第七章 进程控制

控制进程上下文的系统调用包括三类：

1、与存储相关的

fork exec brk

2、与同步相关的

exit wait signal kill

3、与属性相关的

setpgrp setuit

7.1 创建进程 fork

pid = fork( )

fork算法主要完成以下操作：

1、检查系统资源，为新进程在核心进程表proc表中分配一个空表项。

2、为子进程分配一个唯一的进程标识数PID，确认用户的总进程数没有超限。

3、拷贝父进程的上下文到子进程中，包括真正用户标识号、有效用户标识号、进程组号、调度优先权等。初始化子进程中的相关参数，如初始优先权数、初始CPU使用时间、计时域等。调整文件的引用计数等。

4、增加与该进程相关的file表项和活动inode表项的引用计数。

5、对父进程返回子进程的进程号；对子进程则返回0。

算法 fork

输入：无

输出：对父进程是子进程的PID

对子进程是0

{

检查可用的核心资源（如内外存空间、页表等）；

取一个空闲的进程表项和唯一的PID号；

检查用户没有过多的运行进程；

将子进程的状态设为“创建”状态；

将父进程的进程表项中的数据拷贝到子进程的进程表项中；

当前目录的索引节点和改变的根目录的引用计数加一；

系统打开文件表中相关表项的引用计数加一；

在内存中作父进程上下文的拷贝（包括u区、正文、数据、堆栈）；

算法 fork （续）

```
在子进程的系统级上下文中压入虚设的系统级上下文层
/*  虚设的上下文层中含有使子进程能够识别自己的数据，
 *  并使子进程被调度时从这里开始
 */
if (正在执行的进程是父进程)
{
        将子进程的状态设置为“就绪”状态；
        return（子进程的PID）；        /* 从系统态到用户态 */
}
else    /*  当前正在执行的进程是子进程  */
{
        初始化u区的计时域；
        return（0）；        /*  从系统态返回到用户态 */
}
}
```

实例1：双进程文件拷贝

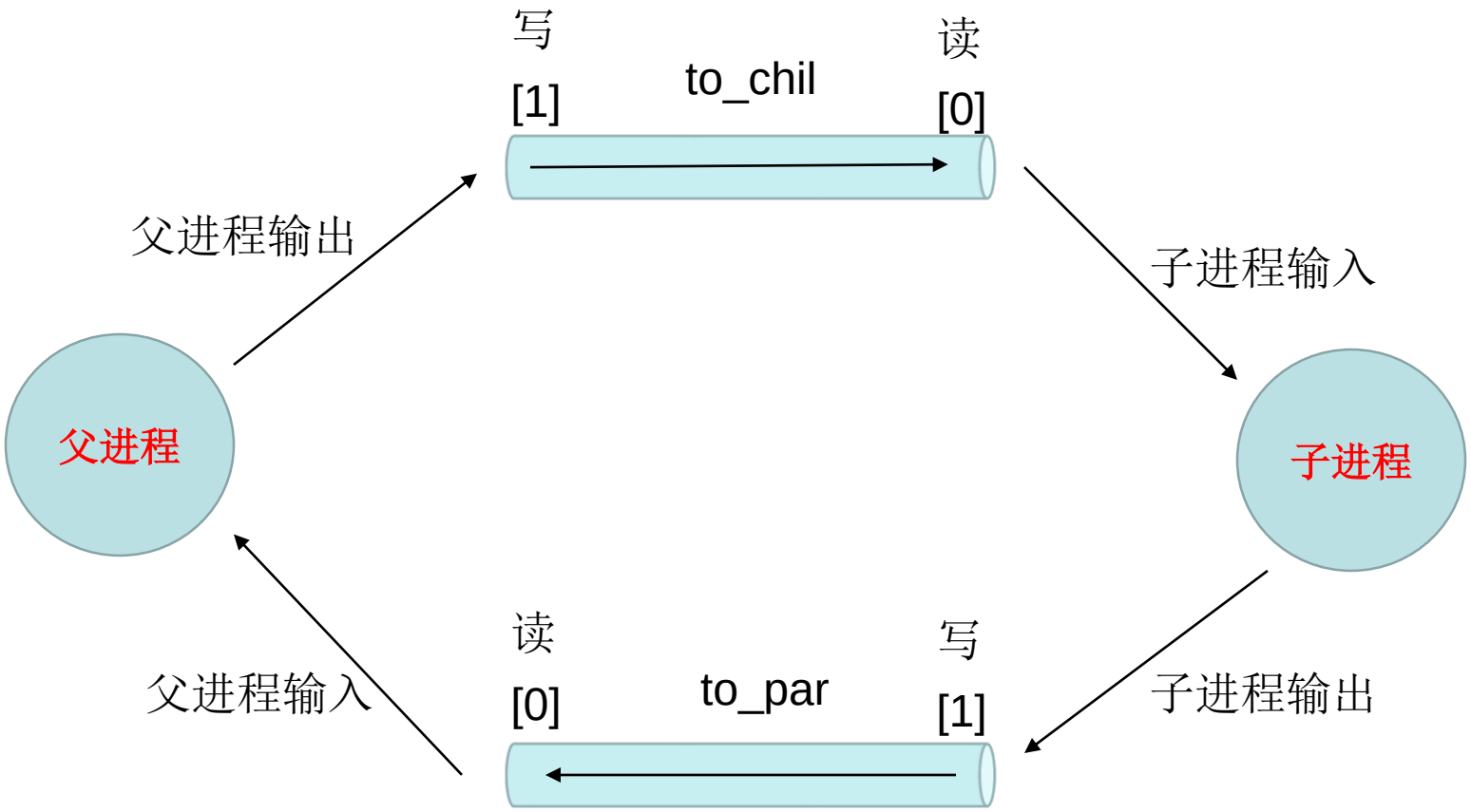
两个进程共享读指针和写指针，但运行的结果并不能保证新文件中的内容与老文件中的完全相同，这取决于父子两个进程的调度顺序。

—— 多进程环境下的数据一致性问题。

```
main(int argc, *argv[ ])
{
        fdrd = open(argv[1], O_RDONLY);
        fdwt = create(argv[2], 0666);
        fork( );
        /* 父子进程执行下面相同的代码 */
        rdwrt( );
        exit(0);
}

rdwrt( )
{
        for(; ;)
        {
                if(read(fdrd, &c, 1) != 1)
                        return;
                write(fdwt, &c, 1);
        }
}
```

实例2：确保数据一致性的双进程数据传输



在本例中，父子进程分别把自己的标准输入和标准输出重新定向到to_chil和to_par管道中了，无论这两个进程执行相应的系统调用的顺序如何，两个管道中的数据顺序不会发生变化，因此两个进程最终的读写结果不会改变。

```
#include <string.h>
char string[ ] = “hello,world”;
main( )
{
    int count, i;
    int to_par[2], to_chil[2];          /*  到父、子进程的管道  */
    char buf[256];
    pipe(to_par);
    pipe(to_chil);
    if (fork( ) == 0)
    {     /*  子进程从此处开始运行  */
        close(0);                       /*  关闭老的标准输入  */
        dup(to_chil[0]);                /*  把管道to_chil的读指针复制到标准输入  */
        close(1);                       /*  关闭老的标准输出  */
        dup(to_par[1]);                 /*  把管道to_par的写指针复制到标准输出  */
        close(to_par[1]);               /*  关闭不再需要的管道描述符  */
        close(to_chil[0]);
        close(to_par[0]);
        close(to_chil[1]);
        for (; ;)
        {
            if((count=read(0, buf, sizeof(buf)) == 0)
                exit();
            write(1, buf, count);
         }
```

```
    /* 父进程从此处开始运行 */
    close(1);           /* 重新设置父进程的标准输入和输出 */
    dup(to_chil[1]);
    close(0);
    dup(to_par[0]);
    close(to_chil[1]);
    close(to_par[0]);
    close(to_chil[0]);
    close(to_par[1]);
    for (i=0; i<15; i++)
    {
        write(1, string, strlen(string));   /* 每次向to_chil管道写11个字符 */
        read(0, buf, sizeof(buf));          /* 每次从to_par管道读256个字符 */
    }
}
```

7.2 软中断信号

1、软中断信号的作用：

通知进程发生了异步事件需要处理。

2、软中断信号的发送：

进程之间相互发送；

进程之内给自己发送。

3、发送软中断信号的方法：

kill 系统调用

4、软中断信号的分类：

（1）、与进程终止相关的软中断信号

例如进程退出时

（2）、与进程例外事件相关的软中断信号

例如地址越界、写只读内存区

（3）、在系统调用期间遇到不可恢复的条件相关的软中断信号

例如执行exec而系统资源已用完

（4）、在系统调用时遇到的非预测错误条件产生的软中断信号

例如调用不存在的系统调用、向无读进程的管道写数据

（5）、由在用户态下运行的进程发出的软中断信号

例如用kill向自己或其他进程发出的软中断信号

（6）、与终端交互有关的软中断信号

终端上按的break键或delete键等

（7）、跟踪进程执行的软中断信号

5、软中断信号的标识

给一个进程发软中断信号时，核心在接收进程的核心进程控制表proc中，按所要接收的信号类型设置软中断信号域中的某一位。

当该接收进程睡眠在一个可被中断的优先级上时，核心就唤醒该进程。

6、检查软中断信号的时间

（1）当一个进程即将从核心态返回到用户态时；

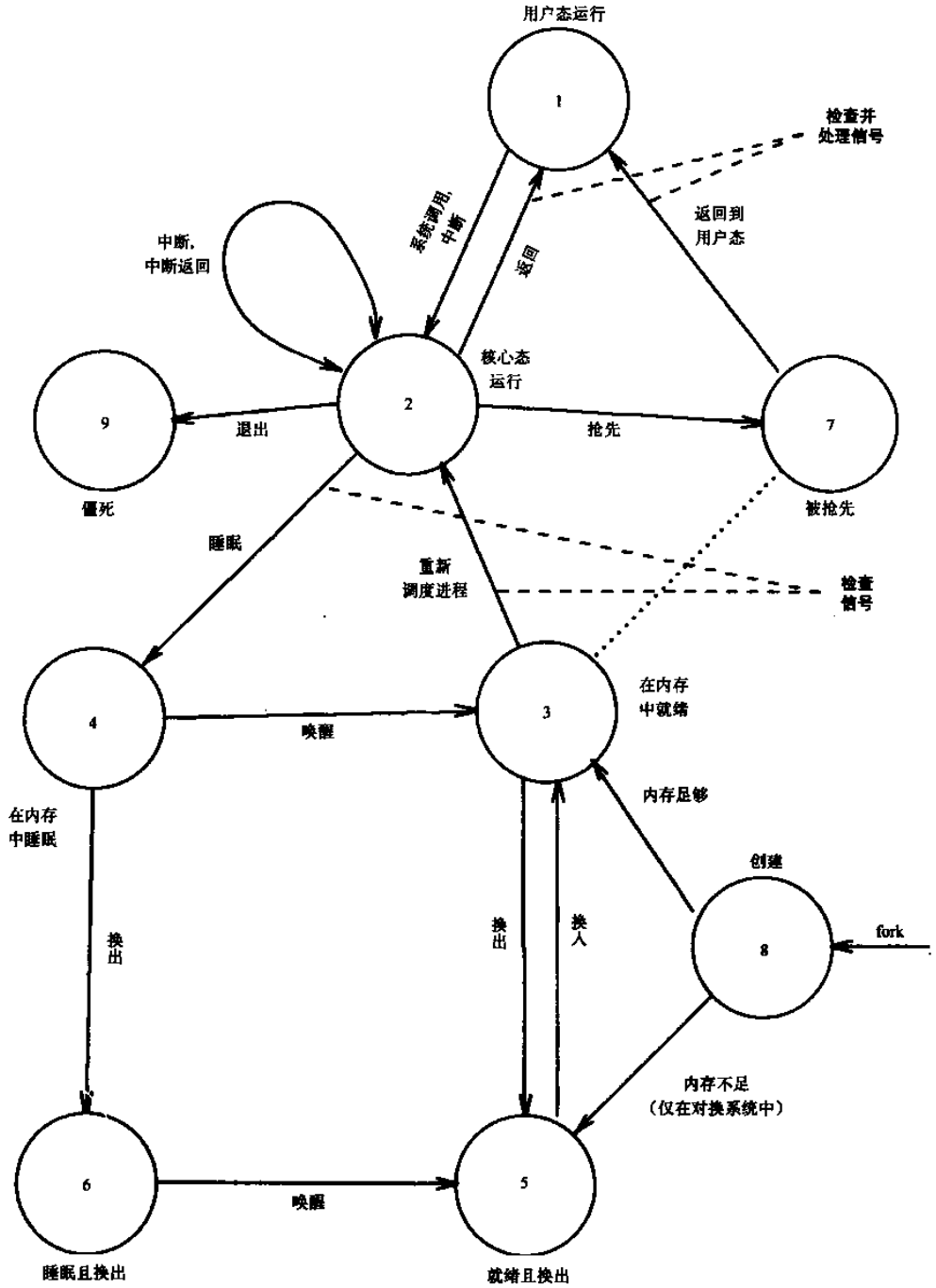
（2）当一个进程即将进入或退出一个较低调度优先级的睡眠状态时。

7、处理软中断信号的时刻

仅当进程从核心态返回用户态时才处理软中断信号。

由于软中断信号主要由应用程序运行时发出，不是影响系统正常运行的特别紧急事件，故不在核心态下处理——当进程在核心态下运行时，软中断信号并不立即起作用。

并且，核心也保证了进程在核心态下运行的时间不会太长，能够保证进程能很快在从核心态返回用户态时，对可能的软中断信号进行处理，从而确保对应用程序的响应时间要求。


用户态运行
1
检查并
处理信号
返回到
用户态
系统调用
中断
返回
中断，
中断返回
核心态
运行
2
9
退出
抢先
7
僵死
被抢先
睡眠
重新
调度进程
检查
信号
4
唤醒
3
在内存
中就绪
在内存
中睡眠
内存足够
创建
换
出
换
出
换
入
8
fork
内存不足
（仅在对换系统中）
6
唤醒
5
睡眠且换出
就绪且换出

```
算法 issg     /* 检查是否收到软中断信号 */
输入：无
输出：如果进程收到不忽略的软中断信号，则为真；否则为假
{
    while (proc表项中收到的信号的域不为0)
    {
        找出发送给进程的一个软中断信号号——是第几号中断；
        if (该信号是“子进程死”)
        {
            if (忽略“子进程死”信号)
                释放僵死的子进程的proc表项；
            else  if (捕俘“子进程死”信号)
                return (真)；
        }
        else  if (不忽略信号)
                return (真)；
        关掉proc表项中该信号域的（要忽略的）信号位
    }
    return (假)
}
```

8、软中断信号的处理

进程在确认收到软中断信号后，有三种处理方式：

（1）进程退出（exit）—— 缺省动作；

（2）进程忽略信号，就像没有收到该信号一样；

（3）进程收到信号后执行一个特殊的用户函数。通过系统调用signal来定义进程的动作。

oldfunction = signal (signum, function)

signum —— 软中断信号号

function —— 函数地址。

function=0 进程在核心态下退出（exit）

function=1 进程忽略以后出现的该信号

oldfunction —— 最近一次为信号signum所定义的函数的值

```
算法  psig        /*  识别出软中断信号后处理信号  */
输入：无
输出：无
{
    取进程表项中设置的信号号；
    清理进程表项中的该信号号；        /*  表明该信号已经被处理过了    */
    if (用户已调用signal来忽略该信号)      /*  即function=1    */
        return；      /*   完成   */
    if (用户指定了处理该信号的函数)
    {
        取u区中信号捕俘程序的虚地址；
        清除u区中存放信号捕俘程序地址的项；
        修改用户级上下文：创建用户栈来模拟对信号捕俘程序的调用；
        修改系统级上下文：将信号捕俘程序的地址写入用户保存的寄存器上下文中程序计数
器域；
        return；
    }
    if (信号类型要求转储进程的内存映像)
        在当前目录中创建core文件，将用户级上下文中的内容写入core文件中；
    立即调用算法 exit；
{
```

捕俘一个软中断信号的例子

```
#include  <signal.h>
main()
{
        extern  catcher();
        signal(SIGINT, catcher);
        ……
        kill(0, SIGINT);
}

catcher()
{
}
```

进程向自己及其所有子进程发送软中断信号

本例中程序捕俘interrupt软中断信号（SIGINT），并向自己发一个interrupt软中断信号——即系统调用kill的结果。

注意：当进程处理软了中断信号且在其返回用户态之前，核心要清除u区中含有用户软中断信号处理函数的地址的域。如果进程要再次处理该信号，它必须再次调用系统调用signal。

下面的例子演示了捕俘软中断信号时的竞争条件：

```
#include  <signal.h>
sigcatcher()
{
        printf("PID  %d caught one\n",  getpid());       /*  打印进程标识号  */
        signal(SIGINT,  sigcatcher);
}

main()
{
        int  ppid;
        signal(SIGINT,  sigcatcher);
        if (fork() == 0)
        {
                sleep(5);                       /*  为父子进程留足够的准备时间  */
                ppid = getppid();               /*  获取父进程的标识号  */
                for (; ;)
                        if (kill(ppid, SIGINT) == -1)
                                exit();         /*  父进程不存在了，kill出错返回  */
        }
        nice(10);               /* 降低优先权，增大出现竞争的机会  */
        for (; ;)
                ;
}
```

7.3 进程组

1、进程组标识号用于标识一组相关的进程，这组进程对于某些事件将收到共同的信号。

2、系统调用**setpgrp**初始化一个进程的进程组号，将组号设置为与该进程的进程标识号相同的值。

```
grp = setpgrp( )
```

其中**grp**为新的进程组号，在系统调用**fork**期间，子进程继承其父进程的进程组号。

7.4 从进程发送软中断信号

进程使用系统调用kill来发送软中断信号：

kill (pid，signum)

其中signum是要发送的软中断信号号，pid为软中断信号接收进程，pid的值与对应进程的关系如下：

1、pid为正值，信号发送给进程号为pid的进程；

2、pie为0，信号发送给与发送者同组的所有进程；

3、pid为-1，信号发送给真正用户标识号等于发送者的有效用户标识号的所有进程。

4、pid为负数但非-1，信号发送给组号为pid绝对值所有进程。

使用系统调用setpgrp的例子：

```
#include  <signal.h>
main( )
{
      register  int  i;
      setpgrp( );
      for ( i=0;  i< 10;  i++ )
      {
            if ( fork( ) == 0 )
            {
                  /*  子进程  */
                  if ( i & 1 )
                        setpgrp( );
                  printf("pid = %d  pgrp = %d\n", getpid( ), getpgrp( ));
                  pause( );   /*  挂起进程执行的系统调用  */
            }
       }
       kill (0, SIGINT);
}
```

UNIX系统中的信号定义

信号号	信号含义
0	正常的程序终止
1	挂断（拆线）
2	中断（break、delete、^C）
3	退出（FS字符，类似于break）
4	合法结构
5	跟踪陷进（被跟踪程序试图执行exec）
6	IOT结构
7	EMT结构
8	除浮点外
9	不可捕俘的kill信号
10	总线错误
11	溢出（地址越界）
12	系统调用时自变量非法
13	输出给管道，但管道没有接受者
14	警告
15	kill信号
16	用户定义的信号1
17	用户定义的信号2
18	子进程的僵死不应该被捕俘，否则引起管道问题
19	掉电

其中 2、3、15 号信号是常由人工发出的软中断信号

命令行上的信号捕俘和处理命令 trap

1、捕俘信号，并执行指定命令：

trap “command” signal1 signal2 signal3 …

例如：

trap “echo Receive a signal” 2 3 15

2、复位，恢复信号原有功能：

trap signal1 signal2 signal3 …

例如：

trap 2 3 15

trap命令应用实例：

1、建立具有root权限的turnoff用户，主目录：/home/turnoff，在.profile文件中设置一条sysoff命令
2、建立/bin/sysoff 命令，owner为root，sysoff程序如下：

```
trap  ‘ ‘  1  2  3  15
clear
echo “Would you want to shutdown the system?  (y/n)”
read  answer
if [ “$answer” = “y” ]
then
        sync
        shutdown
else
        clear
        kill  -9  0
fi
```

普通用户登录无口令的turnoff账号，即可完成关机任务。

7.5 进程的终止

UNIX系统中的进程都是执行系统调用exit来终止运行。进程进入僵死状态后，释放占用的资源，拆除进程的上下文，但保留进程的proc表项。

exit (status)

其中status是僵死进程发送给父进程的状态值。进程可以显式地调用exit，也可以在程序的结尾隐含地调用。

算法　exit

输入：给父进程的返回码

输出：无

{

忽略所有软中断信号；

if （ 是与控制终端关联的进程组组长）

{

向该进程组中的所有组员发送挂起信号；

将所有组员的进程组号设置为0；

}

关闭所有打开的文件；

释放当前工作目录；

释放改变的根目录（如果设置有“当前根”的话）；

释放进程占用的内存；

写记账记录；

设进程状态为僵死状态；

将所有子进程的父进程设置为1号进程（init）；

若有任何本进程的子进程僵死，则向init发送“子进程死”信号；

向父进程发送“子进程死”信号；

上下文切换；

}

不再接收任何中断信号了

告诉子进程准备结束运行

避免后续某新进程获得本进程释放的进程号，又成为新的进程组组长，从而与本进程组原来的组员混淆

将本进程及所有子进程的运行时间、内存、I/O累计到proc表项和全局记账文件中

使本进程从原进程树中断开，不再管理子进程了。

```
main( )
{
        int  child;
        if (( child = fork( ) == 0)
        {
                printf("child PID %d\n", getpid( ));
                pause( );
        }
        printf("child PID %d\n", child);
        exit(child);
}
```

exit的例子：

一个进程创建一个子进程。子进程打印自己的进程号后，执行系统调用**pause**挂起自己，直到收到一个软中断信号。

父进程打印子进程的进程号后退出，并返回子进程的**PID**作为状态码。

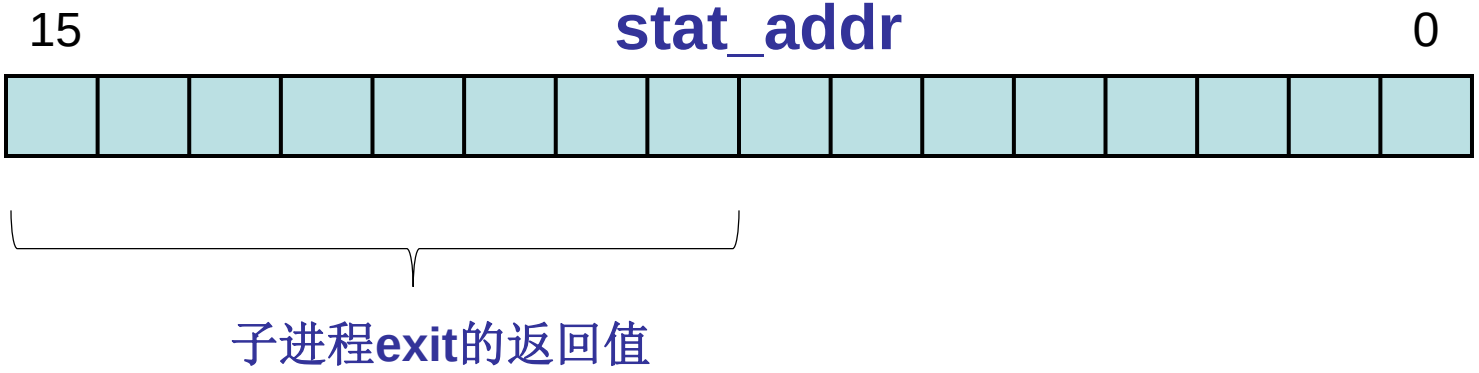
对子进程而言，尽管父进程已死，子进程仍可继续运行下去，直到收到软中断信号为止。

7.6 等待进程的终止

进程通过系统调用**wait**使自己的执行与子进程的终止同步：

pid = wait(stat_addr)

其中**pid**是僵死子进程的进程号，**stat_addr**是一个整数在用户空间的地址，它含有子进程的退出状态码，即**exit**的返回值。



算法 wait

输入：存放退出进程的状态的变量地址
输出：子进程标识号，子进程退出码

```
{
    if（本进程没有子进程）
        return（错误）；
    for（；；）
    {
        if（本进程有僵死子进程）
        {
            取任意一个僵死子进程；
            将子进程的CPU使用量累加到父进程；
            释放子进程的进程表项proc；
            return（子进程的进程标识号，子进程的退出码）；
        }
        if（本进程没有子进程）
            return（错误）；
        睡眠在可中断的优先级上；
    }
}
```

并且不忽略“子进程死”软中断信号

僵死子进程不会自己释放proc表项，见exit()

虽进入for前检测到有子进程，但要忽略“子进程死”信号

并无特定的睡眠等待事件，而是被“子进程死”软中断信号唤醒

等待并忽略子进程死软中断信号的例子

```
#include  <signal.h>
main(int argc, *argv[ ])
{
      int  i,  ret_pid,  ret_code;
      if ( argc > 1)
            signal(SIGCLD, SIG_IGN);    /* 忽略”子进程死”软中断信号/*
      for ( i=0;  i<15;  i++)
            if ( fork( ) == 0)
            {
                  printf(“child proc %x\n”, getpid( ));
                  exit (i);     /* 把当前的 i值(子进程顺序号)返回给父进程 */
            }
      ret_pid = wait (&ret_code);
      printf(“wait  ret_pid  %x  ret_code  %x\n”, ret_pid, ret_code);
}
```

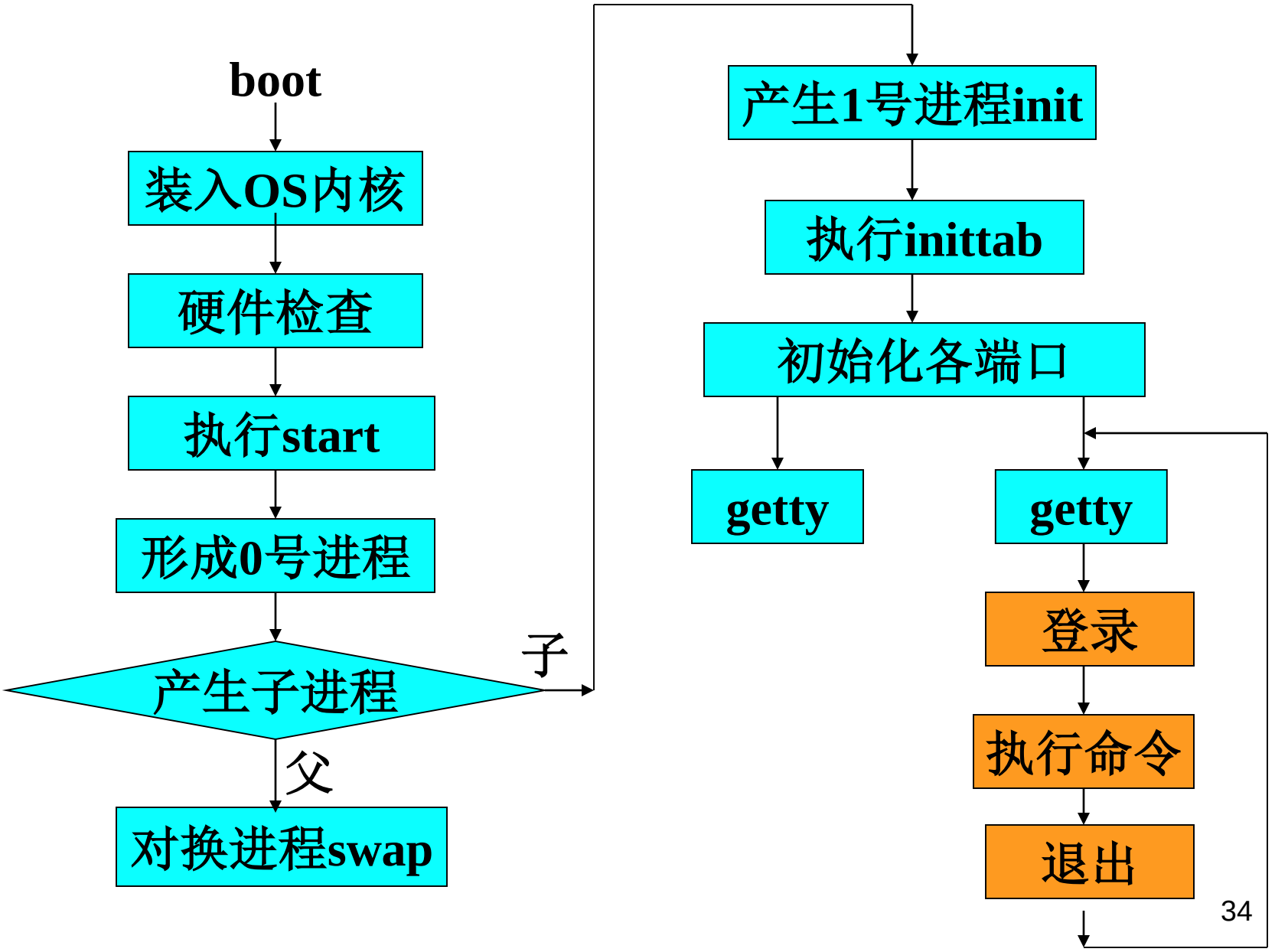
通过不同的运行参数的设置——确定进程对软中断信号的处理方式——影响进程间的同步关系和运行时序。

7.7 shell程序

```
/*  读命令行直到文件尾EOF  */
while ( read(stdin, buffer, numcnars))
{
      /* 分析命令行 */
      if (命令行中含有 &)
            amper = 1;
      else
            amper = 0;
      /* 对于非shell的内部命令，即是操作系统的命令 */
      if (fork( )  == 0)      /* 由子进程来执行命令 */
      {
            /* 是否为标准输入输出重定向？  */
            if(标准输出重定向)
            {
                  fd = creat(newfile, fmask);
                  close(stdout);
                  dup(fd);
                  close(fd);
            }
            if(建立管道)
            {
                  pipe(fildes);
```

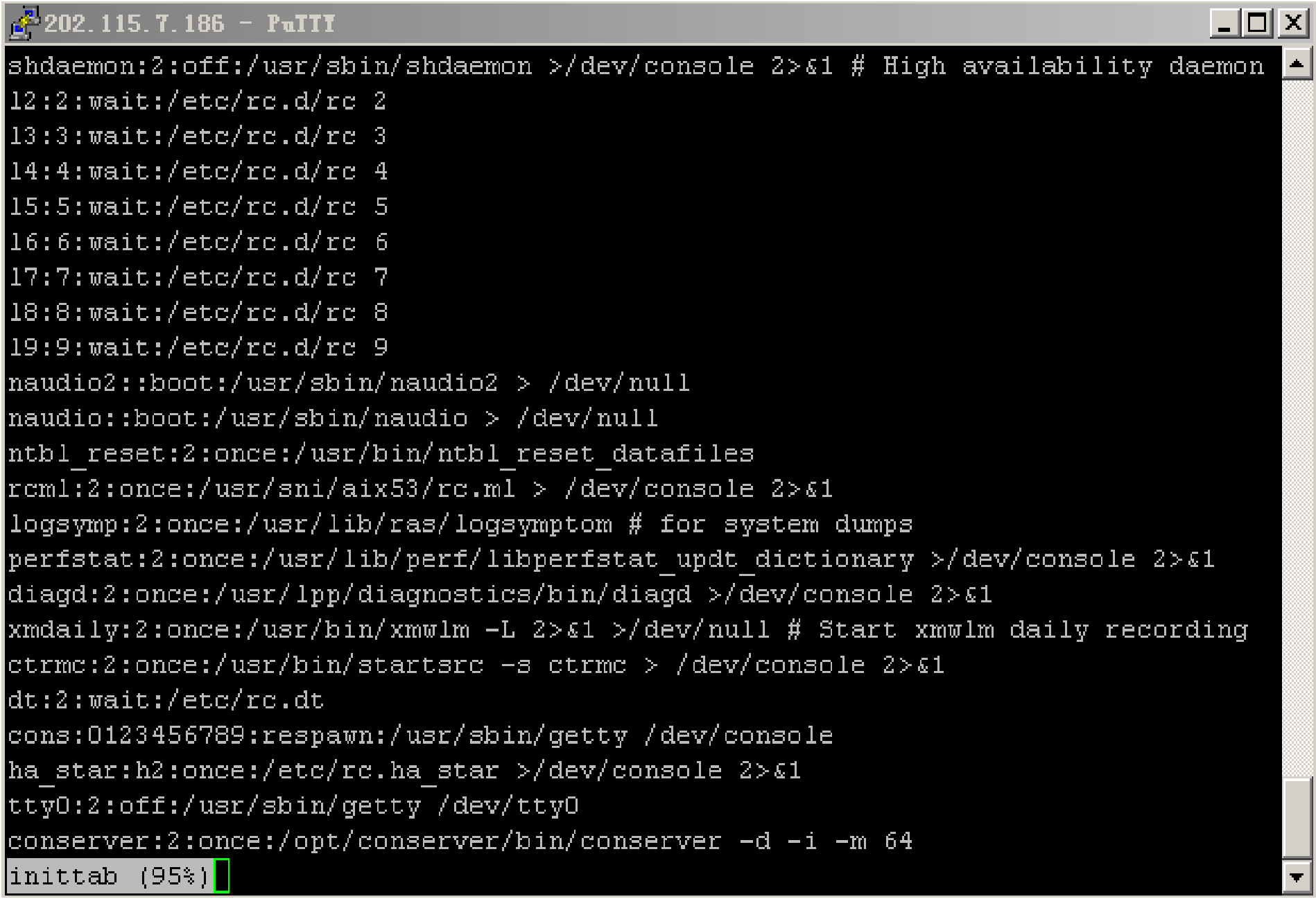
```
            if(fork( ) == 0)
            {
                /* 管道前面的命令，重定向标准输出到管道去 */
                close(stdout);
                dup(fildes[1]);
                close(fildes[1]);
                close(fildes[0]);
                /* 子进程执行命令 */
                execlp(command1, command1, 0);
            }
            /* 管道后面的命令，重定向标准输入从管道来 */
            close(stdin);
            dup(fildes[0]);
            close(fildes[0]);
            close(fildes[1]);
        }
        execve(command2, command2, 0);    /* 管道后的命令 */
    }
    /* 父进程shell从此处继续运行.....
     * 如果需要则等待子进程退出
     */
    if( amper == 0)
        retid = wait(&status);
} /* 回到while循环，重新读入下一个命令行 */
```

7.8 系统自举和init进程



```
算法  start      /* 系统初始化进程 */
输入：无
输出：无
{
    初始化全部核心数据结构，包括缓冲区、活动inode表、file表等；
    安装根文件系统；
    设置进程0的环境，初始化proc表和u区等；
    产生（fork）进程1；
    {
        /* 对于进程1 */
        为进程1分配地址空间；
        将用来执行init的代码（exec）从核心空间拷贝到用户空间，形成用户级上下文；
        改变状态：从核心态返回用户态；
        进程1调用exec执行/etc/init(模拟正常的系统调用)；
    }
    /* 对于进程0 */
    产生(fork)核心进程，执行系统服务；
    如果需要，调用对换程序(swapper)进行内外存交换；
}
```

init根据/etc/inittab文件来进行系统初始化，并产生其他进程。下面是inittab的样本：



```
算法 init        /* 系统1号进程 */
输入：无
输出：无
{
    fd = open("/etc/inittab", O_RDONLY);
    while (从文件中读入一行到缓冲区)
    {
        /* 读inittab的每一行 */
        if(调用状态 != 缓冲区中标定的状态)
            continue;
        /* 状态匹配 */
        if(fork( ) == 0)
        {
            调用exec执行缓冲区中规定的程序；
            exit( );
        }
        /* init进程不等待，继续while循环 */
    }
    while((id = wait((int *)0)) != -1);
    {
        检查如果是本进程派生的子进程死，则考虑是否需要重新派生该子进程；
        否则，继续while循环；
    }
}
```