一、绪论

1. 操作系统的类型：

   批处理操作系统、分时操作系统、实时操作系统、个人计算机操作系统、网络操作系统、分布式操作系统

2. 计算机的基本硬件：

   处理器、存储器、输入输出控制与总线、外部设备

3. 寄存器类别

   1) 用户可编程寄存器：

      数据寄存器、地址寄存器、标志寄存器

   2) 控制与状态寄存器：

      程序计数器（Program Counter）：

      存有下一周期执行指令的地址

      指令寄存器（Instruction Register）：

      存有待执行的指令

      程序状态字（Program Status Word）寄存器：

      各比特位代表系统中当前的各种不同的状态/信息

      中断现场保护寄存器：

      一组中断现场保护寄存器以保护被中断程序的现场和断恢复处

      过程调用堆栈：

      存放过程调用时的调用名、调用参数、返回地址

4. 寄存器的访问速度：

   寄存器>高速缓存>内存>硬盘缓存>硬盘>光盘|磁盘

5. 指令的执行与中断：

   1) 指令的执行：

      - 读入：根据 Program Counter 所指地址读入下一条指令：
      - 执行：执行 Instruction Register 中当前指令

2) 指令的中断：

中断：暂停正在执行的程序，转去处理相应的紧急事件，待处理完毕后再返回原处继续执行

中断具体过程：系统发生中断时，处理机收到中断信号，从而不能继续执行程序计数器中所指的原程序。这时处理机将保存当前的执行现场（也就是各寄存器中的值）并调用新的程序到处理机上执行。

6. 操作系统的启动：

启动电源 -> 计算机硬件产生中断信号 -> 触发 CPU 中的一段指令执行 -> 发现外部存储设备中的操作系统引导区(boot block)的位置 -> b.b.中的代码自动导入内存中执行 -> 将操作系统程序加载到内存中的指定区域，并初始化计算机的有关硬件 -> 操作系统程序启动

二、操作系统用户界面

1. 作业：

在一次应用业务处理过程中，从输入开始到输出结束，用户要求计算机所做的有关该次业务处理的全部工作成为一个作业。作业主要分为三个部分：程序、数据和作业说明书。

2. 系统调用：

1）设备管理

2）文件管理

3）进程控制

4）进程通信

5）存储管理

6）线程管理

3. 陷阱（Trap）:

在系统中为控制系统调用服务的处理机构称为陷阱处理机构;由于系统调用引起处理机中断的指令称为陷阱指令

4. Linux 下的系统调用(案例)

使用系统调用实现文件的打开、读、写、关闭等操作。

使用到的头文件：

1) fcntl.h : file control

是 Unix 标准中通用的头文件，包含相关函数有 open、fcntl、shutdown、unlink、fclose 等，一般还有 unistd.h 头文件对应更多的函数原型。（与 windows.h 对应）

1.1） open 函数：https://blog.csdn.net/zjhkobe/article/details/6633435

int open(const char* pathname, int oflag,…/*mode_t mode*/);

-返回值：成功则返回文件描述符（0:input; 1:output; 2:error）,否则为-1；

-pathname:待打开/创建文件的路径名；

-oflag:指定文件的打开/创建模式这个参数可由以下常量通过 || 得到：

O_RDONLY:只读模式

O_WRONLY:只写模式

O_RDWR:读写模式

在打开/创建文件时，至少得使用上述三个常量中的一个，并且有以下可选项：

O_APPEND:每次写操作都写入文件的末尾

O_CREAT:如果指定文件不存在，则创建这个文件

O_EXCL:如果要创建的文件存在，则返回-1，并且修改 errno 的值

O_TRUNC:如果文件存在，并且以只写/读写方式打开，则清除文件所有内容

-mode_t mode: 仅当创建新文件（O_CREAT）时,用于指定文件的访问权限位

S_IRWXU 00700 权限，代表该文件所有者具有可读、可写及可执行的权限。

S_IRUSR 或 S_IREAD， 00400 权限，代表该文件所有者具有可读取的权限。

S_IWUSR 或 S_IWRITE，00200 权限，代表该文件所有者具有可写入的权限。

S_IXUSR 或 S_IEXEC， 00100 权限，代表该文件所有者具有可执行的权限。

S_IRWXG 00070 权限，代表该文件用户组具有可读、可写及可执行的权限。

S_IRGRP 00040 权限，代表该文件用户组具有可读的权限。

S_IWGRP 00020 权限，代表该文件用户组具有可写入的权限。

S_IXGRP 00010 权限，代表该文件用户组具有可执行的权限。

S_IRWXO 00007 权限，代表其他用户具有可读、可写及可执行的权限。

S_IROTH 00004 权限，代表其他用户具有可读的权限

S_IWOTH 00002 权限，代表其他用户具有可写入的权限。

S_IXOTH 00001 权限，代表其他用户具有可执行的权限。

1.2） read 函数：

size_t read(int filedes, void *buf,size_t nbytes);

read 函数从 filedes 指定的已打开文件中读取 nbytes 字节到 buf 中。

返回值：读取到的字节数，0 代表 EOF,-1 表示出错。

例如，

```
#define SIZE 1
……
char *FileName1 = ”PATH_OF_FILE1” ;
char *FileName2 = “PATH_OF_FILE2”;
char Buffer[SIZE];
int InFile, OutFile, Cnt;
if (InFile=open(FileName1,O_RDONLY)==-1)printf(“Error in InFIle”);
if (OutFile=open(FileName2,O_RDWR)==-1)printf(“Error in OutFIle”);
while ((Cnt=read(InFile,Buffer,sizeof(Buffer)))>0){//读入数据到 Buffer 中
/*DO SOMETHING WITH BUFFER*/
}
```

1.3） write 函数:

size_t write(int filedes, const void *buf, size_t nbyte);

write 函数向 filedes 中写入 nbytes 字节的数据，数据来源于 buf

返回值一般等于 nbytes，否则出错。

比如在/* DO SOMETHING WITH BUFFER */中写入：

```
if (write(OutFile,Buffer,Cnt)!=Cnt) printf(“Error in write OutFile”);
if(Cnt==-1) printf(“Error in read InFile”);
```

1.4） 在调用文件的结尾，一定要关闭打开的文件以释放该文件所在资源

int close(int filedes);

返回值 0：顺利关闭；-1：发生错误

比如在上述的结尾：

if (close(inFile)==-1)printf(“Error in close InFile”);

2） stdio.h 中的读写文件操作（FILE 结构体定义在 stdio.h 中，主要用来定义带缓冲的文件指针）

2.1） fopen 函数：

FILE *fopen(char *filename, char *mode);

-filename:文件路径名

-mode:打开方式。

返回值：fopen 函数会获取文件名、文件状态、当前读写位置等信息，并将这些信息保存到一个 FILE 类型的结构体变量中，将该变量的地址返回。若文件读错时，返回空指针 NULL。比如：

```
FILE *fp;
if( (fp=fopen("D:\\demo.txt","rb") == NULL ){
    printf("Fail to open file!\n");
    exit(0);   //退出程序（结束程序）
}
```

mode 的方式：

控制读写权限的字符串（必须指明）	
打开方式	说明
"r"	以“只读”方式打开文件。只允许读取，不允许写入。文件必须存在，否则打开失败。
"w"	以“写入”方式打开文件。如果文件不存在，那么创建一个新文件；如果文件存在，那么清空文件内容（相当于删除原文件，再创建一个新文件）。
"a"	以“追加”方式打开文件。如果文件不存在，那么创建一个新文件；如果文件存在，那么将写入的数据追加到文件的末尾（文件原有的内容保留）。
"r+"	以“读写”方式打开文件。既可以读取也可以写入，也就是随意更新文件。文件必须存在，否则打开失败。
"w+"	以“写入/更新”方式打开文件，相当于 w 和 r+ 叠加的效果。既可以读取也可以写入，也就是随意更新文件。如果文件不存在，那么创建一个新文件；如果文件存在，那么清空文件内容（相当于删除原文件，再创建一个新文件）。
"a+"	以“追加/更新”方式打开文件，相当于a和r+叠加的效果。既可以读取也可以写入，也就是随意更新文件。如果文件不存在，那么创建一个新文件；如果文件存在，那么将写入的数据追加到文件的末尾（文件原有的内容保留）。
控制读写方式的字符串（可以不写）	
打开方式	说明
"t"	文本文件。如果不写，默认为 "t"。
"b"	二进制文件。

调用 fopen 函数时必须指明读写权限，但可以不指明读写方式（默认为 t）。

读写权限和读写方式可以组合使用，但是必须将读写方式放在读写权限的中间或者尾部（换句话说，不能将读写方式放在读写权限的开头）。例如：

- 将读写方式放在读写权限的末尾："rb"、"wt"、"ab"、"r+b"、"w+t"、"a+t"
- 将读写方式放在读写权限的中间："rb+"、"wt+"、"ab+"、

  r(read)　w(write)　a(append)　t(text)　b(binary)　+(read&write)

2.2） fclose 函数：文件使用结束，使用该函数以释放资源。

int fclose(FILE *fp);

-fp: 要关闭的文件指针

返回值：0：正常关闭；　　非 0 值：关闭异常

2.3） fwrite 函数：将数组中的元素写入到文件流中

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

-ptr:指向要被写入的元素数组的指针。

-size:要被写入的每个元素的大小，以字节为单位。

-nmemb:要被写入的元素的个数，每个元素的大小为 size 字节

-stream:指向 FILE 结构体指针，标识了一个输出流

返回值：如果成功，该函数返回一个 size_t 对象，表示元素的总数，。如果该数字与 nmemb 参数不同，则会显示一个错误。

例如：

```
#include<stdio.h>
int main (){
    FILE *fp;
    char str[] = "This is runoob.com";
    fp = fopen( "file.txt" , "w" );
    fwrite(str, sizeof(str) , 1, fp );
    fclose(fp);
    return(0);
}
```

2.4）fseek 函数：重定位流上的文件指针

int fseek(FILE *stream, long int offset, int fromwhere);

函数设置文件指针 stream 的位置, stream 流指向以 fromwhere 为基准，偏移 offset 个字节的位置（+为向后，-为向前）。若执行失败，则不改变 stream 的指向。

-stream： 指向 FILE 结构体的指针，指定要操作的文件流

-offset： 相对偏移量，以字节为单位

-fromwhere: 开始偏移 offset 的基准值，通常指定为以下常量：

常量	描述
SEEK_SET	文件的开头
SEEK_CUR	文件指针的当前位置
SEEK_END	文件的末尾

返回值：0：成功 非 0：失败

2.5） fgets 函数：

char* fgets(char* str,int num, FILE *stream);

函数 fgets()从流 stream 中读取一行(num-1 个字符|遇到\n|遇到 EOF)，并将其存储到指向的字符数组 str 中。

-str: 指向字符数组的指针，该数组存储了要读取的字符串。

-num:这是要读取的最大字符数（包括最后的\0）,通常是使用以 str 传递的数组长度（sizeof(str)）

-stream:指向 FILE 对象的指针，该 FILE 对象标识了要从中读取的字符的值。、

返回值：若成功：返回 str 指针；若达到 EOF 或未读到字符，返回空指针并 str 不发生改变；发生错误：返回空指针。

stream 文件流指针体指向文件内容地址的偏移原则：

如果使用 fgets()读取某个文件，第一次读取的 bufsize 为 5，而文件的第一行有 10 个字符（算上'\n'），那么读取文件的指针会偏移至当前读取完的这个字符之后的位置。也就是第二次再用 fgets()读取文件的时候，则会继续读取其后的字符。而，如果使用 fgets() 读取文件的时候 bufsize 大于该行的字符总数加 2（多出来的两个，一个保存文件本身的'\n'换行，一个保存字符串本身的结束标识'\0'），文件并不会继续读下去，仅仅只是这一行读取完，随后指向文件的指针会自动偏移至下一行。

#include<string.h>

#include<stdio.h>

```
int main (){
    FILE*stream;
    char str []="Thisisatest";
    char Buffer[20];
    stream=fopen("text.txt","w+");// 打开文件
    fwrite(str,strlen(str),1,stream);//写入文件
    fseek(stream,0,SEEK_SET);//重定位到文件头部
    fgets(Buffer,strlen(str)+1,stream);//读取文件的每一行
    printf("%s",Buffer);
    fclose(stream);//关闭文件
    return 0;
}
```

2.6） fread 函数：将文件流中的数据读取到相应的数组中

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

-ptr: 指向带有最小尺寸 size*nmemb 字节的内存块的指针

-size:要读取的每个元素的大小，以字节为单位

-nmemb:要读取的元素的个数，每个元素大小为 size 字节

-stream：指向 FILE 结构体的指针，标识了要读取的文件流

返回值：成功返回值与 nmemb 大小相同，否则失败。

例如：

```
#include <stdio.h>
#include <string.h>
int main(){
    FILE *fp;
    char c[] = "This is runoob";
    char buffer[20];
    /* 打开文件用于读写 */
```

```
    fp = fopen("file.txt", "w+");
    /*  写入数据到文件  */
    fwrite(c, strlen(c) + 1, 1, fp);
    /*  查找文件的开头  */
    fseek(fp, 0, SEEK_SET);
    /*  读取并显示数据  */
    fread(buffer, strlen(c)+1, 1, fp);
    printf("%s\n", buffer);
    fclose(fp);
    return(0);
}
```

2.7) fputs 函数：把字符串写入到指定的流 stream 中，但不包含空字符

```
int fputs(const char *str, FILE *stream);
```

-str: 这是一个数组，包含了要写入的以空字符终止的字符序列

-stream:指向 FILE 结构体的指针，标识了一个文件输出流

返回值：正常：非负值　　异常：EOF

5.windows 下的系统调用：

参看 www.msdn.com

三、进程管理

1. 程序的并发执行（Concurrent）：

并发执行，可分为两种：

1.1. 多道程序系统的程序执行环境变化所引起的多道程序的并发执行。

1.2. 在某道程序的几个程序段中包含着一部分可以同时执行或顺序颠倒的代码

并发执行定义： 一组在逻辑上互相独立的程序或程序段在执行过程中，其执行时间在客观上互相重叠，即一个程序段的执行尚未结束，另一个程序段的执行已经开始的这种执行方式。

并行执行：一组程序按独立的、异步的速度进行执行（不等于时间上的重叠）

2. 两个相邻语句 S1 和 S2 可以并发执行的条件：（***Bernstein 条件***）

将程序中任一语句 Si 划分为两个变量的集合 R(Si)和 W(Si)。其中，**R(Si)**={a1,a2,…,am}，aj 是语句 Si 在执行期间必须对其进行**读写**的变量；**W(Si)**={b1,b2,…,bn},bj 是语句 Si 在执行期间必须对其进行**修改、访问**的变量。

如果对于两个语句 S1 和 S2,满足：

① $R(S1) \cap W(S2) = \emptyset$

② $R(S2) \cap W(S1) = \emptyset$

③ $W(S1) \cap W(S2) = \emptyset$

则语句 S1 和 S2 能够并发执行。（读写变量互不影响，写变量之间也不影响）

3. 进程：

定义：并发执行的程序在执行过程中分配和管理资源的基本单位。是一个程序对某个数据集的执行过程。

进程与程序的区别：

进程是一个动态概念（是一个执行的过程），程序是一个静态概念（是一个建立的过程）

进程具有并发性，而程序没有

进程是竞争计算机系统资源的基本单位，从而其并发性受到系统自己的制约

不同的进程可以包含同一程序，只要该程序所对应的数据集不同。

4. 进程的静态描述 ： 系统中描述进程存在和反映其变化的实体。

进程的静态描述包含三部分：

进程控制块（PCB/Process Control Block）<常驻内存>

有关程序段：描述进程所要完成的功能 <外存>

数据结构集：程序执行的工作区和操作对象 <外存>

PCB 中包含了有关进程的描述信息、控制信息以及资源信息，是进程动态特征的集中体现。系统根据 PCB 感知进程的存在，通过 PCB 中包含的各项变量的变化掌握进程所处的状态，以达到控制进程活动的目的。（PCB 结构是常驻内存的）

5. 进程控制块（PCB）：集中反映一个进程的动态特性

PCB 包含一个进程的描述信息、控制信息以及资源信息，有些系统中还有【进程调度等待所使用的】现场保护区。在创建一个进程时，首先创建一个 PCB,然后根据 PCB 中的信息管理/控制进程，当进程结束时系统释放 PCB,进程从而消亡。

6. PCB 中包含的信息：

1） 描述信息：

（1） 进程名或进程标识号：

具有唯一性，并代表该进程。

（2） 用户名或用户标识号：

每个进程都隶属于某个用户，用户名或用户标识号有利于资源共享和保护。

（3） 家族关系：

PCB 中对应的项描述进程之间互成的家族关系。

2） 控制信息：

（1） 进程当前状态：

分为**初始态、就绪态、执行态、等待状态和终止状态**

初始态：该进程刚被创建，由于其他进程正占有处理机而得不到执行

执行态：该进程占有处理机

就绪态：该进程准备占有处理机

等待态：进程因某种原因不能占有处理机

终止态：进程在执行结束后，将退出执行而被终止

（2） 进程优先级：

是选取进程占有处理机的重要依据，与进程优先级有关的 PCB 表项：占有 CPU 时间、进程优先级偏移、占据内存时间

（3） 程序开始地址：

规定该进程的程序以此地址开始执行

（4） 各种计时信息：

给出进程占有和利用资源的有关情况

（5） 通信信息：

说明该进程在执行过程中与别的进程所发生的信息交换情况

3） 资源管理信息：

PCB 中包含最多的信息。包括有关存储器的信息、使用输入输出设备的信息、有关文件系统的信息。具体有：

（1） 占用内存大小及其管理用数据结构指针，例如内存管理中用到的进程页表指针。

（2） 对换或覆盖用的信息，如对换程序段长度和对换外存地址。这些信息在内存申请、释放内存时使用

（3） 共享程序段的大小和起始地址

（4） 输入输出设备的设备号，所要传送的数据长度、缓冲区地址，所用设备的有关数据结构指针。这些信息在进程申请释放设备进行数据传输时使用

（5） 指向文件系统的指针及有关标识。通过这些信息对文件系统进行操作

4） CPU 现场保护结构：

当前进程因等待某个事件而**进入等待状态或**因某种事件发生**被中止**在处理机上的**执行**时，**为了以后该进程能在被打断处恢复执行，需要保护当前进程的 CPU 现场**。PCB 中设有专门的 CPU 现场保护结构，已存储退出执行时的进程现场数据。

【Remark】PCB 是系统感知进程存在的唯一实体！通过对 PCB 的操作，系统为有关进程分配资源从而使得有关进程得以被调度执行；从而完成进程所要求功能的程序段的有关地址，以及现场保护信息；在进程结束后，则通过释放 PCB 来释放进程所占有的各种资源。

7. 进程上下文：（与程序段和数据集相关）

是进程执行过程中顺序关联的静态描述。包含了每个进程执行过的/执行时的/待执行的指令和数据，在指令寄存器、堆栈（存放各调用子程序的返回点和参数）和状态字寄存器中的内容。

上文：已执行过的进程指令和数据在相关寄存器与堆栈中的内容

正文：正在执行的进程指令和数据在相关寄存器与堆栈中的内容

下文：将要执行的进程指令和数据在相关寄存器与堆栈中的内容

在不发生进程调度时，进程上下文的改变都是在同一进程内的，每条指令的执行对进程上下文的改变较小。

同一进程的上下文的结构由下面部分与 PCB 结构构成：

（1） 与执行该进程有关的各种**寄存器中的值**

（2） 程序段经过编译后形成的**机器指令代码集（正文段）**

（3） **数据集**

（4） 各种**堆栈值**

（5） **PCB**

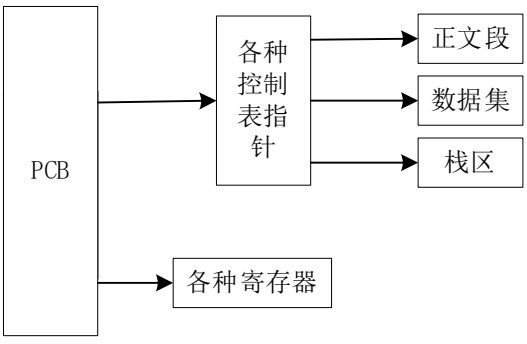


Fig: 上下文结构

在 UNIX 系统中，进程上下文可被分为：用户级上下文、寄存器上下文、系统级上下文

（1） 用户级上下文：

用户正文段、用户数据集、用户栈

（2） 寄存器上下文：

程序计数器（PC）的值、状态字寄存器（PSW）的值、栈指针的值、通用寄存器的值

PC: 给出 CPU 要执行的下条指令的虚地址

PSW: 给出机器与该进程相关联时的硬件信息

栈指针：指向下一项的当前地址

通用寄存器：用于不同执行模式之间的参数传递

（3） 系统级上下文：

分为**静态部分**和**动态部分。**

动态部分的意义：

指进入和退出不同的上下文层次时，系统为各层上下文相关联的寄存器值所保存和恢复的记录

- 静态部分包括：

  PCB 结构

  将进程虚地址空间映射到物理空间用的有关表格和核心栈（地址变换用表格）

【Remark】核心栈主要用来装载进程中所用的系统调用序列

- 动态部分（与寄存器上下文相关联）：

  进程上下文的层次概念主要体现在动态部分，即系统级上下文的**动态部分可看成是一些数量变化的层次组成的**，变化规则满足先进后出的堆栈方式，每个上下文层次在占中各占一项。

8. 进程上下文切换：<Mark>

进程上下文切换发生在**不同的进程之间**而不是同一个进程内！

进程上下文切换过程一般包括 3 个部分涉及 3 个进程：

- **第一部分：**

  保存*被切换进程*的正文部分（或当前状态）至有关存储区，如该进程 PCB 中

- **第二部分：**

  操作*系统进程*中有关调度和资源分配程序执行，并选取新的进程

- **第三部分：**

  将*被选中进程*的原来被保存的正文部分从有关存储区中取出，并送到有关寄存器与堆栈中，激活被选中进程执行

9. 进程空间与大小 （进程的大小就是进程空间的大小）

任一进程，都有一个自己的地址空间，该空间被称为进程空间或虚空间。

进程空间的大小只与处理机的位数有关！如 32 位处理机的进程空间大小为2^{32}。

程序的执行都在程序空间内进行。

用户程序、进程的各种控制表格等都按一定的结构排列在进程空间中。

在 LINUX 中，进程空间还被分为**用户空间**和**系统空间**两大部分。用户程序在用户空间中执行，操作系统内核程序在系统空间中执行。

【Remark】防止用户程序访问系统空间造成访问错误，计算机系统还通过程序状态寄存器等设置不同的执行模式，即用户执行模式（用户态）和系统执行模式（系统态）进行保护

10. 进程状态：

一个进程的生命期可划分为一组状态。

一个进程至少具有 5 个基本状态：初始态、执行态、等待态、就绪态、终止态

就绪态的进程：已得到除 CPU 之外的其他资源，只要由调度得到处理机，便可执行

单 CPU 系统中，任一时刻处于执行态的进程只能有一个。只有处于就绪态的进程经调度选中后方可进入执行态。

执行态可被分为用户执行态（用户态）和系统执行态（系统态/核心态）。

- 用户态：

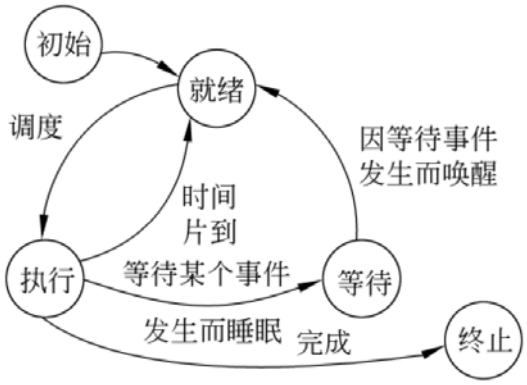
  进程的用户程序段在执行时

- 系统态：

  进程的系统程序段在执行时

为什么划分系统态和用户态：将用户程序和系统程序区分开，以利于程序的共享和保护

11. 进程状态转换：



12. 进程控制：

概念：系统使用一些具有特定功能的程序段来创建、撤销进程以及完成进程间各状态间的转换，从而达到多进程高效率并发执行和协调、实现资源共享的目的

原语：系统态下执行的某些具有特定功能的程序段

原语的分类：

- 机器指令级:

  执行期间不允许中断，是不可分割的基本单位

- 功能级：

  作为原语的程序段不允许并发执行

【Remark】通常将进程控制用程序段做成原语。

用于进程控制的原语有：

创建原语

撤销原语

阻塞原语

唤醒原语

1） 进程的创建

进程的创建方式有以下几种：、

（1） 由系统程序统一创建

这样创建的进程之间关系是平等的，它们之间一般不存在资源继承关系。

（2） 由父进程创建

父进程与父进程创建的进程之间存在隶属关系，且相互构成树形结构的家族关系。属于某个家族的一个进程可以继承其父进程所拥有的资源。

这两种创建方式都必须**由操作系统创建**一部分承担系统资源分配和管理工作的***系统进程***

创建进程必须通过***创建原语***实现：

创建原语扫描系统的 PCB 链表，在找到一定的 PCB 链表后，填入调用者提供的有关参数，最后形成代表进程的 PCB 结构。

有关参数：

进程名、进程优先级 P0、进程正文段起始地址 d0、资源清单 R0

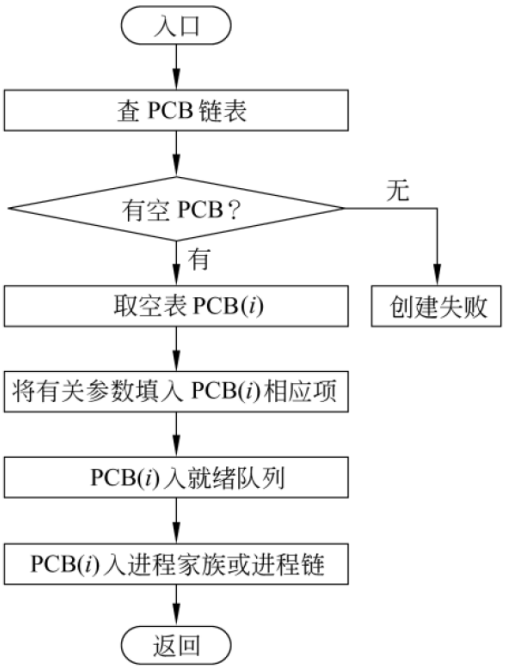


Fig: 创建原语流程图

2） 进程撤销：

导致进程撤销的情况：

（1） 该进程已完成所要求的功能而正常终止

（2） 由于某种错误导致非正常终止

（3） 祖先进程要求撤销某个子进程

进程被撤销，则需释放它所占有的各种资源和 PCB 结构本身，以利于资源的有效利用。

撤销进程的流程：

首先检查 PCB 进程链或进程家族，寻找所要撤销的进程是否存在。如果找到了所要撤销的进程的 PCB 结构，则撤销原语释放该进程所占有的资源之后，把对应的 PCB 结构从进程链或进程家族中摘下并返回给 PCB 空队列。如果撤销的进程有自己的子进程，则撤销原语先撤销其子进程的 PCB 结构并释放子进程所占有的资源后，再撤销当前进程的 PCB 结构并释放资源。

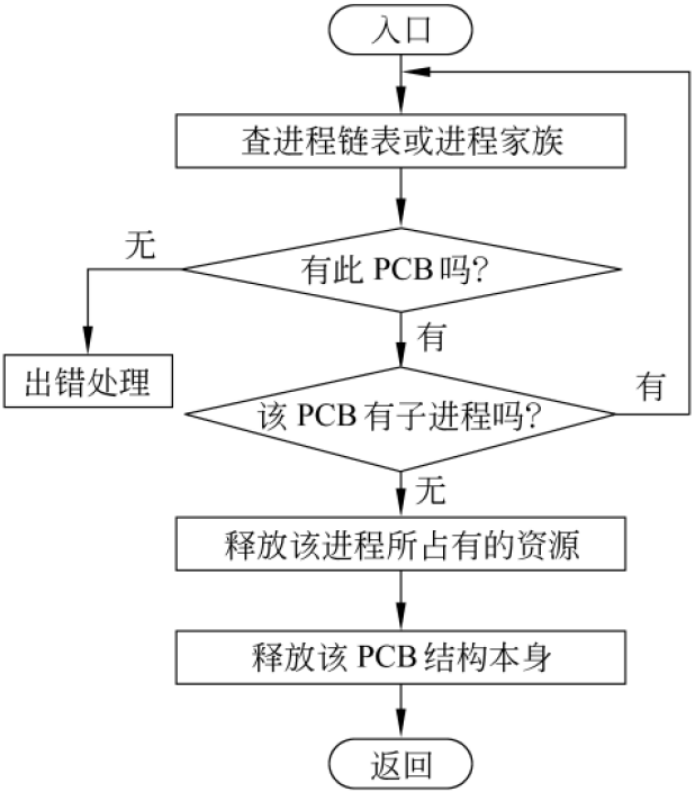


Fig:进程撤销|释放原语

3） 进程的阻塞：

阻塞原语：进程从***执行态***到***等待态***

【Remark】阻塞原语在一个进程期待某一事件（如键盘输入数据、写盘、其他进程发来数据 et.）发生，但发生条件尚不具备时，被该进程自己调用自己来阻塞自己

阻塞原语流程：

在阻塞一个进程时，由于该进程正处于执行状态，故应先中断处理机和保存该进程的 CPU 现场。然后将被阻塞进程置为“阻塞”状态后插入等待队列中，再转进程调度程序选择新的就绪进程投入运行（否则会出现空转而浪费资源）。

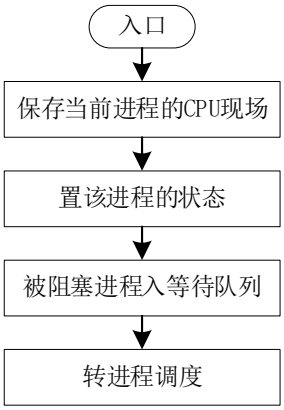


Fig:进程阻塞|阻塞原语

4） 进程的唤醒：

唤醒原语：进程从***等待态***到***就绪态***

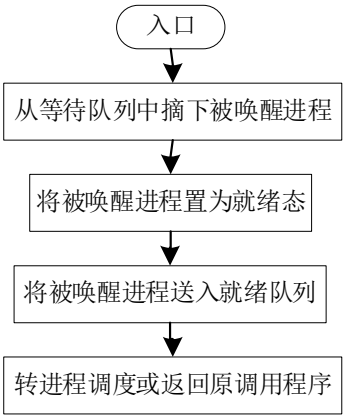
【Remark】当等待队列中的进程所等待的事件发生时，等待该事件的所有进程都将被唤醒。

一个阻塞状态的进程不能唤醒自己。

唤醒一个进程的方法：

（1） 系统进程唤醒

（2） 事件发生进程唤醒



Fig：唤醒进程|唤醒原语

13. 进程互斥：

1） 临界区（临界部分）：（访问公共数据的那段代码）

概念：不允许多个并发进程交叉执行的一段程序

起因：不同并发进程的程序段共享共用数据或共用数据变量

注意：临界区不能通过增加硬件的方法来解决

2） 间接制约：（指受公共资源的制约而非进程间直接制约）

概念：把由于共享某一公共资源而引起的在临界区内不允许并发进程交叉的现象，称为由共享公共资源而造成的对并发进程执行速度的间接制约，简称间接制约。

14. 互斥：

概念：一组并发进程中的一个或多个程序段，因共享某一公共资源而导致它们必须以一个不允许交叉执行的单位执行。或者说，不允许两个以上的共享该资源的并发进程同时进入临界区。

纯过程：在执行过程中不改变过程自身代码的一类过程

【Remark】纯过程可以被多个并发进程所访问

互斥执行的准则:

1) 不能假设各并发进程的相对执行速度（都可能进入临界区）
2) 某个进程不在临界区时，不阻止其他进程进入临界区
3) 若干进程申请进入临界区时，只允许一个进程进入
4) 某个进程从申请进入临界区开始，应在有限时间内进入临界区<不发生死锁>

15. 互斥的实现（加锁）：

对临界区加锁以实现互斥。当某个进程进入临界区，它将锁上临界区，直到它退出临界区为止。

并发进程申请进入临界区时，首先测试临界区是否是上锁的，如果临界区已被锁住，则该进程需等到开锁之后再申请临界区。

一个简单的实现伪码为：

```
Lock(x)：begin local v
         repeat
         v← x
         until v = 1
         x← 0 //locked flag
end
```

【Remark】尽管用加锁的方式可以实现进程之间的互斥，但这种方法仍然存在一些影响系统可靠性和执行效率的问题

16. 信号量（semaphore）：在对互斥处理时使用的是共用信号量

概念：信号量管理相应临界区的公共资源，它代表可用资源实体

使用：在操作系统中，信号量 sem 是个整数，在 sem≥ 0时代表可供并发进程使用的资源实体数；sem< 0时表示正在等待使用临界区的进程数。

意义：信号量 sem 初始值应大于 0，而建立一个信号量必须说明所建立信号量代表的意义，赋初值，以及建立相应的数据结构，以便指向那些等待使用该临界区的进程。

17. P、V 原语：（执行期间不允许中断）

信号量的数值仅能由 P、V 原语操作改变。

1 次 P 原语操作使得信号量 sem 减 1

1 次 V 原语操作使得信号量 sem 加 1

【Remark】当某个进程正在临界区内执行时，其他进程如果执行了 P 原语操作，则该进程并不像调用 lock 时那样因进不了临界区而返回 lock 的起点以等之后重新测试，而是在等待队列中等待有其他进程做 V 原语操作释放资源后，进入临界区，这时，P 原语的执行才算真正结束。

【Remark】当好几个进程执行 P 原语未通过而进入等待状态之后，如果有进程执行了 V 原语操作，则等待进程中的一个可以进入临界区，但其他进程必须等待。

P 原语的主要动作：

1） sem －= 1

2） 若 sem≥ 0，则 P 原语返回，该进程继续执行

3） 若 sem< 0，则该进程被阻塞后进入与该信号相对应的队列中，然后转进程调度

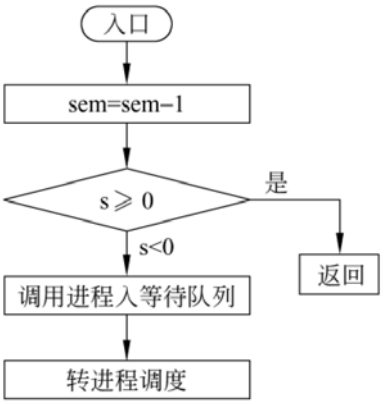


Fig:P 原语操作流程

V 原语的主要动作：

1） sem += 1

2） 若 sem> 0，V 原语停止执行，该进程返回调用处，继续执行<无等待进程>

3） 若 sem≤ 0，则从该信号的等待队列中唤醒一个等待进程，然后在返回原进程继续执行或转进程调度。<存在等待进程>

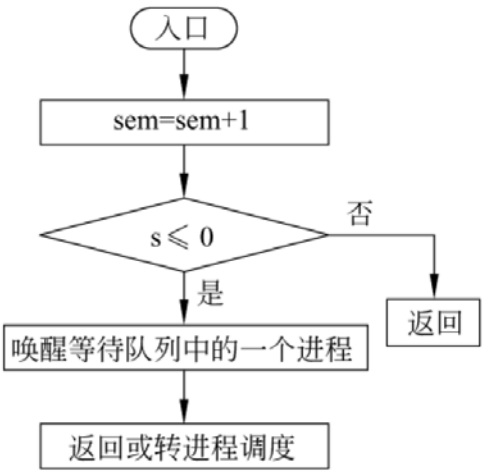


Fig:V 原语操作功能

加锁法的 P、V 原语实现：

```
P(sem):
    begin
        封锁中断;
        lock(lockbit)
        val[sem] = val[sem] - 1
        if val[sem] <0
            保护当前进程 CPU 现场
            当前进程状态置为"等待"
            将当前进程插入信号 sem 等待队列
            转进程调度
        fi
        unlock(lockbit);开放中断
    end
V(sem):
    begin
        封锁中断;
        lock(lockbit)
        val[sem] = val[sem] +1
        if val[sem] ≤0
            localk

        从 sem 等待队列中选取一等待进程,将其指针置入 k 中
        将 k 插入就绪队列
        进程状态置"就绪"
    fi
    unlock(lockbit);开放中断
end
```

18. P、V 原语实现进程互斥

使用 P、V 原语和信号量，可以方便的解决并发进程的互斥问题

两个并发进程 Pa,Pb 互斥实现：

（1） 设 sem 为互斥信号量，初值为 1（不论是几个并发进程），取值范围为（1，0，-1）。其中 sem=1 表示进程 Pa 和 Pb 都未进入类名为 S 的临界区，sem=0 表示进程 Pa 或 Pb 已进入类名为 S 的临界区，sem=-1 表示进程 Pa 和 Pb 中，一个进入临界区，一个进入等待队列。

（2） 实现描述;

Pa:

P(sem)

<临界区 S>

V(sem)

Pb:

P(sem)

<临界区 S>

V(sem)

---

19. 进程同步：

并发进程间的直接制约：一组在异步环境下的并发进程，各自的执行结果互为对方的执行条件，从而限制各进程的执行速度的过程。

异步环境：各并发进程的执行时间的随机性和执行速度的独立性。

进程同步：异步环境下的一组并发进程因直接制约而互相发送信息，从而进行互相合作、互相等待，使得各进程按一定的速度执行的过程

合作进程：具有合作关系的一组并发进程

消息（事件）：合作进程间互相发送的信号

使用 wait(消息名)和 signal(消息名)的方式描述 Pc 和 Pp 进程的同步关系：

（1） 设消息名 **Bufempty** 表示 **Buf** 为空，消息名 **Buffull** 表示 **Buf** 中装满数据

（2） 初始化 **Bufempty**=true,**Buffull**=false

```
P_C  :
     A: wait(Bufempty)
          计算
          Buf ← 计算结果
          Bufempty ← false
          signal(Buffull)
          Goto A
P_P  :
     B: wait(Buffull)
          打印 Buf 中的数据
          清除 Buf 中的数据
          Buffull ← false
          signal(Bufempty)
          Goto B
```

- wait 的功能是等待到消息名为 true 的进程继续执行
- signal 的功能是向合作进程发送合作所需的消息名，并将其置为 true

20. 私用信号量(private semaphore)：

信号量只与制约线程及被制约线程有关，而不是与整组并发进程有关，称为私用信号量

一个进程 P 的私用信号量 Sem 是从制约进程发送来的进程 P 的执行条件所需的消息

21. P、V 原语实现同步：

主要步骤：

（1） 为个并发进程设置私用信号量

（2） 为私用信号量赋初值

（3） 利用 P、V 原语和私用信号量规定各进程的执行顺序

比如：

```
P_A:  deposit(data):
      begin local x
          P(Bufempty);
          按 FIFO 方式选择一个空缓冲区 Buf(x);
          Buf(x)← dat
          Buf(x)置满标记
          V(Buffull)
      end

P_B:  remove(data):
      Begin local x
          P (Buffull);
          按 FIFO 方式选择一个装满数据的缓冲区 Buf(x)
          data ← Buf(x)
          Buf(x)置空标记
          V (Bufempty)
      End
```

22. 生产者-消费者问题(Producer-Customer Problems)

资源：外设、内存、缓冲区等硬件资源；临界区、数据、例程等软件资源

消费者：系统中**使用**某一类资源的进程称为该资源的消费者

生产者：系统中**释放**同类资源的进程称为该资源的生产者

生产者-消费者问题是一个同步问题，满足以下条件：

（1） 消费者想接受数据，有界缓存区中至少一个单元是满的

（2） 生产者想发送数据，有界缓存区中至少一个单元是空的

（3） 各生产者和各消费者之间必须互斥（有界缓存区是临界资源）

公共信号量 mutex：

- 保证生产者进程和消费者进程之间的互斥
- 表示可用有界缓冲区的个数，初值为 1

私用信号量 avail：

- 生产者进程的私用信号量
- 表示有界缓冲区的空单元数，初值为 n

私用信号量 full：

- 消费者进程的私用信号量
- 表示有界缓冲区的非空单元数，初值为 0

从而得到 **生产者过程（deposit）** 和 **消费者过程（remove）**

```
deposit(data):
    begin
        P(avail)
        P(mutex)
        送数据入缓冲区某单元
        V(full)
        V(mutex)
    end
remove(data):
    begin
        P(full)
        P(mutex)
        取缓冲区中某单元数据
        V(avail)
        V(mutex)
    End
```

【Remark】V 原语释放资源可以任序；P 原语不可任序否则死锁

23. 进程通信:

低级通信：进程间控制信息的交换

- 通常只传送一个或几个字节的信息
- **目的：**达到控制进程执行速度的作用

高级通信：进程间大批量数据的交换

- 通常要传送大量的数据
- **目的：**交换信息

24. 单机系统中的进程通信方式：

（1） 主从式：（如：终端控制进程 和 终端进程）

- 主进程可自由的使用从进程的资源或数据
- 从进程的动作受主进程的控制
- 主进程和从进程的关系是固定的

（2） 会话式：（如 用户进程 和 磁盘管理进程）

- 通信进程的双方为**使用进程**和**服务进程，**使用进程调用服务进程提供的服务
- 使用进程在使用服务进程所提供的服务之前，必须得到服务进程的许可
- 服务进程根据使用进程的要求提供服务，但对所提供服务的控制由服务进程是自身完成
- 使用进程和服务进程在进行通讯时有固定连接关系

（3） 消息或邮箱机制：（两个互相通信的进程地位平等）

- 无论接收进程是否已经准备好接收消息，发送进程都将把所要发送的消息发送到缓冲区或邮箱
- 消息的构成：发送进程名 – 接收进程名 – 数据 – 有关数据的操作
- 只要存在空缓冲区或邮箱，发送进程就可以发送消息
- 发送进程和接收进程之间无直接连接关系，接收进程可能在收到某个发送进程发来的消息之后，又转去接收下一个发送进程发来的消息
- 发送进程和接收进程之间存在缓冲区或邮箱用以存放传送消息

（4） 共享存储区方式

- 该方式不要求移动数据。两个需要互相交换信息的进程通过对同一共享数据区的操作来达到互相通信的目的。
- **共享数据区**是每个相互通进程的一个组成部分

25. 消息与邮箱机制：消息缓冲机制

机制描述：

- 发送进程在发送消息前，先在自己的内存空间设置一个发送区，把欲发送的消息填入其中，然后再用发送过程将其发送出去。
- 接收进程在接收消息之前，在自己的内存空间内设置相应的接收区，然后用接收过程接收消息

【Remark】消息缓冲机制中使用的缓冲区是**公共缓冲区**

两个进程需要满足的条件：

（1） 在发送进程把消息写入缓冲区和把缓冲区中挂入消息队列时，禁止其他进程对该缓冲区消息队列的访问（否则引起消息队列的混乱）

（2） 在接收进程从消息队列中取出消息缓冲时，禁止其他进进程对该队列的访问

（3） 当缓冲区中无消息存在时，接收进程不能接收任何消息

缓冲机制的伪码描述：

- 公共信号量 mutex：控制对缓冲区访问的互斥信号量，初值为 1
- 接收进程的私用信号量 SM：等待接收的消息个数，初值为 0
- Send(m)：发送进程将消息 m 发送到缓冲区
- Receive(m)：接收进程将消息 m 从缓冲区读往自己的数据区
- 伪码描述为

```
send(m):
    begin
        向系统申请一个消息缓冲区
        P(mutex)
        将发送区消息 m 送入新申请的消息缓冲区
        把消息缓冲区挂入接收进程的消息队列
        V(mutex)
        V(SM)
    end
receive(n):
    begin
        P(SM)
        P(mutex)
        摘下消息队列中的消息 n
        将消息 n 从缓冲区复制到接收区
        释放缓冲区
        V(mutex)
    end
```

【Remark】无法在 send 过程中用 P 操作来判断信号量 SM

26. 邮箱通信：

概念：由发送进程建立一个与接收进程链接的邮箱。发送进程把消息送往邮箱，接收进程从邮箱中取出邮箱，从而完成进程间的通信。

优点：发送进程和接收进程之间没有时间上的限制

邮箱与缓冲区的区别：

- 邮箱可考虑成发送进程与接收进程之间大小固定的私有数据结构，
- 缓冲区被系统内所有进程共享

邮箱的构成：

- 邮箱头：邮箱名称、邮箱大小、邮箱方向、拥有该邮箱的进程名
- 邮箱体：存放消息

对于只有一个发送进程和一个接收进程的邮箱通信条件：

（1） 发送进程发送消息时，邮箱中至少要有一个空格能存放该消息

（2） 接收进程接收消息时，邮箱中至少有一个消息存在

邮箱的伪码描述：

- deposit(m): 发送进程将消息 m 发送到邮箱
- remove(m): 接收进程将消息 m 从邮箱取出
- 信号量 fromnum：发送进程的私用信号量，初值为信箱的空格数
- 信号量 mesnum：接收进程的私用信号量，初值为 0
- 伪码描述：

```
deposit(m):
    begin local x
        P(fromnum)
        选择空格 x
        将消息 m 放入空格 x 中
        置格 x 的标志为满
        V(mesnum)
    end
remove(m):
    begin local x
        P(mesnum)
        选择满格 x
        把满格 x 中的消息取出放 m 中
        置格 x 标志为空
        V(fromnum)
    end
```

【Remark】deposit 和 remove 之间存在同步制约关系而不是互斥制约关系

27. 进程通信的实例 – 管道（pipeline/pipe）：

分类：有名管道 和 无名管道

无名管道：为建立管道的进程及其子孙提供了一条以比特流方式传送消息的管道通信。

无名管道构成：

- 在逻辑上被看做是管道文件
- 在物理上则由文件系统的高速缓存区构成，而很少启动外设

发送进程：

- 利用文件系统的系统调用 write(fd[1],buf,size)把 buf 中的长度为 size 个字符的消息送入管道入口 fd[1]

接收进程：

- 利用文件系统的系统调用 read(fd[0],buf,size)从管道出口 fd[0]读出 size 个字符的消息置入 buf 中

【Remark】管道按 FIFO 方式传递消息，只能单向传递消息

建立管道：

- 使用 pipe 系统调用建立一条同步通信管道

```
int fd[2];
pipe(fd);
/*其中 fd [1]为写入端，fd[0]为读出端*/
```

【例1】 建立一个管道，同时父进程生成一个子进程，子进程向管道中写入一个字符串，父进程从管道中读出该字符串

```
#include <stdio.h>
void main(){
        int x,fd[2];
        char buf[30],s[30];
        pipe(fd);//父进程建立管道
        while((x=fork())==-1);//创建子进程失败时，循环【注 1】
        if (x==0){//子进程
            sprintf(buf,”This is an example/n”);//将格式化字符串写入到缓冲区中
            write(fd[1],buf,30);//将 buf 中的字符写入管道
            exit(0);//正常退出进程
```

```
        }
        else{//父进程返回
            wait(0);//阻塞自己，保证子进程先完成【注 2】
            read(fd[0],s,30);
            printf("%s",s);//父进程读管道的字符
        }
    }
```

【1】 pid_t fork(void); /* https://baike.baidu.com/item/fork/7143171?fr=aladdin*/

- 包含头文件：unistd.h sys/types.h
- fork 系统调用用于创建一个新进程，称为子进程，它与进程（称为系统调用 fork 的进程）同时运行，此进程称为父进程。
- 创建新的子进程后，两个进程将执行 fork()系统调用之后的下一条指令。子进程使用相同的 PC（程序计数器），相同的 CPU 寄存器，在父进程中使用的相同打开文件。
- 返回值：

  负值：创建子进程失败

  零：返回到新创建的子进程

  正值：返回父进程或调用者。该值包含新创建的子进程 ID
- fork()函数一次调用返回两次!!!

【2】 int wait(int *status);/*wait 和 fork 配套使用*/

- 包含头文件 wait.h
- 阻塞父进程以等到子进程处理完成（销毁僵尸进程）
- 父进程一旦调用了 wait 就立即阻塞自己，由 wait 自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait 就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait 就会一直阻塞在这里，直到有一个出现为止。
- 返回值：正常返回子进程 PID 异常返回负值
- status 是指向 int 类型的指针，用来保存被收集进程退出时的状态，若不关心退出的状态，设为 NULL 或者 0

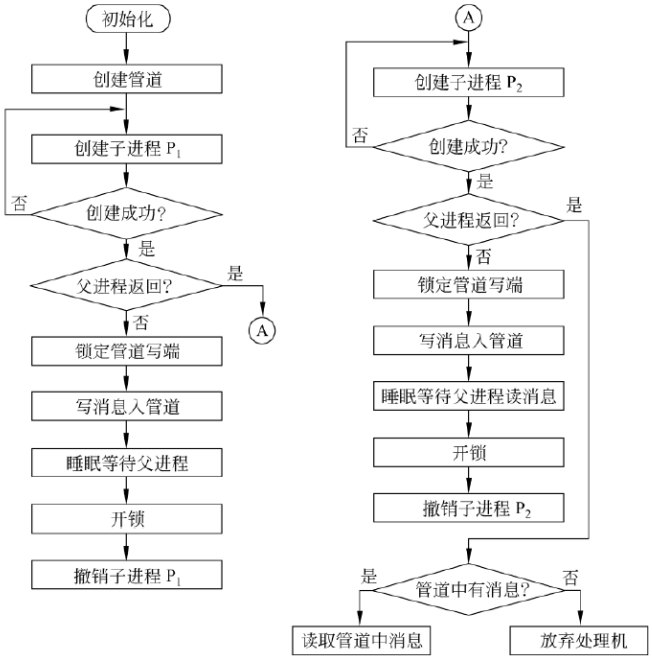
【3】 exit(0):

- 包含头文件：stdlib.h
- 退出进程

【例2】 建立一个管道。父进程生成子进程 P1 和 P2,这两个子进程分别向管道中写入各自的字符串，父进程读出它们。

```
#include <stdio.h>
void main(){
    int i, r, p1, p2, fd[2];
    char buf[50],s[50];
    pipe(fd);//父进程建立管道
    while((p1=fork())==-1);//创建子进程 p1,失败时循环
    if (p1==0){
        lockf(fd[1],1,0);//lock file:加锁锁定写入端【注 4】
        sprintf(buf,"child process P1 is sennding messages\n");
        write(fd[1],buf,50);//将 buf 中的 50 个字符写入管道
        sleep(5);//睡眠 5s，让父进程读
        lockf(fd[1],0,0);//unlock file:释放管道输入端
        exit(0);//关闭 P1 子进程
    }
    else{//从父进程返回，执行父进程
        while((p2=fork())==-1);//创建子进程 p2,失败时循环
        if (p2==0){//从子进程 p2 返回，执行 p2
            lockf(fd[1],1,0);//lock file:加锁锁定写入端
            write(fd[1],buf,50);//将 buf 中的 50 个字符写入管道
            sleep(5);//睡眠 5s，让父进程读
            lockf(fd[1],0,0);//unlock file:释放管道输入端
            exit(0);//关闭 P2 子进程
        }
        wait(0);//阻塞进程，等待 P1 和 P2 子进程结束
```

```
        if (r=read(fd[0],s,50)==-1)//父进程读取子进程字符串（P1 或 P2）
            printf("can't read pipeline\n");
        else printf("%s",s);
        if (r=read(fd[0],s,50)==-1)//父进程读子进程字符串（P2 或 P1）
            printf("can't read pipeline\n");
        else printf("%s",s);
        exit(0);//退出父进程
    }
}
```



【4】 int lockf(int fd, int cmd,off_t len);

- 目的：锁定文件及解锁文件
- fd:文件描述符；cmd:0 解锁，1 互斥锁定；len:锁定区域，0 为当前偏移量到文末
- 返回值：调用成功 0；失败-1

【5】 lockf 为保证进程互斥使用管道的系统调用

【6】 sleep 为保证当前进程睡眠以转让处理机的系统调用

28. 死锁问题

- **概念**：各并发进程互相等待对方所拥有的资源，且这些并发进程在得到对方的资源之前不会释放自己所拥有的资源。从而造成大家想得到资源但又得不到资源，各个并发进程无法继续推进的状态。|| **互相需要彼此的资源，但都不放手**
- **起因**：由并发进程的资源竞争造成。
- **产生的必要条件**：

1. 互斥条件：

   资源只能由一个进程所享有

2. 不剥夺条件：

   进程所获得的资源在未使用结束之前不能被其他进程所剥夺

3. 部分分配：

   进程每次申请所需资源时，在等待新资源的同时，继续占用已分配到的资源

4. 环路条件：

   存在一种进程回环链，链中每个进程已获得的资源同时被下一个进程所请求

- 消除方法：

（1） 预防方法：

限制并发进程对资源的请求

a) 打破资源的**互斥**和**不可剥夺**条件,如允许并发进程同时访问资源

b) 打破资源的**部分分配**条件，即预先分配并发进程的所有资源

c) 打破资源的**环路条件**，即将资源分类按顺序排列，使之不成回环

（2） 避免方法（动态预防）：资源分配调度策略

系统在分配资源时，根据资源的使用情况进行预测

基本模式：把进程分为多个步，其中每个步使用的资源是固定的，且在一个步中，进程所保持的资源数不变。即**进程的资源请求、使用和释放要依赖不同的步完成。**

使用请求向量 Wi、分配向量 Ai、释放向量 Bi、空闲资源向量 Fi、系统能力向量 Ci,进行进程分步时，进程的请求向量不大于空闲资源和该进程准备释放的资源，则安全

（3） 检测和恢复

设置专门的机构，检测死锁的位置和原因，并外力破坏死锁的发生条件

检测：有限状态转移图和 PetriNet 技术

恢复：终止各锁住的进程或按一定顺序终止进程序列

29. 线程的概念;

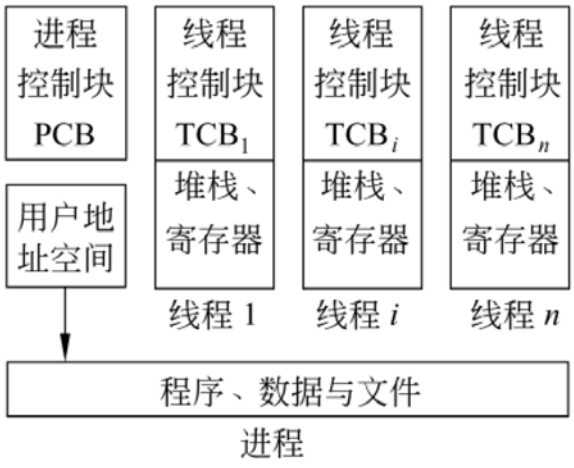
原由：为提高 CPU 执行效率，减少因为程序等待带来的 CPU 空转以及其他计算机软硬件资源的浪费（有多任务需要处理机处理时减少处理机的切换时间）

概念：

- 线程是进程的一部分，有时把线程称为轻量级进程或轻权进程
- 线程也是 CPU 调度的一个基本单位
- 没有线程的进程可被看做是单线程的，即进程的执行过程是线状的
- 如果在一个进程中拥有多个线程，则进程的执行过程由多条线状执行过程组成

线程与进程的区别：

- 线程的改变只代表了 CPU 执行过程的改变，而计算机内的软硬件资源的分配与线程无关，线程只能共享它所属进程的资源
- 线程有线程控制块（TCB）,TCB 中所保存的线程状态信息主要是相关指针用堆栈(系统栈和用户栈)以及寄存器中的状态数据
- TCB 信息少于 PCB 信息
- 进程是系统中所有资源分配的基本单位，有 1 个完整的虚拟内存地址
- 线程是进程的一部分，没有自己的地址空间
- 进程中的多个线程一起共享分配给该进程的资源



线程的使用范围：

- 多处理机系统、网络系统、分布式系统
- 在用户程序可以按功能分为不同的小段时，单处理机系统也可因使用线程而简化程序的结构和提高执行效率

应用场景：

- 服务器中文件管理或通信控制
- 前后台处理（实时性不高的程序安排到处理机空闲时处理）
- 异步处理（程序中的两部分没有顺序规定）

线程的分类：

- **用户级线程：（不需要操作系统内核的特殊支持，只需要线程库）**

  1） 管理过程（调度算法和调度过程）全部由用户程序完成

  2） 操作系统内核只对进程进行管理

  3） 操作系统提供在用户空间执行的线程库（创建/调度/撤销线程）

  4） 只使用用户堆栈和分配给所属进程的用户寄存器

  5） 当一个线程被派生时，线程库为其生成相应的线程控制块（TCB）等数据结构，并为 TCB 的参量赋值和把该线程置为就绪态

  6） 操作系统内核的调度单位是进程

  7） 调度算法只进行线程上下文切换而不进行处理机切换

  8） 线程上下文切换不涉及处理机状态（只在用户栈、用户寄存器中）

  9） 相关进程阻塞或等待，但所属线程可能却在执行（上下文切换与内核无关）

- **系统级（内核级）线程：（需要操作系统内核支持）**

  1） 由操作系统内核进行管理

  2） 操作系统内核给应用程序提供相应的系统调用和应用程序接口（API），以使得用户能够创建、执行、撤销线程

  3） 既可以调度到一个处理机上并发执行，也可以被调度到不同的处理机上并行执行

  4） 操作系统内核既负责进程的调度，也负责进程内不同线程的调度

  5） 不会出现进程堵塞或等待而线程却能执行的情况

  6） 内核级线程可用于内核程序本身，提高操作系统内核程序执行效率

  7） 内核级线程上下文切换时间要大于用户级线程的上下文切换时间

- **线程的执行特性：**

1） 基本特性：执行、就绪、阻塞

2） 只与内存和寄存器相关的概念，它的内容不会因交换而进入外存

3） 5 中基本操作：

a) **派生：**

线程在进程中派生出来，既可由进程派生，也可由线程派生 clone()|Create-thread(),新派生的线程放到就绪队列中

b) **阻塞：**

线程在执行过程中需要等待某个事件发生，则被阻塞

c) **激活：**

如果阻塞线程的事件发生，则该线程被激活进入就绪队列中

d) **调度：**

选择一个就绪线程进入到执行状态

e) **结束：**

线程执行结束，它的寄存器上下文及堆栈被释放

【Remark】 线程阻塞可能导致进程阻塞

【注】线程的一个执行特性是同步。由于同一进程中的所有线程共享该进程的所有资源和地址空间，任何线程对资源的操作都会对其他相关线程造成影响。因此系统必须为线程操作提供同步控制机制。

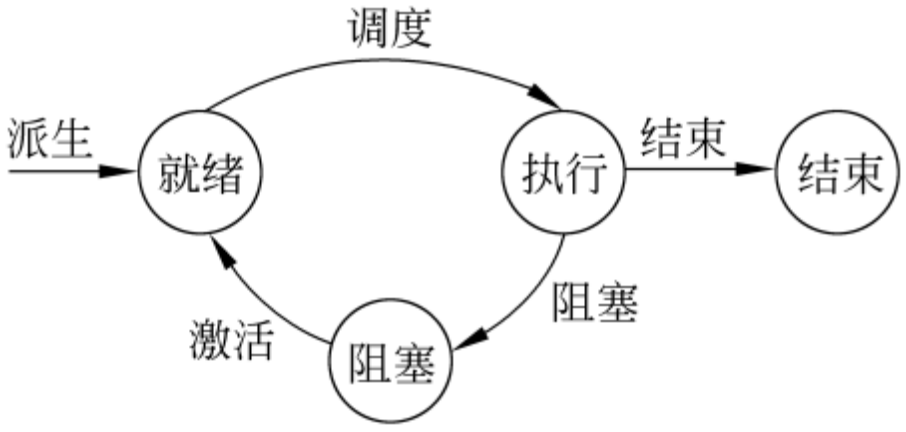


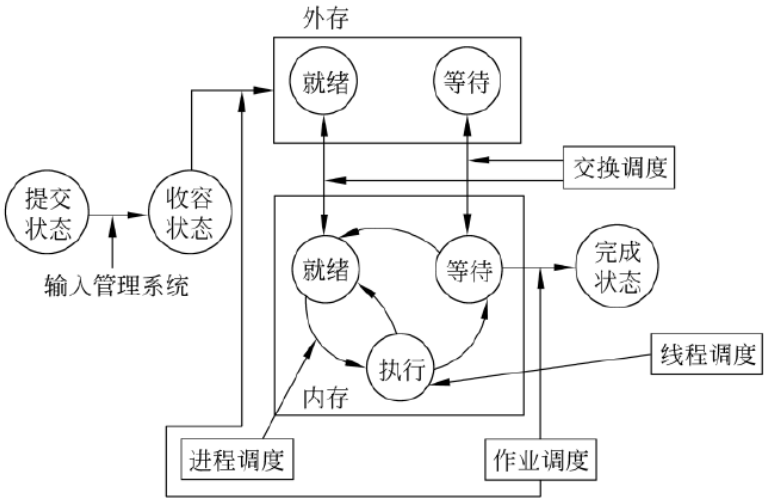
Fig: 线程的状态和操作

四、处理机调度

1. 衡量调度策略的指标：

   - 周转时间（作业时间）
   - 吞吐量
   - 相应时间（命令时间）
   - 设备利用率

作业：一次业务处理的全部时间，包括 提交、执行、输出



提交状态：作业从输入设备进入外存的过程

收容状态：作业全部传入到输入井（外存中存放作业处理信息）还未被调度执行

执行状态：通过调度使得作业在内存中执行

完成状态 ：作业运行完毕，但资源未回收

2. 调度可分为 4 级：

   - 作业调度：（不存在于分时和实时系统中）

     从输入井中选择作业并分配作业资源，分配根进程（使该作业能竞争处理机），执行完成后回收资源。**（收容->执行，执行->完成）**

   - 交换调度：

     将外存就绪态或等待态进程调入内存；将内存就绪态或等待态进程调入外存

   - 进程调度：

     选用就绪态进程进入处理机，并进行进程的上下文切换

- 线程调度

  为多个线程分配 CPU 的使用

3. 作业和进程的关系：

   作业：用户向计算机提交任务的任务实体

   进程：计算机为完成用户任务而设置的执行实体（系统分配资源的基本单位）

   一个作业由一个以上进程组成

4. 作业分解为进程：

   - 系统为作业建立一个根进程
   - 根据任务要求，在执行任务控制语句时，系统/根进程创建子进程
   - 给子进程分配资源 & 调度子进程

5. 作业调度功能：

   （1） 记录系统中各作业的状况，包括执行阶段的有关情况：

   通过作业作业控制块（JCB）记录有关信息。在进入收容态后建立 JCB，使得作业被作业调度程序感知

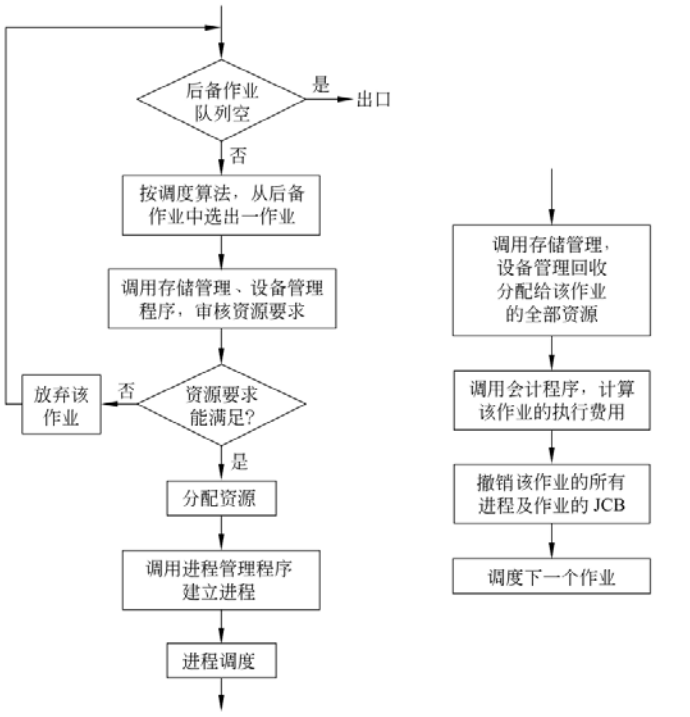
作业名
作业类型
资源要求
资源使用情况
优先级(数)
当前状态
其他

   Fig: JCB 结构

   （2） 从后备队列中选出一部分作业投入执行

   （3） 为被选中的作业做好执行前的准备工作

   （4） 在任务结束时做善后处理工作



(a) 作业从后备状态到执行状态　　(b) 作业从执行状态到完成状态

Fig： 作业调度的状态转换过程

6. 作业调度算法的评价指标：

（1） 周转时间：

包含等待时间和执行时间

（2） 带权周转时间;

作业周转时间/作业执行时间

7. 进程调度的功能：

（1） 记录系统中所有进程的执行情况

（2） 选择占有处理机的进程

（3） 进行进程的上下文切换

8. 进程调度的评价指标：

通过模拟或测试系统响应时间

9. 调度算法：

（1） 先来先服务（FCFS）调度算法：只考虑等待时间

将用户作业和就绪进程按照提交的顺序或变为就绪状态的先后排成**队列**，并按照先来先服务（First Come First Serve）的方式进行调度处理。

（2） 轮转法（round robin）： （作业调度不适用）

让每个进程在就绪队列中的**等待时间**与**享受服务的时间**成比例。

将 CPU 的处理时间分成固定大小的时间片，若一个进程在被调度选中之后用完了系统规定的时间片，但未完成要求的任务，则它自行释放自己所占 CPU 而排到就绪队列的末尾，等待下一次调度。

作业调度中包含了不可抢占的资源，不能使用轮转法。

（3） 多级反馈轮转法（round robin with multiple feedback）:

将由于不同的原因进入就绪队列（1.时间片用完，2.从等待进入就绪，3.新建进程进入就绪）分别对待。

（4） 优先级法：作业/进程调度

系统或用户通过以某种原则设置一个优先级来表示该作业或进程所享有的调度优先级。

- 静态优先级：一旦执行不能更改优先级

作业调度的优先级原则：

1） 由用户自己根据作业的紧急程度输入合适的优先级

2） 由系统/操作员根据作业类型指定优先级

3） 系统根据作业要求资源情况确定优先级

进程调度的优先级原则：

1） 按照进程的类型基于不同的优先级

2） 将作业的静态优先级作为所属进程的优先级

- 动态优先级：将静态特性和动态特性相结合以动态的改变优先级

进程的动态优先级原则：

1） 根据进程**占有 CPU 的时间**长短来决定：

占据时间愈长，在阻塞之后再次调度的优先级愈低

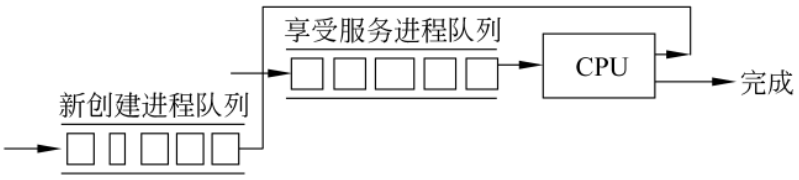
2） 根据就绪进程等待 CPU 的时间长短来决定：

等待时间愈长，获得的优先级愈高

- 线性优先级调度策略（Selfish Round Robin/SRR）

原理：新创建的进程按 FCFS 策略排成就绪队列（优先级随时间增长而增高$P(t) = at$），而其他已得到过时间片服务的进程也按照 FCFS 策略排成另一个就绪队列（享受服务队列）（享受服务队列的优先级$P(t) = bt$）,a>b。

当新建就绪队列的 top 进程/作业 P1 的优先级与享受服务队列的 bottom 进程/作业 P2 的优先级相同时，将 P1 加入到享受服务队列中。



（5） 最短作业优先法（Shortest Job First/SJF）：只考虑执行时间

选择执行时间最短的作业投入执行（吞吐量会比较大）

（6） 最高响应比优先法（Highest Response-ratio Next/HRN）：

考虑每个作业的等待时间和执行时间，从中选出响应比最高的作业投入使用。

其中响应比$R = (W + T)/T$，W:等待时间。T:执行时间

10. 算法评价：

（1） FCFS:使用泊松过程进行分析。

对于 M/M1（顾客到达服务器的时间间隔满足泊松过程<参数为λ>，顾客离开服务器的时间间隔满足泊松过程<参数为μ>，服务器只有 1 台）

设**响应时间 R 为顾客到达等待队列后开始到离开服务器的时间**，则在稳定状态后，系统中的顾客数$n = \lambda R$,其中λ为顾客的平均到达率。（***Little Result***）

同时，令$\rho = \lambda/\mu$，$n = \rho(1 - \rho)$。当ρ越大，响应时间急剧变大，系统性能变差；

当$\rho < 1/2$时，等待队列中为空的可能性较大。

对短作业是不利的。

（2） Round Robin：

所需服务时间短的顾客的响应时间小于所需服务时间长的响应时间。因此轮转法在响应时间上要优于 FCFS 调度方法。

（3） SRR:

介于 FCFS 和轮转法之间的一种调度策略，

（4） 响应时间的比较：

平均响应时间：

- FCFS：$R_{fc} = 1/(\mu - \lambda)$
- 轮转法：$R_{rr} = kq\mu/(\mu - \lambda)$ (q 为时间片，k 为顾客平均需要时间片数)
- SRR：$R_{sr} = \frac{1}{\mu - \lambda'} - \frac{1 - kq\mu}{\mu - \lambda}$ （λ:到达就绪队列的到达率；λ′:到达享受服务队列的达到率）

在长作业时，有：$R_{fc} < R_{sr} < R_{rr}$

在短作业时，有：$R_{rr} < R_{sr} < R_{sr}$

11. 实时系统：

特点：

- 处理和控制的正确性取决于：1）计算的逻辑结果 2）计算和处理结果产生的时间。
- 外部任务可分为周期性的和非周期性的。

硬实时任务（hard real time task）：要求系统必须完全满足任务的时限要求

软实时任务（soft real time task）：允许系统对任务的时限要求有一定的延迟，时限要求只是一个相对条件。

非周期性任务：必定存在一个完成或开始处理的时限。

周期性任务：只要求在周期 T 内完成或开始进行处理。

总的特点：

- **有限等待时间**：所有进程在处理事件时在有限时间内开始
- 有限响应时间：从系统响应外部事件开始，必须在有限时间内处理完毕
- 用户控制：控制进程的优先级和选择相应的调度算法
- 可靠性高：基本不出现错误
- 系统出错处理能力强：出错后也不能影响正在执行的用户使用

12. 实时系统的调度算法：

- 静态表格驱动类：对可能的参数和调度条件分析得到调度结果以处理周期性任务
- 静态优先级驱动抢先式调度算法：静态分析以指定优先级
- 动态计划调度算法类：在处理任务之前排出调度任务并进行分析是否满足时限
- 尽力而为类：只对到达的事件和相关任务指定相应的优先级并调度

13. 时限调度算法：（静态表格驱动类/抢先式）

按用户的时限要求顺序设置优先级，优先级高者占据处理机。

14， 频率单调调度算法：（静态优先级驱动）

频率越低（周期越长）的任务优先级越低

五、存储管理

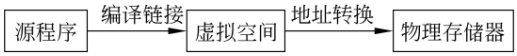
1. 存储器；CPU 要通过启动相应的输入输出设备后才能使外存和内存交换信息
    - 内存：由顺序编址的块组成，每块包含相应的物理单元。
    - 外存
2. 虚拟存储器：

    缘由：在物理上只受内存和外存总容量的限制的存储系统，该存储系统只把进程执行时频繁使用和立即需要的指令与数据等存放到内存中，而把暂时不需要的部分放到外存中，在需要时自动调入。

    编译链接程序把用户源程序编译后链接到一个以 0 地址为始地址的线性或多维虚拟地址空间。这里**链接**可以**是**程序执行以前由链接程序完成的**静态链接**，也可以**是**程序执行过程中由于需要而进行的**动态链接**。每个进程都有这么一个空间。每个指令或数据单元都在这个虚拟空间中有确定的地址，这个地址称为**虚拟地址**。

    （进程在虚拟地址中可以是非连续的）

    实际物理地址由**虚拟地址到实际物理地址的变换得到。**

    虚拟存储器：进程中的目标代码、数据等的**虚拟地址组成的虚拟空间**
    - 不考虑物理存储器的大小和信息存储的实际位置，**只规定每个进程中互相关联的信息的相对位置**
    - 每个进程都有各自的虚拟存储器
3. 地址变换：

    内存空间：内存地址的集合
    - 在内存中，每一个存储单元都与相应的内存地址编号相对应
    - 内存空间是一维线性空间

    虚拟地址划分：与系统结构有关

    地址映射（地址重定位）：把虚拟空间中已链接和划分好的内容装进内存
    - 静态地址重定位

      在虚拟空间程序执行之前由装配程序完成地址映射工作

      只完成首地址不同的连续地址变换

要求所有待执行的程序必须在执行之前完成它们之间的链接

不需要硬件支持

不能实现虚拟存储器

需要占用连续的内存空间

- 动态地址重定位

  在程序执行过程中，在 CPU 访问内存之前，将要访问的指令或数据地址转换为内存地址

  内存地址为基地址寄存器 BR 和虚拟地址寄存器 VR 的加和

  可以对内存进行非连续分配

  提供了实现虚拟存储器的基础

  有利于程序段的共享

4. 内外存数据传输的控制

- 覆盖：用户程序控制内外存数据交换

  不能实现虚拟存储器

- 操作系统控制：

  1）交换：

  由操作系统把那些在内存中处于等待状态的进程换出内存，而将等待条件满足而进入就绪态的进程还如内存

  2）请求调入和预调入：（能够实现虚拟存储器）

  请求调入：在程序执行时，如果所要访问的程序段或数据段不在内存中，操作系统则自动从外存中将其调入内存

  预调入：操作系统预测在不久的将来会访问某些程序段或数据段，并在它们被访问之前选择合适的时机将它们调入内存

5. 内存的分配与回收：

存储管理模块为每一个并发执行的进程分配内存空间，并当进程结束之后及时回收该进程所占用的内存资源，以便给其他进程分配空间

- 分配结构：登记内存使用情况以及供分配程序使用的表格和链表
- 放置策略：确定调入内存的程序和数据在内存中的位置（选择内存空闲区的方法）
- 交换策略：确定将内存中的某些程序段和数据段移除内存
- 回收策略：回收的时机以及对空闲区的调整

6. 内存信息共享与保护：

   内存信息保护方法：

   - 硬件保护法：

     上下界保护法是常用的一类方法，为每个进程设置一对上下界寄存器，其中装有被保护程序和数据段的起始地址和终止地址。

   - 软件保护法：

     保护键法是常用的一类方法，在程序状态字中设置相应的保护键开关字。

   - 软硬件结合法：（UNIX）

     界限寄存器与 CPU 的用户态/核心态工作方式相结合。核心态进程可访问所有内存地址空间，用户态只能访问界限寄存器界定的内存空间。

7. 分区存储管理：（硬件支持少，内存利用率不高，各区之间不能信息共享）

   概念：将内存分为若干大小不等的区域，除操作系统占用一个区域外，其他由并发进程所共享。（**一般与覆盖和交换技术一起使用以扩充内存**）

   原理：**给每一个内存中的进程划分一块适当大小的存储区**，以连续存储各进程的程序和数据，使各进程能够并发执行。

   - **固定分区法：**

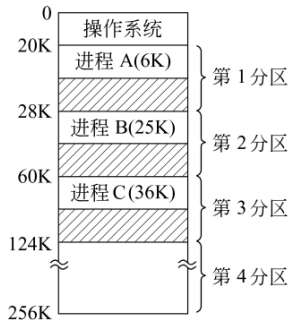
     把内存固定的划分为若干个大小不等的区域。

     分区一旦确定，则在执行过程中每个分区的长度和内存的总分区个数保持不变。

     通过**分区说明表**管理和控制内存分区。

区号	分区长度	起始地址	状态
1	8K	20K	已分配
2	32K	28K	已分配
3	64K	60K	已分配
4	132K	124K	未分配

(a) 分区说明表



(b) 内存状态

   - **动态分区法：**

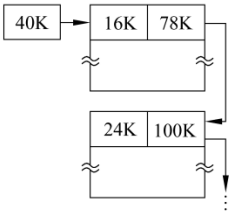
     分区的建立是在作业的处理过程中进行，且其大小随作业或进程对内存的要求而改变。提高了内存的利用率。

在系统初启时，除了操作系统中常驻内存部分，只有一个空闲分区。随后分配程序将该区依次划分给调度选中的作业或进程。随着进程的执行，会出现一系列的分配和释放。

可以使用**可用表、自由链、请求表**等结构进行表示。

区号	分区长度	起始地址
1	16K	40K
3	24K	78K
5	9K	100K

(a) 可用表



(b) 自由链

作业(进程)号	请求长度
P_1	13K
P_2	20K
⋮	

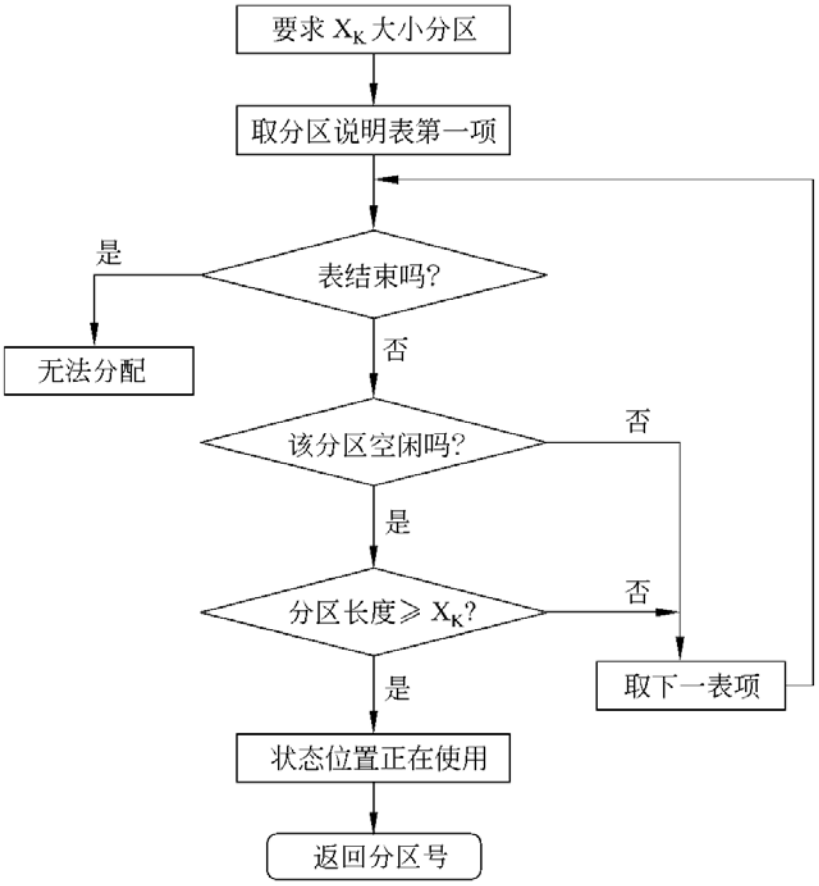
(c) 请求表

8. 分区的分配与回收：

- 固定分区时的分配与回收：

分配：当用户程序要装入执行时，**通过请求表提出内存分配**要求的内存空间大小。**存储管理程序**根据请求表**查询分区说明表**，从中**找到空闲分区进行**分配

（程序执行前，根据请求表查询可用表，找到空闲分区进行分配）



回收：进程执行完毕后，管理程序将对应的分区状态置为未使用

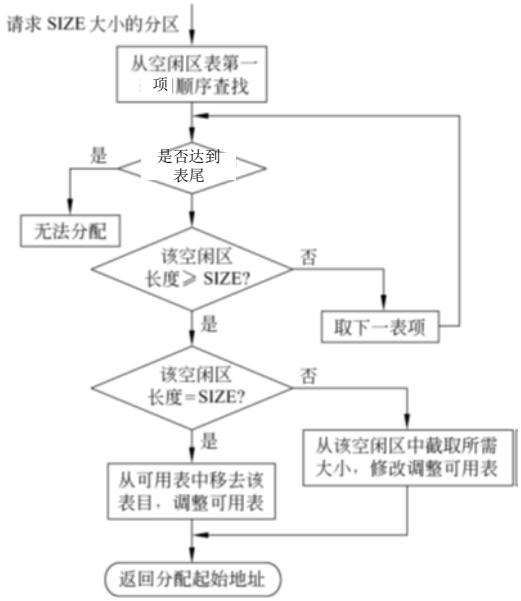
- 动态分区时的分配与回收：

分配：

I. 根据请求表，从可用表/自由链中寻找合适的空闲区进行分配：

- **最先适应算法：（搜索最优）**

  要求可用表/自由链按**起始地址递增**顺序排列。



- **最佳适应算法：（找到的空闲区是最佳的）**

  要求可用表/自由链按**内存空间递增**的次序组成可用表/自由链。

- **最坏适应算法：（不留下碎片空间）**

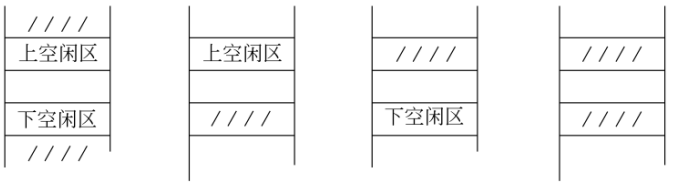
  要求可用表/自由链按**内存空间递减**的次序组成可用表/自由链。

II. 分配空闲区后更新可用表/自由链

回收：

进程执行完毕，存储管理程序回收已使用完毕的空闲区（释放区），并将其插入到可用表/自由链中。并将回收的空闲区进行拼接。

回收的空闲区与上下相邻区间的关系：



(a) 上下相邻区都是空闲区　(b) 上相邻区为空闲区　(c) 下相邻区为空闲区　(d) 上下相邻区都不是空闲区

拼接算法：相邻空闲区合并，其初始地址改为合并后的开始地址。

9. 覆盖技术：（可与分区技术一起使用）

   覆盖技术作用： 在多道环境下进行扩充内存

   思想：一个程序不需要一开始就把它的全部指令和数据装入内存后执行。

   将程序划分为若干个功能上相对独立的**程序段**，按照程序的逻辑结构让那些不会同时执行的程序段共享同一内存区。**程序段保存在外存中，当有关程序段的先头程序段执行结束后，再把后续程序段调入内存覆盖前面的程序段。**从而达到扩大内存的目的。

10. 交换技术：（可与分区技术一起使用）

    交换技术作用： 在多道环境下进行扩充内存，用在小型机或微机系统中

    先将内存某部分的程序或数据写入外存交换区，再从外存交换区中调入指定的程序或数据到内存中并执行。（不要求覆盖结构）

    分为换出（swap out）和换入（swap in）两部分：

    换出：将内存中的数据和程序换到外存交换区

    换入：将外存交换区的数据和程序换到内存中

11. 页式管理：

    目的：减少碎片；只在内存中存储那些反复执行或即将执行的程序段与数据部分，而将那些不经常执行的程序段或数据部分存放在外存中，待执行时调入，以提高内存利用率

    原理：各进程的虚拟空间被划分为若干个长度为相等的页（Page）,从而进程的虚地址变为页号 P 与页内地址 W 所组成。同时，页式管理也把内存空间按页的大小划分为**片或页面（page frame）。**这些页面为所有进程所共享。

    - 分页管理时，**用户进程**在内存空间内除了在**页面内地址连续**，每个**页面之间不再连续**。
    - 碎片减少
    - 非连续存储

    地址映射：页式管理将页式虚地址与内存页面物理地址建立映射关系，并用相应的硬件地址变换机构解决离散地址变换问题

12. **静态页式管理**：在作业/进程开始执行之前，将该作业/进程的程序段和数据全部装入内存的各个页面中，并通过页表（page mapping table）和硬件地址变换机构实现虚拟地址到内存物理地址的地址映射。

- 内存页面分配和回收：

i. 页表：

由页号和页面号组成。在内存中占有固定的存储区。在页式管理中，每个进程至少有一个页表。连续的页号对应不连续的页面号。

ii. 请求表：

用来确认作业/进程的虚拟空间中的各页在内存中的实际位置。整个系统需要一张请求表

iii. 存储页面表：

整个系统一张，指出内存各页面是否已被分配出去以及未分配页面总数

- **位示图**表示法
- **空闲页面链**表示法

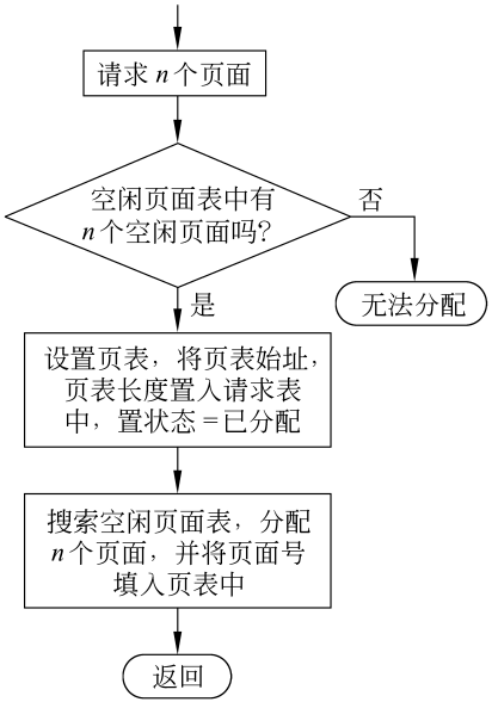
- **页面分配算法：**

1） 请求表给出进程/作业所需的页面数

2） 存储页面表查看是否有空闲页面，若没有空闲页面则无法分配；若有，则首先分配设置页表，并填写请求表中的相应表项，按一定的查找算法搜索出所需的空闲页面，并将对应的页面号填入页表中，

- **页面回收算法：**

进程执行结束后，拆除相应的页表，并将页表中的各页面插入存储页面表中。



- **地址变换**：怎样由页号和页内相对地址变换到内存物理地址

  系统将所调度执行的**进程页表起始地址和长度**从**请求页表**中取出放入**控制寄存器**中。

13. **动态页式管理**：

定义：只将常用的工作区部分的代码和数据装入内存，其他部分则在执行时动态载入。主要分为请求页式管理和预调入页式管理，这里主要介绍**请求式页式管理**

请求式页式管理：在执行到某条指令需要从外存中调入内容时再调入到内存

需解决的问题：如何发现这些不在内存中的虚页(断页)，以及如何处理

发现断页：使用扩充页表，增设中断位（判断该页是否在内存中）、外存始址（该页在外存中的副本的起始地址）和改变位（判断该页的内容是否改变<改变的淘汰页需要写回外存>）

页号	页面号	中断位	外存始址	改变位

Fig: 扩充页表

14. 动态页表的管理流程：

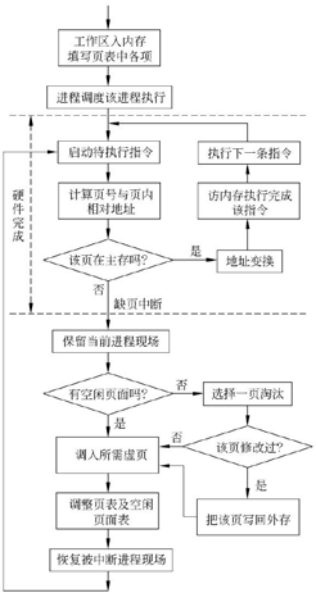


Fig:请求式动态管理

15. 请求页式管理中的置换算法：

适用场景：在内存中没有空闲页面时，选出一个被淘汰的页面。

置换：置换出那些访问概率最低的页，将它们移除内存。

1) **随机淘汰算法：**

在无法确定哪些页访问概率较低时，使用随机淘汰的策略

2) **轮转法**（round robin）和**先进先出**（FIFO）算法

RR：循环换出内存中一个可以被换出的页，无论该页换进内存多长时间。

FIFO: 换出在内存中待的时间最长的页

<缺点> **内存利用率不高：**CPU 很多时候不是线性的访问地址空间，有些(长时间待在内存中的)页面的访问概率依旧很高。

Belady 现象:分配的页面数增多，但缺页次数反而增多（FIFIO 的缺点）

Belady 现象产生原因：未考虑程序执行的动态特性

3) 最近最久未使用（least recently used/LRU）页面置换算法

基本思想：当需要淘汰某一页时，选择离当前时间最近的一段时间内最久没有使用过的页先淘汰。

近似 LRU 算法：

a) 最不经常使用(least frequency used/LFU)页面淘汰算法

淘汰到当前时间为止被访问次数最少的那页

b) 最近没有使用(Not Used Recently/NUR)页面淘汰算法

淘汰最近一个时期内未被访问的页中任选一个

16. **页式管理的存储保护:**

a) 地址越界保护：

通过比对控制寄存器中的页表长度值和要访问的虚地址进行保护

b) 存取控制保护：

在页表中增加相应的保护位

17. **页式管理的优缺点：**

a) 优点：

解决碎片问题； 提供了内存和外存统一管理的虚存实现方式，使得用户可利用的存储空间得到大大的增加

b) 缺点：

需要相应的硬件支持； 增加了系统开销； 可能出现抖动现象（页面的来回调入调出内存）； 每个作业/进程的最后一页总有一部分空间得不到利用。

18. **段式与段页式管理：**

目的：为用户提供一个方便灵活的程序设计环境（动态链接）

思想： 把程序按内容或过程（函数）关系分成段，每段有自己的名字。一个用户作业/进程所包含的段对应于一个二维线性虚拟空间，段式管理程序以段为单位分配内存，然后通过地址映射机构将段式虚拟地址转换为物理地址。

原理：

a) **段式虚拟空间：**虚地址是一个二维结构，即段号 s 和段内相对地址 w。

段号不要求是连续的。

段的长度不是固定的。

每个段定义一组逻辑上完整的程序/数据。

每个段是一个首地址为 0 的，连续的 1 维线性空间。根据需要而动态的增长。

b) **内存分配和释放：**

以段为单位进行分配内存，每段分配一个连续的内存区。

分配的内存区大小不一。

内存区不要求连续。

动态的分配内存和释放内存。

c) **地址变换：**

段式管理只存放部分用户信息副本在内存中，大部分信息在外存中。

使用段式地址变换机构能够处理:

1)缺段情况 2)二维虚地址空间到一维物理地址空间转换

i. **段表：**

通过段表进行管理内存：

段号	始址	长度	存取方式	内外	访问位

Fig: 段表

其中，

“段号”：与用户指定的段名一一对应。

“始址”：该段在内存或外存中的起始物理地址

“长度”：该段在内存或外存中的物理地址的长度

“存取方式”：对该段进行存取保护，只有处理机状态字中的存取控制位和段表中的存取方式一致时才能访问该段。

“内外”：该段现在存储在内存还是外存。如果在外存则发生中断

“访问位”：根据淘汰算法的需要而设的。（NUR）

ii. **动态地址变换：**

在内存中给出一块固定的区域存放段表。

当某段进程开始执行后时，管理程序将该**进程的段表始址**放到**段表地址寄存器**中。通过访问段表寄存器，管理程序得到该进程的段表始址。由虚地址中的段号 s 为索引，查段表。若该**段在内存**，则判断其存取方式是否有错。若存取方式正确，则从段表的相应表项中查出该段在内存中的始址，并与段表相对地址相加得到实际的物理地址。

若段**不在内存**中，则产生缺段中断，将 CPU 控制权交给内存分配程序

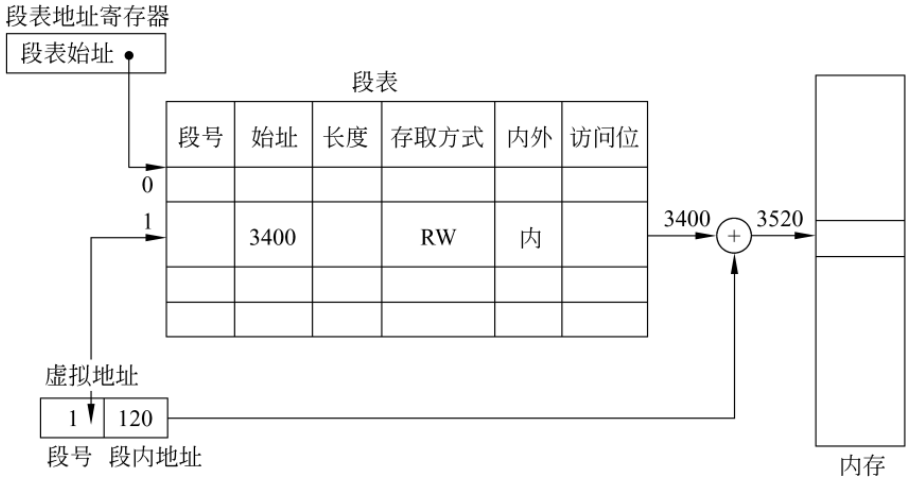


Fig:段式地址变换

以及缺段处理方法为：

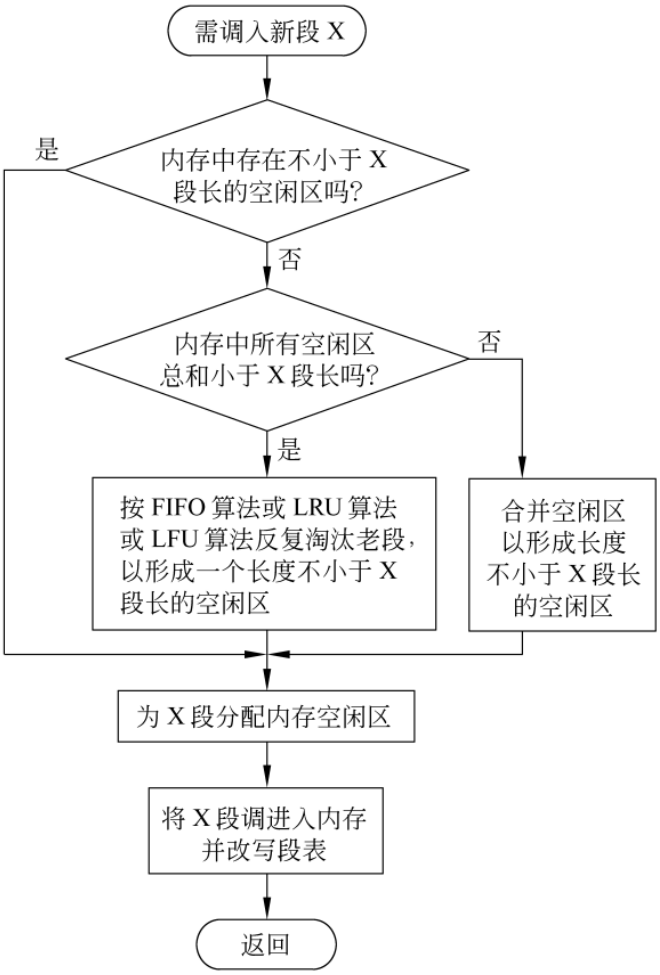


Fig:缺段处理方法

19. **段的共享和保护：**

段是按逻辑意义来划分的，可以按段名进行访问

共享：在内存中只保留一个(程序/数据)副本，供多个用户进行使用。

使用相同的段名，就可以在新的段表中填入已存在于内存之中的段的起始地址，并置以适当的读写权限，就可以共享内存段。

保护：地址越界保护 和 存取方式控制保护

20. **段式管理的优缺点：**

a) **优点：**

内外存统一管理的虚存实现；

每次交换有意义的段；

信息共享；

动态链接

b) **缺点：**

硬件支持

动态增长带来的开销

21. **段页式管理：**

结合段式管理和页式管理的优点，进行结合，但开销较大。

a) **虚地址**

一个进程拥有一个自己的**二维地址**。

一个进程中拥有独立逻辑功能的**程序/数据分为段**，并具有自己的段号 s。

对于**段**内的程序/数据，按照一定的大小**划分**为不同的**页**。

虚拟地址由：**段号 s，页号 p** 和**页内相对地址 d** 组成。其中对编程人员可见的是段号 s 和段内相对地址 w。w=p+d 得到。

最小单位：**页**

22. **段表和页表：**

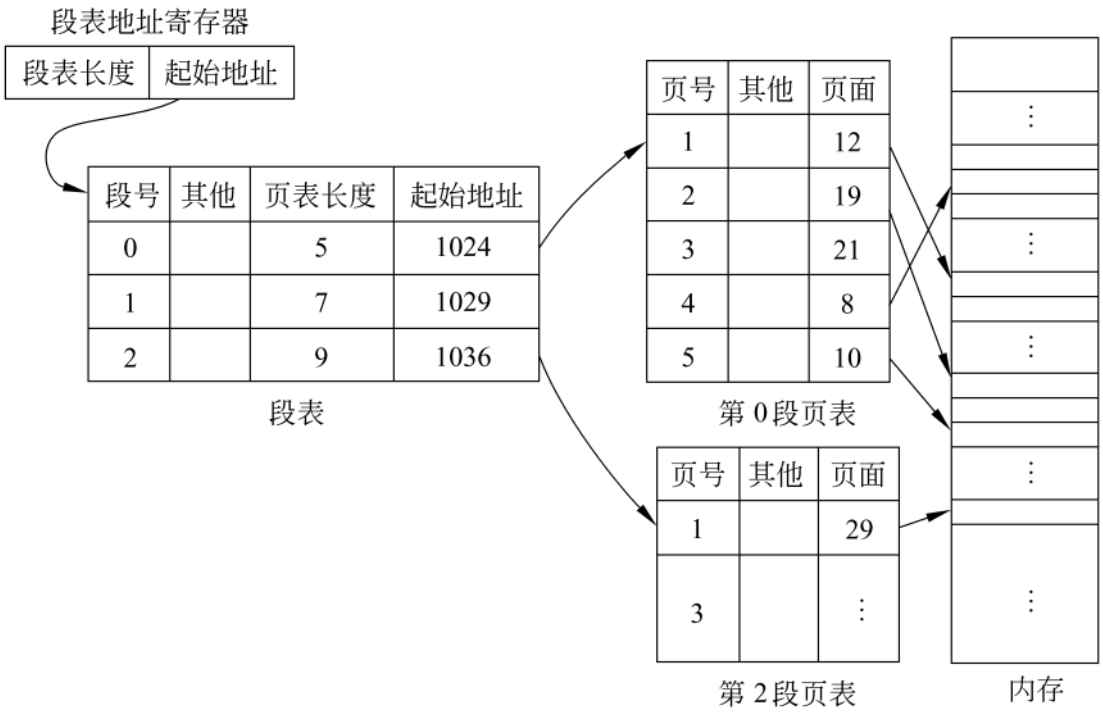


Fig:段表和页表

每个作业/进程有一个段表，每个段包含多个页，即建立多个页表。

23. **动态地址变换过程：**

在内存中开辟一块固定的区域存放段表和页表。

对内存中的指令/数据进行一次存取，需要访问 **3 次以上**内存。

i. 段表地址寄存器得到段表始址**访问段表**，取出该段的页表地址

ii. 访问页表得到所要访问的物理地址

iii. 在访问段表和页表之后，真正访问物理单元

为解决 3 次访问带来的速度较慢的问题，设置**高速联想寄存器**。将当前最常用的段号 s，页号 p 和对应的内存页面放入高速联想寄存器。

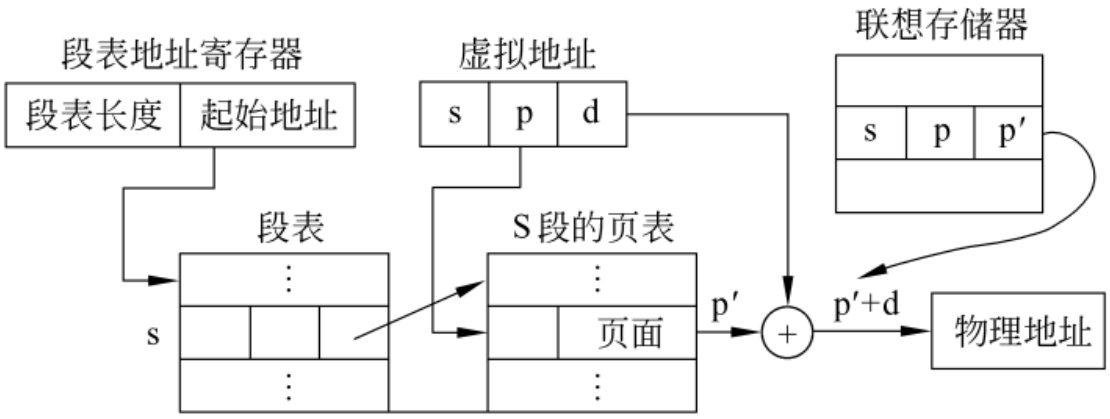


Fig:段页式管理

24. **局部性原理：**

在几乎所有的程序执行中，**在一段时间内，CPU 总是集中的访问程序中的某一部分。**

六、进程与存储管理示例

1. Linux 进程和存储管理简介：

1.1 安装过程：

为了使 OS 内核能够在每次开机都能装入内存，用户必须事先将 LINUX 操作系统的执行代码以文件形式存储到 PC 硬盘设备上，并对 PC 系统的相应资源进行分配（引导程序、交换区等）

1.2 进程间关系：

0 号进程（idle 进程）：OS 核心加载到内存并初始化，即建立静态进程**<核心态>**

1 号进程（init 进程）：用来控制终端进程和 SHELL 进程

除了 0 号进程之外其他的进程都是从 init 进程衍生出来的**<核心态/用户态>**

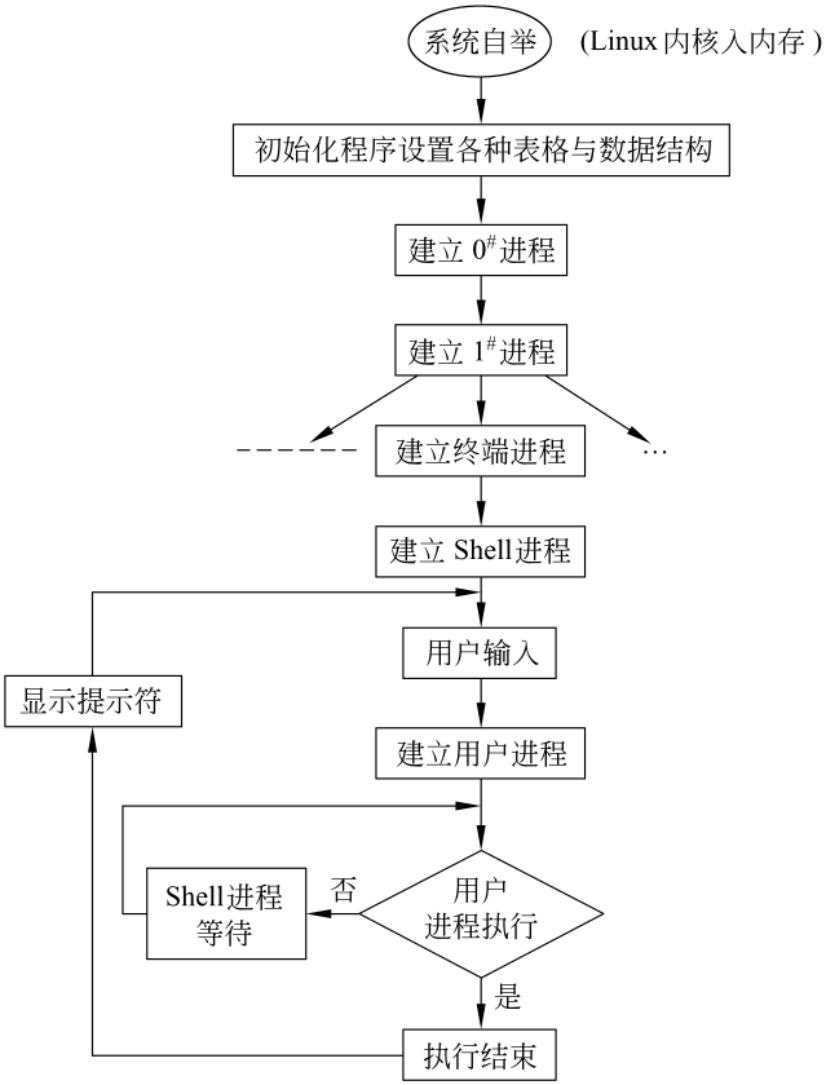


Fig:各进程之间的关系

1.3 **核心态和用户态：**

目的：用来达到操作系统和用户进程之间的安全隔离

用户态堆栈：在用户态下使用的堆栈区

核心态堆栈：在系统调用下使用的堆栈区

系统调用：进程通过请求操作系统进入核心态的机制

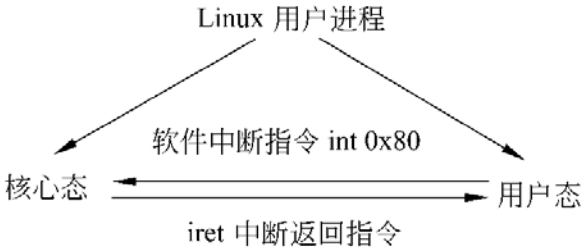


Fig:核心态和用户态之间的转换

核心线程：只在核心态下执行的进程

进程控制系统：由 4 个模块组成：

- 与文件系统的接口部分
- 进程本身的控制部分
- 进程间控制部分
- 存储管理部分

2. LINUX 进程结构：

LINUX 进程：

- 一个进程是对一个程序的执行（**动态特性**）
- 一个进程的存在意味着一个 task_struct 结构(包含进程控制信息)（**静态特性**）
- 一个进程可以生成&消灭子进程
- 一个进程是获得和释放各种系统资源的基本单位

3. **LINUX 进程控制段：**

七、Windows 的进程与内存管理

八、文件系统

九、设备管理

十、Linux 文件系统

十一、Windows 的设备管理和文件系统

十二、嵌入式操作系统简介