華中科技大學
计算机科学与技术学院
School of Computer Science & Technology, HUST

操作系统原理

第四章 进程及进程管理

授课人：郑 然

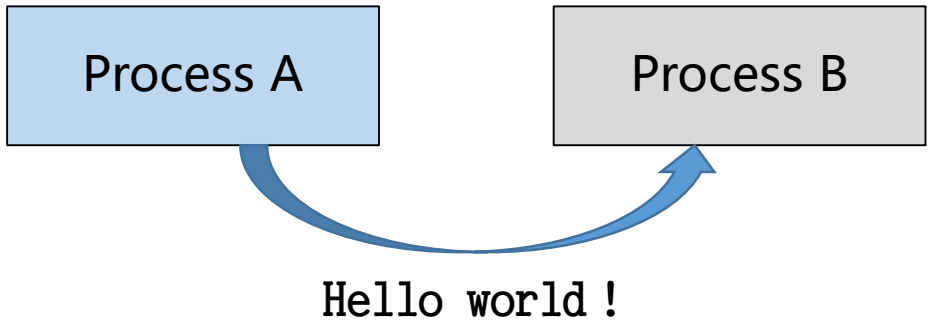
1. 进程通信的概念

协作进程可能影响另一个进程的执行或被另一个进程执行影响。

协作进程需要一种进程间通信的机制来允许进程间交换数据与信息。

信号灯及P、V操作实现的是进程之间的低级通信，只能传递简单的信号，不能传递交换大量信息。

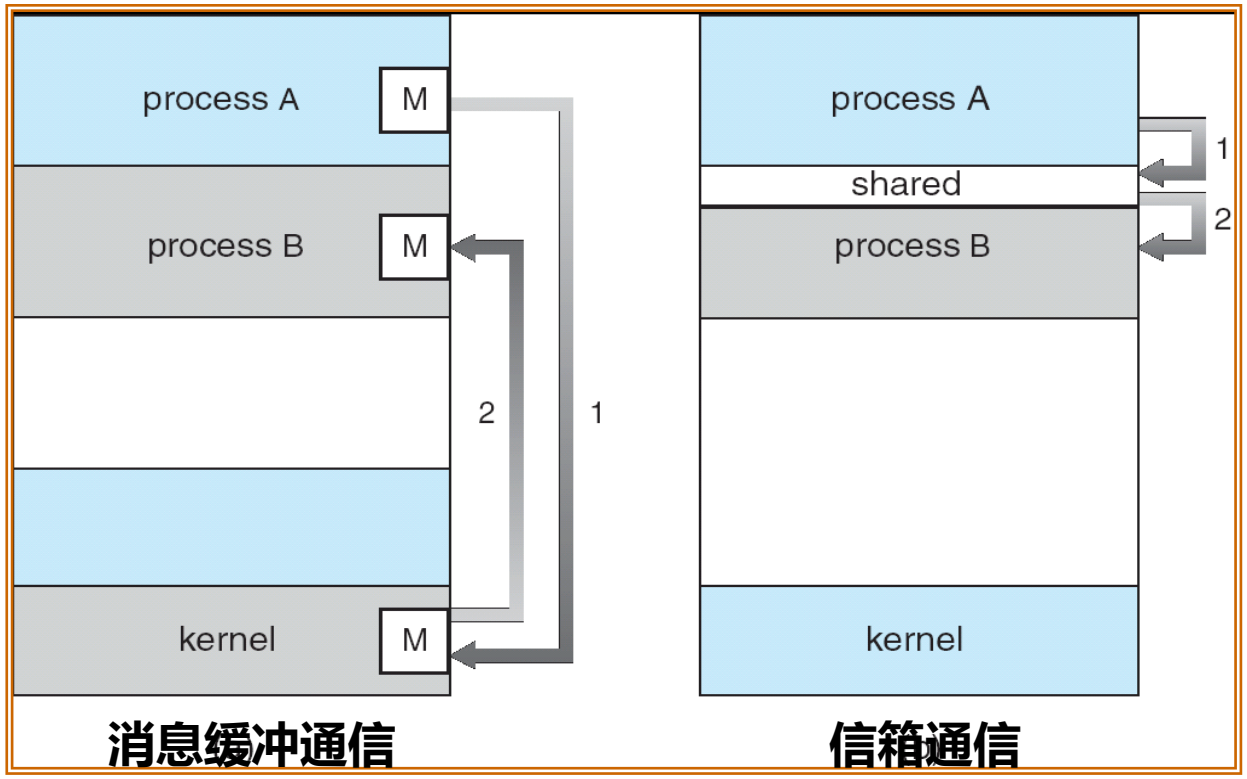
进程通信是指进程之间直接以较高的效率传递较多数据的信息交互方式。



2. 进程通信方式

以消息为单位：

- 接收方和发送方之间有明确的协议和消息格式
- 包括消息缓冲、发送原语和接收原语



共享存储区：

- 需要定义信箱结构，还包括消息发送和接收功能模块，提供发送原语和接收原语
- 使用的信箱可以位于用户空间中，是接收进程地址空间的一部分；也可以放置在操作系统的空间中

信号通信机制（只能发送单个信号而不能传递数据）
管道通信机制（只能在进程家族内使用）
消息传递通信机制
信号量通信机制
共享主存通信机制

提纲

- 进程的引入
- 进程概念
- 进程控制
- 进程的相互制约关系
- 进程同步机构
- 进程互斥与同步的实现
- **线程**
- 进程调度

1. 引入线程的原因

例：编写一个MP3播放软件。核心功能模块有三个：

（1）从MP3音频文件当中读取数据 (Read)

（2）对数据进行解压缩 (Decompress)

（3）把解压缩后的音频数据播放出来 (Play)

单进程实现

```
main( ) {
    while(TRUE) {
        Read( );
        Decompress( );
        Play( );
    }
}
```

Read( ); → I/O

Decompress( ); → CPU

问题：

1. 播放的效果?
2. 资源的使用效率?

多进程的实现方法

程序1

```
main( ) {
   while(TRUE) {
      Read( );
   }
}
```

程序2

```
main( ) {
   while(TRUE) {
      Decompress( );
   }
}
```

程序3

```
main( ) {
   while(TRUE)  {
      Play( );
   }
}
```

存在的问题：

1. 进程之间如何通信，共享数据?
2. 进程是一个独立运行的活动单位，也是竞争系统资源的基本单位，导致进程的创建、撤销和切换开销极大
3. 系统中的进程数不宜过多，进程切换的频率也不宜过高，限制了并发程度的进一步提高

多线程的解决思路

在进程内部增加一类实体，满足以下特性：

（1）实体之间可以并发执行

（2）实体之间共享相同的地址空间

这种实体就是线程（Thread）

引入线程的动机

- 分离进程的功能：作为独立运行的活动单位，但不作为竞争系统资源的基本单位（避免频繁的切换）
- 减少程序并发执行时的时空开销，使得并发粒度更细、并发性更好

2. 什么是线程

(1) 线程定义

线程是比进程更小的活动单位，它是进程中的一个执行路径。

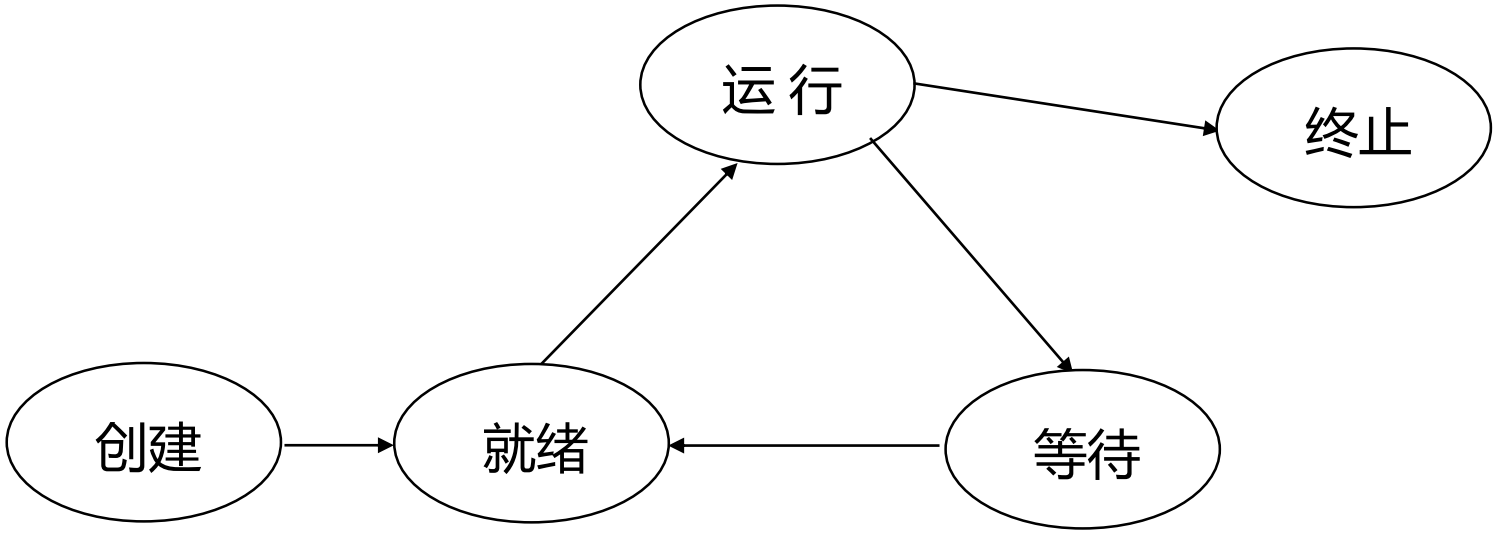
(2) 线程可以这样来描述

- 进程中的一条执行路径；
- 它有自己私用的堆栈和处理机执行环境 ；
- 它与父进程共享分配给父进程的主存；
- 它是单个进程所创建的许多个同时存在的线程中的一个。

3. 线程的特点

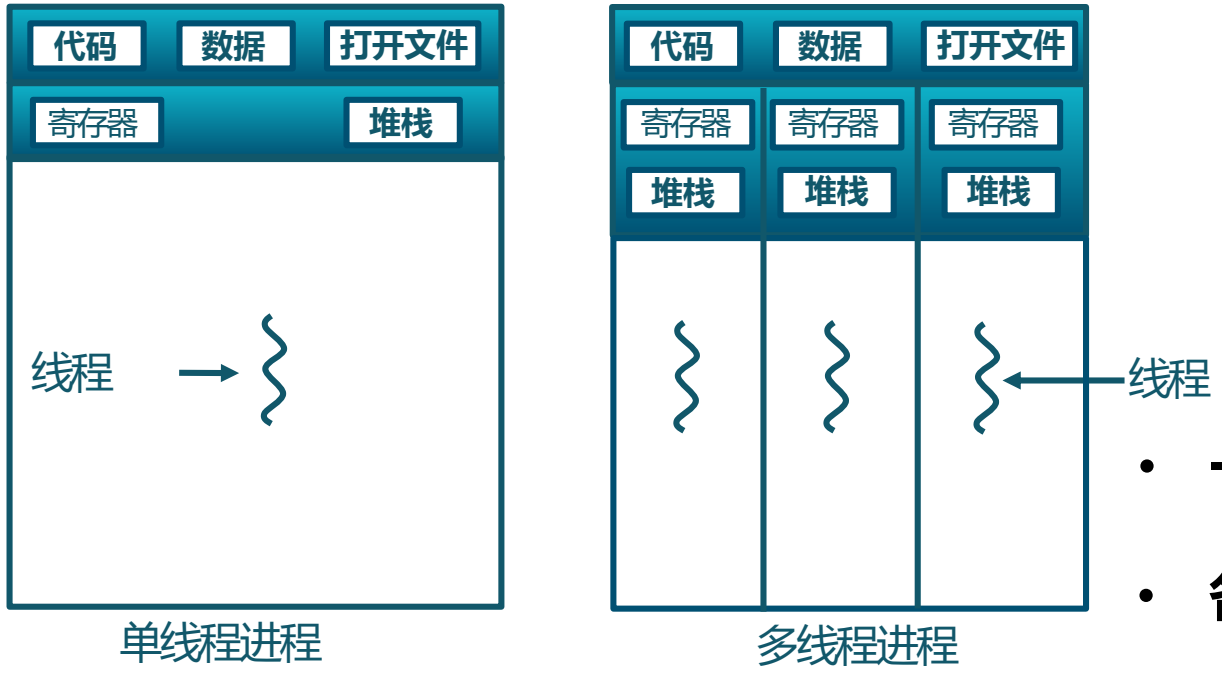
- 线程是比进程更小的活动单位，它是进程中的一个执行路径。 创建一个线程比创建一个进程开销要小得多。
- 实现线程间通信十分方便，因为一个进程创建的多个线程可以共享地址区域和数据。
- 线程是一个动态的概念。
- 在进程内创建多线程，可以提高系统的并行处理能力，加快进程的处理速度。

4. 线程的状态变迁



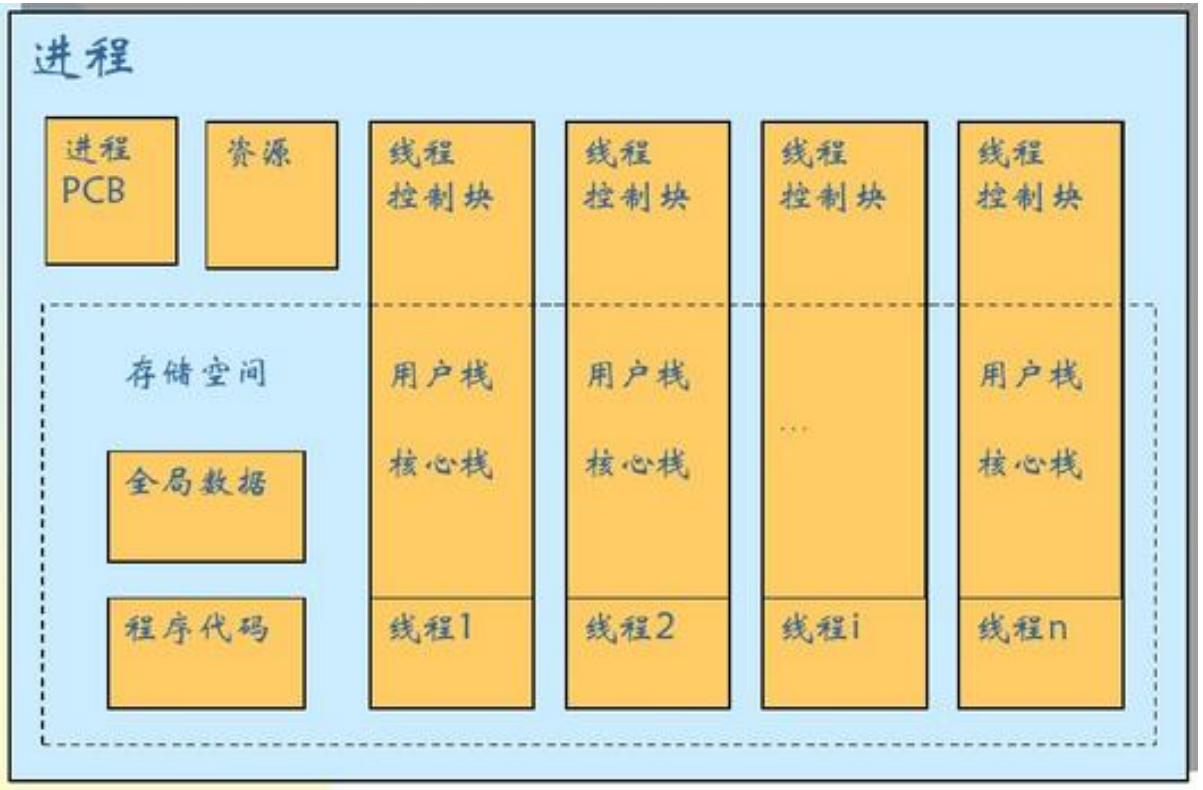
线程的状态变迁图

5. 进程和线程的关系



- 一个进程中可以同时存在多个线程
- 各个线程之间可以并发地执行
- 各个线程之间可以共享地址空间和文件等资源
- 一个线程崩溃，会导致所属进程的所有线程崩溃

多线程结构的进程



- 创建一个线程比创建进程开销要小得多（线程终止亦同）
- 两个线程的切换开销也小得多
- 同一进程内的多个线程共享内存和数据，相互的通信无须调用内核，十分方便

6. 线程与进程的比较

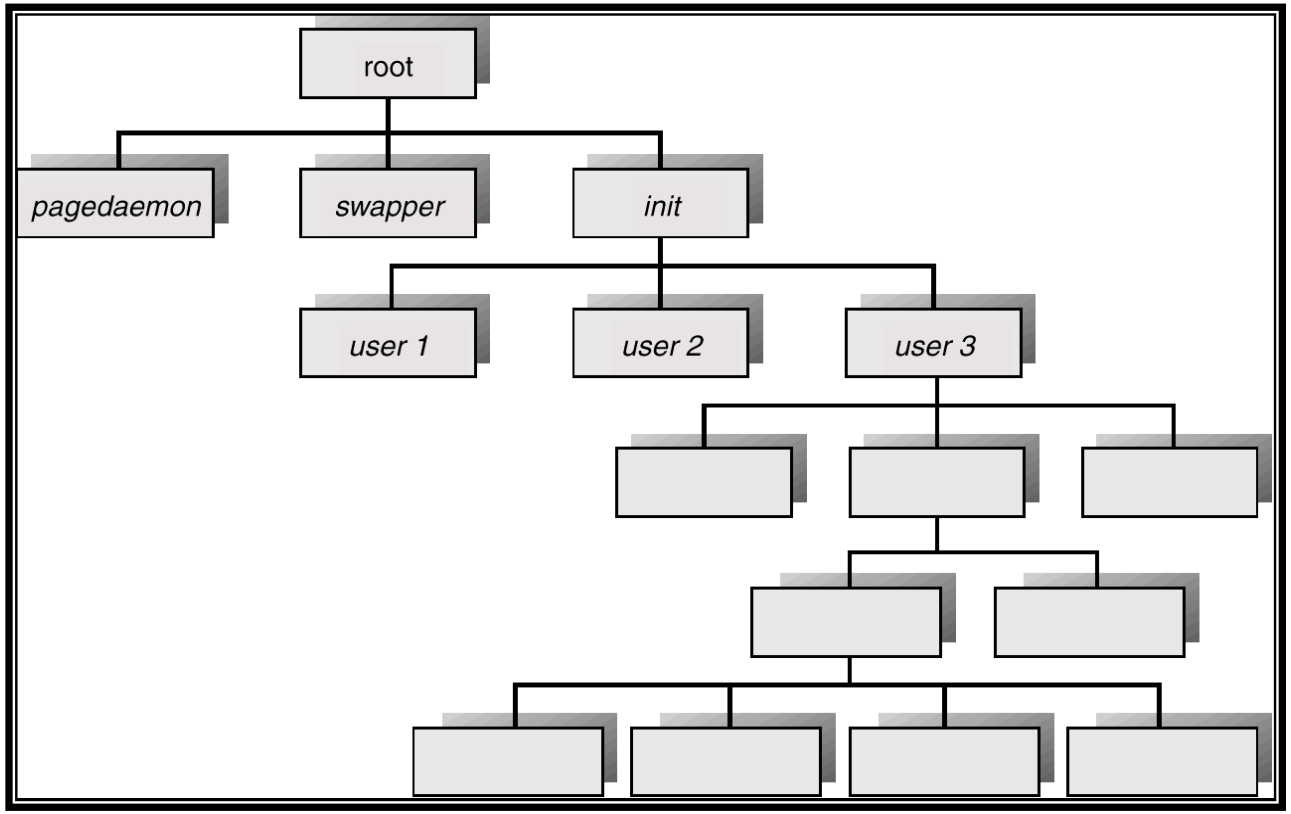
线程通常又称为轻量级进程

在引入线程的操作系统中：

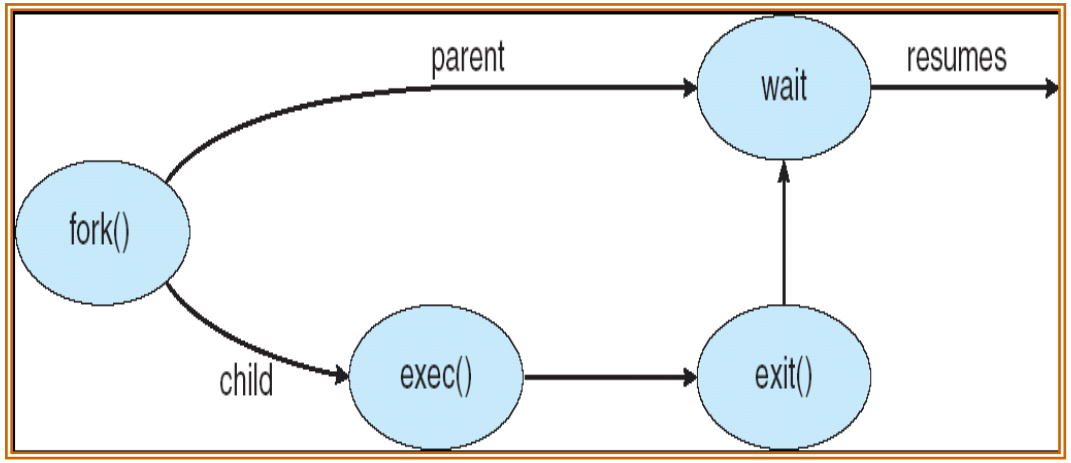
- 每个进程都拥有一个或多个线程
- 线程是调度和分派的基本单位，进程是拥有资源的基本单位
- 同一进程内的线程切换不会产生进程切换，一个进程内的线程切换到另一个进程内的线程会引起进程切换
- 进程可以并发执行，同一进程内的线程也可以并发执行
- 线程一般不拥有系统资源，但可以访问隶属于进程的资源（进程内的所有线程共享使用）
- 线程创建、撤销和切换开销远小于进程，同一进程内的各线程之间共享内存和文件资源，可不通过内核进行直接通信

进程及进程管理——操作系统的并发机制实例

UNIX/LINUX系统



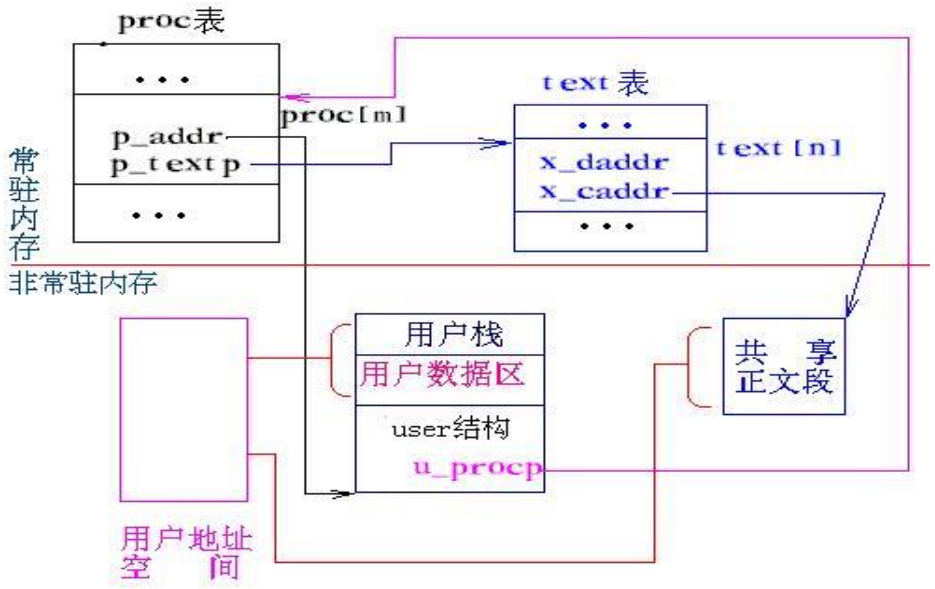
UNIX系统中的进程树



进程及进程管理——操作系统的并发机制实例

Unix进程由proc结构、数据段和正文段三部分组成,合称进程映像

- **基本进程控制块：proc结构，存放进程的最基本的管理和控制信息**
- **正文段 (仅共享时存在)**
- **数据段 (又称上下文，context)**
  - 数据区和用户栈区
  - 共享正文段(text表，正文控制表)
  - U区(进程数据区)：核心栈、User结构



Unix的进程控制块(PCB)

- proc结构　proc.h (基本的)
  - 进程控制和管理的最基本的信息
  - 常驻内存
- user结构　user.h (扩充的)
  - 进程运行时才用到的数据和状态信息
  - 通常和数据段一起放在磁盘上，需要时调入
- 正文控制表　text.h
  - 管理共享正文区
  - 内存无共享进程映像时调出

1. 创建进程及应用实例

(1) 调用形式

pid=fork();

功能：创建一个子进程，被创建的子进程是父进程的进程映像的一个副本 (除proc结构外)，在UNIX系统中，除了0#进程外，其它进程都是通过调用进程创建系统调用创建的。

(2) 系统调用 fork 完成的操作

UNIX/Linux系统的核心为系统调用fork 完成下列操作：

① 为新进程分配一个新的pcb结构；

② 为子进程赋一个唯一的进程标识号 (PID)；

③ 做一个父进程上下文的逻辑副本。由于进程的正文区 (代码段) 可被几个进程所共享，所以核心只要增加某个正文区的引用数即可，而不是真的将该区拷贝到一个新的内存物理区。这就意味着父子进程将执行相同的代码。数据段和堆栈段属于进程的私有数据，需要拷贝到新的内存区中。

④ 增加与该进程相关联的文件表和索引节点表的引用数。这就意味着父进程打开的文件子进程可以继续使用。

⑤ 对父进程返回子进程的进程号，对子进程返回零。

(3) 执行这个程序可能的结果

```
例：  main()
      {int x;
       while((x=fork())= = - 1);
       if(x= =0)
            printf("a");
         else
            printf("b");

         printf("c");
       }
```

结果？

pid=fork();

pid < 0： 进程创建失败

pid > 0：子进程id，此时为父进程代码

pid==0：此时为子进程代码

getpid()、getppid()查询

(3) 执行这个程序可能的结果

```
main()
{
        pid_t p1;
        pid_t t1;

      p1=fork();
        if (p1 == 0)
      {
            puts("sub1 created\n");
        }
      else       //main
      {
          t1=waitpid(p1,&status,0);
        }
}
```

父进程p1>0

子进程p1==0

```
main()
{       pid_t p1;
        pid_t t1;

      p1=fork();
        if (p1 == 0)
        {    puts("sub1 created\n");
        }
      else
      {  t1=waitpid(p1,&status,0);
      }
}
```

父进程实际执行的程序段

```
main()
{       pid_t p1;
        pid_t t1;

      p1=fork();
      if (p1 == 0)
      {  puts("sub1 created\n");
      }
      else       //main
      {   t1=waitpid(p1,&status,0);
        }
}
```

子进程实际执行的程序段

进程及进程管理——操作系统的并发机制实例

```
例：  main()
      {int x;
       while((x=fork())= = - 1);
       if(x= =0)
            printf(“a”);
       else
            printf(“b”);

       printf(“c”);
      }
```

结果？

pid=fork();

pid < 0 ： 进程创建失败

pid > 0：子进程id，此时为父进程代码

pid==0：此时为子进程代码

getpid()、getppid()查询

abcc?
bcac?
abcc?
acbc?
cabc?

进程及进程管理——操作系统的并发机制实例

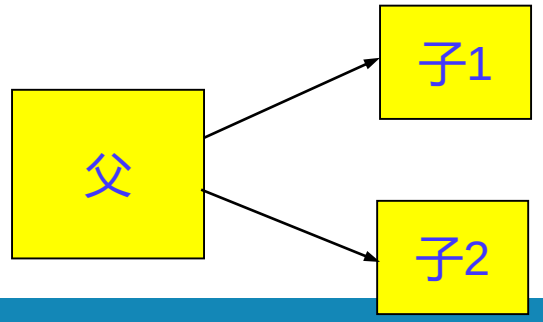

```
main( ) {
   int child, i=2;
   if ((child=fork()) == –1) {
      printf("fork error. ");
      exit();      }
   if(child==0) {
     i=i+3;
     printf(“i=%d\n”,i);     }
   i=i+5;
   printf(“i=%d\n”,i);
}
```

插入else呢?

1.fork error
2 . i=5
    i=10
    i=7
3. i=7
    i=5
    i=10
4. i=5
    i=7
    i=10

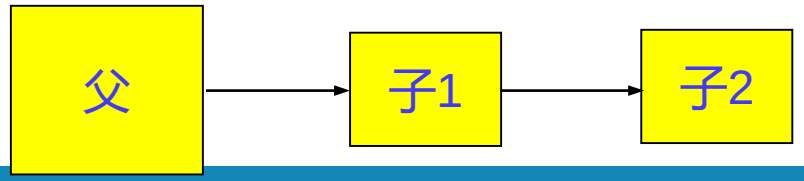
进程及进程管理——操作系统的并发机制实例

```
main( )
{
  if (fork()==0)
    { 子1的代码段 }
  else {
   if (fork()==0)
    { 子2的代码段 }
    else
    { 父代码段 }
  }
}
```



去掉这个else
谁执行这一段?

```
main( ) {
  if (fork()==0) {
    子1的代码段;
    if (fork()==0)
      {子2的代码段}
    else
      { 子1的代码段}
    }
 else
    {父代码段}
}
```



(4) exec-执行文件，启动新程序运行

① 更换进程执行代码，更换正文段，数据段

② 调用格式：exec (文件名，参数表，环境变量表)

③ 例　　execlp(“./max”,15,18,10,0);

　　execvp(“max”,argp)

```
main()
{
   if(fork()==0)
      {   printf(“a”);
          execlp(“file1”,0);
          printf(“b”);
      }
   printf(“c”);
}
```

```
file1:
main()
{
    printf(“d”);
}
```

acd?
cad?
adc?
abdc?
adbcc?

2. 创建线程及应用实例

(1) 调用形式

pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

pthread_join(pthread_t th, void **thread_retrun);

作用：挂起当前线程直到由参数th指定的线程被终止为止。

(2) 程序范例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void thread(void)
{
 int i;
 for(i=0;i<3;i++)
 printf("This is a
    pthread.\n");
}
int main(void)
{
pthread_tid;
int i,ret;

ret=pthread_create(&id,NULL,(void *)
thread,NULL);
if(ret!=0){
    printf ("Create pthread error!\n");
    exit (1);
}
for(i=0;i<3;i++)
   printf("This is the main process.\n");

pthread_join(id,NULL);
return (0);
}
```

(3) 运行结果?

3. 等待进程、线程的终止及其应用

(1) 等待进程终止

wait( ); waitpid( );

① wait() 语法格式

pid=wait(status);

wait( )函数使父进程暂停执行(阻塞自己），直到它的一个子进程结束为止，该函数的返回值是终止运行的子进程的PID。参数status所指向的变量存放子进程的退出码，即从子进程的main函数返回的值或子进程中exit( )函数的参数。如果status不是一个空指针，状态信息将被写入它指向的变量。

② waitpid() 语法格式

waitpid(pid_t pid, int * status, int options)

用于等待某个特定进程结束。参数pid指明要等待的子进程的PID，参数status的含义与wait()函数中的status相同。

pid>0：等待进程ID为pid的子进程

pid=-1：等待任一子进程退出，等同wait函数

pid=0：等待与调用者进程组ID相同的任意子进程

pid<-1：等待进程组ID与pid绝对值相等的任意子进程

options：WNOHANG(不阻塞) / WUNTRACED(子进程暂停则返回)

(2) wait--等待子进程结束 与 exit---终止进程的使用方法

① 例1

```
main( ) {
  int  n;
   ….
   if (fork( )==0) {
        printf(“a”);
        exit(0);

      }
  wait(&n);
  printf(“b”);
}
```

进程及进程管理——操作系统的并发机制实例

② 例2

```
main( )
{    int p1,p2,p3,p4,p5,pp1,pp2;

    printf("程序开始执行");
    if ((p1=fork( )== 0） {
       printf("进程proc1执行");
       exit(1);
    } else if ((p2=fork( ) )== 0) {
            printf("进程proc2执行");
            exit(1);
    }
    pp1=wait(&pp1);  /* 等待，直到子进程终止 */
    pp2=wait(&pp2);  /* 等待，直到子进程终止 */
    if ((p3=fork( ))== 0） {
      printf("进程proc3执行");
      } else if ((p4=fork( ))== 0) {
         printf("进程proc4执行");
      } else if ((p5=fork( ))== 0) {
         printf("进程proc5执行");
         exit(1);
      }
    printf("整个程序终止");
    exit(0);
}
```

ii **试回答如下问题**

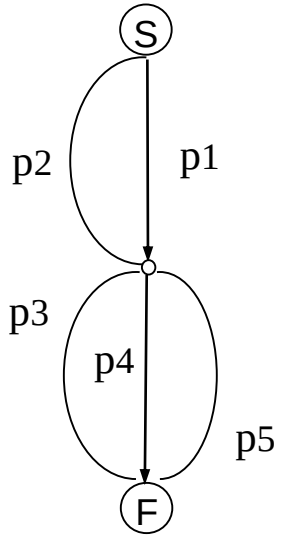
a. 画出描述子进程执行先后次序的进程流图。(各进程分别用其对应的函数名或包含其进程号的符号名标识)。

b. 这个程序执行时最多可能有几个进程同时存在？同时存在的进程数最多时分别是哪几个进程?

c. 程序执行时，“整个程序终止”被输出几次？分别是哪些进程输出的?

iii 问题答案

a.



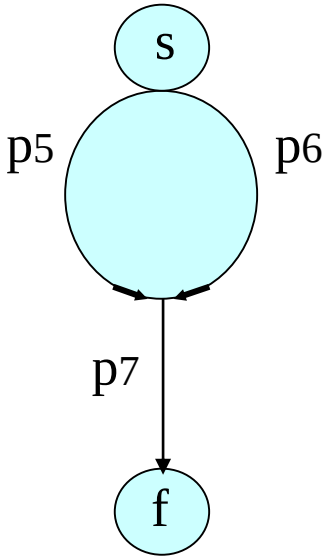
子进程进程执行的流图图

b. 最多4个进程同时存在，分别是main、p3、p4、p5 。

c. 3次，main、p3、p4 。

进程及进程管理——操作系统的并发机制实例

③ 应用实例1



3个合作进程
的进程流图

```
#include  <sys/types.h>
#include  <sys/wait.h>
int  main ( )   {
    pid_t  pid;
    int status;
    pid=fork( );
    if  (pid==0) {
        p6( );  exit( );
        } else{
            pid=fork( );
            if  (pid==0 )  {
                p5( );  exit( );
            }
        }
        wait(&status);
        wait(&status);
        p7( );
  }
```

④ 应用实例2

```
#include <stdio.h>
#include <pthread.h>
int A;
void subp1( )  {
    printf(“A in thread is %d\n”,A);
    A = 10;
}
main( )
{
    pthread_t  p1;
    int pid;
    A = 0;
    pid = fork();
    if (pid==0) {
        printf("A in son process is %d\n",A);
        A=100;
        exit(0);
    }
    wait( );
    pthread_create(&p1,NULL,subp1,NULL);
    pthread_join(p1,NULL);
    printf(“A in father process is %d\n”,A);
}
```

运行结果?

A in son process is 0
A in thread is 0
A in father process is 10

4. 信号量及其使用方法

Linux信号量函数在通用的信号量数组上进行操作，而不是在一个单一的信号量上进行操作。这些系统调用主要包括：semget、semop和semctl。

(1) 信号量的创建

① 功能

创建一个新的信号量或是获得一个已存在的信号量键值。

② 原型

int semget(key_t key, int num_sems, int sem_flags)键值

i 参数key是一个用来允许多个进程访问相同信号量的整数值，它们通过相同的key值来调用semget。

ii 参数num_sems参数是所需要的信号量数目。semget创建的是一个信号量数组，数组元素的个数即为num_sems。

iii sem_flags参数是一个标记集合，与open函数的标记十分类似。低九位是信号的权限，其作用与文件权限类似。另外，这些标记可以与IPC_CREAT进行或操作来创建新的信号量。一般用：IPC_CREAT | 0666

(2) 信号量的控制

原型

int semctl(int sem_id, int sem_num, int command, ...)

i 参数sem_id，是由semget所获得的信号量标识符。

ii 参数sem_num参数是信号量数组元素的下标，即指定对第几个信号量进行控制。

iii command参数是要执行的动作，有多个不同的ommand值可以用于semctl。常用的两个command值为:

SETVAL: 用于为信号量赋初值，其值通过第四个参数指定。

IPC_RMID：当信号量不再需要时用于删除一个信号量标识。

iv 如果有第四个参数，则是union semun，该联合定义如下：

```
union semun {        int val;
                     struct semid_ds *buf;
                     unsigned short *array;
}
```

(3) 信号量的操作

① 原型

int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops)

i 参数sem_id，是由semget函数所返回的信号量标识符。

ii 参数sem_ops是一个指向结构数组的指针，该结构定义如下：

iii num_sem_ops 操作次数，一般为1

```
struct sembuf {
        short sem_num;  //数组下标
        short sem_op;  //操作，-1或+1
        short sem_flg; //0
}
```

② P操作

```
void P(int semid, int index)
{      struct sembuf sem;
       sem.sem_num = index;
       sem.sem_op = -1;
       sem.sem_flg = 0;//：操作标记：0或IPC_NOWAIT等
       semop(semid,&sem,1);  //1:表示执行命令的个数
       return;
}
```

③ V操作

```
void V(int semid, int index)
  {    struct sembuf sem;
       sem.sem_num = index;
       sem.sem_op =  1;
       sem.sem_flg = 0;
      semop(semid,&sem,1);
      return;
}
```

5. 共享内存

(1) 功能

共享内存允许两个或更多进程访问同一块内存，就如同malloc() 函数向不同进程返回了指向同一个物理内存区域的指针。当一个进程改变了这块地址中的内容的时候，其它进程都会察觉到这个更改。

(2) 共享内存创建

int shmget(key_t key,int size,int shmflg)

其中:

① key: 键值，多个需要使用此共享内存的进程用相同的key来创建

② shmflg: IPC_CREAT|0666

(3) 共享内存绑定

int shmat (int shmid, char *shmaddr, int shmflg)

其中：

① shmid：共享内存句柄，shmget调用的返回值；

② shmaddr：一般用NUL;

③ shmflg： SHM_R|SHM_W

S = (char *)shmat(shmid1,NULL,SHM_R|SHM_W)

一旦绑定，对共享内存的操作即转化为对局部变量S的操作。

(4) 共享内存的释放

系统调用格式：int shmctl(shmid,cmd,buf);

其中：

shmid：共享内存句柄，shmget调用的返回值；

cmd：操作命令shmctl(shmid,IPC_RMID,0)

提纲

- 进程的引入
- 进程概念
- 进程控制
- 进程的相互制约关系
- 进程同步机构
- 进程互斥与同步的实现
- 线程
- 进程调度

1. 调度/分派结构

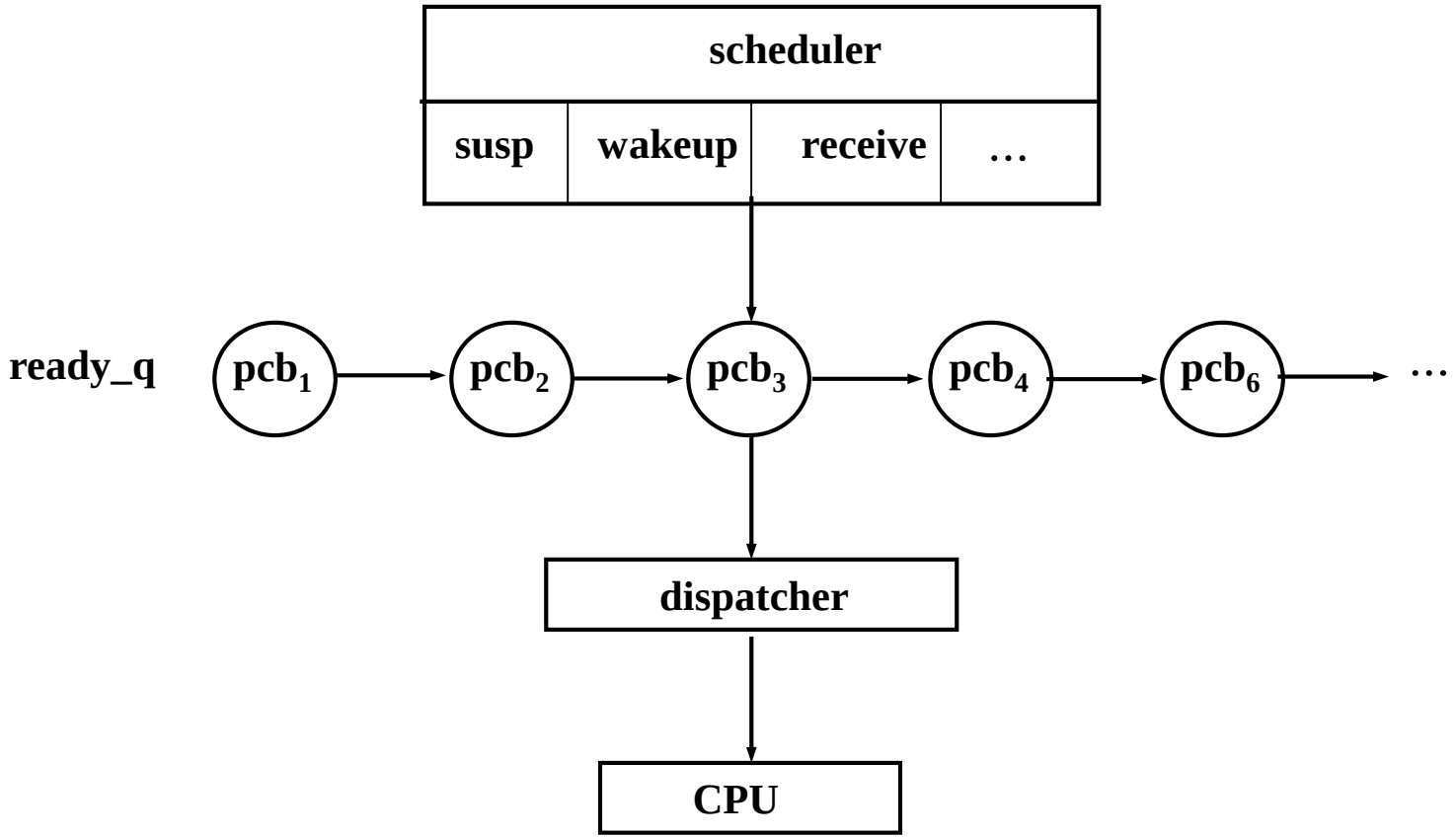
(1) 调度

在众多处于就绪状态的进程中，按一定的原则选择一个进程。

(2) 分派

当处理机空闲时，移出就绪队列中第一个进程，并赋予它使用处理机的权利。

(3) 调度分派结构图



调度/分派结构示意图

2. 进程调度的功能

(1) 进程管理的数据结构

(2) 决定调度策略

① 优先调度

就绪队列按进程优先级高低排序

② 先来先服务

就绪队列按进程来到的先后次序排序

(3) 实施处理机的分配和回收

3. 进程调度的方式

(1) 什么是调度方式

当一进程正在处理机上执行时，若有某个更为“重要而紧迫”的进程需要运行，系统如何分配处理机。

(2) 非剥夺方式

当“重要而紧迫”的进程来到时，让正在执行的进程继续执行，直到该进程完成或发生某事件而进入“完成”或“阻塞”状态时，才把处理机分配给“重要而紧迫”的进程。

优点：实现简单，系统开销小
缺点：难以满足紧急任务的要求

(3) 剥夺方式

当“重要而紧迫”的进程来到时，便暂停正在执行的进程，立即把处理机分配给优先级更高的进程。

优点：及时响应紧急任务
缺点：增加了系统开销

实现中还可采用选择可抢占策略。

4. 进程调度算法

(1) 进程优先数调度算法

① 什么是进程优先数调度算法

预先确定各进程的优先数，系统把处理机的使用权赋予就绪队列中具备最高优先权 (优先数和一定的优先级相对应)的就绪进程。

采用这种调度算法的关键是如何确定进程的优先数、进程的优先数确定之后是固定的，还是随着该进程运行的情况的变化而变化。

② 优先数的分类及确定

i 静态优先数

在进程被创建时确定，且一经确定后在整个进程运行期间不再改变。

ii 静态优先数的确定

- 优先数根据进程所需使用的资源来计算
- 优先数基于程序运行时间的估计
- 优先数基于进程的类型

iii 动态优先数

进程优先数在进程运行期间可以改变。

iv 动态优先数的确定

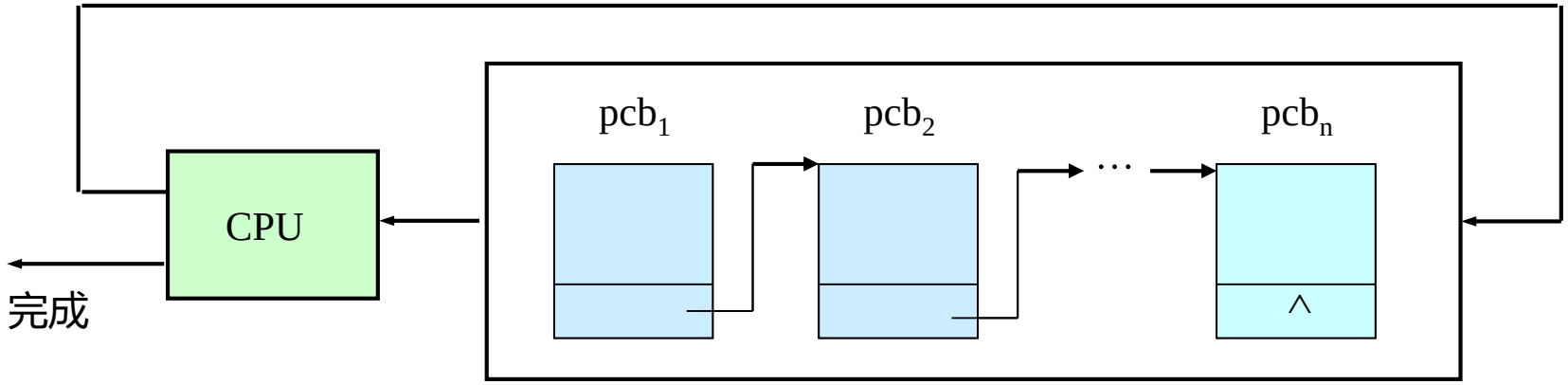
- 进程使用CPU超过一定数值时，降低优先数
- 进程I/O操作后，增加优先数
- 进程等待时间超过一定数值时，提高优先数

(2) 循环轮转调度算法

① 什么是循环轮转调度算法

当CPU空闲时，选取就绪队列首元素，赋予一个时间片，当时间片用完时，该进程转为就绪态并进入就绪队列末端。

该队列排序的原则是什么？



简单循环轮转调度算法示意图

② 简单循环轮转调度算法

就绪队列中的所有进程以等速度向前进展。

$q = t/n$

t 为响应时间，n为进入系统的进程数目

q 值的影响?

③ 循环轮转调度算法的发展

- 可变时间片轮转调度
- 多重时间片循环调度

UNIX系统采用优先数调度算法

- 进程有一个进程优先数p_pri
- p_pri取值范围是 - 127～127，其值越小，进程的优先级越高

目的

充分利用系统资源，即提高系统资源的使用效率

① 优先数的确定

- 0＃进程（ - 100）；
- 资源请求得不到满足的进程，磁盘（ - 80），打印机（ - 20），...;
- 等待块设备I/O完成的进程(- 50);
- 等待字符设备I/O完成的进程(0～20）；
- 所有处于用户态运行进程同步（一般情况下为大于100）。

② 优先数的计算

（1）计算公式：

p_pri = min{127, (p_cpu/16-p_nice+PUSER)}

大量的统计工作，并且要做浮点运算

其中：

p_cpu 进程占用CPU的程度R_1:

$R_1 = T_u/(T_u + T_{nu})$

T_u:进程创建后占用CPU的累计值

T_{nu}：进程生成后没有占用CPU的累计值

p_nice 用户通过系统调用nice()设置的进程优先数

PUSER 常数，其值为100

③ 优先数的计算

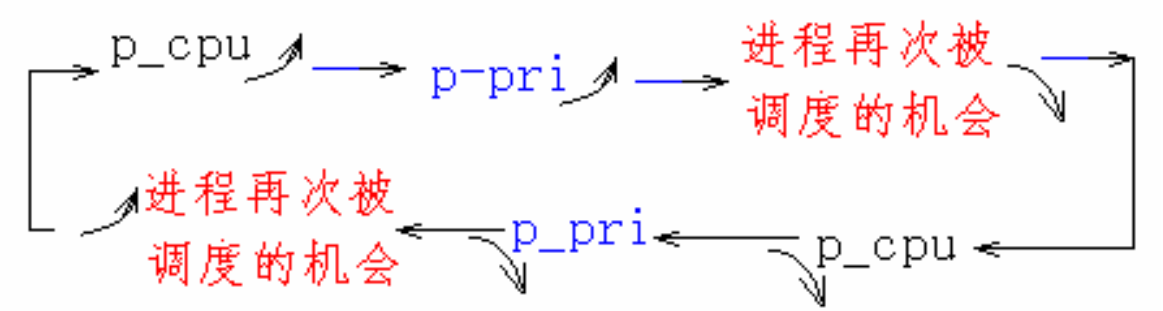
UINX系统中对p_cpu的处理：

每个时钟中断当前占用处理机的进程的 p_cpu++;

每秒钟（时钟中断）（对所有进程）：

```
p_cpu=p_cpu-10;
if (p_cpu-SCHMAG < 0 )
  p_cpu = 0;
```

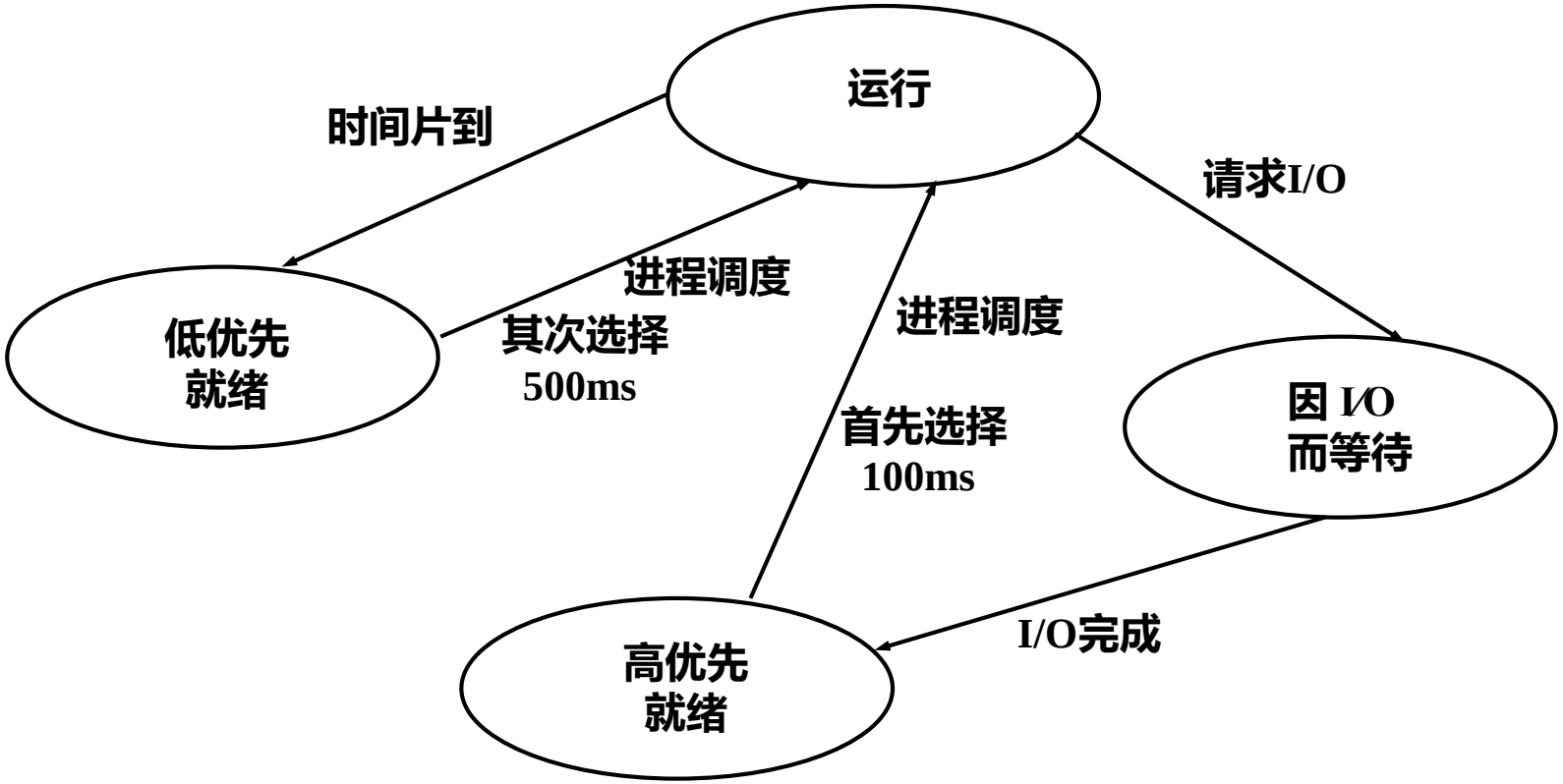
其中：SCHMAG =10



这种负反馈的效果使得系统中在用户态下运行的进程能均衡地得到处理机

5. 调度用的进程状态变迁图

(1) 一个调度用的进程状态变迁图的实例



调度用的进程状态变迁图

(2) 调度用进程状态变迁图实例的分析

① 进程状态

- 运行状态
- 低优先就绪状态
- 高优先就绪状态
- 因I/O而等待状态

② 队列结构

- 低优先就绪队列
- 高优先就绪队列
- 因I/O而等待队列

③ 进程调度算法

优先调度与时间片调度相结合的调度算法

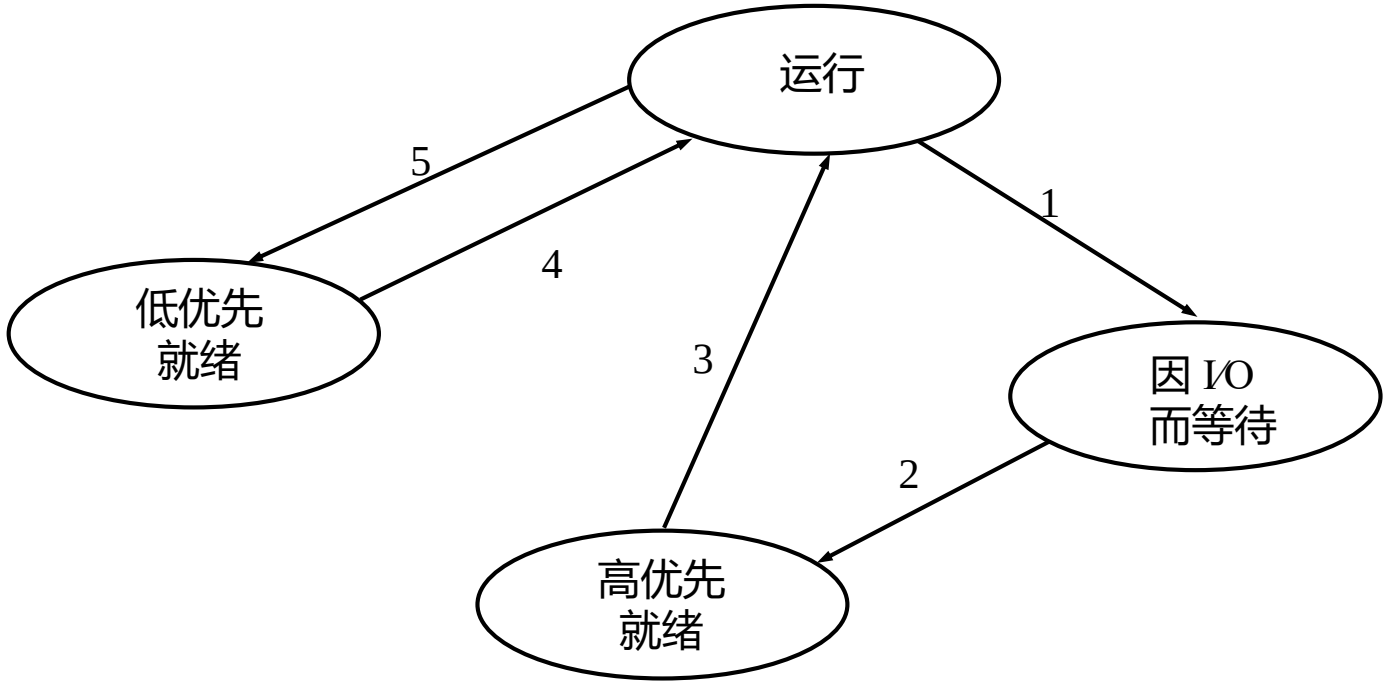
i 当CPU空闲时，若高优先就绪队列非空，则从高优先就绪队列中选择一个进程运行，分配时间片为100ms。

ii 当CPU空闲时，若高优先就绪队列为空，则从低优先就绪队列中选择一个进程运行，分配时间片为500ms。

④ 调度效果

优先照顾I/O量大的进程；适当照顾计算量大的进程。

(3) 较复杂进程状态的讨论



进程状态变迁图

变迁1 → 变迁3

变迁1 → 变迁4

变迁2 → 变迁3

进程及进程管理——小结

进程概念（掌握）

- 进程引入
  - 程序的顺序执行　定义　特点
  - 程序的并发执行　定义　特点
- 进程定义
  - 定义
  - 进程与程序的区别
- 进程状态
  - 三个基本状态、状态变迁图
  - 不同操作系统类型的进程状态变迁图
- 进程描述
  - PCB的定义与作用
  - 进程的组成
- 线程定义

进程控制（理解）

- 进程控制原语
  - 基本进程控制原语
  - 进程控制原语的执行与进程状态的变化
- 进程创建、进程撤销原语的功能
- 进程等待、进程唤醒原语的功能

进程的相互制约关系（掌握）

- 进程互斥
  - 临界资源
  - 互斥
  - 临界区
- 进程同步
  - 进程同步的概念
  - 进程同步的例

进程及进程管理——小结

- **进程同步机构（掌握）**
  - 锁、上锁原语、 开锁原语
  - 信号灯及P、V操作
- **进程同步与互斥的实现（掌握）**
  - 用信号灯的P、V操作实现进程互斥
  - 两类同步问题的解答
    - 合作进程的执行次序
    - 共享缓冲区的合作进程的同步
  - 生产者——消费者问题及解答
- **操作系统的并发控制机制（掌握）**
  - 创建进程、创建线程及其使用
  - 等待进程、线程的终止及其使用
  - 信号量与使用方法
  - 共享内存与使用方法
- **进程调度（掌握）**
  - 进程调度的功能
  - 调度方式　非剥夺方式　剥夺方式
  - 常用的进程调度算法
  - 调度用的进程状态变迁图的分析