



# 第三章 进程管理

---

陆松年 [snlu@sjtu.edu.cn](mailto:snlu@sjtu.edu.cn)

# 作业管理的功能

作业调度从预先存放在辅助存储设备中的一批用户作业中，按照某种方法选取若干作业，为它们分配必要的资源，决定调入内存的顺序。

- 建立相应的用户作业进程和为其服务的其他系统进程。
- 然后再把这些进程提交给进程调度程序处理。
- 作业管理是宏观的高级管理，进程管理是微观的低级管理。

# 作业的状态变迁

## (1) 后备状态

作业的提交后，操作系统要对作业进行登记，建立并填写一些与作业有关的表格。建立一个作业控制块（**JCB**）。

## (2) 执行状态

它分配必要的资源，提交给进程管理模块

## (3) 完成状态

从作业队列中去掉，回收作业所占用的资源

# 作业调度算法

- **(1) 先来服务 (FCFS)** 有利于长作业而不利于短作业。
- **(2) 短作业优先 (SJF)** 较短的作业平均等待时间，较大的系统吞吐率。
- **(3) 响应比高优先 (HRN)** 求等待时间与执行时间两者时间之比。相对等待时间长优先。
- **(4) 优先级调度** 作业的紧急程度、资源要求、类别等。

**选择调度算法考虑的因素：**最高的吞吐率，最高的资源利用率，合理的作业调度，使各类用户都满意。



# 3.1 进程概述

---

## 3.1.1 进程的概念

### 进程的定义：

进程是程序处于一个执行环境中在一个数据集上的运行过程，它是系统进行资源分配和调度的一个可并发执行的独立单位。



# “进程”与“程序”的关系

- 进程是程序的一次动态执行活动，而程序是进程运行的静态描述文本。
- 一个进程可以执行一个或多个程序。
- 同一程序也可被多个进程同时执行。
- 程序是一种软件资源，它可以长期保存，而进程是一次执行过程，它是暂时存在的，动态地产生和中止的。

## 3.1.2 进程的组成

进程是在一个上下文的执行环境中执行的，这个执行环境称为进程的映像，或称图像。

- 包括处理机中各通用寄存器的值，进程的内存映像，打开文件的状态和进程占用资源的信息等。进程映像的关键部分是存储器映像。
- 进程存储器映像由以下几部分组成：
  - ❖ 进程控制块（PCB）、
  - ❖ 进程执行的程序（code）、
  - ❖ 进程执行时所用的数据、
  - ❖ 进程执行时使用的工作区。



# 进程的基本组成图

进程控制块

共享正文段

数据区

工作区





# 1. 进程控制块

- **PCB**是系统用于查询和控制进程运行的档案，它描述进程的特征，记载进程的历史，决定进程的命运。
- **PCB**可分为两部分：
  - 一部分是**进程基本控制块**。基本控制块要常驻内存。
  - 另一部分是**进程扩充控制块**。当进程不处于执行状态时，操作系统就不会访问这部分信息。扩充控制块可以在盘交换区与内存之间换入换出。

## 2. 共享正文段

共享正文段是可以被多个进程并发地执行的代码，由不可修改的代码和常数部分组成，例如：编辑程序 *vi*。

- 用户用C语言所编的程序，经编译后产生的代码也是作为共享正文段装入内存的。

## 3. 数据段

- ✦ 进程执行时用到的数据，如C程序中的外部变量和静态变量。
- ✦ 如进程执行的程序为非共享程序，则也可构成数据段的一部分。



## 4. 工作区

---

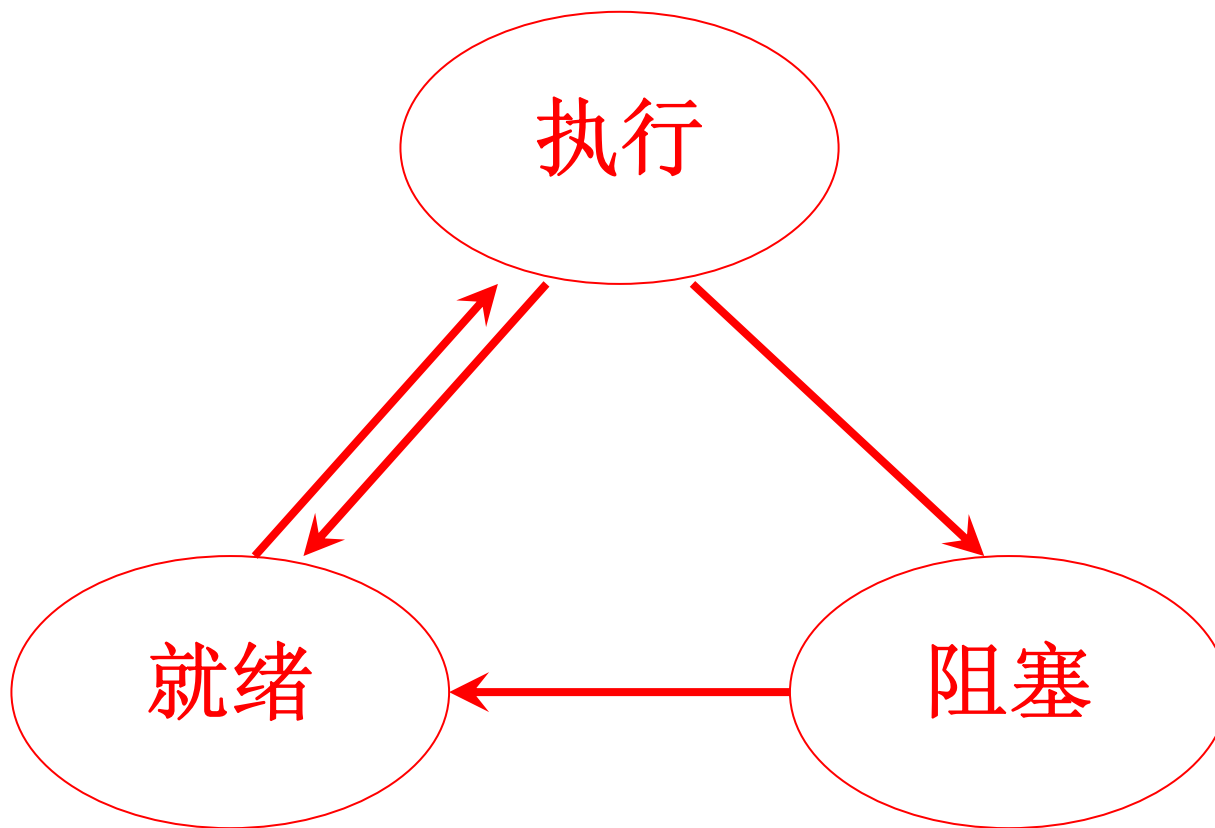
- **核心栈**——进程在核心态运行时的工作区。
- **用户栈**——进程在用户态下运行时的工作区。
- 在调用核心的函数或用户的函数时，两种栈分别用于传递参数、存放返回地址、保护现场以及为局部动态变量提供存储空间。



## 3.1.3 进程的状态及其变化

- 进程具有生存期，它有一个创建、活动及消亡的过程。
- 进程在其整个生存期间可处于不同的状态，有一些不同的特征，其中最基本的状态有以下三种。
  - 执行 (Running) 状态
  - 就绪 (Ready) 状态
  - 阻塞 (Blocked) 状态

# 进程状态的转换图





## 3.2 进程控制块

---

在PCB中一般包括以下的信息：

- 进程的标识信息
- 进程的特征
- 现场保护区
- 资源信息
- 运行管理信息
- 进程的状态
- 进程的位置和大小
- 进程通信信息
- 进程间联系

# 1. proc结构

UNIX中常驻内存的PCB部分称为proc结构。

```
struct proc {  
    char    p_stat;    /* 进程状态 */  
    char    p_flag;    /* 进程标志 */  
    char    p_pri;    /* 进程运行的优先数 */  
    char    p_time;    /* 进程驻留时间 */  
    char    p_cpu;    /* 进程使用CPU的量 */  
    char    p_nice;    /* 进程优先数偏置值 */  
    ushort  p_uid;    /* 实际用户标识数 */  
    ushort  p_suid;    /* 有效用户标识数 */  
    short   p_pgrp;    /* 所在进程组的首进程标识数 */  
};
```

## struct proc (续)

```
short    p_pid;        /* 进程标识数 */
short    p_ppid;       /* 父进程标识数 */
short    p_addr;       /* 相应user结构的起始页面号 */
short    p_size;       /* 可交换存储映像的大小 */
short    p_swaddr;     /* 交换区的磁盘地址 */
short    p_sig;        /* 进程收到的软中断信号 */
caddr_t  p_wchan;      /* 进程睡眠原因 */
struct text *p_textp;
                /* 指向正文段控制结构的指针 */
struct proc *p_link;   /* 运行队列进程或各睡眠队列进程的链接指针 */

.....
};
```



# text结构

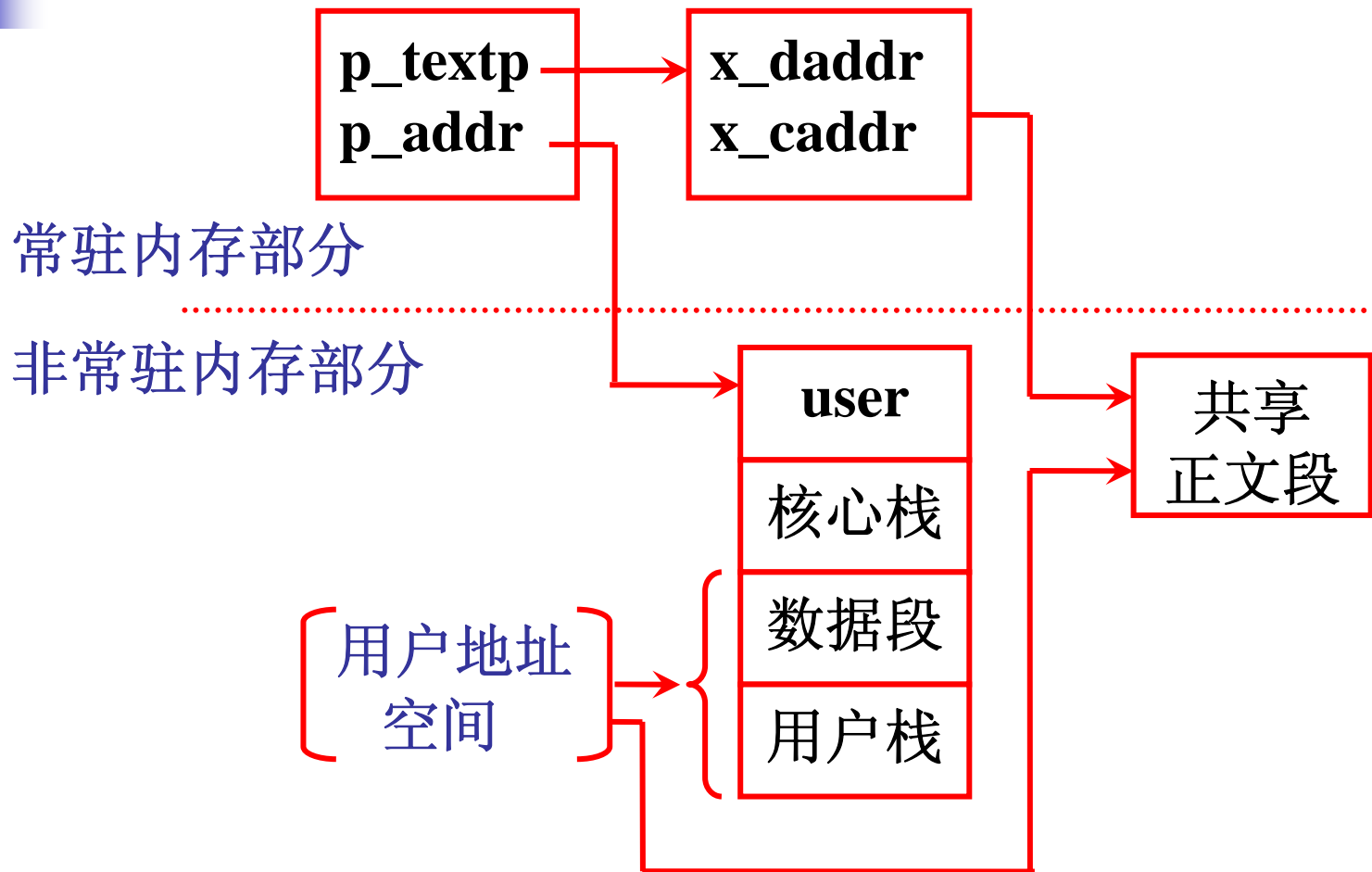
系统分配一个控制块**text**结构，以便于多个进程共享一个可执行程序 and 常数段。

```
struct text {
    short    x_daddr;           /* 正文段在盘交换区地址 */
    short    x_size;           /* 正文段块数 */
    struct proc *x_caddr;      /* 指向链接的proc结构 */
    struct inode *x_iptr;     /* 指向正文段所在文件的
                               内存索引节点的指针 */
    char     x_count;          /* 共享该正文段的进程数 */
    char     x_ccount;         /* 共享该正文段且映像在
                               内存的进程数 */
    char     x_flag;           /* 标志 */
};
```

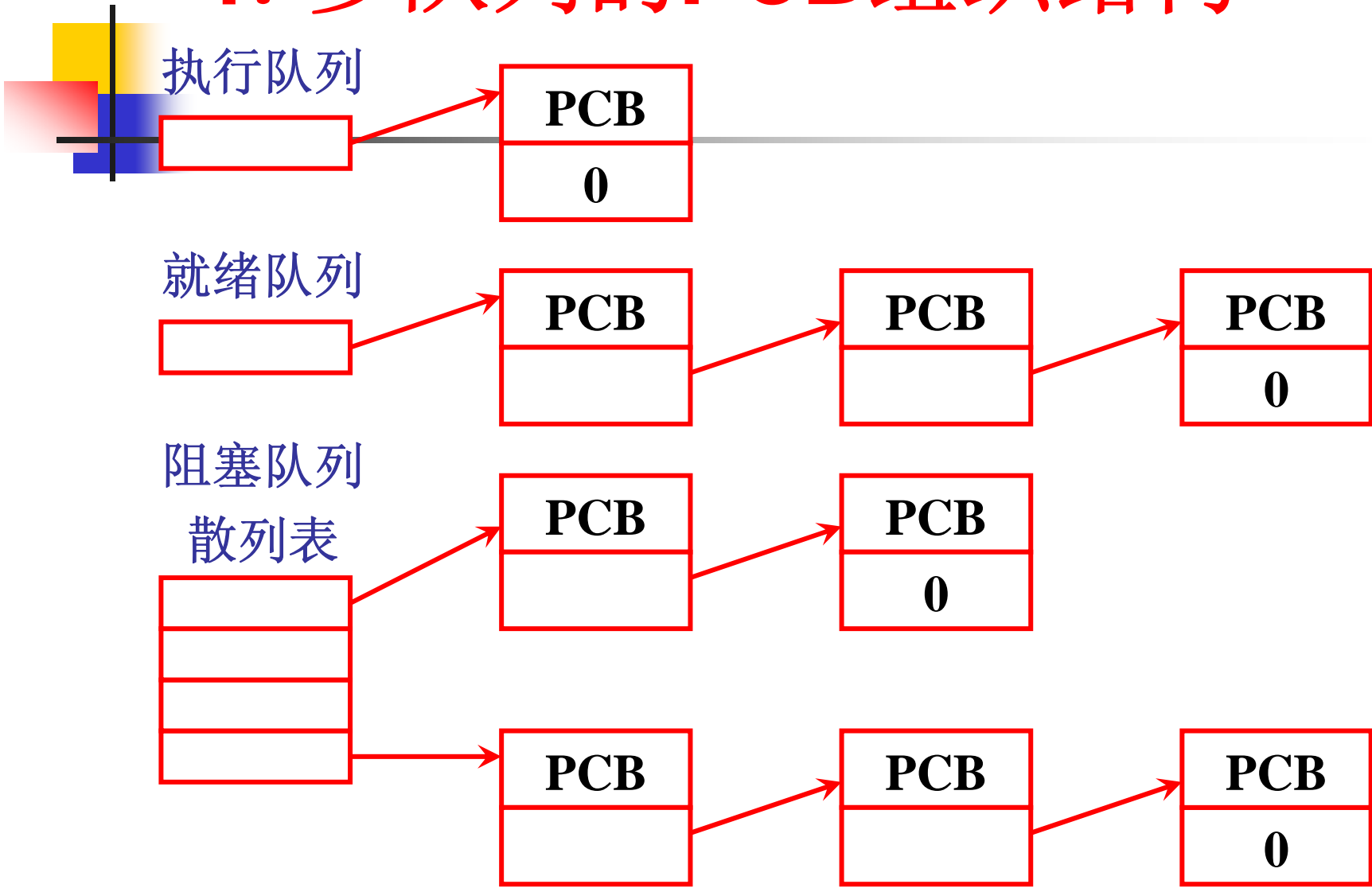
## 2. user结构

```
struct user {
    struct pcb u_pcb;      /* 进程切换时保存硬件环境区域 */
    ushort    u_uid;      /* 有效用户标识数 */
    ushort    u_gid;      /* 有效用户组标识数 */
    ushort    u_ruid;     /* 实际用户标识数 */
    ushort    u_rgid;     /* 实际用户组标识数 */
    struct proc *u_procp; /* 指向相应proc结构的指针 */
    struct file *u_ofile[NOFILE]; /* 用户打开文件表 */
    unsigned   u_tsize;    /* 代码段长度 */
    unsigned   u_dsize;    /* 数据段长度 */
    unsigned   u_ssize;    /* 堆栈段长度 */
    int u_signal[NSIG];    /* 软中断信号的处理函数地址表 */
    time_t     u_otime;    /* 进程用户态运行时间累计值 */
    time_t     u_stime;    /* 进程核心态运行时间累计值 */
};
```

# 3. 进程映像的基本结构



# 4. 多队列的PCB组织结构

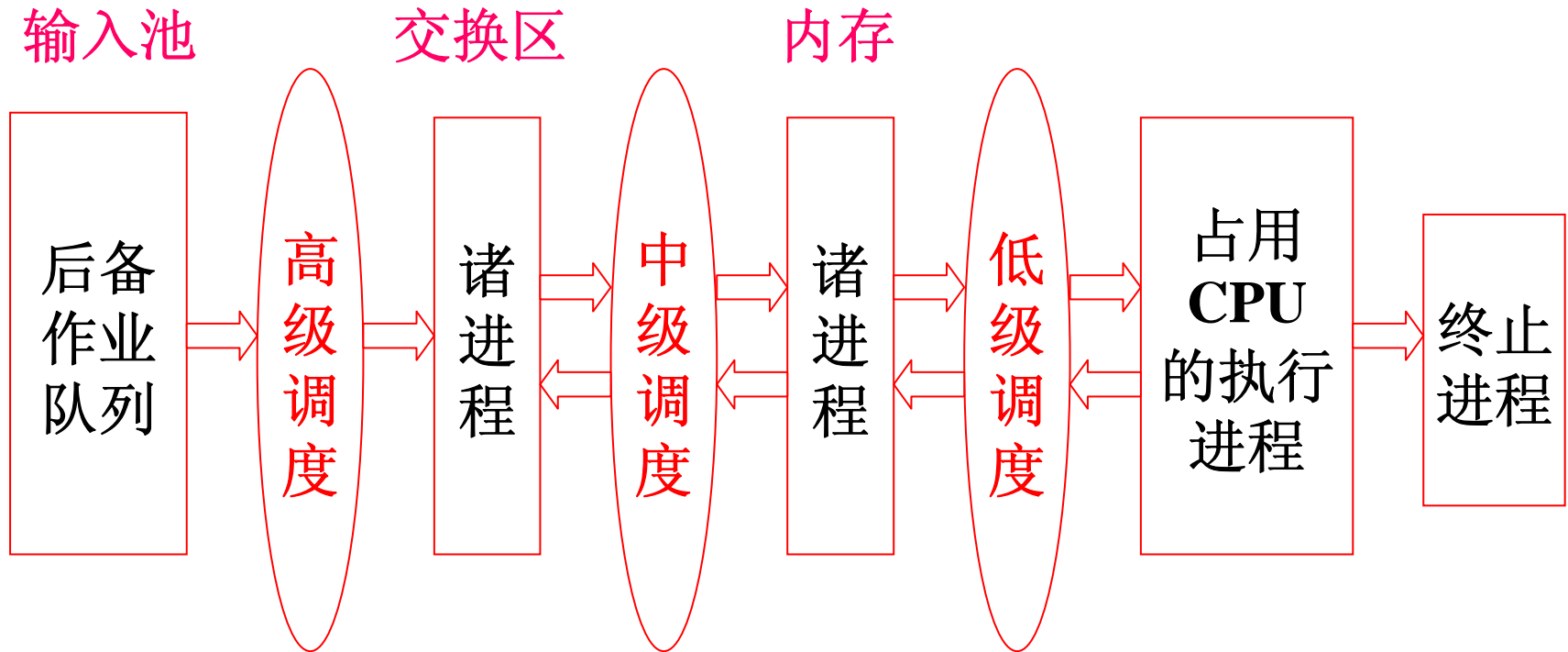


# 3.3 调度

## 3.3.1 调度概述

- **高级调度**：又称作业调度，它决定处于输入池中的哪个后备作业可以调入主系统，成为一个或一组就绪进程。
- **中级调度**：又称对换调度，它决定处于交换区中的就绪进程中哪一个可以调入内存，以便直接参与对**CPU**的竞争。在内存资源紧张时，将内存中处于阻塞状态的进程调至交换区。
- **低级调度**：又称进程调度或处理机调度，它决定驻在内存中的哪一个就绪进程可以占用**CPU**，使其获得实实在在的**执行权力**。

# 三种调度的工作情况图





## 3.3.2 进程调度策略

- 进程切换的两种方式：
  - 不可剥夺（或不可抢占）方式
  - 可剥夺方式
- 调度策略的权衡因素：
  - 考虑不同的的设计目标。
  - 批处理系统，提高运行效率，取得最大的作业吞吐量和减少作业平均周转时间；
  - 交互式分时系统，能及时响应用户的请求；
  - 实时系统，能对紧急事件作出及时处理和安全可靠。



## 调度策略的权衡因素（续）

- 调度算法应能充分使用系统中各种类型资源，使多个设备并行地工作。
- 既能公平地对待各个进程，使它们能均衡地使用处理机，也能考虑不同类型进程具有不同的优先权利。
- 合理的系统开销。





## 3.3.3 进程调度算法

---

- 先来先服务（**FIFO**）调度算法
- 时间片轮转法
- 优先级调度算法
- 多级反馈队列调度算法
- 实时系统调度策略



# 1. FIFO调度算法

- 按照进程达到就绪队列的时间次序分配处理机，这是一种不可强占式的简单算法。一旦进程占用了处理机，它可一直运行到结束或因阻塞而自动放弃处理机。
- 缺点是当一个大进程运行时会使后到的小进程等待很长时间，这就增加了作业平均等待时间。
- 对于I/O繁忙的进程，每进行一次I/O都要等待其他进程一个运行周期结束后才能再次获得处理机，故大大延长了该类作业运行的总时间，也不能有效利用各种外部设备资源。



## 2. 时间片轮转法

- 基本思想是按进程到达的时间排在一个**FIFO**就绪队列中，每次选择队首的进程占用处理机并运行一段称为“时间片”的固定时间间隔。
- 在时间片内，如进程运行任务完成或因I/O等原因进入阻塞状态，该进程就提前退出就绪队列，调度程序就使就绪队列中的下一个进程占用处理机，使用一个时间片。
- 当一个进程耗费完一个时间片而尚未执行完毕，调度程序就强迫它放弃处理机，使其重新排到就绪队列末尾。

## 2. 时间片轮转法（续）

时间片轮转法较适合于交互式分时系统。

- 系统的效率与时间片大小的设置有关。如时间片过大，系统与用户间的交互性就差；如时间片太小，进程间切换过于频繁，系统开销就增大。
- 在系统中可设置时间片大小不同的 $n$ 个队列。
- 将运行时间短、交互性强或I/O繁忙的进程安排在时间片小的队列，这样可以提高系统的响应速度和减少周转时间。
- 将需要连续占用处理机的进程安排在时间片长的队列中，这样可减少进程切换的开销。根据运行状况，进程可以从时间片小的队列中转入时间片较长的队列。



## 3. 优先级调度算法

---

### 两类优先级调度算法：

- ▶ **静态优先级法：** 在进程创建时就赋予一个优先数，在进程运行期间该优先数保持不变。
- ▶ **动态优先级法：** 反映进程在运行过程中不同阶段的优先级变化情况。



# (1) 静态优先数的确定

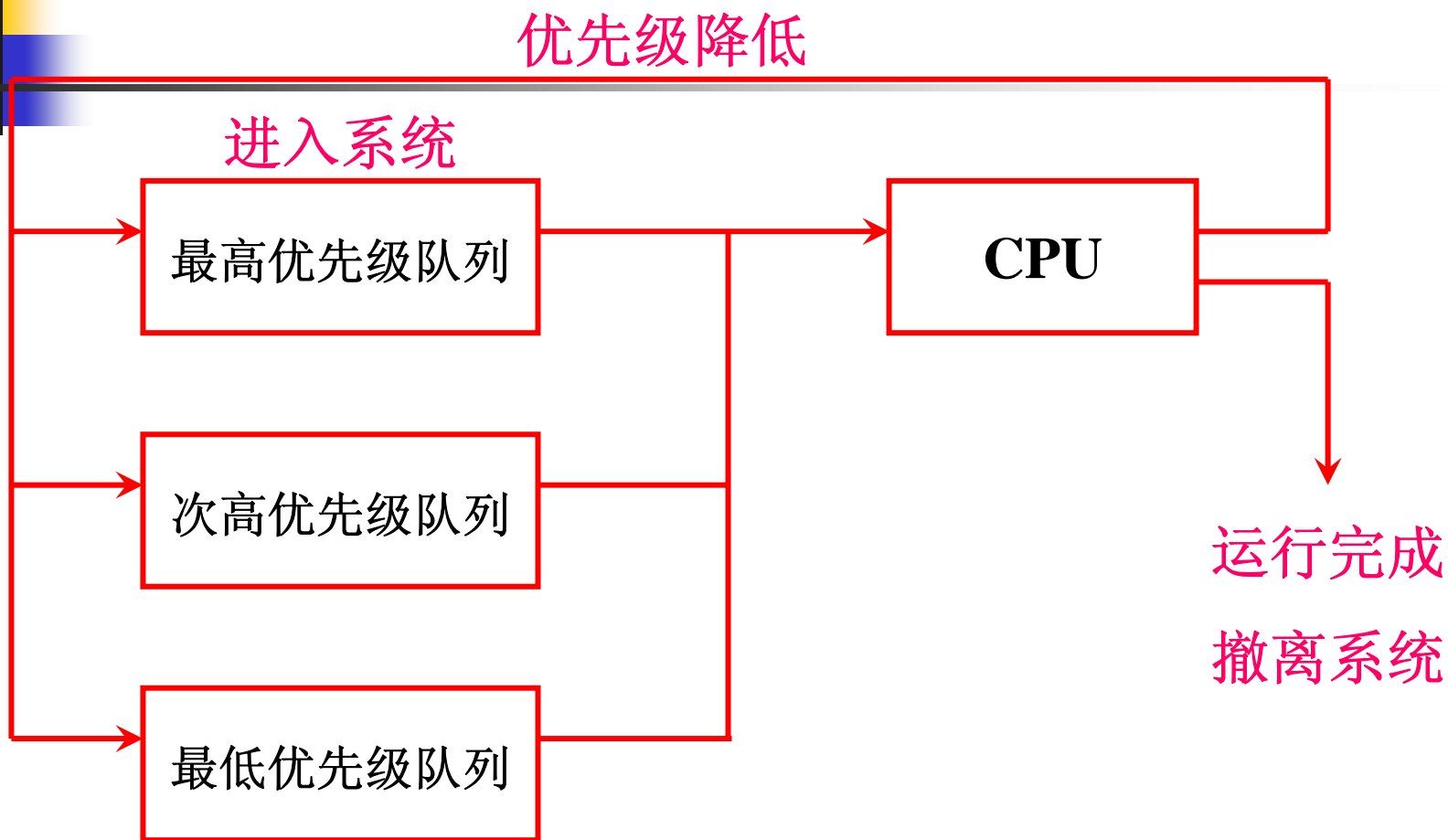
- 系统进程应当赋予比用户进程高的优先级。
- 短作业的进程可以赋予较高的优先级。
- **I/O**繁忙的进程应当优先获得**CPU**。
- 根据用户作业的申请，调整进程的优先级。
- 静态优先权法较适合于实时系统，其优先级可根据事件的紧迫程度事先设定。



## (2) 动态优先级的考虑

- 对于一个总体**CPU**忙的进程，在其**I/O**阶段就应提高其优先级，而在其**CPU**繁忙阶段，可以降低该进程的优先级。
- 对于运行到某一阶段的进程，需要和用户交互才能正确运行下去，也应当在该阶段提高优先级，以减少用户等待的时间。
- 进程占用**CPU**时间越长，就可降低其优先级；反之一个进程在就绪队列中等待的时间越长，就可升高其优先级。
- 也可根据进程在运行阶段占用的系统资源，如内存、外部设备的数量 and 变化来改变优先级。

# 4. 多级反馈队列调度算法





# 3.4 UNIX系统的进程调度

## 3.4.1 进程的切换调度算法

### 1. UNIX的切换调度策略

- 采用动态优先权算法：在一个适当的时机，选择一个优先权最高，也即优先数（`p_pri`）最小的就绪进程，使其占用处理机。
- 进程可以处于两种运行状态，即用户态和核心态。
- 系统对在用户态下运行的进程，可以根据优先数的大小，对它们进行切换调度。
- 对在核心态下运行的进程，一般不会对它们进行切换调度，即UNIX核心本质上是不可重入的。

## 2. 优先数的计算

对用户态的进程，核心在适当时机用下列公式计算进程的优先数：

$$p\_pri = p\_cpu/2 + p\_nice + PUSER + NZERO$$

- **p\_cpu**是进程占用处理机的量度，每次时钟中断，使当前执行进程的**p\_cpu**加1，但最多加到80；每1秒钟使所有就绪状态进程的**p\_cpu**衰减一半。
- 形成一个负反馈的过程，使用户态的诸进程能比较均衡地使用处理机。
- **p\_nice**是用户设置的进程优先数偏置值。
- **PUSER**和**NZERO**是两个常量，用于分界不同类型的优先数。



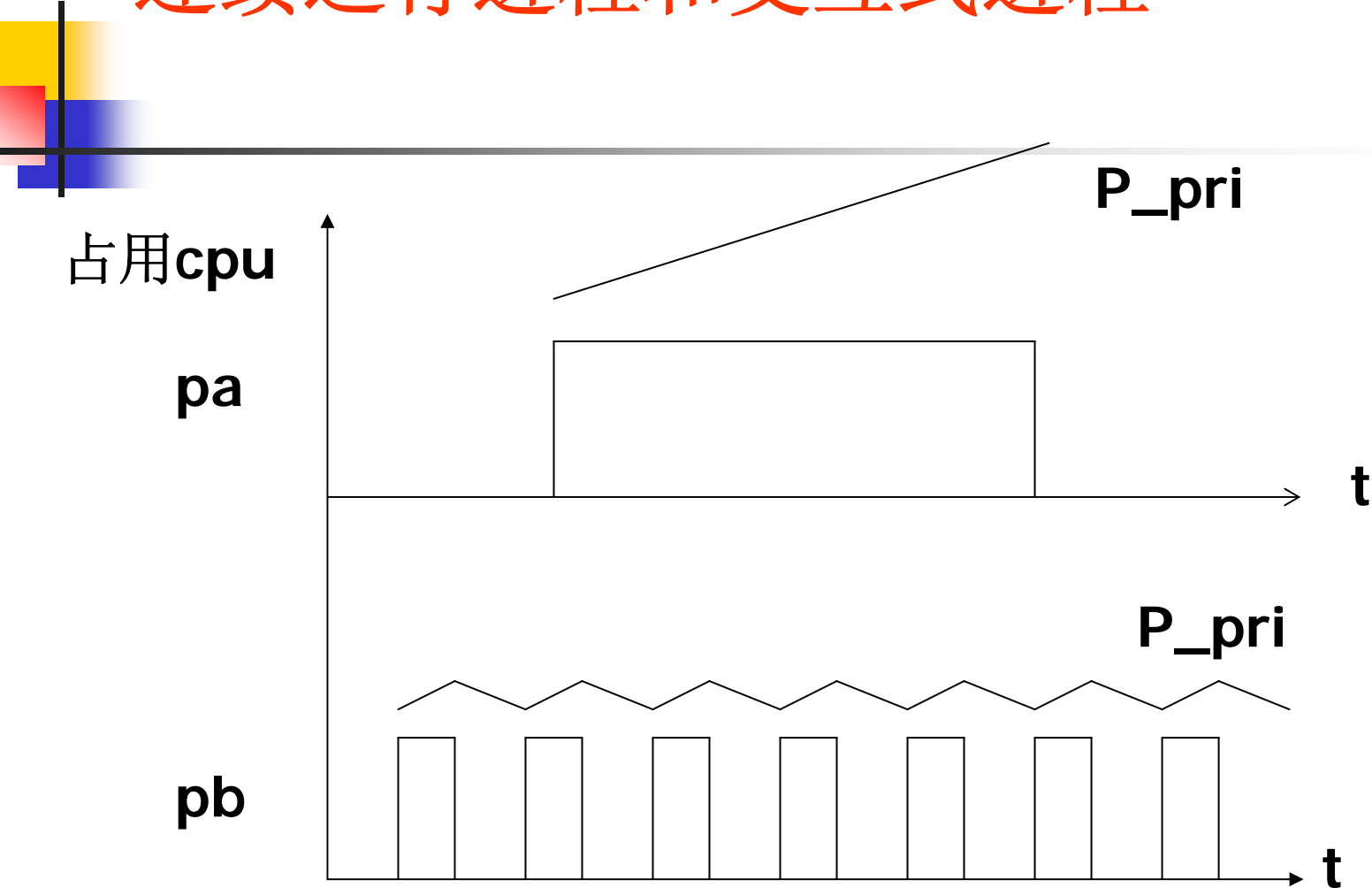
## 3. 优先数的设置

- 在核心态下运行的进程，不会被强迫剥夺处理机。只有当它因等待系统资源等原因进入睡眠状态时，系统才分配给其一个与事件相关的优先数。只有当它被唤醒之后，才以设置的优先数参与竞争处理机。
- 核心态进程根据被设置优先数的大小，可分为可中断和不可中断两类优先级。

# 4. UNIX进程优先级的排列

- **PSWP (0)** 对换进程
- **PINOD (10)** 等待磁盘I/O、管道；  
申请空闲磁盘块，空闲I节点
- **PRIBIO (20)** 等待缓冲区
- **NI\_PRILEV (25)** 有关网络机制的睡眠
- **NZERO (25)** 可中断和不可中断的分界常数
- **PMSG (27)** 因等待消息而睡眠
- **TTIPRI (28)** 等待TTY输入
- **TTOPRI (29)** 等待TTY输出
- **PWAIT (30)** 等待子进程终止
- **PSLEP (39)** 因系统调用**pause**，**sleep**而入睡
- **PUSER (60)** 核心态与用户态的分界常数
- 若干用户态进程优先级 由公式计算而得到
- **PIDLE (127)** 机器空转的最低优先级

# 连续运行进程和交互式进程



# 5. 进程切换调度的时机

进程自愿放弃处理机而引起切换调度：

- 进程完成了预定任务；
- 进程因等待某种资源进入了睡眠状态；
- 进程因同步或互斥的需要，暂停执行。
- 系统发现可能更适合占用处理机的进程，设置了强迫调度标志**runrun**：
  - 在唤醒睡眠进程时，发现该进程的优先数小于现运行进程。
  - 进程由系统调用返回用户态时，重新计算自己的优先数，发现自己的优先数上升了。
  - 在时钟中断程序中每一秒对所有优先数大于（**PUSER-NZERO**）的进程（一般原先在用户态下运行）重新计算优先数，通常总要设置**runrun**标志。



## 3.4.2 切换调度程序

---

实施进程切换调度的程序是`switch`，其主要任务是：

- 保存现运行进程的现场信息；
- 在就绪队列中选择一个在内存且优先数 `p_pri` 最小的进程，使其占用处理机，如找不到这样的进程，机器就空转等待；
- 为新选中的进程恢复现场。

# swtch第二阶段的有关程序段

```
swtch ( )
```

```
{
```

```
    register n;
```

```
    register struct proc *p, *q, *pp, *pq;
```

```
    :
```

```
    p = curproc;    /* 全局变量，记住现运行进程 */
```

```
    switch (p->p_stat) {
```

```
        case SZOMB:
```

```
            memfree (p->p_spt + p->p_nspt*  
                    128-USIZE, USIZE);
```

```
                /* 释放核心栈和user区空间 */
```

```
            sptfree (p->p_spt, p->p_nspt, 1);
```

```
                /* 释放系统页表项和用户页表页面 */
```

```
            break;
```

```
    }
```



```

loop: sp16 (); /* 提高处理机优先级，置为6级 */
runrun = 0; /* 清强迫调度标志 */
pp = NULL;
q = NULL;
n = 128; /* 最大的优先数 */
if (p = runq) do { /* 就绪队列首指针；如不空 */
    if ((p->p_flag&SLOAD)&& p->p_pri <= n) {
        /* 找在内存且优先数最小的进程 */
        pp = p; /* 当前找到的进程 */
        pq = q; /* 记住找到进程的前驱 */
        n = p->p_pri; /* 记住当前最小的优先数 */
    }
    q = p;
} while (p = p->p_link); /* 队列中下一个进程 */
p = pp; /* 最终找到的进程 */

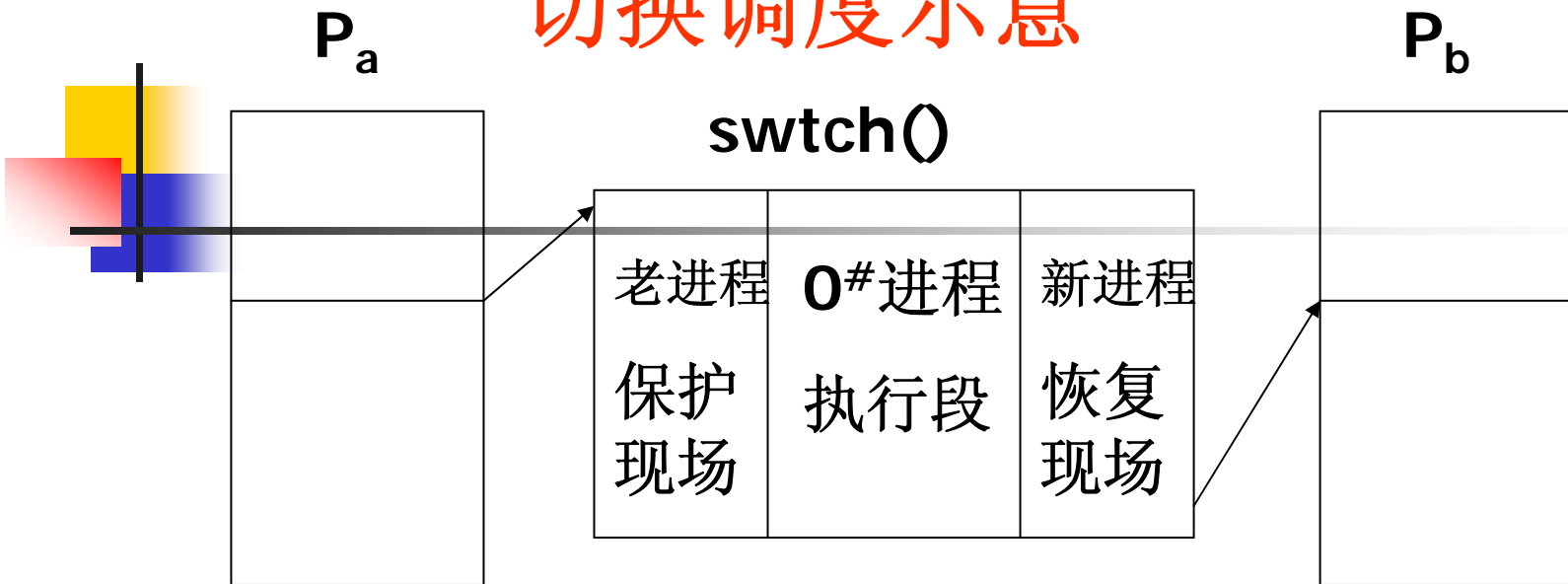
```

```

if (p == NULL) { /* 如找不到任何符合条件的进程 */
    curpri = PIDLE; /* 最大的优先数 */
    curproc = &proc[0]; /* 暂以0号为现运行进程 */
    idle (); /* 空转等待中断 */
    goto loop;
}
q = pq; /* 找到进程的前驱 */
if (q == NULL) /* 找到进程处于队首 */
    runq = p->p_link; /* 删除队首进程 */
else
    q->p_link = p->p_link; /* 删除队列中的找到进程 */
curpri = n; /* 记住现运行进程的优先数 */
curproc = p; /* 新的现运行进程 */
sp10 (); /* 将处理机优先级降为0 */
:
}

```

# 切换调度示意



老 → 0# → 新

0# → 0# → 新

老 → 0# → 老

0# → 0# → 0#

老 → 0# → 0#

## 3.4.3 UNIX的对换调度

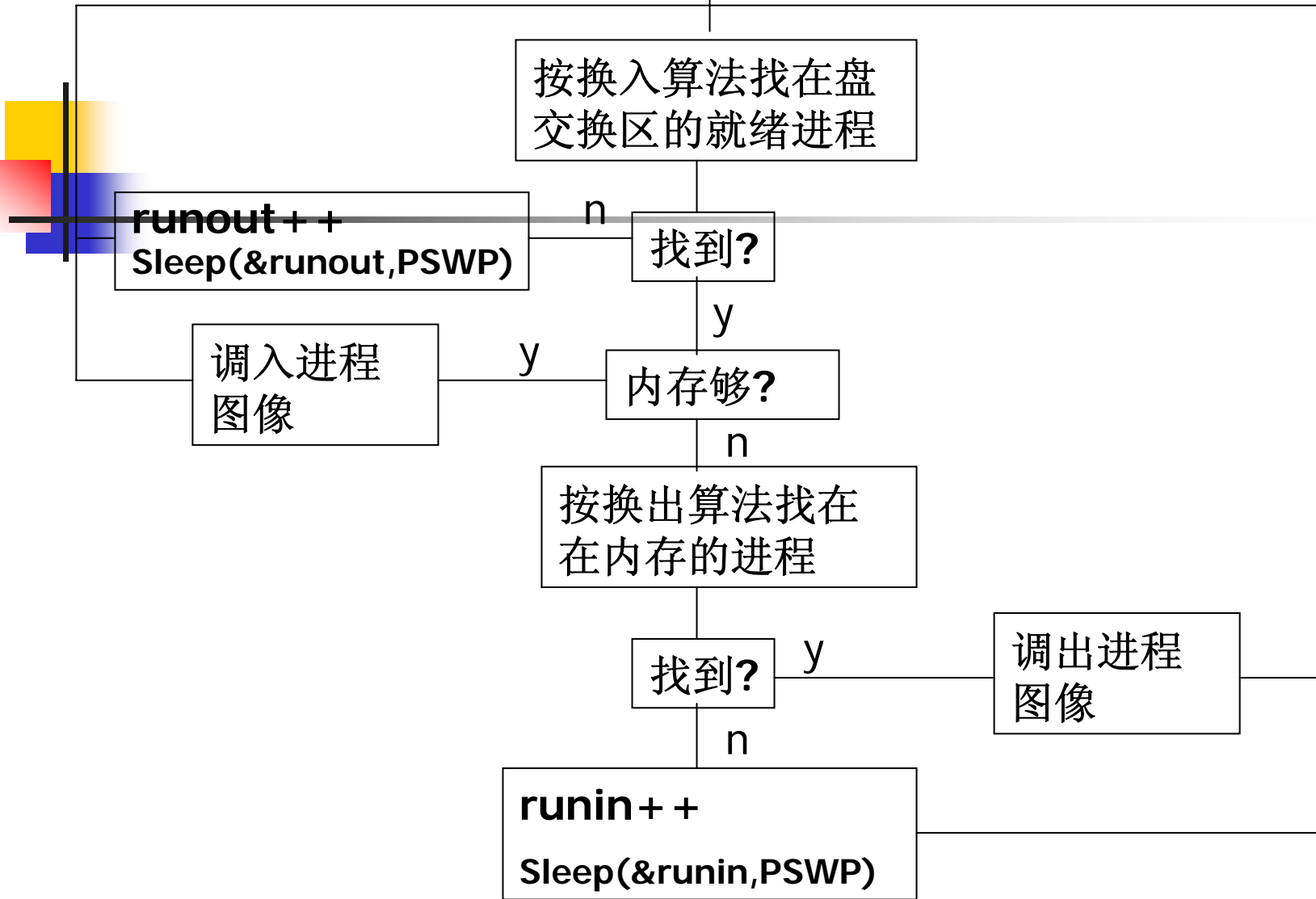
在磁盘开辟一块空间——进程图像的交换区，作为内存的逻辑扩充。**0**号进程的任务是按照换入换出算法在内存和盘交换区之间传输进程的图像。

- **换入算法**：找在盘交换区的就绪进程，按它们在外存驻留时间**p\_time**从长到短的次序逐个换入内存，直至全部调入或内存无足够空闲区为止。
  - 当**0**号进程将交换区中的就绪进程全部调入内存后，它就暂时无事可干，将全局标志变量**runout**置位，进入睡眠状态。
  - 换入过程中如发现内存无足够空间，则要将内存中的进程换出，以便为要换入的进程腾出空间。

# 进程图像的换出算法

- 先将部分图像已换出的进程从内存中完全换出，然后考虑处于睡眠或被跟踪的进程，最后是在内存驻留时间最长的进程，包括就绪状态的进程，但p\_flag中包括SSYS或SLOCK的进程不能换出。
- 考虑换出最后一类进程时，要求换入进程在交换区驻留时间应大于3秒，换出进程在内存驻留时间应大于2秒。
- 当找不到合适的换出进程时，0号进程就将全局标志runin置位，进入睡眠状态。
- runin标志置位表示盘交换区有就绪的进程要调入内存，但内存没空，且无合适的进程可换出。

# Sched()





## 3.5 进程的控制

### 3.5.1 进程的挂起

- 1. 请求系统服务或请求分配资源时得不到满足。
- 2. 同步等待I/O的完成。由中断处理程序解除该进程的阻塞状态。
- 3. 进程间互相配合共同完成一个任务时，一个进程往往要同步等待另一进程完成某一阶段的工作。
- 4. 一进程在等待某一进程发消息时，也进入了阻塞状态。



## 5. 进程将自己挂起或进程将自己某个子进程挂起

- 挂起是对进程进行控制的手段，如原进程处于就绪状态，则转变为静止就绪状态；如原进程处于阻塞状态，就转变为静止阻塞状态。
- 进程在进入阻塞状态时，要将进程的运行现场存入保护区，改变自己的进程状态，并将该进程排到对应事件的阻塞队列中，然后由调度程序重新在就绪队列中挑选一个进程投入运行。



## 3.5.2 UNIX系统中的进程睡眠和唤醒

### 1. 进程睡眠

- 核心中，进程通过调用**sleep**程序自愿地进入睡眠状态。其调用形式是：

**sleep (caddr\_t chan , int disp);**

- 参数**chan**是睡眠原因，它实际上是与睡眠事件相关的变量或数据结构的地址。如系统用一个标志变量**event**的值表示一个事件是否发生，睡眠原因就是该变量的地址 **&event**。等待系统分配它一个缓冲区或等待需缓冲的**I/O**完成，那么其睡眠原因就是相应的缓冲区始址。
- 变量**disp**的低7位是根据睡眠原因的缓急程度对睡眠进程设置的优先数，该优先数是在进程被唤醒后参与竞争处理机时起作用的。

# 睡眠等待队列

- UNIX系统V按睡眠原因chan将睡眠进程排列到NHSQUE(64)个睡眠等待队列:

```
struct proc *hsque[NHSQUE]
```

- 其中，散列算法采用:

```
sqhash(X) =
```

```
&hsque[((int)X >> 3) & (NHSQUE-1)]
```

- 其中X是睡眠原因。当正在将一个进程排到睡眠队列中时，核心要提高处理机的优先级来屏蔽所有的中断，以免在操作队列指针时因中断而引起混乱。核心还要将进程proc结构中的p\_stat改为睡眠状态，在p\_wchan中记录相应的睡眠原因。

# 核心的sleep函数

```
sleep (chan, disp)
caddr_t chan;          /* 睡眠原因 */
int disp;              /* 优先数pri */
{
    register struct proc *rp = u.u_procp;
        /* 指向proc结构 */
    register struct proc **q = sqhash (chan);
        /* 指向对应于chan的睡眠队列 */
    register s;
    s = splhi ();      /* 将处理机中断优先级IPL
        提到最高级, 原IPL保存在s中 */
    rp-> p_stat = SSLEEP; /* 设置睡眠状态 */
    rp-> p_wchan = chan; /* 设置睡眠原因 */
    rp-> p_link = *q;    /* 插到睡眠队列首 */
    *q = rp;
}
```

```

f ( (rp ->p_pri = (disp&PMASK) ) > NZERO) {
    /* NZERO为软中断信号能否影响睡眠的优先级分界值 */
    if ( rp->p_sig && issig ( ) ) {
        /* 低优睡眠前如收到不可忽略的信号 */
        rp->p_wchan = 0; /* 清睡眠原因 */
        rp->p_stat = SRUN; /* 恢复就绪状态 */
        *q = rp->p_link; /*恢复睡眠队列原先状况 */
        spl0 ( );
        goto psig; /* 转去处理信号 */
    }
    spl0 ( );
    if (runin != 0) { /* 如runin标志已设置 */
        runin = 0; /* 清该标志 */
        wakeup ( (caddr_t)&runin);
    }
    /* 唤醒睡眠原因为&runin的对换进程 (0#) */
}

```

```

switch ();          /* 切换调度, 暂不返回 */
    /* 进程睡醒, 已被切换程序选中, 从switch返回 */
    if (rp->p_sig && issig ())    /* 如其间收到信号 */
        goto psig;              /* 转去处理信号 */
    } else {                /* 高优睡眠 */
        spl0 ();
        switch ();
    }
    splx (s);              /* 恢复原中断优先级 */
    return (0 );

psig:
    splx (s);              /* 恢复原中断优先级 */
    if (disp&PCATCH)      /* 如捕获位设置 */
        return (1);      /* 返回, 去处理软中断 */
}

```

## 2. 定时睡眠

- **sleep(chan, disp)** 是内核函数，用户是不能直接调用的。**UNIX**向用户另外提供了一个系统调用：

**sleep(seconds)**

**int seconds;**

该系统调用的功能是使用调用进程定时睡眠**seconds**秒，可用于进程间的同步和定时处理事务。

### 3. 进程的唤醒

- ❖ 当引起进程阻塞的原因消除后，核心就可将阻塞进程唤醒。唤醒的方法是根据睡眠原因先算出对应的阻塞队列。
- ❖ 不同的原因可能挂在同一个队列中，在对应的 **hash** 队列中还要核查睡眠原因。
- ❖ 可能有多个进程因同一原因而阻塞，一般将这些进程全部唤醒。将它排到就绪队列中，使其有被调度的资格。
- ❖ 同一原因而唤醒的进程将先竞争处理机，从而才可获得有关的资源。如其中有一个获得等待的资源后，其他竞争相同资源的进程一旦获得处理机，还得再转入阻塞状态。

## wakeup(chan)函数的主要工作

- UNIX核心的唤醒函数首先提高处理机的优先级，以防止函数的执行被中断。
- 然后在睡眠队列hsqueue[NHSQUE]中查找所有p\_wchan等于参数chan的进程，将找到进程的p\_stat置为SRUN（就绪状态），消除原睡眠原因（p\_wchan置为0），将唤醒的进程送入就绪队列中。



```
wakeup (chan)
```

```
caddr_t chan;
```

```
{
```

```
register struct proc *p;
```

```
register struct proc **q;
```

```
register s; /* 睡眠散列队列指针 */
```

```
s = splhi ();
```

```
/* 将处理机中断优先级IPL提到最高级, 原IPL保存至s */
```

```
for (q=sqhash(chan); p=*q; ) /* 在相应的睡眠队列中 */
```

```
if (p->p_wchan==chan &&
```

```
p->p_stat==SSLEEP) {
```

```
/* 查找所有睡眠原因为chan的进程 */
```

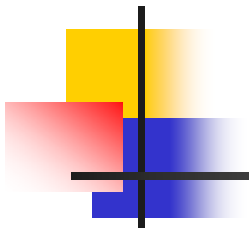
```
p->p_stat = SRUN; /* 置为运行状态 */
```

```
p->p_wchan = 0; /* 清睡眠原因 */
```

```
*q = p->p_link; /* 从睡眠队列中取出 */
```

```
p->p_link = runq; /* 放入运行队列首 */
```

```
runq = p;
```



```

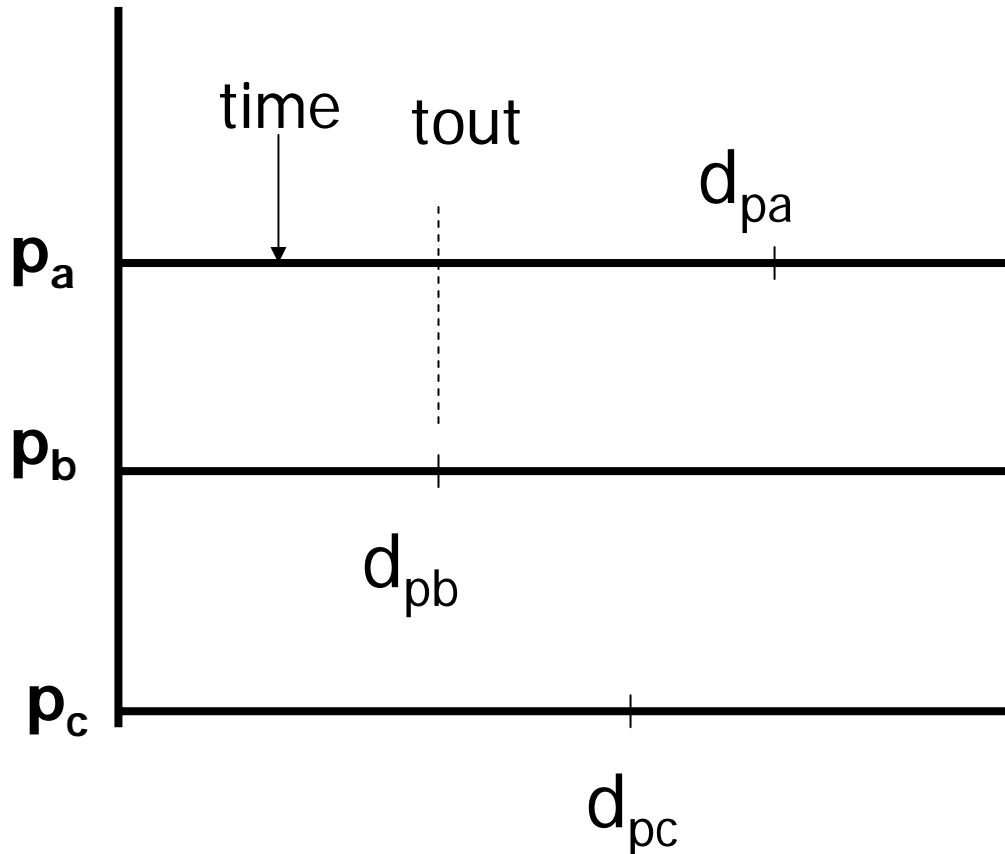
if ( ! ( p->p_flag&SLOAD ) ) {
    p->p_time = 0;
    /* 运行态进程在交换区驻留时间 */
    if ( runout > 0 )
        /* 推迟运行setrun, 以避免中断链表操作 */
        runout = -runout;
    } else if ( p->p_pri < curpri )
        /* 如唤醒进程优先级高于现运行进程优先级 */
        runrun++;          /* 设置强迫调度标志 */
    } else
        q = &p->p_link;    /* 搜索队列中下一个进程 */
if ( runout < 0 ) {      /* 如果runout标志已设置 */
    runout = 0;          /* 清runout标志 */
    setrun(&proc[0] ); /* 使0#进程转为运行状态 */
}
splx(s);                /* 恢复原中断优先级 */
}

```

# 定时睡眠的唤醒

在时钟中断中每隔一秒，比较日历时钟`time`和闹钟设置`tout`：

```
if(time == tout) wakeup (&tout)
```



## 3.5.3 进程的终止和等待终止

### 1. 进程的终止

一个进程在完成了任务后，可用系统调用**exit(int status)**终止自己。**status**左移8位后作为传送给父进程的参数。

- 进程调用**exit**时，关闭所有的打开文件，释放共享正文段、本进程的数据段、用户栈和核心栈的存储空间，暂时保留**proc**结构和盘交换区的**user**结构副本，进程的状态改为**SZOMB**状态。
- 进程终止时如有父进程因等待子进程的终止而处于睡眠状态，就唤醒父进程，最后调用**swtch**程序重新调度。
- 父进程对进入**SZOMB**的子进程作善后处理后，释放该子进程的一切资源，使其生命期最后被终止。



## 2. 父进程等待子进程终止

- 创建子进程的父进程可以通过系统调用**wait**等待它的一个子进程终止。

**pid = wait(&status)**

- 通过返回值**pid**可获得终止进程的进程标识数。**status**的高位部分为子进程传给父进程的参数，**status**的低**8**位部分为核心设置的系统调用状态码。
- 如父进程在调用**wait**时，子进程已先期终止了，那么对子进程作善后处理后，立即返回。

# 3.6进程的创建和图像改换

## 3.6.1 进程的创建

### 1. 进程创建的目的

(1) 用户登录时，核心需要创建一个进程，为用户建立交互命令的环境，接收、分析并执行用户输入的命令。

(2) 执行批处理命令时，核心要产生一个进程分析并处理批命令，对于批处理文件中的每一条命令，也要创造一个子进程来执行该命令。

(3) 当用户进程需要获得系统服务时，核心要创建一个新进程，代表用户程序的利益，完成一种系统的功能。

(4) 用户进程为了并发执行的需要，可在运行时创建一系列的进程，并互相合作，完成总任务。

## 2. 创建进程的主要步骤

- (1) 为新进程分配唯一的进程标识数。接着为新进程分配进程控制块的空间，在进程表中加入新项。
- (2) 为新进程分配进程各部分映像所需的内存空间。
- (3) 初始化进程控制块，如进程标识数、父进程标识数、程序计数器、系统栈指针、进程的状态等，根据进程的性质或缺省值初始化进程的控制信息。进程的运行状态一般初始化为就绪状态
- (4) 子进程复制父进程扩充控制块，子进程将共享父进程的全部打开文件、信号处理方式等。
- (5) 子进程复制了父进程的数据区、核心栈和用户栈，父子进程程序执行的当前位置、状态、数据区、变量的当前值都是相同的。但随着父、子进程的各自独立执行，子进程的映像将会与父进程有明显的差异。

# 如何使父子进程完成不同的任务？

由上可知，当一个子进程刚创建完成，它与父进程共享执行代码，且起始执行位置相同，数据区与栈段也相同，那么两者以后是否只能执行相同的程序段和完成相同的功能呢？如果确实这样，那么创建子进程就显然毫无意义的了。

- **UNIX**采用了在调用创建子进程的系统调用后，使父子进程具有不同的返回值，这样就可以采用判断语句，使父子进程可以执行不同的程序段，以便完成不同的任务。
- 在执行系统调用**fork**后，父进程得到的返回值是所创建子进程的标识数，而子进程的返回值为**0**。下面是一个使用系统调用**fork**的简例。

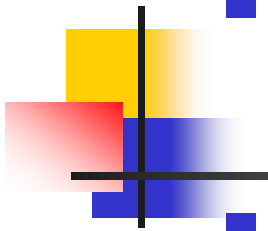


main()

```
{
int pid;
printf("Before fork\n");
while ( (pid=fork() ) == -1 );
if (pid) {
    printf ("Parent process: PID= %d\n",
            getpid());
    printf "Prodeced child's PID= %d\n", pid);
} else
    printf ("Child process.: PID= \n", getpid());
printf("Parent or child process:
        PID= %d\n",getpid());
}
```

## 3.6.2 进程图像的改换

- **fork**是一个很有特色而且非常有用的系统调用，但如果仅有它，那么**UNIX**系统的性能就会受到不可容忍的影响。
- **fork**虽然能产生子进程，将一个进程变成了两个，使它们能执行不同的程序段和完成不同的功能，形成了多进程并发运行的必要条件，但这两个进程的图像基本相同。
- 对于子进程来说，**fork**之前的图像已由父进程执行过了，子进程是不会再从头开始执行的，因而这部分图像的存在对于子进程是毫无意义的；**fork**之后父进程所执行的图像部分的存在对于子进程来说也是一个浪费。父进程也有相似的问题。

- 
- **UNIX**为了配合**fork**，还提供了进程图像改换的**exec**系列的系统调用。
  - **exec**的调用进程用一个可执行文件中的程序和数据取代当前正在运行的程序和数据，从而使主调进程的图像改换成新的图像。
  - 尽管新执行的程序与原进程的执行程序完全不同，但该调用并不形成新进程，因为原进程的**proc**结构和**user**结构并不改换，其进程标识数**p\_pid**与主调进程相同，与父进程的关系也没有改变。

# 图象改换的exec系列系统调用

- **exec**的调用进程用一个可执行文件中的程序和数据取代当前正在运行的程序和数据，从而使主调进程的图像改换成新的图像。
- 尽管新执行的程序与原进程的执行程序完全不同，但该调用并不形成新进程，因为原进程的**proc**结构和**user**结构并不改换，其进程标识数**p\_pid**与主调进程相同，与父进程的关系也没有改变。

# execl系统调用

用于装入一个带路径的可执行文件，用新程序覆盖老程序，然后运行这个“新”进程。

- 一般情况这个系统调用不返回到主调程序，仅当调用出错时（如不存在指定的可执行文件），系统才返回出错误代码。

- **execl**的调用格式是：

```
execl(pathname, cmdname, arg1,  
      arg2, ..., (char*)0);
```

```
char *pathname, *cmdname, *arg1,  
      *arg2, ..., *argn;
```

- `ret=execl (“/bin/lS”, “lS”, “-l”, (char*)0)`

# execv系统调用

`execv (pathname, argv)`

`char *pathanme, *argv[ ];`

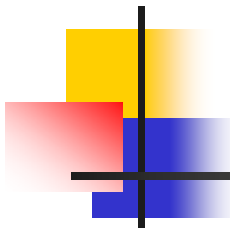
`pathname` 意义上,

`argv`是指针数组, 其中`argv [0]`指向对应于`execl`调用中的`cmdname`,

`argv [1]`指向`arg1`, ..., 指针数组最后一个元素的值是0。

- 分别与`execl`和`execv`对应的另两个系统调用是`execlp`和`execvp`, 使用这两个系统调用与上面两个的区别是主调函数的第一参数不必带路径, 如

`execlp("ls", "ls", "-l", (char*)0);`



# 执行含有元字符的命令

要使**exec**通过标准**shell**来执行，其调用格式为：

```
execl("/bin/sh", "sh", "-c"  
      commandline, (char*)0);
```

```
char *commandline;
```

参数**-c**指明要把下一个参数按整个命令行看待。

下面是 **fork**和**execl**的使用示例：

```
main()
```

```
{
```

```
int i, pid, status = 1;
```

```
① printf (“Before fork call.\n”);
```

```
③ while ((i=fork( ))==-1);
```

```
if (i) { /* 父进程 */
```

```
④ printf (“It is the parent process.\n”);
```

```
⑤ ⑧ pid =wait (&status);
```

```
⑨ printf (“Child process %d, status =%d \n”, pid, status);
```

```
} else { /* 子进程 */
```

```
⑥ printf (“It is the child process.\n”);
```

```
⑦ execl (“/bin/lis”, ”lis”, ”-l”, (char*)0); /* 映象改换 */
```

```
8) printf (“execl error.\n”); /* 映象改换失败 */
```

```
9) exit (2);
```

```
}
```

```
⑩ printf (“Parent process finish. \n”);
```

```
}
```



# 3.7 线程

## 3.7.1 进程和线程

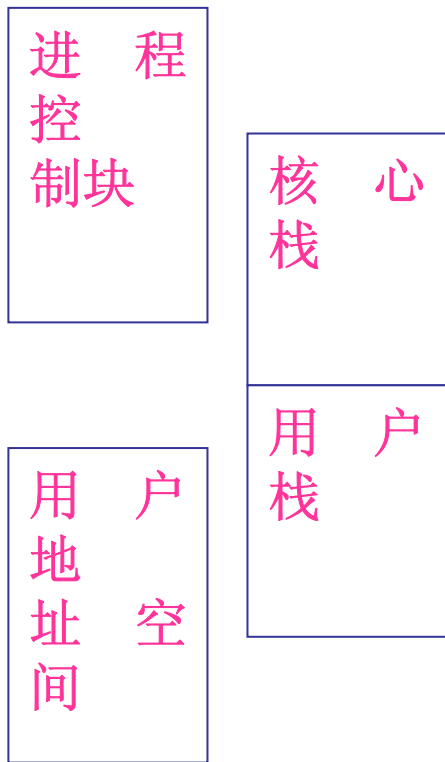
- ❖ 在进程的概述一节中谈到进程包含了下列两个特征：
  - ❖ **资源拥有单位**：进程的虚址空间用于驻留进程的图象，进程可以分配到主存和控制其它的系统资源。
  - ❖ **调度单位**：进程是可以并发执行的独立单元，是操作系统进行调度的一个实体。
- ❖ 大多数操作系统把这两个特征看成不可分割的，是一个进程的基本特征，但实际上这两个特征是独立的。
- ❖ 近年来开发的一些操作系统区别了这两个特征，并创建了一种新的调度和执行的实体单元——**线程 (thread)**，也可称轻量级进程，而原先的资源拥有单位通常还是称为进程或任务。

## 3.7.2 多线程

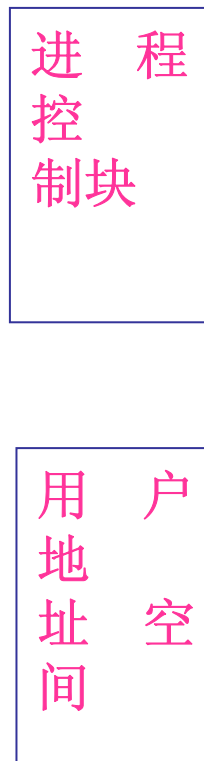
- 多线程指的是操作系统支持在单个进程中执行多个线程的能力。
- 进程被定义为保护单位和资源分配单位，
- 在一个进程内部可以有一至多个线程，每一个线程具有如下特征：
  - 线程的执行状态（运行、就绪等）。
  - 线程上下文环境。可以把线程看成是进程内一个独立的程序计数器的运转。
  - 执行栈。
  - 存取所属进程内的主存和其它资源，在本进程的范围内与所有线程共享这些资源。

# 线程和进程的区别

## 单线程进程模型



## 多线程进程模型



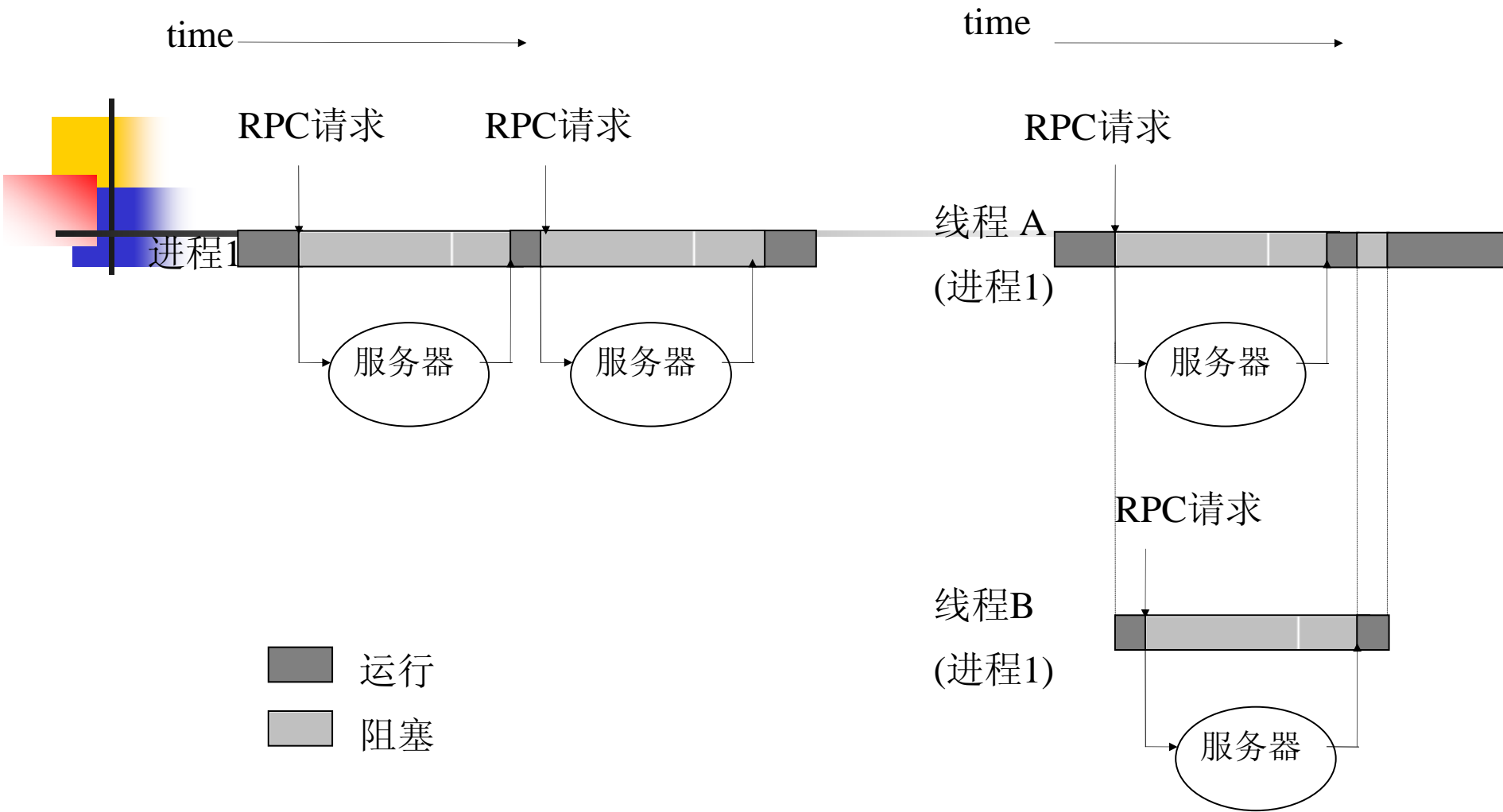
# 线程带来的关键的好处

- 提高了操作系统的性能。在一个现存的进程中创建一个新的线程的时间远小于创建一个新的进程。
- 终止一个线程的时间也较小。
- 在同一个进程内部两个线程的切换开销比进程之间的切换开销小得多。那么用一组线程执行而不是用一组分开的进程执行，其效率就要高得多。
- 服务器是使用线程来提高效率的很好例子。当一个文件服务请求到达时，由于服务器能处理很多请求，在一个短时间内，很多线程将被产生和销毁。



# 线程的并发执行

- 图3-8显示了执行两个对不同的主机的远程过程调用（**RPCs**）以获得的组合的结果。
- 在单线程程序，这两个结果是顺序获得的，故程序要依次等待每一个服务器的响应。
- 如用各自的线程执行**RPC**以获得调用结果的方法重写程序，性能就获得实质性的提高。
- 程序是并行地等待两个主机的响应结果。



(a) 使用单线程的RPC

(b) 对每个服务器使用一个线程的RPC

图3-8 使用线程的远程过程调用



---

## 复习题

p75 1,3, 6, 7, 9,11,17

## 习题

p75 12



---

谢谢

