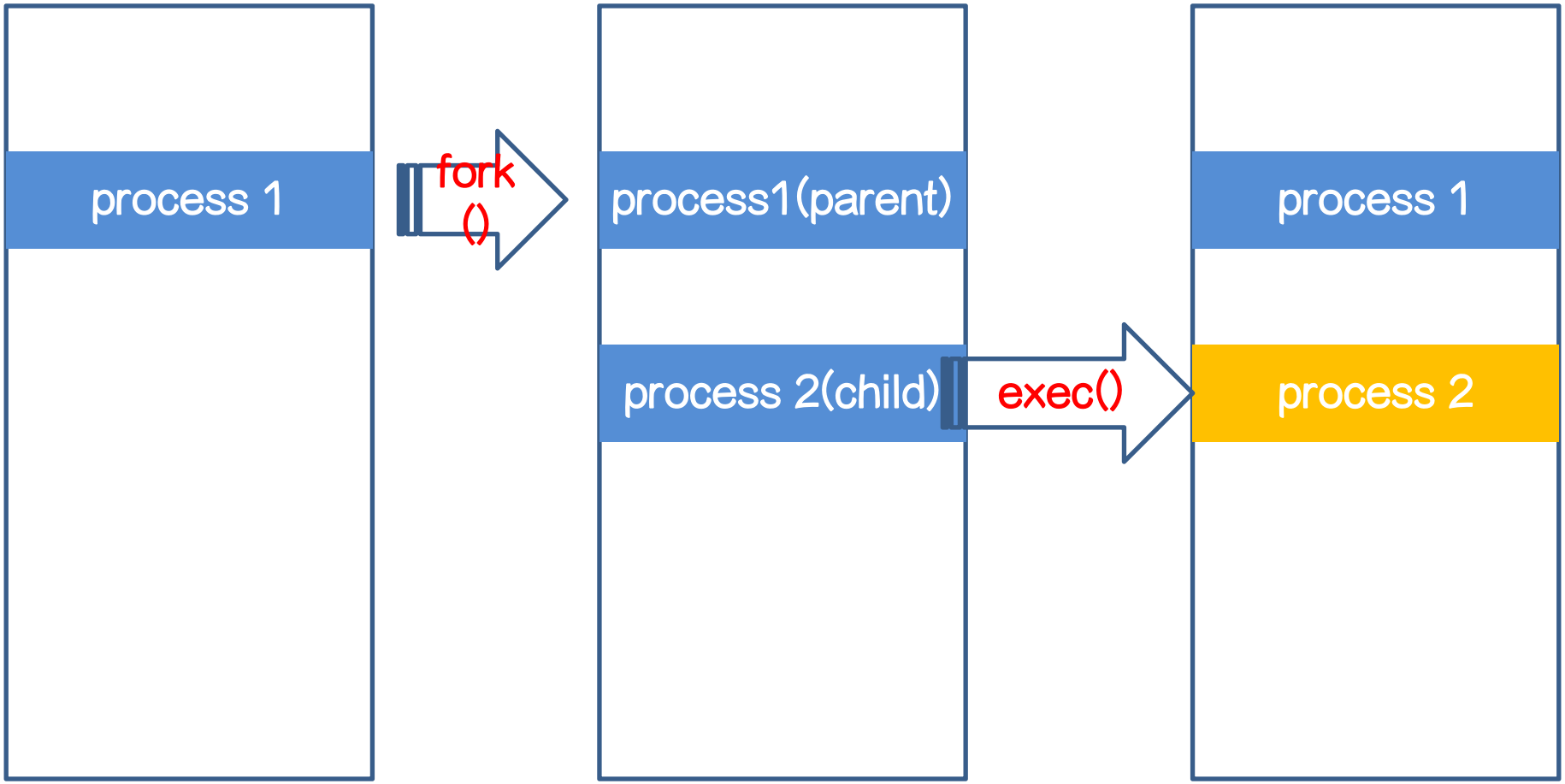
① 父进程和子进程同时执行
② 父进程等待子进程结束

3. 地址空间选项

① 子进程完全复制父进程内容
② 子进程覆盖父进程内存空间

以UNIX 系统为例，提供fork( )创建进程系统调用，用fork( )系统调用创建进程之后，一般是用exec( ) 系统调用覆盖进程内存空间




process 1
fork
()
process1(parent)
process 2(child)
exec()
process 1
process 2

4 Step

1. 在系统内部创建进程控制块
2. 分配内存
3. 载入可执行文件（通过调用exec()系统调用）
4. 初始化程序

For Example:

- fork() 系统调用创建新的进程
  1. 复制父进程PCB
  2. 分配内存空间
- 调用exec() 系统调用，以便重载运行程序
  3. 从磁盘载入二进制程序
  4. 初始化

```
1.  int main()  {
2.                  pid_t  pid;
3.                  /* fork another process */
4.                  pid = fork();
5.                  if (pid < 0) { /* error occurred */
6.                          fprintf(stderr, "Fork Failed");
7.                          exit(-1);
8.                  }
9.                  else if (pid == 0) { /* child process */
10.                         printf("I am a Child\n");
11.                 }
12.                 else { /* parent process */
13.                 /* parent will wait for the child to complete */
14.                         wait (NULL);
15.                         printf ("I am a Parent\n");
16.                         exit(0);
17.                 }
18. }
```

使用 fork( ) 系统调用的C语言程序

执行结果为

```
hbpark@hbpark-VirtualBox:~/src/fork$ ./fork
I am a Child
I am a Parent
```

fork( ) 函数的返回值

1. 子进程：返回值为 0
2. 父进程：返回值为子进程进程标识符 PID

实验一：

运行 test.sh 命令， test.sh 内容如下：

```
1. #!/bin/bash
2. for ((i=1;$i<=100;i++));
3. do
4. echo $i
5. ./fork
6. done
```

实验二：

把 fork.c 程序中的wait(NULL)，即等待子进程结束的代码行去掉，重新运行test.sh

- 比较实验一的结果和实验二的结果
- 运行结果会不同，为什么?

Homework #1

提交纸质版《fork( )系统调用研究分析报告书》，内容包括

1. 提交实验一和实验二的运行结果截图，截图必须包括 full terminal window 界面，如图 in the next slide.
2. 研究分析不同结果的原因
3. 研究分析必须有理论根据

Terminal

00:03

hbpark@hbpark-VirtualBox: ~/src/fork

```
hbpark@hbpark-VirtualBox:~/src/fork$
hbpark@hbpark-VirtualBox:~/src/fork$ ls -al
total 40
drwxrwxr-x  2 hbpark hbpark  4096  3月 13 23:50 .
drwxrwxr-x 11 hbpark hbpark  4096  3月 13 22:18 ..
-rwxrwxr-x  1 hbpark hbpark  8848  3月 13 23:45 fork
-rw-rw-r--  1 hbpark hbpark   578  3月 13 23:46 fork.c
-rwxrwxr-x  1 hbpark hbpark    62  3月 13 23:50 test.sh
-rw-r--r--  1 hbpark hbpark 12288  3月 13 23:49 .test.sh.swp
hbpark@hbpark-VirtualBox:~/src/fork$ vi .test.sh.swp
hbpark@hbpark-VirtualBox:~/src/fork$ rm .test.sh.swp
hbpark@hbpark-VirtualBox:~/src/fork$ ls
fork  fork.c  test.sh
hbpark@hbpark-VirtualBox:~/src/fork$
hbpark@hbpark-VirtualBox:~/src/fork$
hbpark@hbpark-VirtualBox:~/src/fork$ vi test.sh
hbpark@hbpark-VirtualBox:~/src/fork$ ./fork
I am a Child
PID = 10603
I am a Parent
hbpark@hbpark-VirtualBox:~/src/fork$
hbpark@hbpark-VirtualBox:~/src/fork$
hbpark@hbpark-VirtualBox:~/src/fork$
hbpark@hbpark-VirtualBox:~/src/fork$
```

1. 进程完成最后语句，并使用 exit( ) 系统调用请求操作系统删除自身，并终止运行，这时，进程可以通过 wait( ) 函数返回状态值给父进程并释放资源，资源被操作系统收回

2. 父进程可以用 abort( ) 系统调用终止子进程的执行，父进程终止子进程的原因可能如下：

- 子进程使用了超过它所分配到的资源
- 分配给子进程的任务不在需要
- 父进程终止导致子进程终止，即父进程终止，操作系统不允许子进程继续运行

- 如果一个进程终止，它的所有子进程也将终止，这种终止被称为级联终止（Cascading Termination）

- 等待子进程终止的例子代码

```
1. pid_t pid; //声明一个进程标识符变量
2. int status; // 声明一个保存终止进程状态信息的变量
3. pid = wait(&status); // 等待子进程终止
```

- Tips

僵尸进程(zombie process)，

即在僵尸状态的进程，它是进程终止的特殊情况，

即一个进程结束，但是他的父进程没有等待他(调用wait/waitpid)，那么他将成为一个僵尸进程。

用 $ps 命令查看其带有 defunc 的标志，是僵尸状态(Z)，

但是在进程表中占了一个空间。

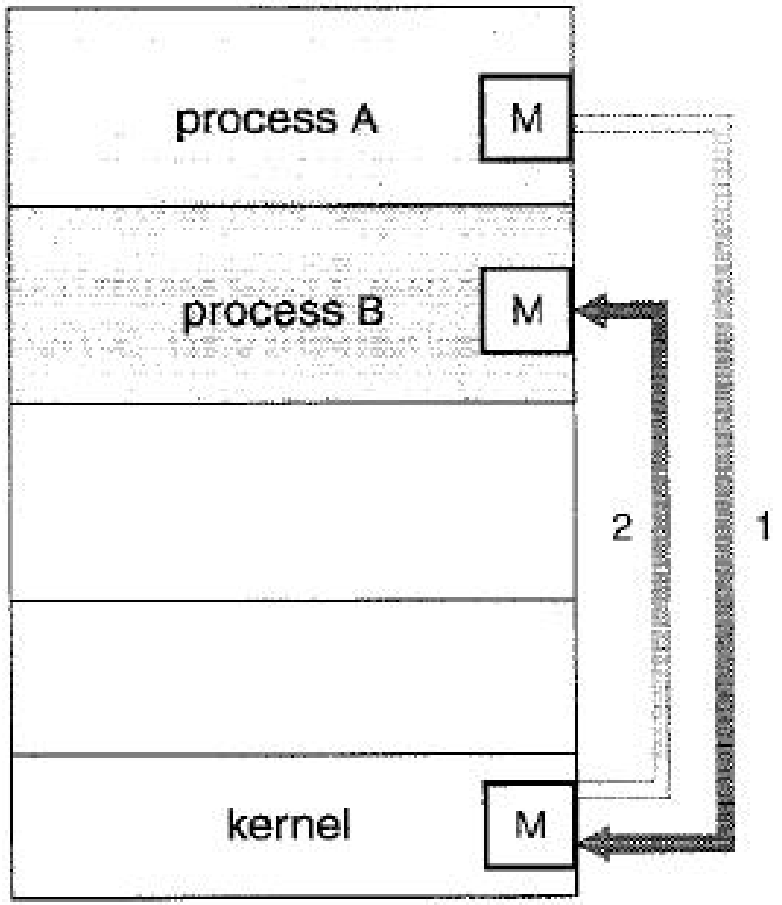
第四节、进程间的通信（IPC: Inter-Process Communication）

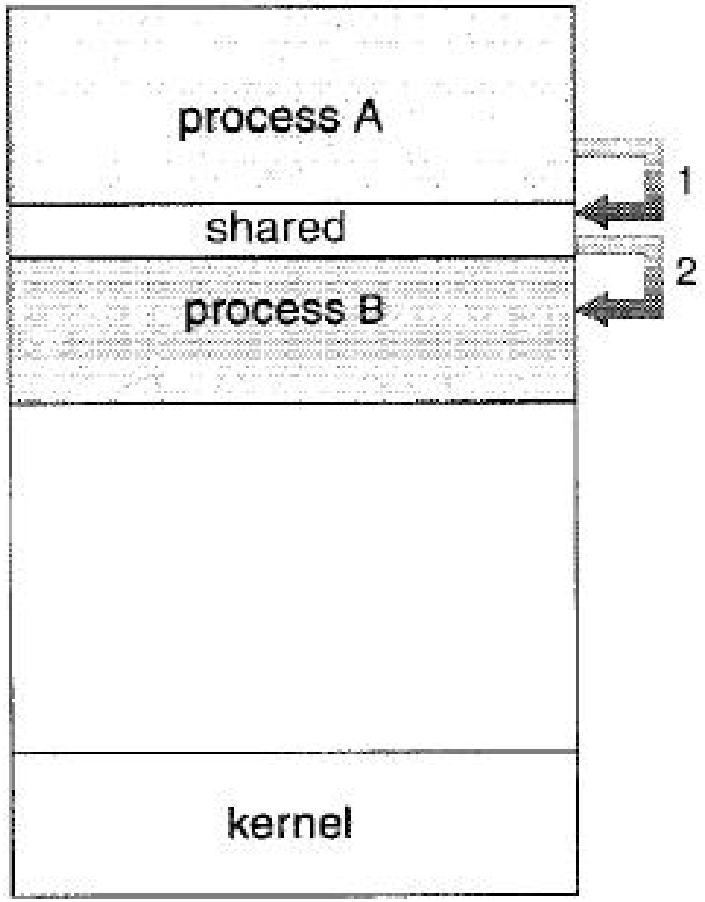
系统内并发运行的进程可以是相互独立或协作工作，进程协作的理由有信息共享、提高运算速度、模块化、方便

1. 如果一个进程不影响其他进程或不被其他进程所影响，那么该进程是独立的，即进程之间没有共享数据

2. 如果一个进程影响其他进程或被其他进程所影响，那么该进程是协作的，即进程之间有数据共享

So,协作进程需要一种进程间的通信机制



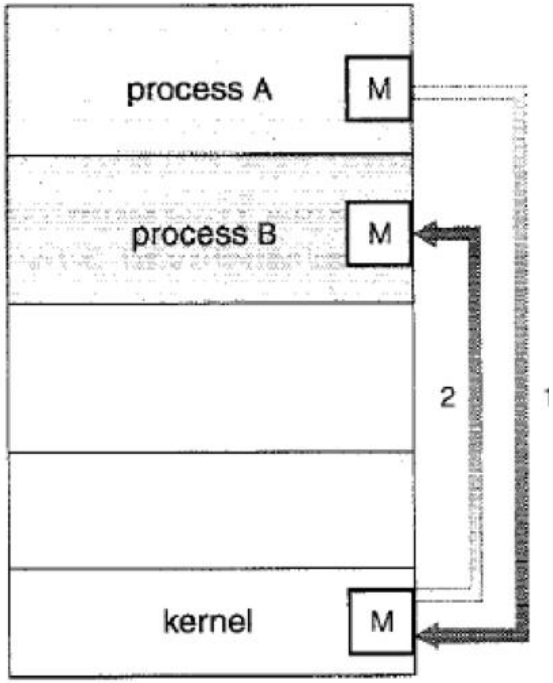


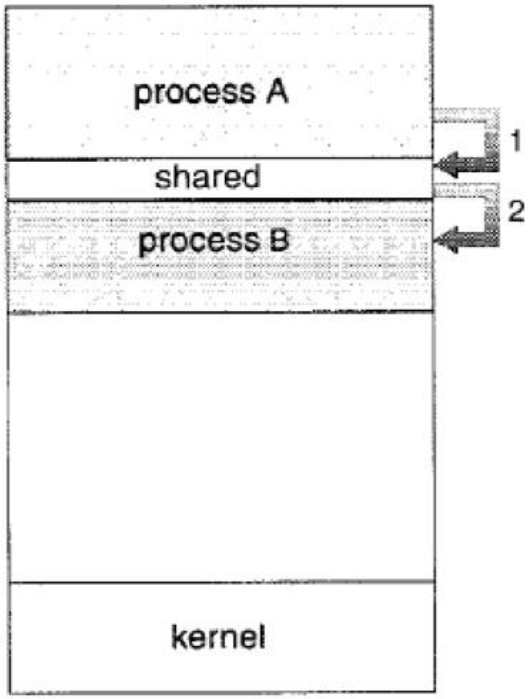
1. 消息传递
message passing

2. 共享内存
shared memory

1、消息传递

- 适合传递较少数量的数据
- 需要内核的干涉
- 易于实现





2、 共享内存

- 允许以最快的速度进行方便的通信
- 不需要内核的干涉
- 比消息传递速度快

我们认识的系统环境一般如下：

1. 每个进程都有自己受保护的内存地址空间
2. 通常操作系统试图阻止一个进程访问另一个进程的内存地址空间

>> 为了实现共享内存、便于两个或多个进程可以访问内存，共享区域应取消这个限制

>> 而且，必须保障不能有两个以上的进程同时向共享区域写入数据，即需要一个同步机制

>> 共享内存典型范例为生产者进程产生信息以供消费者进程消费的问题，即生产者和消费者问题

>> 为了允许生产者进程和消费者进程能并发运行，必须要有缓冲区被生产者和消费者所使用，此缓冲区为共享内存区域，该区域的实现可以采用以下两种方式

1. 无限缓冲（unbounded buffer）：对缓冲大小无限制，如果缓冲为空，消费者必须等待，而生产者总是可以产生新的信息

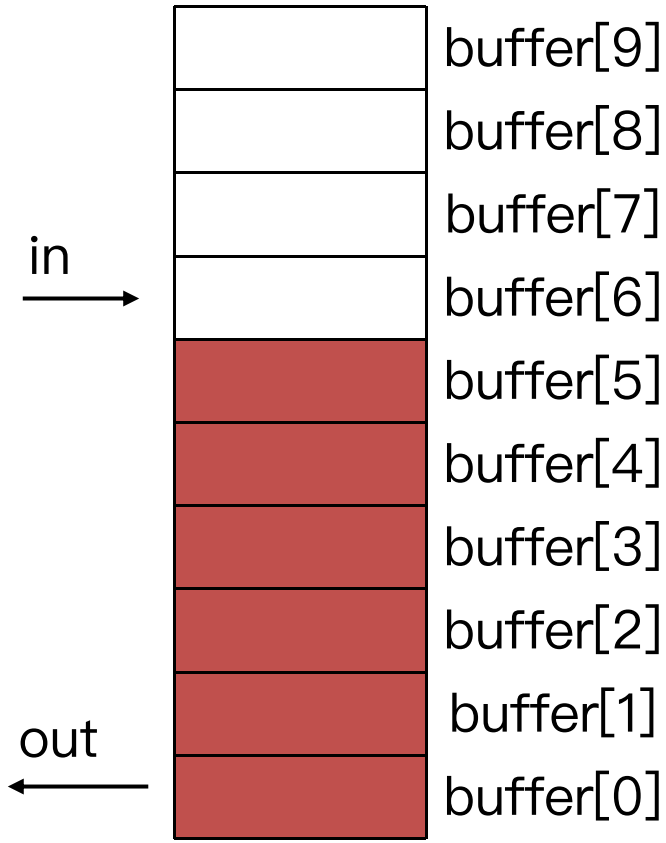
2. 有限缓冲（bounded buffer）：缓冲大小固定，如果缓冲为空，消费者必须等待，如果缓冲为满，生产者不能产生新的信息

有限缓冲

驻留在内存中的数组变量 buffer，由生产者和消费者共享

```
define BUFFER_SIZE 10
typedef struct {
…
} item;

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



- in（写）：指向缓冲中的下一个空位
- out（读）：指向缓冲中的第一个满位
- in == out：表示缓冲为空
- (in + 1) % BUFFER_SIZE == out：表示缓冲为满

有限缓冲

- 生产者进程

```
item nextProduced;

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

--》表示生产的新项

表示缓冲区为满，处于等待状态

--》表示写入

Figure 3.14 The producer process.

有限缓冲

- 消费者进程

```
item nextConsumed;

while (true) {
      while (in == out)
         ; // do nothing

      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      /* consume the item in nextConsumed */
}
```

--》 表示使用的新项

--》共享内存为空，处于等待状态

--》 表示读出

Figure 3.15 The consumer process.

>> 消息传递功能提供了两种操作：发送消息和接收消息 。而且消息可以是定长的或变长的

1. 发送：send(message)
2. 接收：receive(message)

>> 假设，进程 P 和 Q 需要通信，首先需要建立通信线路(communication link)，并相互发送消息和接受消息

>> 通信线路的逻辑实现方法

1. 直接或间接通信（direct or indirect）
2. 同步或异步通信（blocking or non-blocking）
3. 自动或显式缓冲（通信缓冲）

对于直接通信，需要通信的每个进程必须明确地命名通信的接受者和发送者

1. 对称寻址（一对一）

（1）send(P, message)： 向进程 P 发送消息

（2）receive(Q, message)： 接收进程 Q 发来的消息

通信线路的属性：

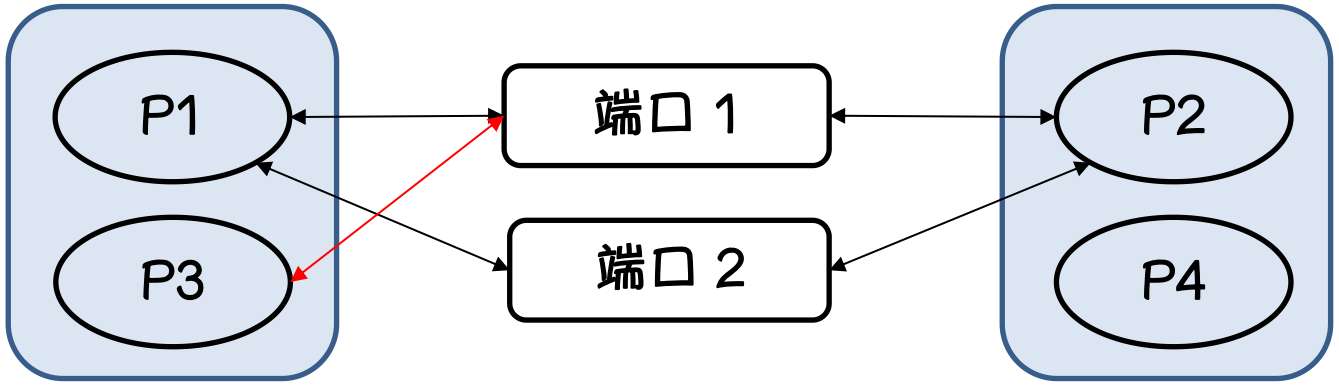
- 线路是自动建立的，
- 一个线路只与两个进程相关，
- 两个进程之间只有一个线路

2. 非对称寻址：接收者不需要命名发送者（多对一）

（1）send(P, message)

（2）receive(id, message)：接收任何进程发送的消息

- 通过端口(port)或邮箱(email)来发送或接受消息
- 每个邮箱都有一个唯一的标识符，进程之间可以通过共享端口来进行通信
- 通信线路具有以下属性
  1. 两个进程共享一个邮箱(端口)就可以建立通信线路
  2. 一个线路(端口)可以与多个进程相关联
  3. 两个通信进程之间可有多个通信线路，每个线路对应一个邮箱(端口)



4.2 消息传递：间接通信

- 操作系统拥有的邮箱（端口）是独立存在的，操作系统必须提供机制以允许进程进行如下操作

  1. 创建或删除邮箱（端口）
  2. 通过邮箱（端口）发送和接受消息

- 世界标准邮箱（端口），

如 网页WEB使用80号端口，FTP使用21号端口，Telnet 使用 23号端口, ssh 使用22号等

- 消息传递可以是同步或异步，又称为阻塞(blocking)或非阻塞(non-blocking)
- 发送和接收类型

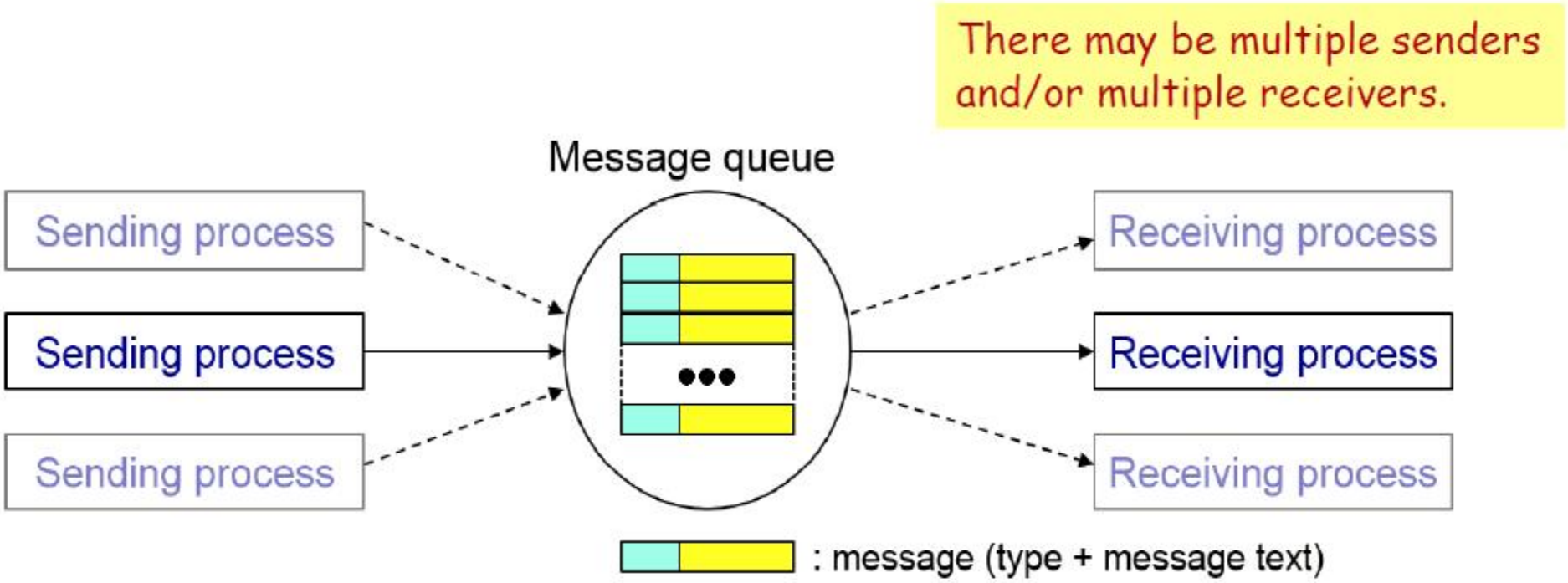
发送者:

1. 阻塞(同步)发送：发送进程阻塞，直到消息被对方进程接收
2. 非阻塞(异步)发送：发送进程发送消息并继续操作

接受者:

1. 阻塞(同步)接收：接收者阻塞，直到有消息可用
2. 非阻塞(异步)接收：接收者收到一个有效消息或空消息

不管通信是直接或间接的，通信进程所交换的消息都驻留在临时的队列中，即缓冲区里。



消息队列的实现方式有以下三种：

1、零容量（zero capacity）：

队列的最大长度为0。因此，线路中没有等待的消息。（阻塞）

2、有限容量（bounded capacity）

队列长度为有限的。如果线路为满，发送者必须等待，直到队列中的空间可用为止

3、无限容量（unbounded capacity）

队列长度为无限。不管多少消息在等待，从不阻塞发送者，即发送者不必等待

POSIX 共享内存系统调用

1. 进程创建共享内存对象

   shm_fd=shm_open(name,O_CREAT|O_RDRW,0666)

2. 设置对象大小

   ftruncate(shm_fd, 4096)

3. 内存映射共享内存对象

   ptr = mmap(0, 4096, PROT_WRITE, MAP_SHARED, shm_fd, 0)

4. 向共享内存写入

   sprintf(ptr, "Writing to shared memory")

5. 移除共享内存对象

   shm_unlink(name)

IPC POSIX Producer

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.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 shm_fd;
/* pointer to shared memory obect */
void *ptr;
```

IPC POSIX 生产者


```
    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_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 消费者


```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.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 shm_fd;
/* pointer to shared memory obect */
void *ptr;
```

```
    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

编译命令：gcc –o xxx xxx.c -lrt

hbpark@hbpark-VirtualBox: ~/src/shm

```
include <stdio.h>
include <sys/shm.h>
include <sys/stat.h>
include <unistd.h>

nt main()

        int segment_id ;
        char * shared_memory;
        const int size = 4096;

        segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
        shared_memory = (char *) shmat(segment_id, NULL, 0);

        pid_t pid = fork();
        if (pid == 0) {
                shared_memory = (char *) shmat(segment_id, NULL, 0);
                printf("Child Write: Hi There!\n");
                sprintf(shared_memory, "Hi There!");
        }
        else {
                wait(NULL);
                printf("Parent Read: %s\n", shared_memory);
                shmdt(shared_memory);
                shmctl(segment_id, IPC_RMID, NULL);
        }
/       shmdt(shared_memory);
/       shmctl(segment_id, IPC_RMID, NULL);
        return 0;
```

Shared Memory

7,1 Top

Message Passing

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <time.h>
#include <strings.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msg {
    long msg_types;
    char msg_buf[511];
};

int main (void) {
    int qid, pid, len;
    struct msg pmsg;
    sprintf ( pmsg.msg_buf, "Hello! This is :%d\n", getpid());
    len = strlen ( pmsg.msg_buf );

    if ( (qid = msgget( IPC_PRIVATE, IPC_CREAT | 0666)) < 0 ) {
        perror( "msgget failed!");
        exit (1);
    }

    if ( (msgsnd(qid, &pmsg, len, 0)) < 0) {
        perror("msgsnd failed!");
        exit(1);
    }

    printf("Send a message to the queue successfully : %d\n", qid);
    exit(1);
}
```

Windows XP IPC系统-消息传递实例

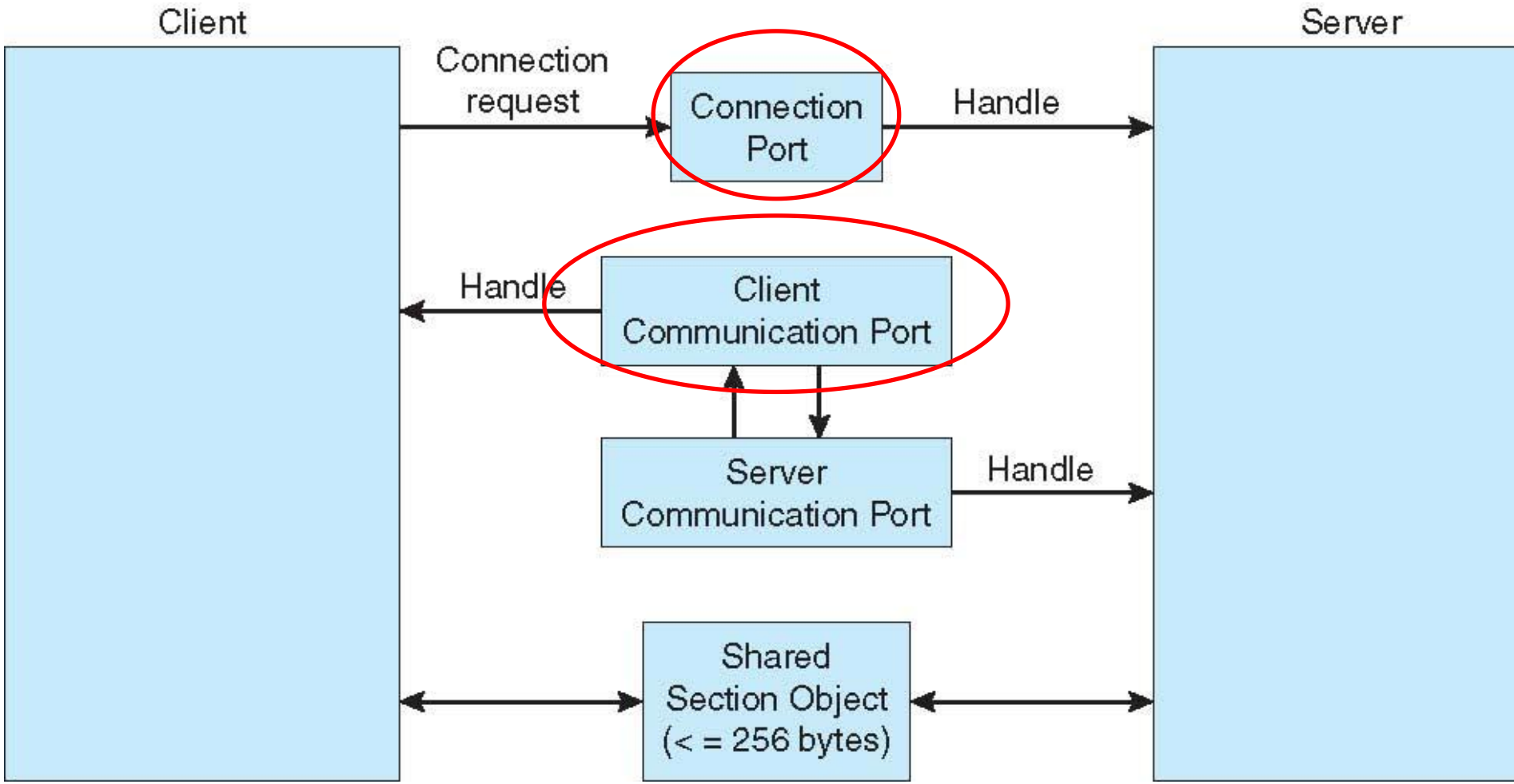
Windows中的消息传递称为 LPC(Local Procedure Call)，用于同一系统内的进程之间进行通信

- 使用端口建立和维护进程之间的链接

通信过程如下：

1. 客户机获得链接服务器的端口对象的句柄(handler)
2. 客户机发送连接请求
3. 服务器创建两个私有通信端口，并返回其中之一的句柄给客户机
4. 客户机和服务器使用相应端口句柄进行通信

Windows XP IPC系统-消息传递实例



第五节、
客户机与服务器系统间通信

Sockets（套接字） SOCKET

A **network socket** is a software structure within a network node of a computer network that serves as an endpoint for sending and receiving data across the network. The structure and properties of a socket are defined by an application programming interface (API) for the networking architecture. Sockets are created only during the lifetime of a process of an application running in the node.

I. 远程过程调用
RPC: Remote Procedure Calls

II. 远程方法调用
RMI: Remote Method Invocation (in Java)

5.1 套接字（Sockets）

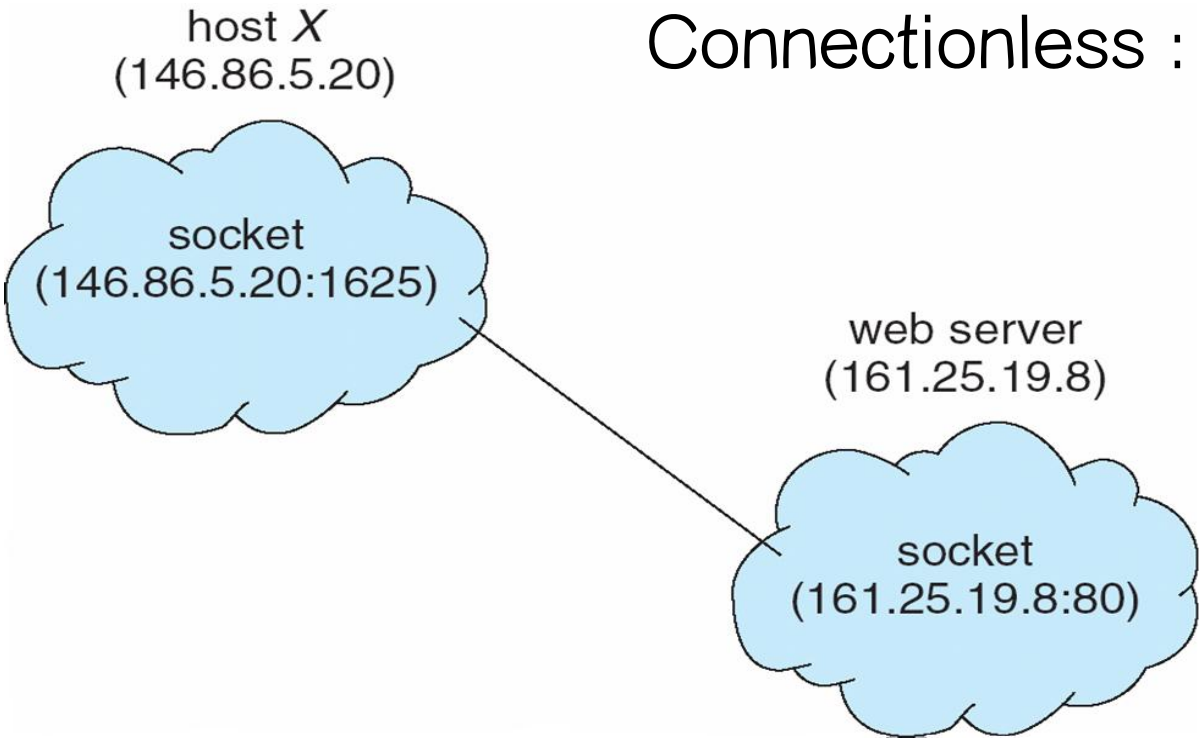
- 套接字被定义为通信的端点
- 套接字由IP地址和端口号连接组成
- 如，套接字161.25.19.8:1625 指的是主机161.25.19.8上的1625端口
- 连接由一对套接字组成

1、 面向连接套接字

Connection oriented ： TCP Socket

2、无连接套接字

Connectionless : UDP Socket



5.2 远程过程调用(RPC)

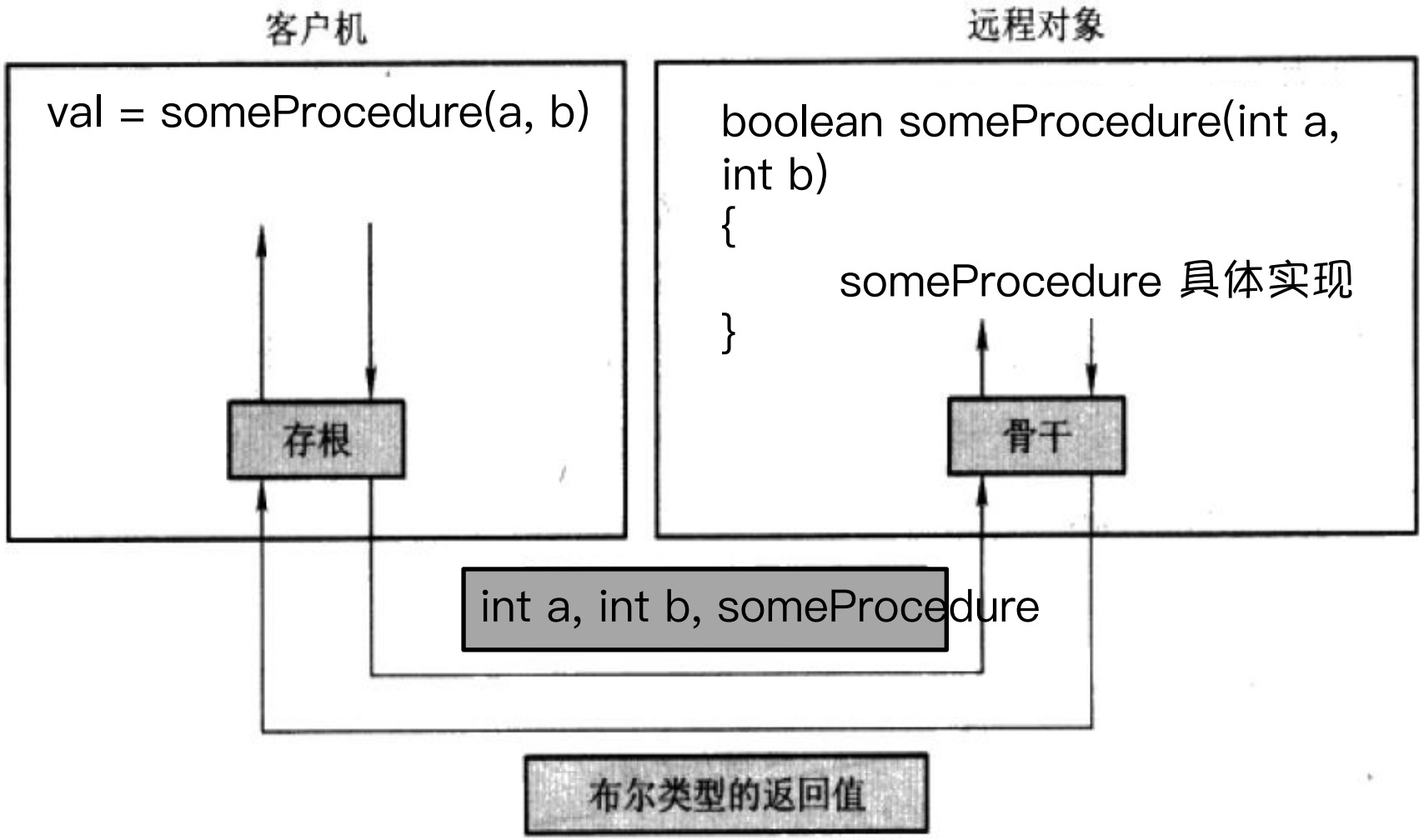
- 远程过程调用(Remote Procedure Call:RPC)抽象化了通过网络连接的进程之间过程调用。
- Stubs(存根)：远程过程的代理，隐藏了通信发生的细节
- 每个独立的远程过程都有一个存根

- 客户机调用位于远程主机上的过程时，

  (1) 客户端存根编组参数，并向服务器相应存根发送参数

  (2) 服务器存根接收参数，解组参数，并调用相应过程

远程过程调用


客户机
远程对象
val = someProcedure(a, b)
boolean someProcedure(int a,
int b)
{
someProcedure 具体实现
}
存根
骨干
int a, int b, someProcedure
布尔类型的返回值

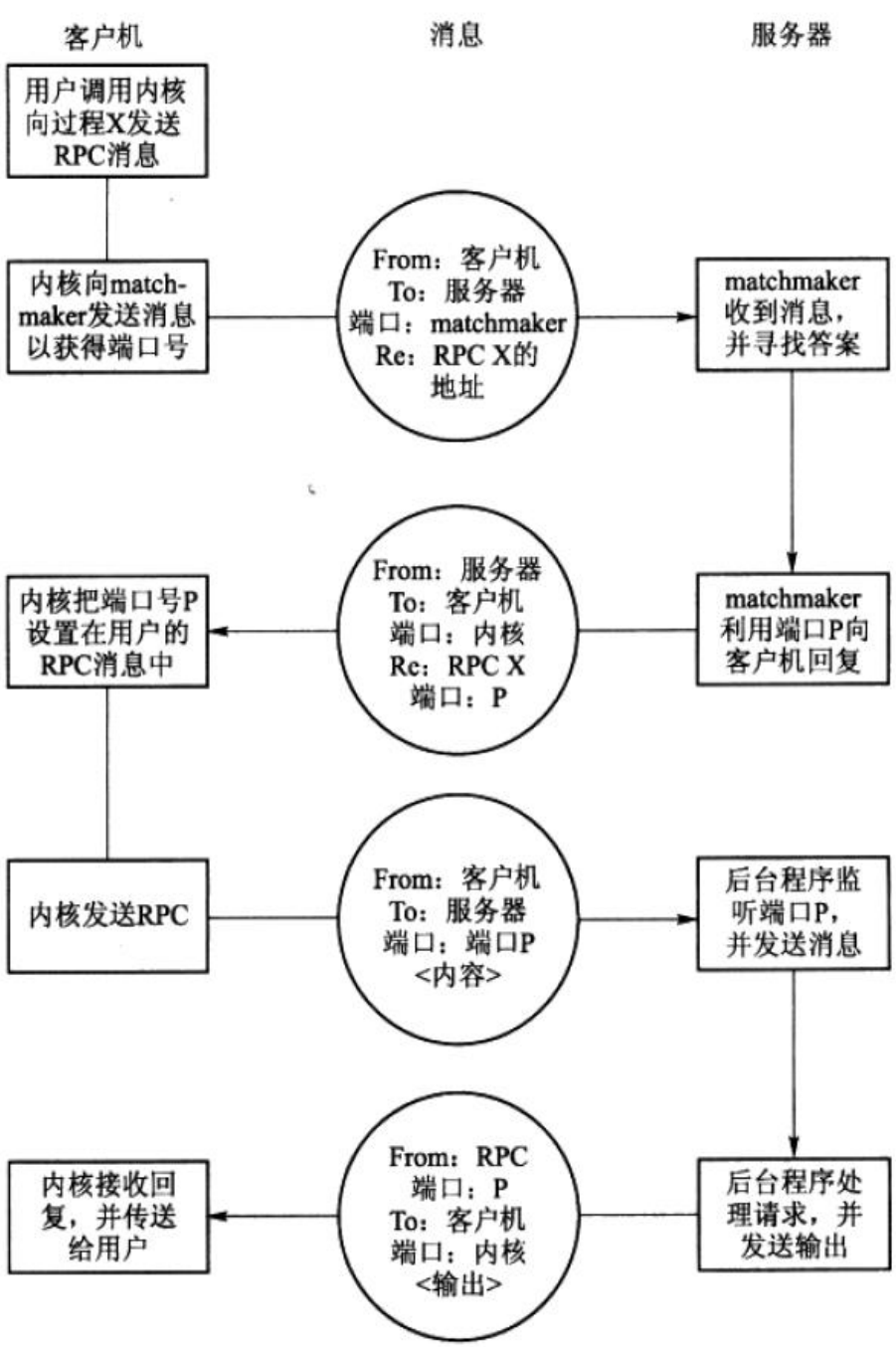


图 3.21　远程过程调用(RPC)的执行

远程过程调用操作

客户机与服务器之间的端口的绑定

1. 预先固定
2. 通过集合点机制动态绑定

集合点服务程序(matchmaker)

RMI（Remote Method Invocation）类似于远程过程调用，是RPC的JAVA特性。 与RPC的不同

- RPC调用远程子程序或函数，RMI调用远程对象的方法
- RPC参数传递方式是普通数据结构，RMI参数传递方式可以是对象

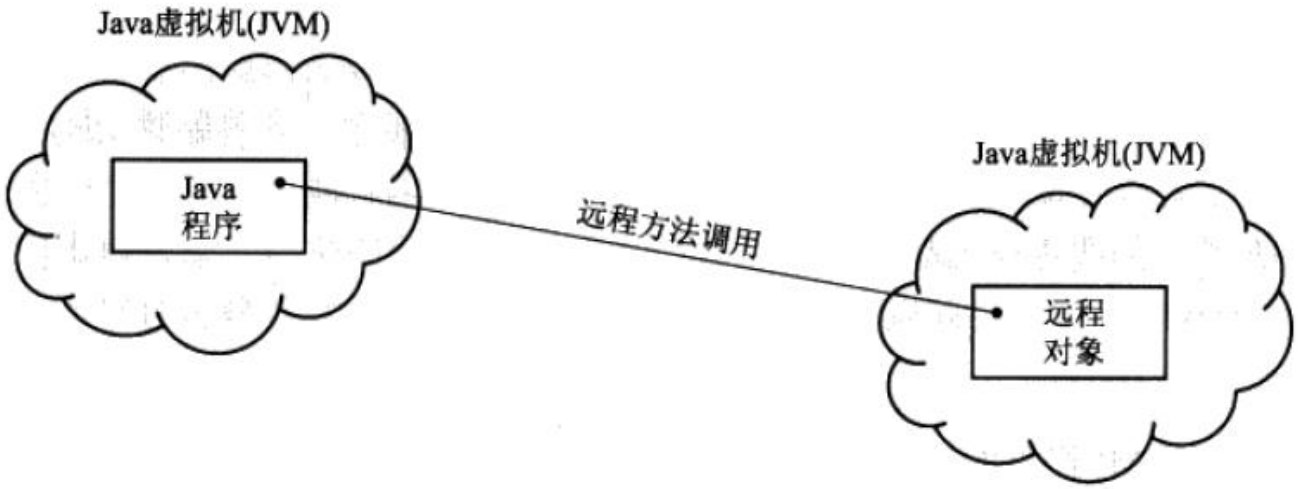
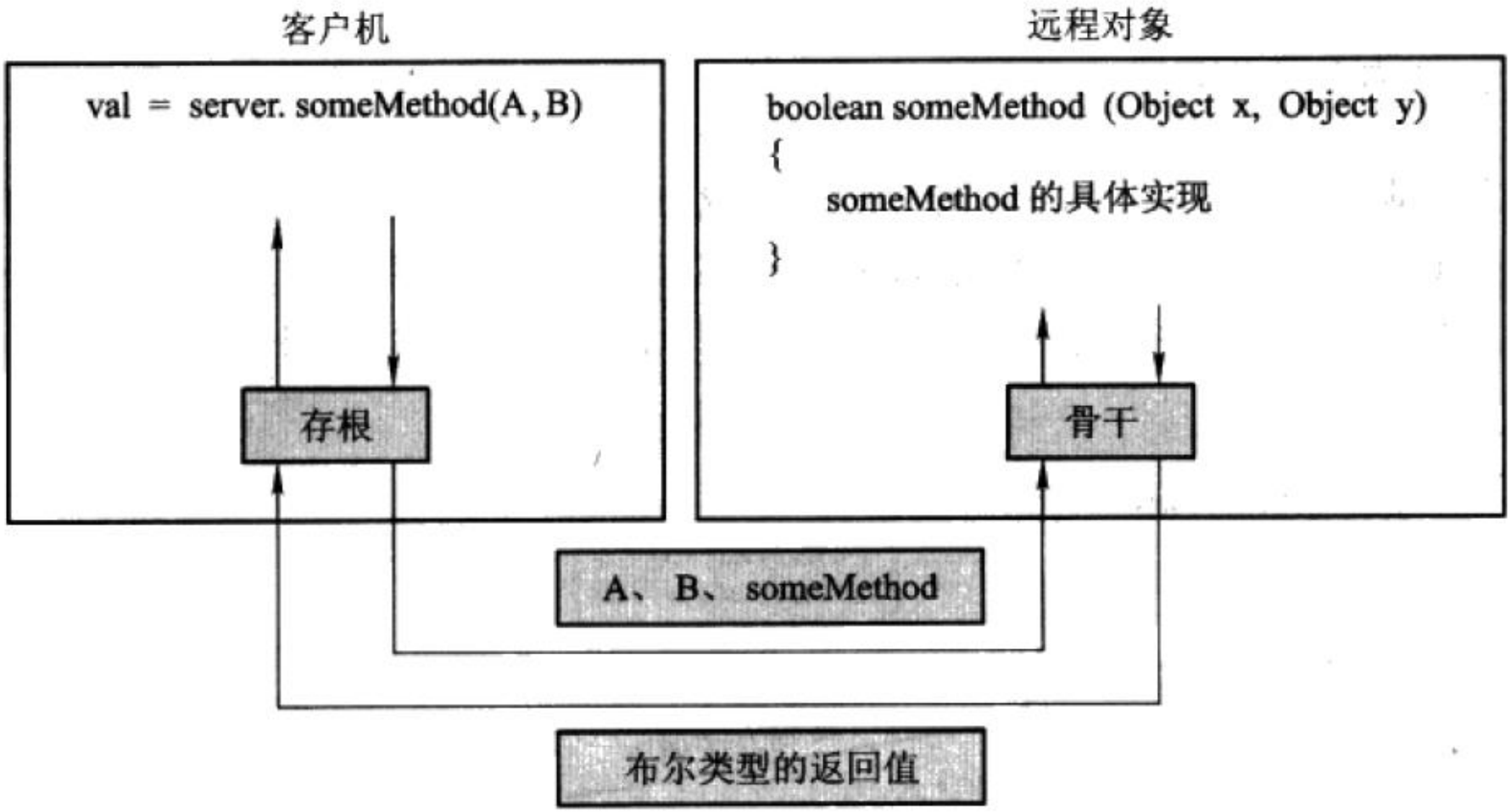


图 3.22 远程方法调用

远程方法调用


客户机
远程对象
val = server. someMethod(A, B)
boolean someMethod (Object x, Object y)
{
someMethod 的具体实现
}
存根
骨干
A、B、someMethod
布尔类型的返回值

第六节、管道 PIPE（IPC: Inter-Process Communication）

6. 管道(Pipe)

- 进程之间进行通信的另一种方式，管道通信方式的中间介质是文件，通常称这种文件为管道文件。
- 如两个进程利用管道文件进行通信时，一个进程为写进程，另一个进程为读进程。需要提供进程同步(synchronization)运行机制
- 管道分为匿名管道和命名管道

1. 匿名管道： Unamed pipes (ordinary/anonymous pipes)
2. 命名管道： Named pipes(FIFO pipe)

```
[root@iZuf620xwaci92tjy2fucbZ pipe]# ls -al
total 8
drwxr-xr-x  2 root root 4096 Nov 16 21:41 .
drwxr-xr-x 13 root root 4096 Nov 16 21:39 ..
prw-r--r--  1 root root    0 Nov 16 21:40 pipe1
prw-r--r--  1 root root    0 Nov 16 21:41 pipe2
prw-r--r--  1 root root    0 Nov 16 21:41 pipe3
[root@iZuf620xwaci92tjy2fucbZ pipe]#
```

9,8,7,6,5,4,3,2,1,0位

9位

- d：目录
- b：块特殊文件
- c：字符特殊文件
- l：符号链接文件
- p：命名管道文件FIFO
- s：套接字文件

8–6 位表所有者权限，

5–3 位表示同组其它用户权限，

2–0：其它用户权限

1、匿名管道: unnamed pipe

管道是半双工的，数据只能单向通信；双方通信时，需要建立起两个管道；只能用于父子进程或者兄弟进程之间(具有亲缘关系的进程)

2、命名管道： named pipe

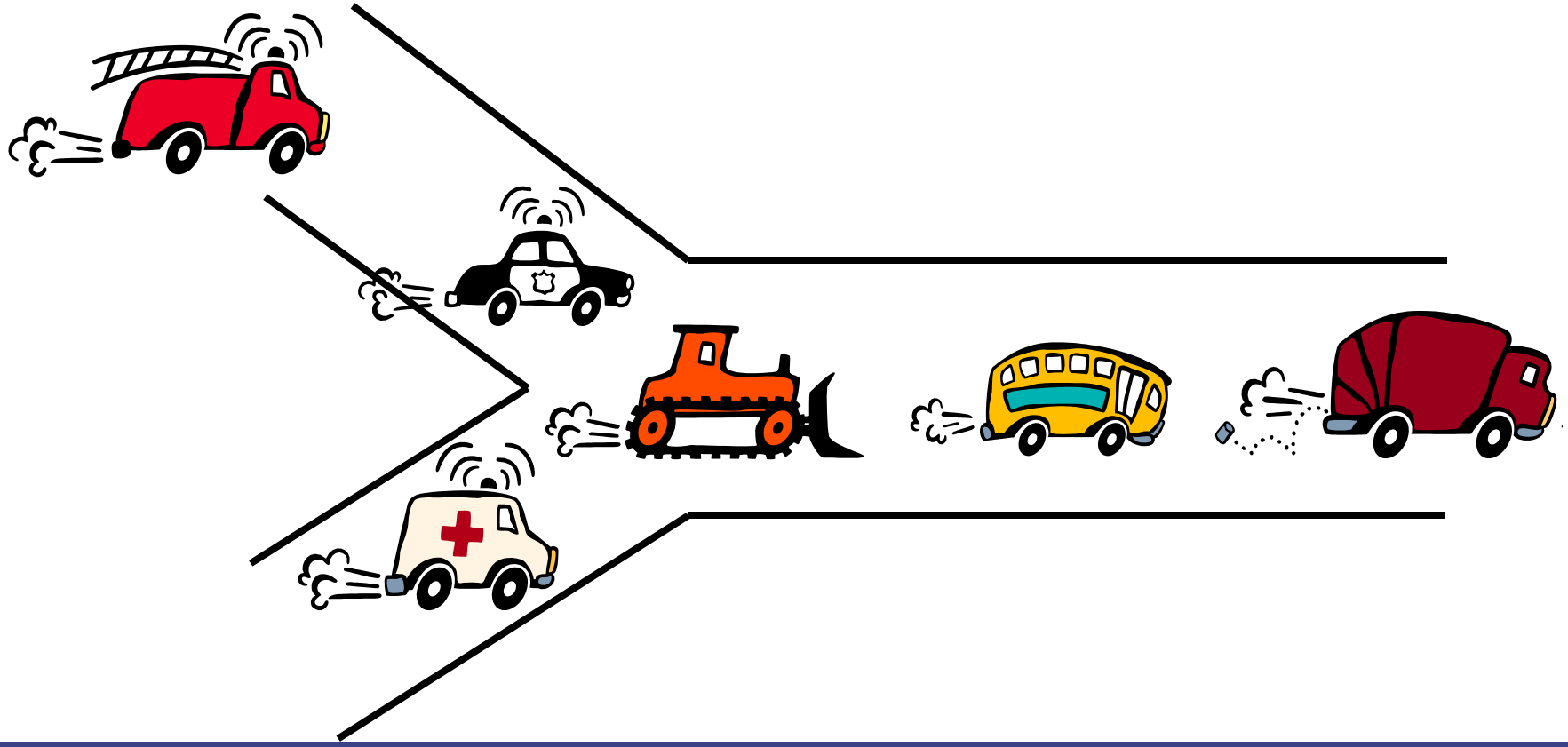
可在同一台计算机的不同进程之间或在跨越一个网络的不同计算机的不同进程之间，支持单向或双向的数据通信

管道的实现：FIFO

- FIFO (First In, First Out)缓冲区就像一个队列
- 在队列的一端添加元素，并以相同的顺序从另一端退出
- 任何一个元素都不可能先于另一个元素前进

Multiple Inputs-多输入管道

- 一个管道可以有多个输入
- 但除了先进先出，没有顺序的保证



匿名管道

- 普通(未命名)管道的使用与我们在Unix I中看到的一样

```
$cat myfile | grep key | sort | lpr
```

- 父进程（创建管道的shell或shell脚本）还生成访问管道的子进程
  - cat, grep, sort, and lpr in this case
  - Note: the shell or script process that sets up the pipes CANNOT access the pipes itself!

命名管道

- 命名管道可以被任何“知道其名称”的进程访问。
- 命名管道出现在用户的目录列表中

```
$ ls –l
pwr_r__r__   1   krf   0   Mar 27 19:33   mypipe
```

- 命名管道使用mknod或mkfifo命令创建

```
$  mkfifo name
$  mkfifo —m mode name
$  mknod name p
```

- 确保在使用后需删除(rm)管道

命名管道实例

- First, create your pipes
  - $ mknod pipe1 p
  - $ mknod pipe2 p
  - $ mknod pipe3 p
- Then, attach a data source to your pipes
  - $ ls –l >> pipe1 &
  - $ cat myfile >> pipe2 &
  - $ who >> pipe3 &
- Then, read from the pipes with your reader process
  - $ cat < pipe1
  - $ spell < pipe2
  - $ sort < pipe3
- Finally, delete your pipes
  - $ rm pipe[1–3]

Thank You!