第四章

进程管理

主 要 内 容

4.1 UNIX时钟中断与异常

4.2 UNIX系统调用

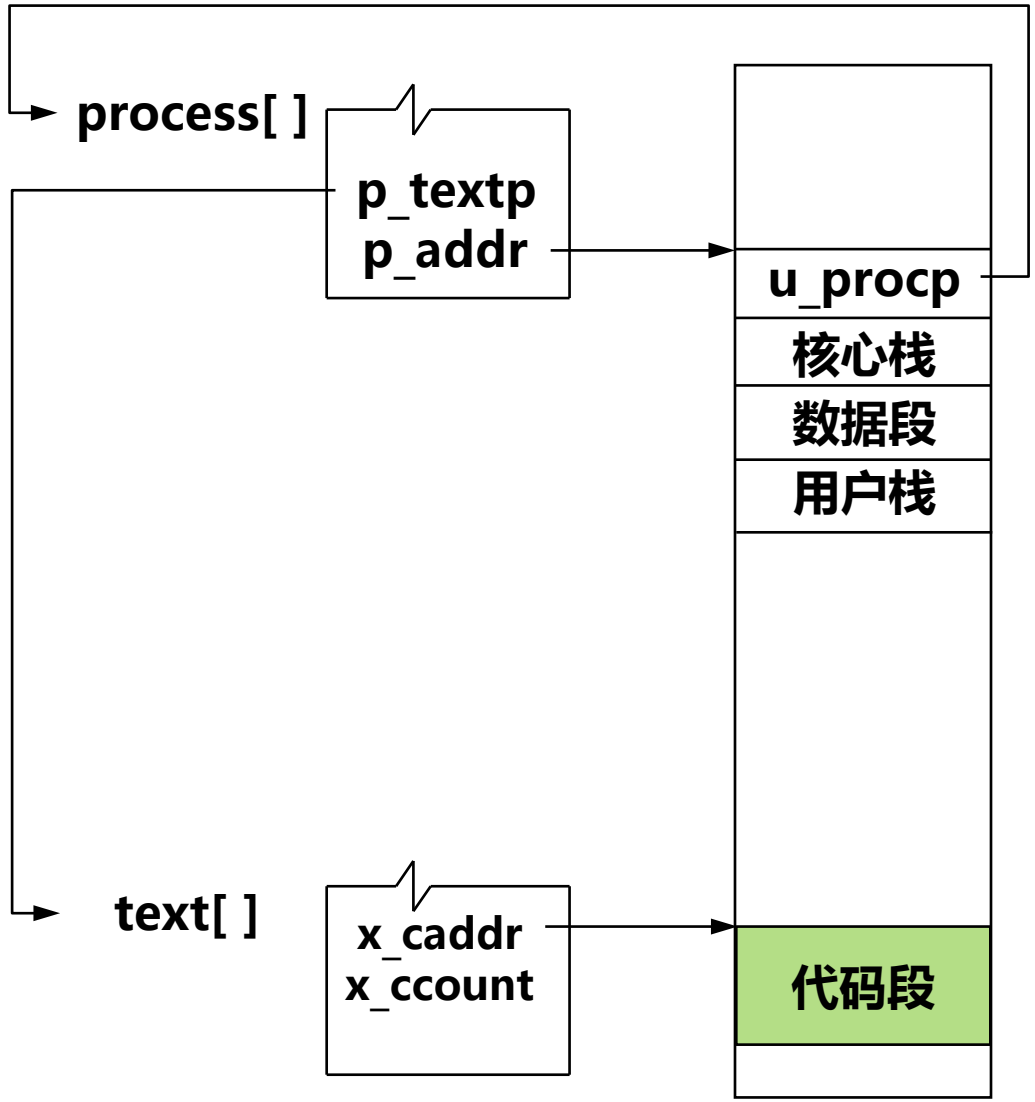
4.3 UNIX的进程调度状态

4.4 UNIX的进程调度控制

- 进程切换调度
- 进程创建与终止
- 进程图像交换

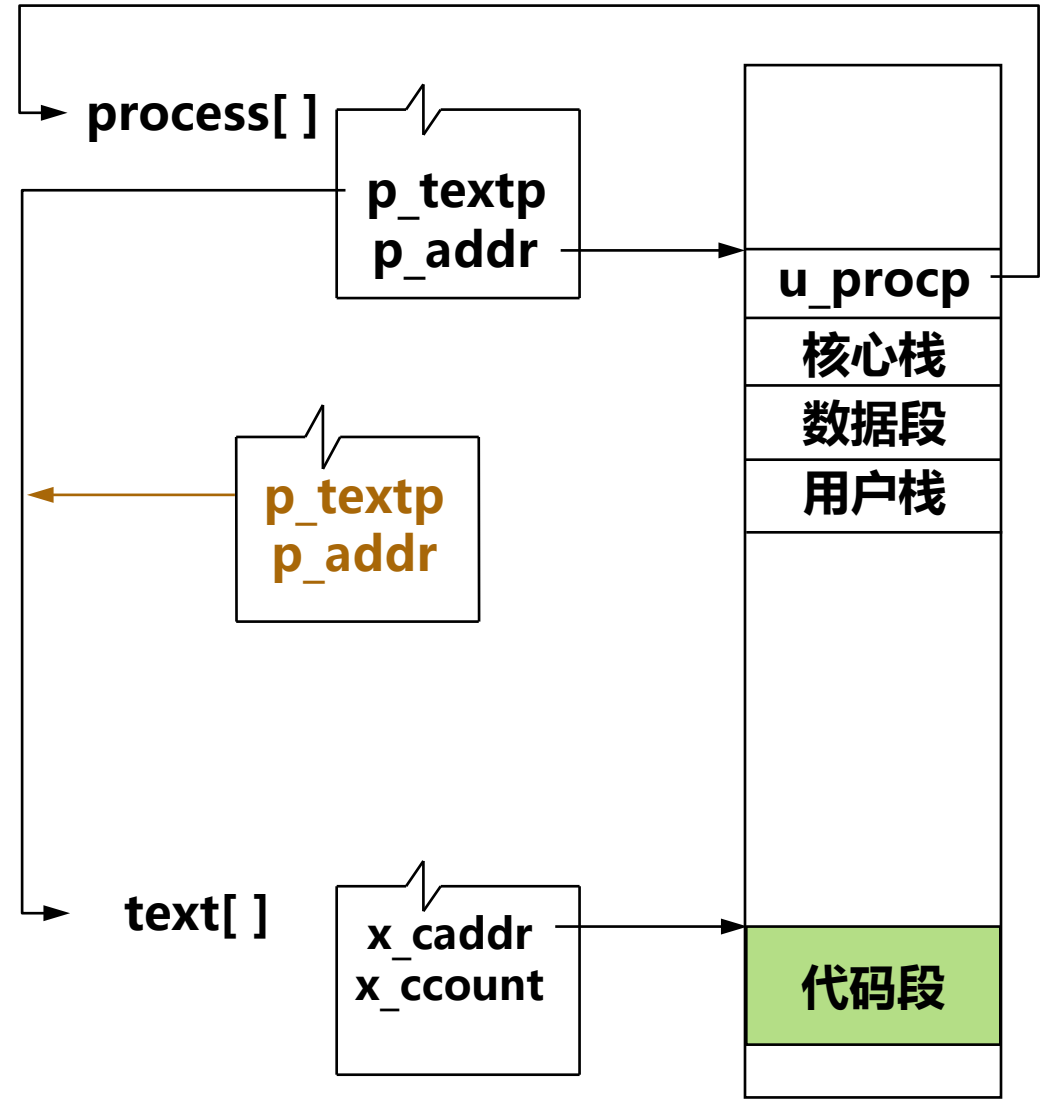
进程的创建与终止

创建子进程的关键：
为子进程构造一个能正确上台的图像



进程的创建与终止

创建子进程的关键：
为子进程构造一个能正确上台的图像

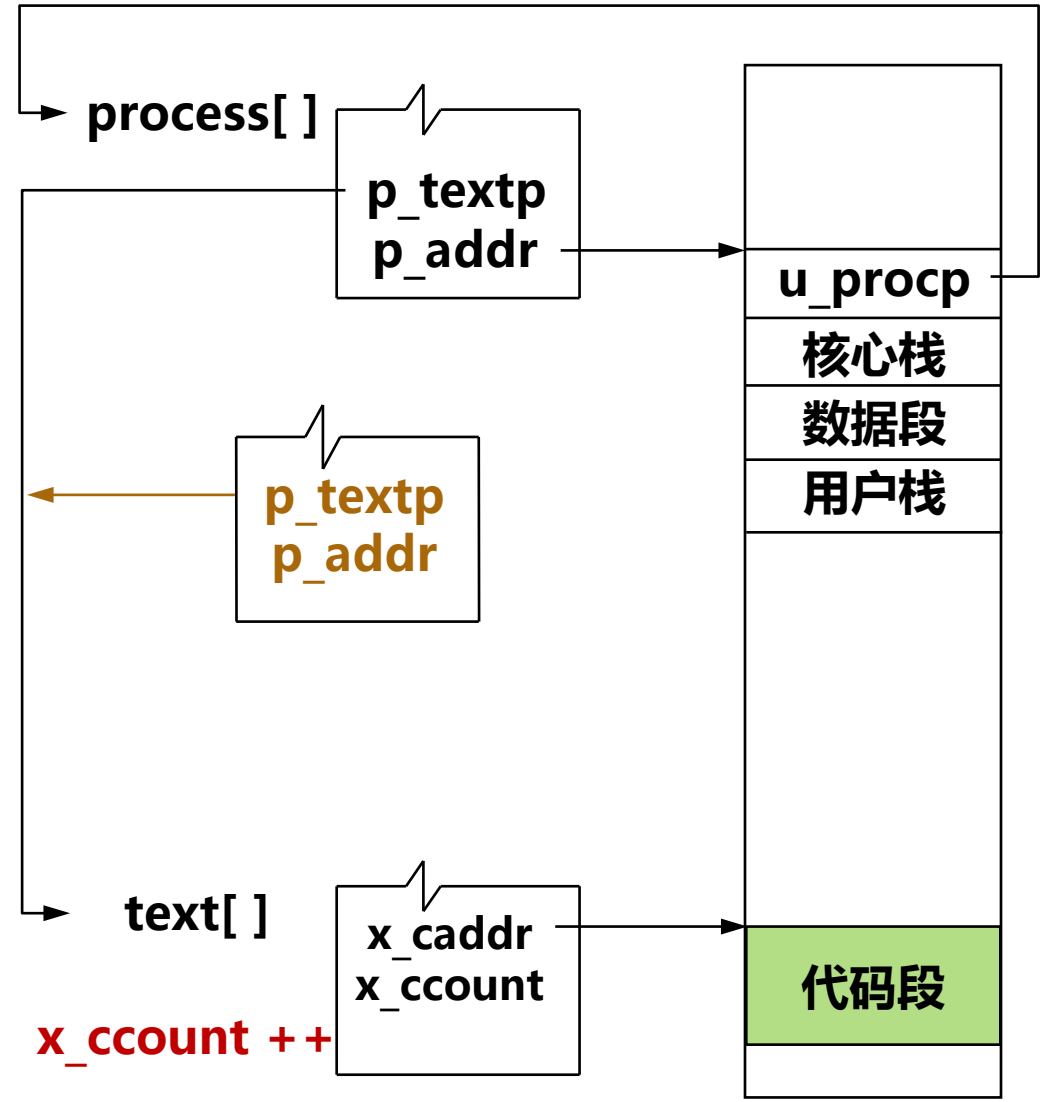


1

复制：p_size，p_stat，p_flag，p_uid，p_nice，p_textp
设置：p_pri = 0；p_time = 0；p_pid = 新标识数；p_ppid = 当前进程；

进程的创建与终止

创建子进程的关键：
为子进程构造一个能正确上台的图像



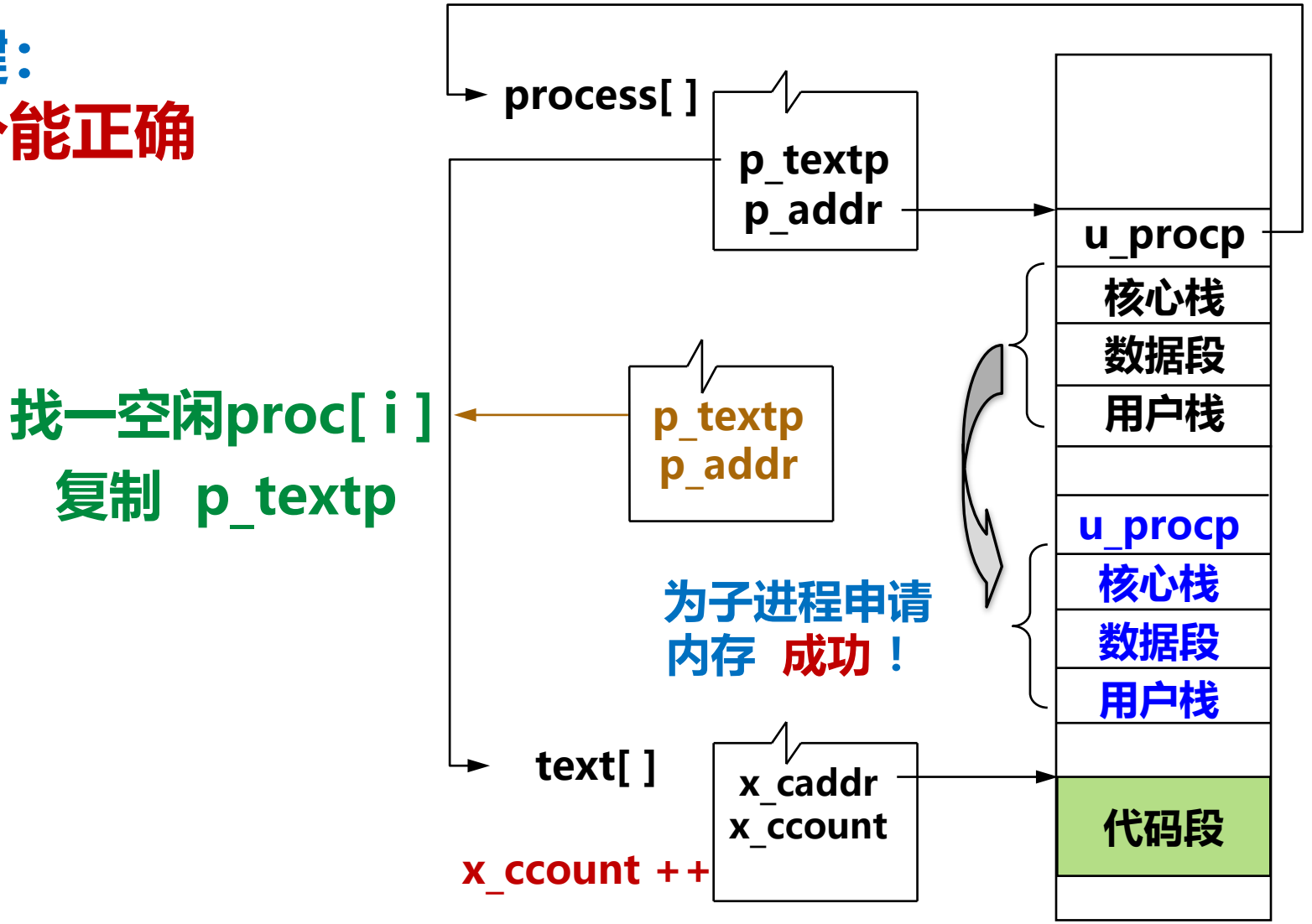
2

各种计数增1：

1. x_count，x_ccount（代码共享）
2. File结构引用数（文件共享）
3. u_cdir 指向的 i_count（目录共享）

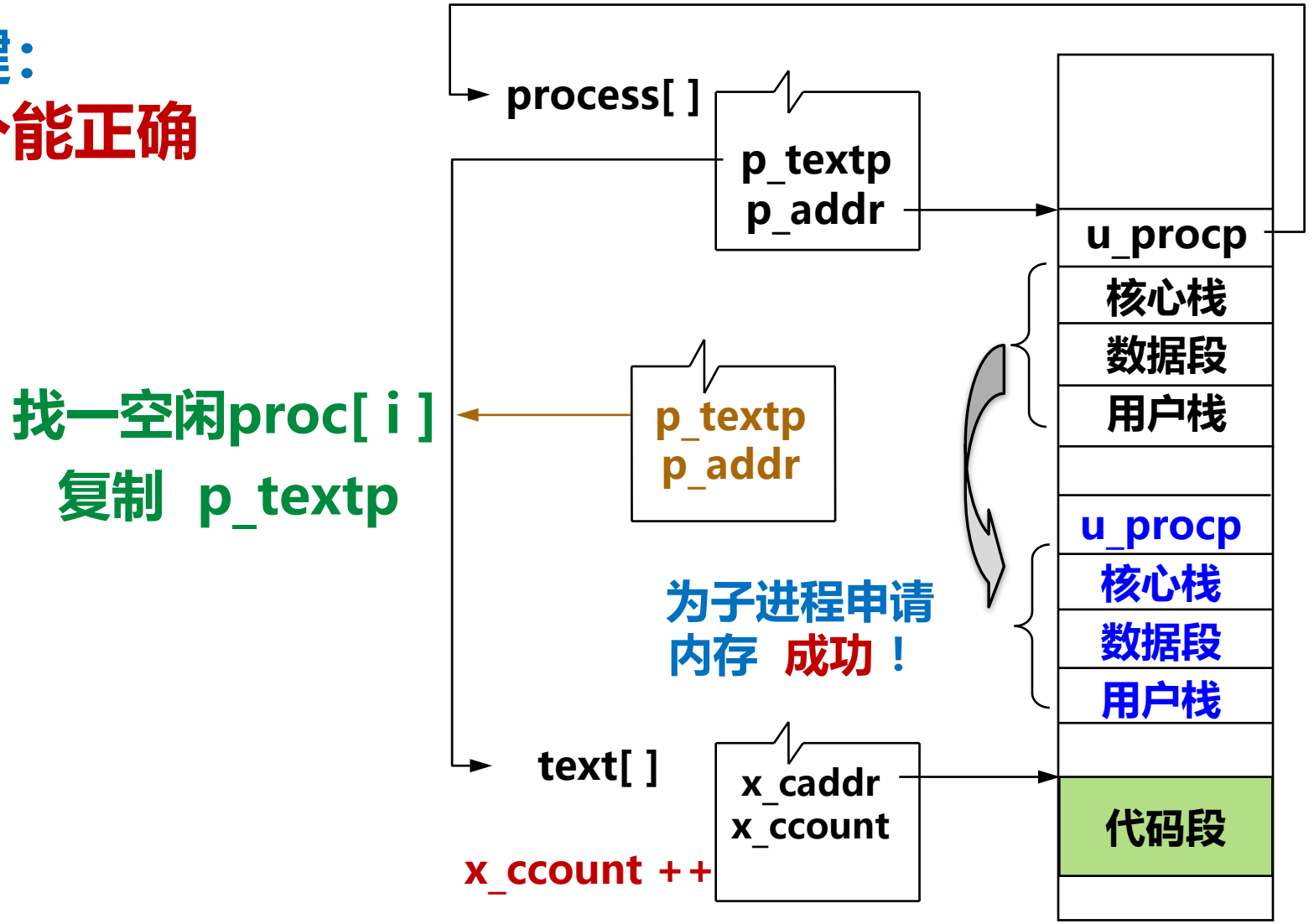
进程的创建与终止

创建子进程的关键：
为子进程构造一个能正确上台的图像



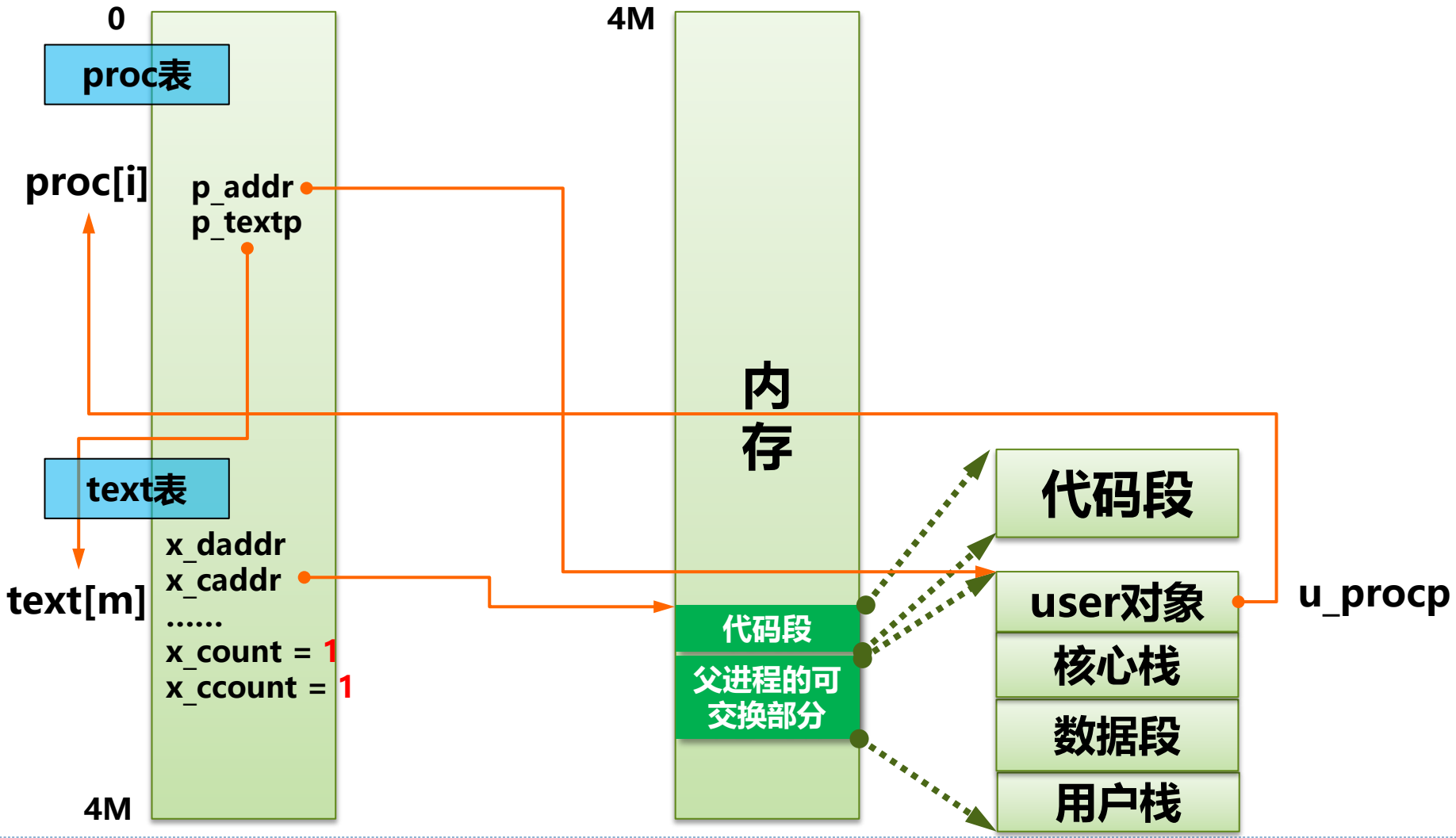
进程的创建与终止

创建子进程的关键：
为子进程构造一个能正确上台的图像

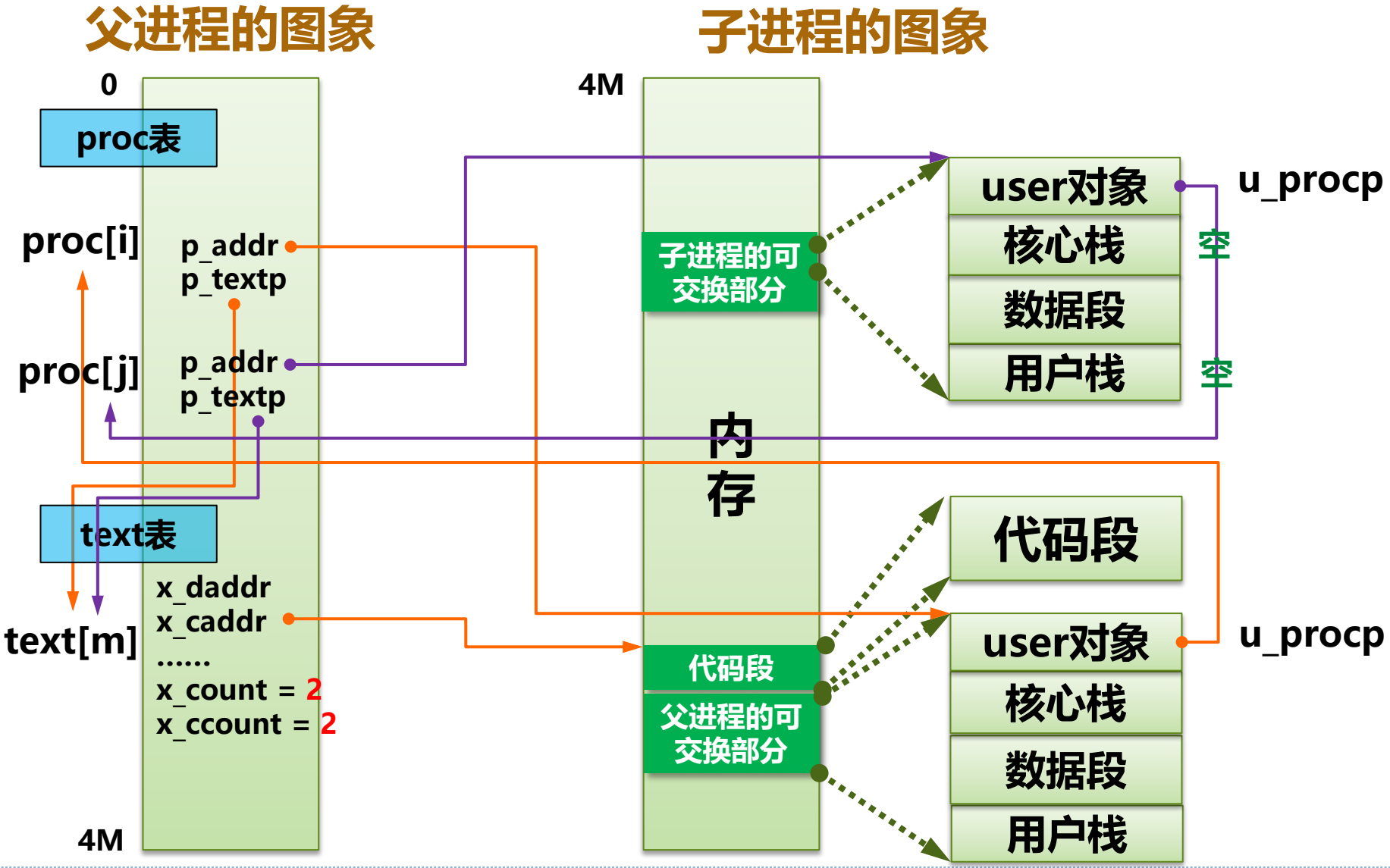


进程的创建与终止

父进程的图象

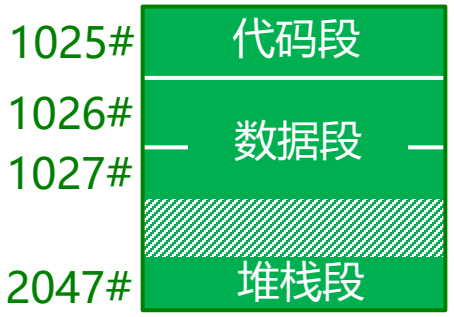


进程的创建与终止


父进程的图象
子进程的图象
0
4M
proc表
proc[i]
p_addr
p_textp
proc[j]
p_addr
p_textp
text表
text[m]
x_daddr
x_caddr
.....
x_count = 2
x_ccount = 2
4M
子进程的可交换部分
内存
代码段
父进程的可交换部分
user对象
核心栈
数据段
用户栈
u_procp
共享
代码段
user对象
核心栈
数据段
用户栈
u_procp

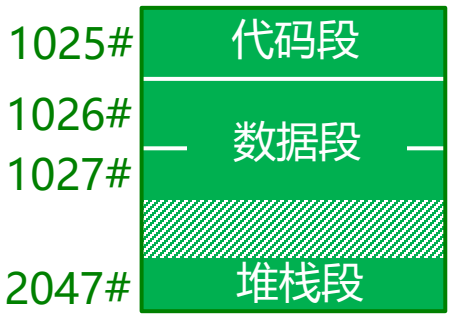
进程的创建与终止

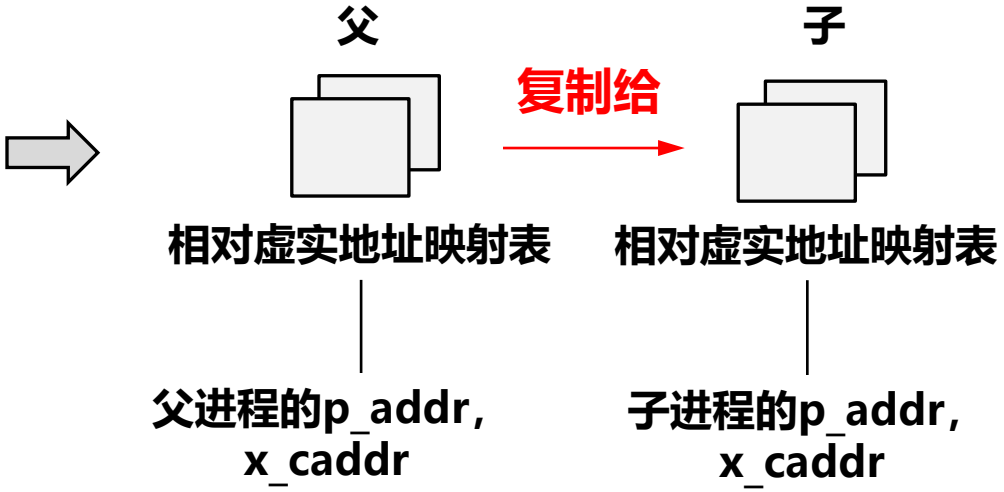
父进程的图象



=

子进程的图象





Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p	
0#	0		0	1	1	√
1#	1		0	1	1	√
	.....					
1022#	1022		0	1	1	
1023#	XXXX		0	1	1	×

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p	
0#	/	/	/	/	/	
1#	XXXX		1	0	1	√
	XXXX		1	1	1	×
	XXXX		1	1	1	×
1023#	XXXX		1	1	1	×

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p	
0#	0		0	1	1	√
1#	1		0	1	1	√
	.....					
1022#	1022		0	1	1	
1023#	XXXX		0	1	1	×

Page Table 1# (0x203号页框)

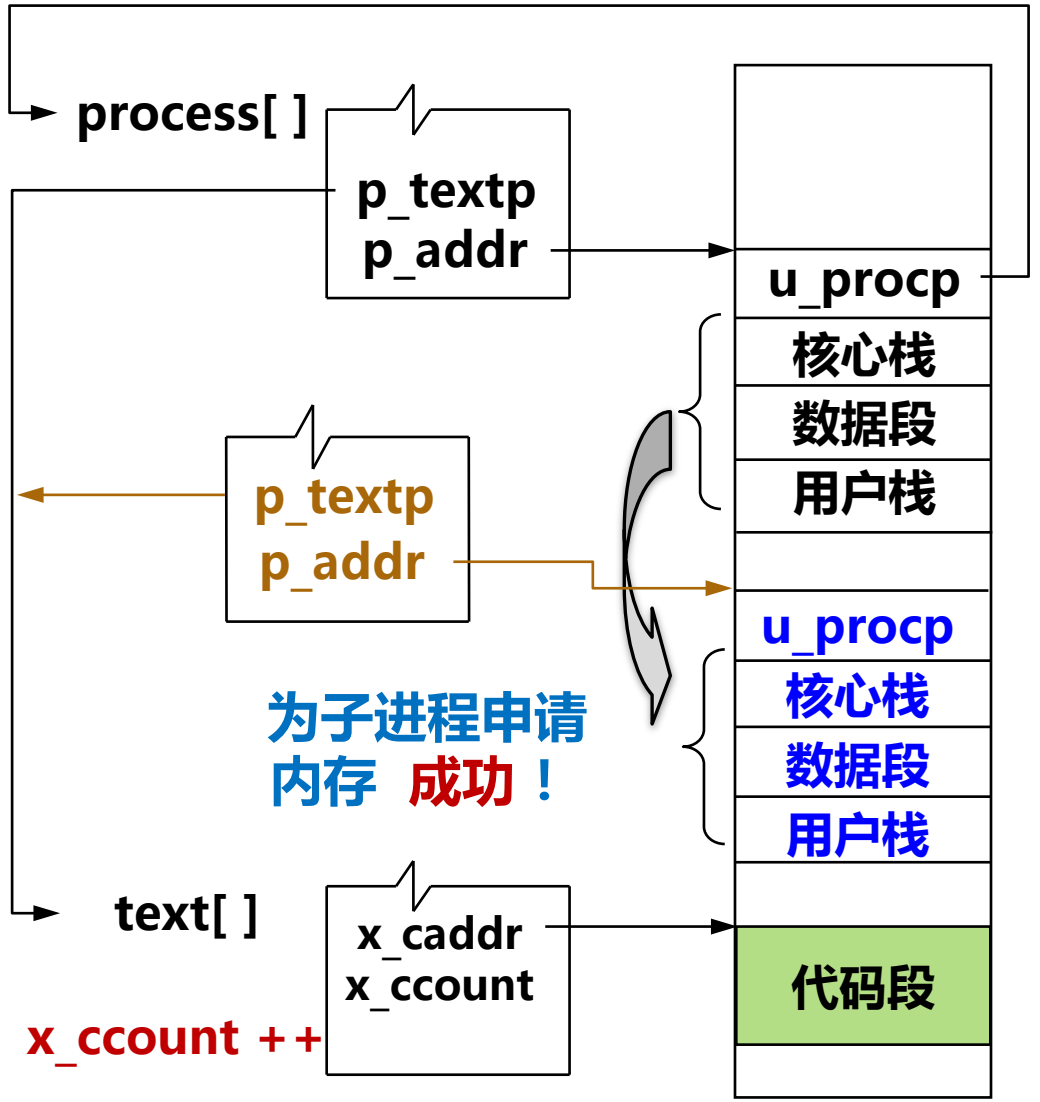
	Page Base Address		s/u	r/w	p	
0#	/	/	/	/	/	
1#	XXXX		1	0	1	√
	XXXX		1	1	1	×
	XXXX		1	1	1	×
1023#	XXXX		1	1	1	×

进程的创建与终止

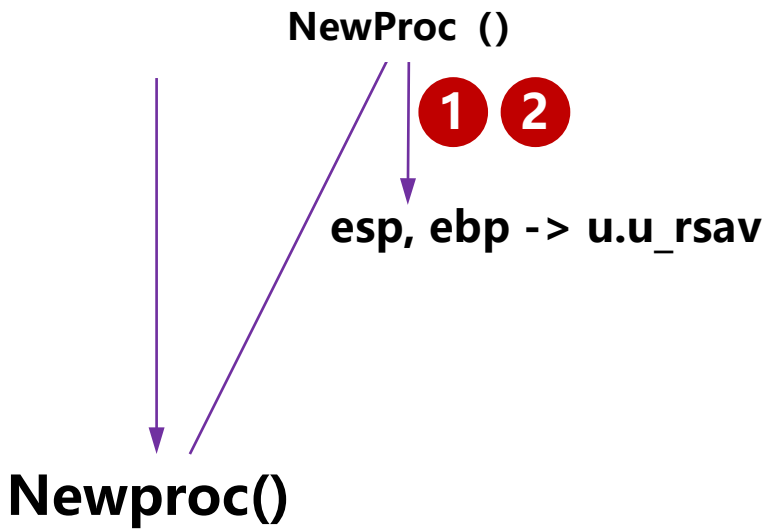
创建子进程的关键：
为子进程构造一个能正确上台的图像

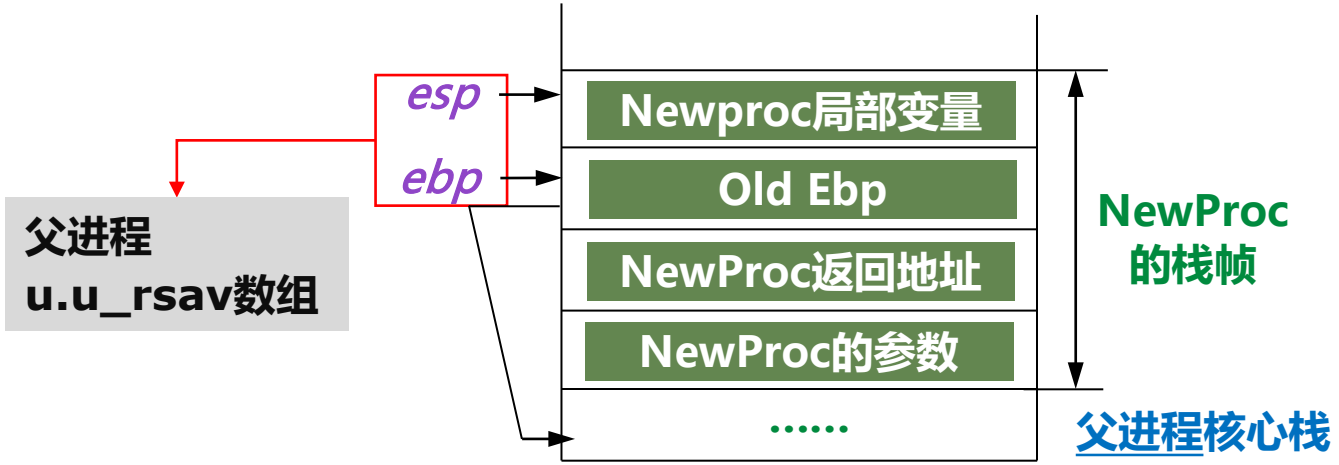
子进程得到了一张和父进程基本一样的图像

如何让父子进程完成不同的工作?

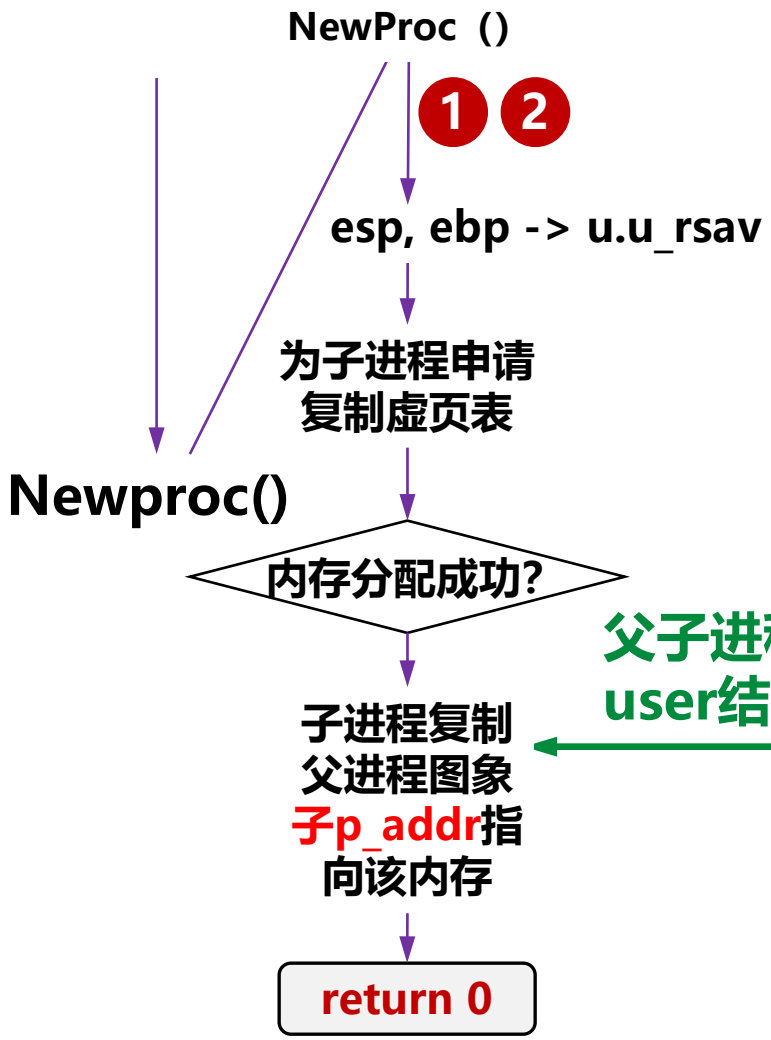


进程的创建与终止

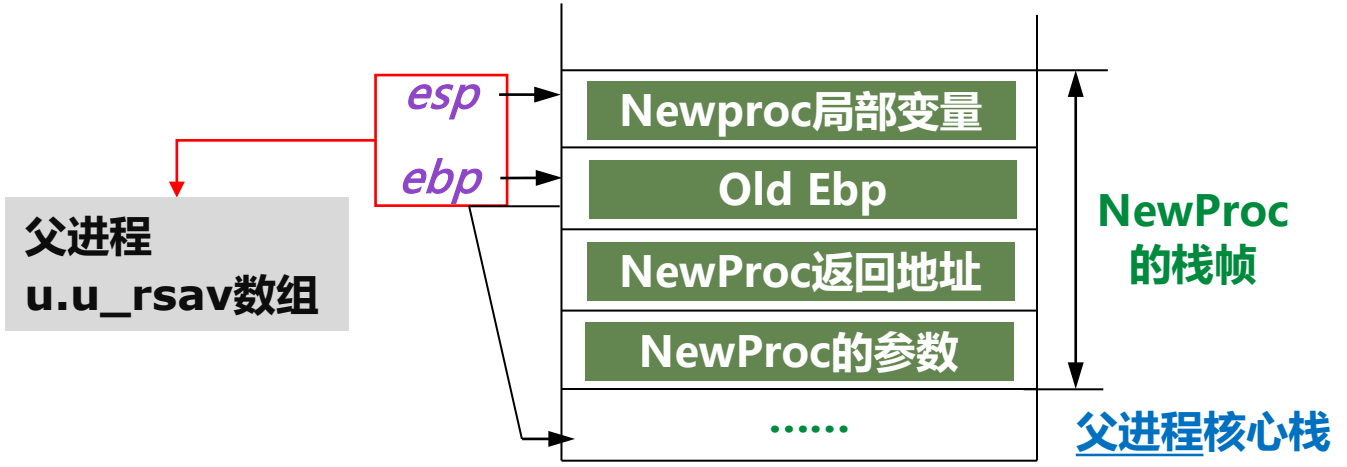

NewProc ()
1
2
esp, ebp -> u.u_rsav
Newproc()


esp
ebp
父进程
u.u_rsav数组
Newproc局部变量
Old Ebp
NewProc返回地址
NewProc的参数
.....
NewProc
的栈帧
父进程核心栈

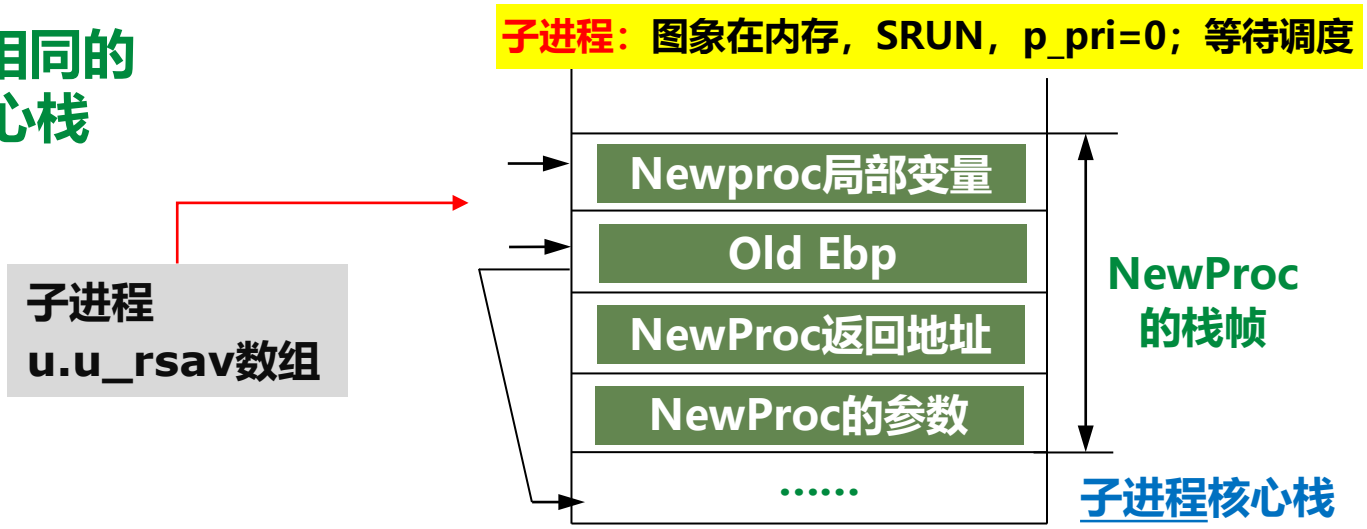
进程的创建与终止



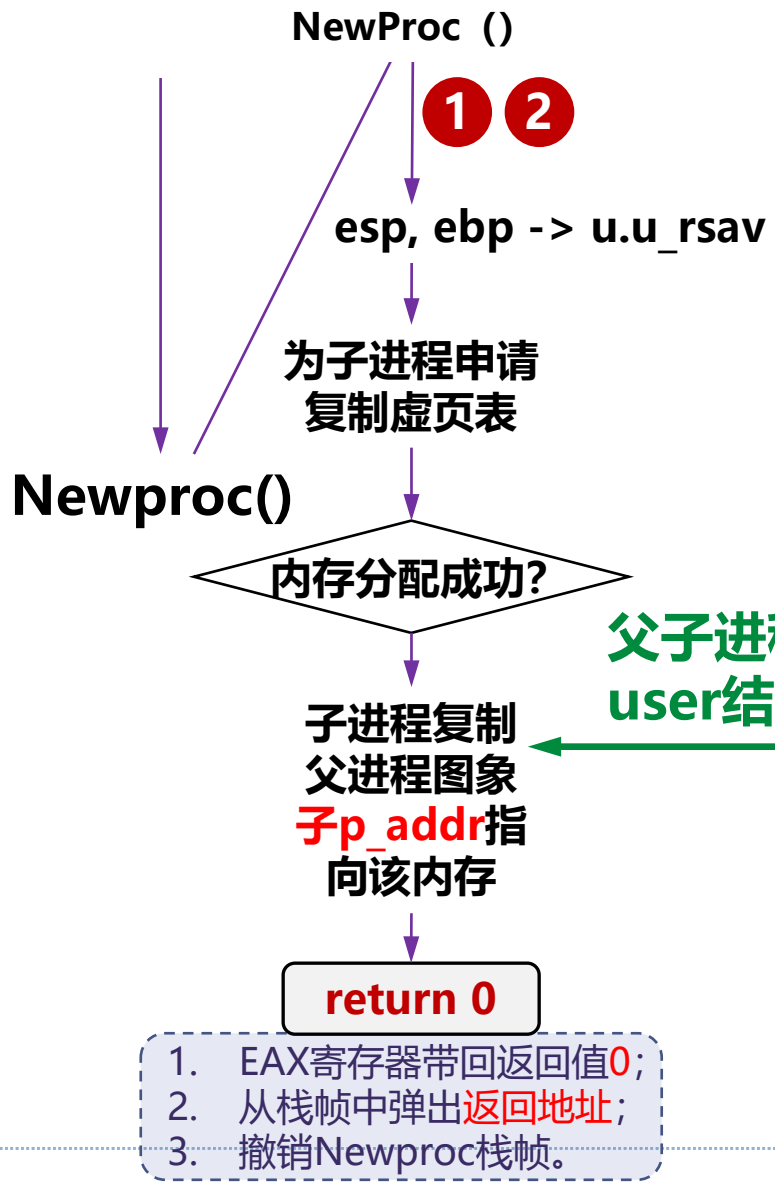
父子进程完全相同的
user结构和核心栈



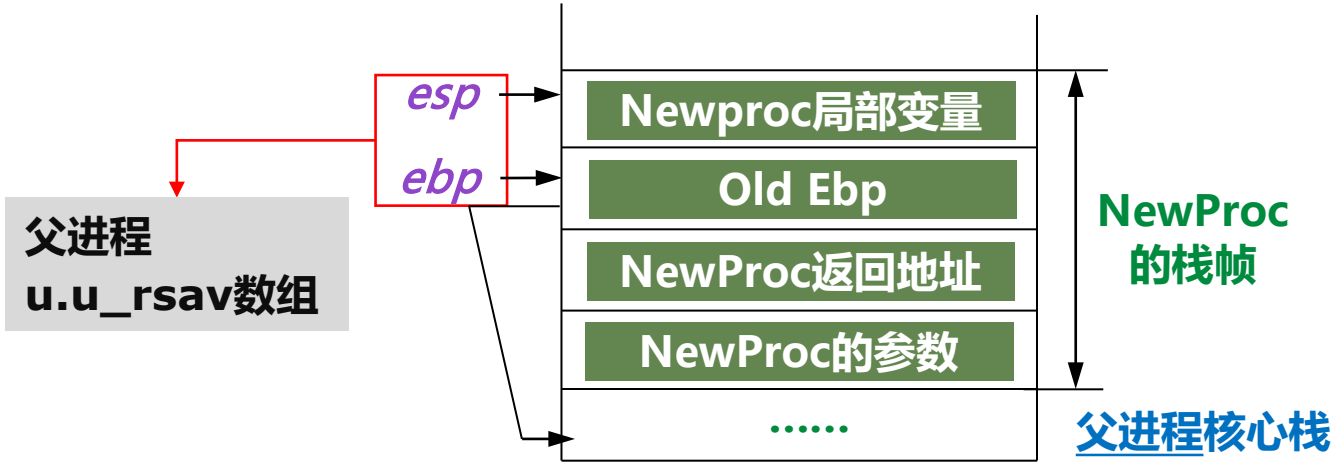
子进程：图象在内存，SRUN，p_pri=0；等待调度



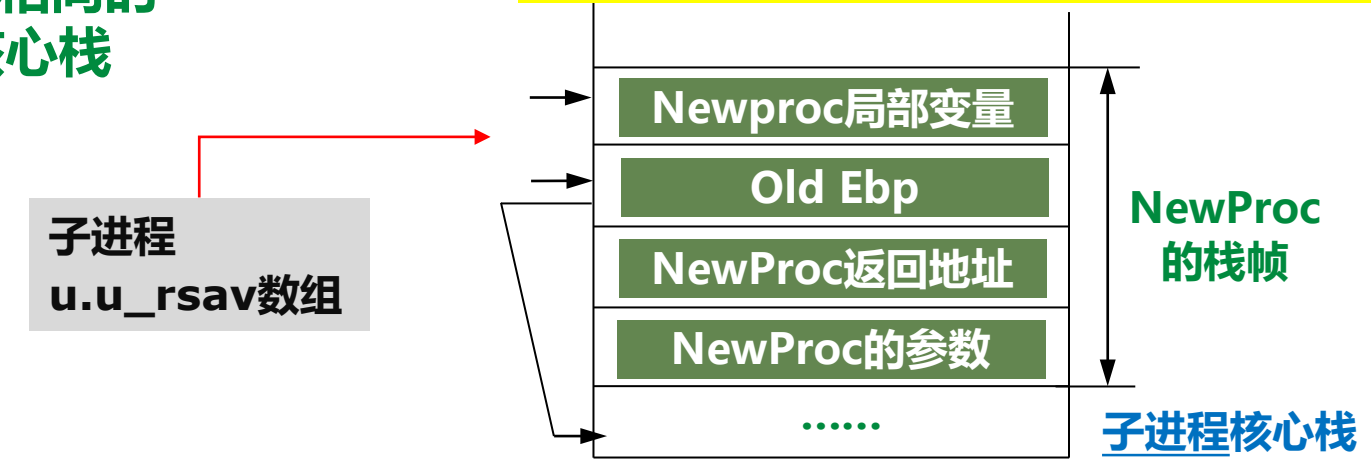
进程的创建与终止



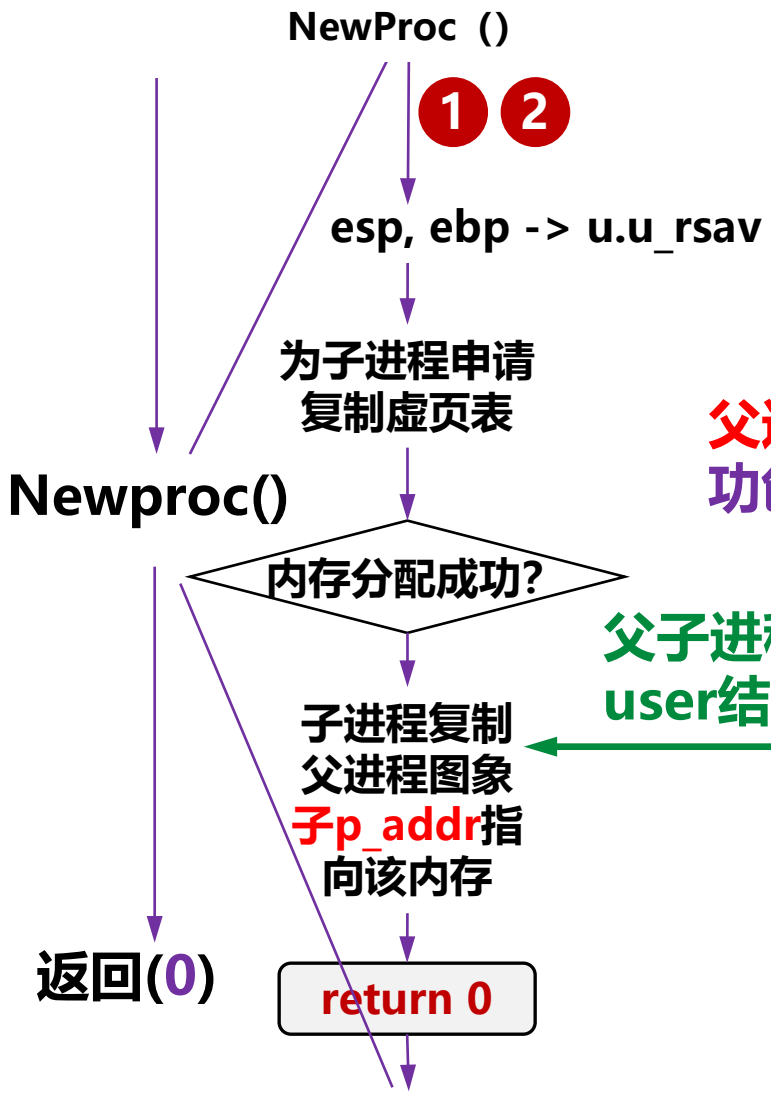
父子进程完全相同的user结构和核心栈



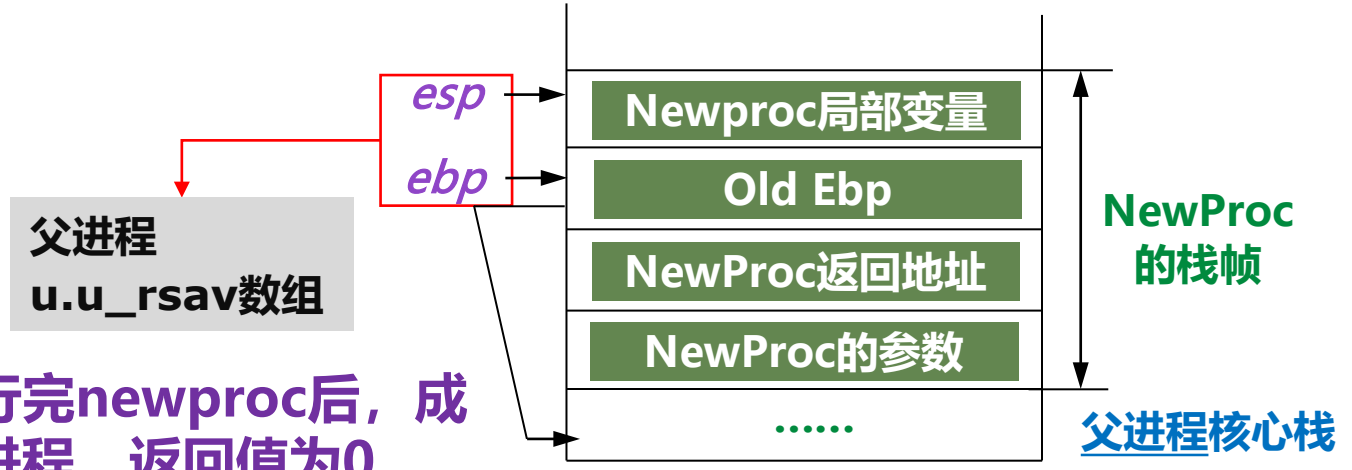
子进程：图象在内存，SRUN，p_pri=0；等待调度



进程的创建与终止

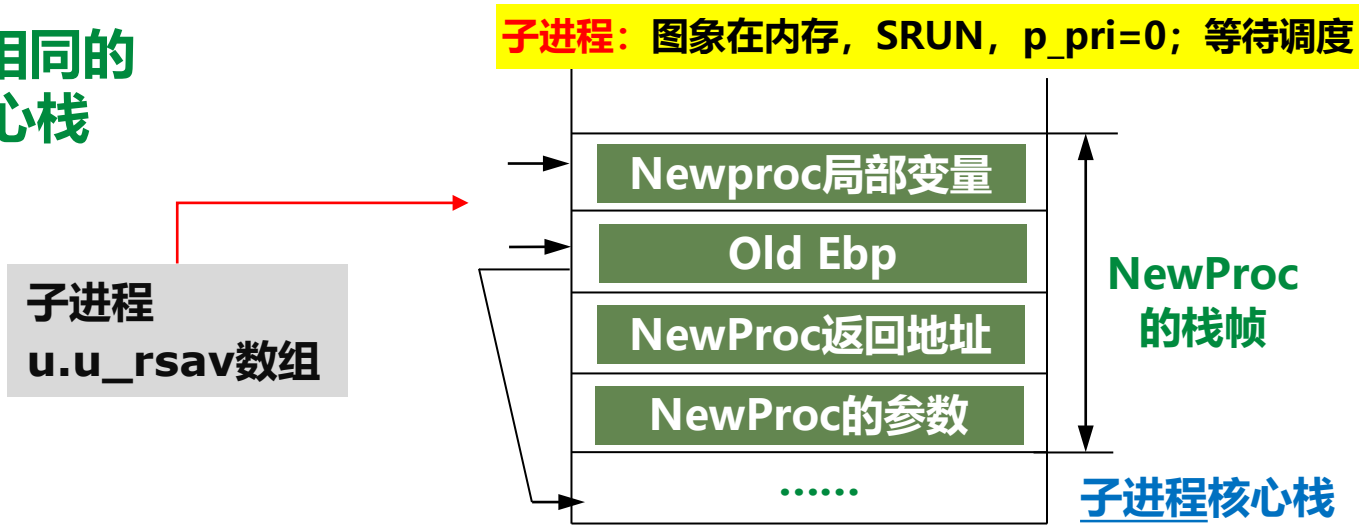


父子进程完全相同的
user结构和核心栈

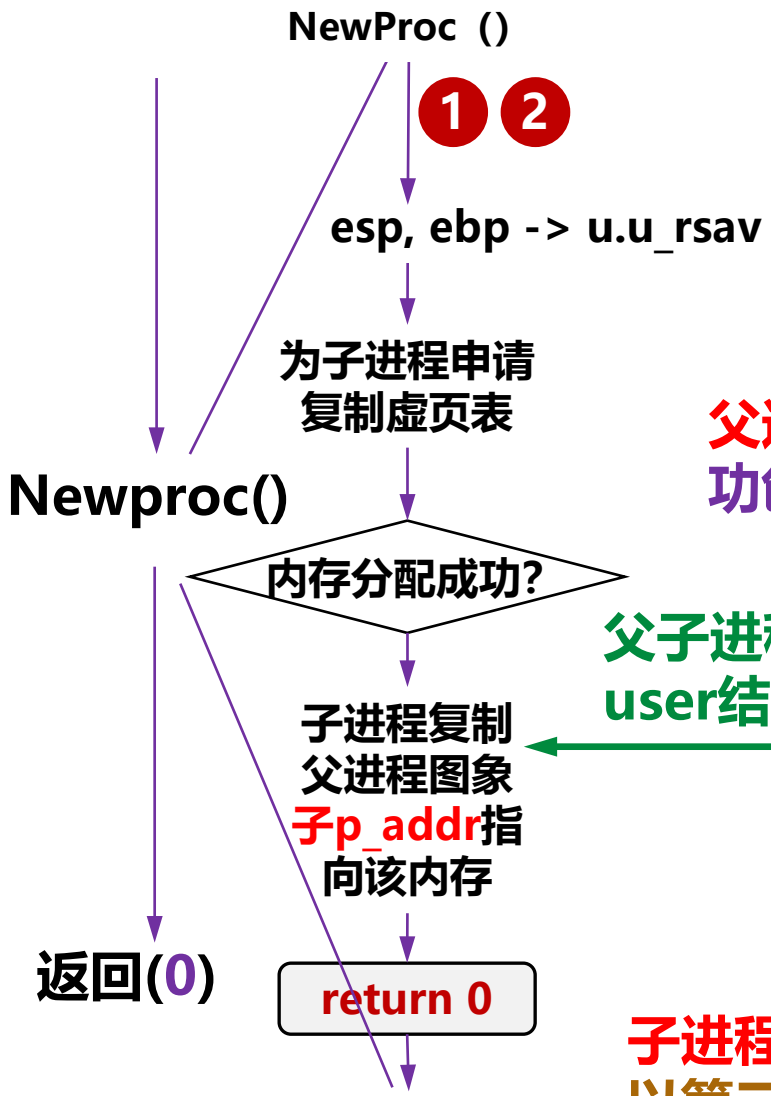


父进程执行完newproc后，成功创建子进程，返回值为0

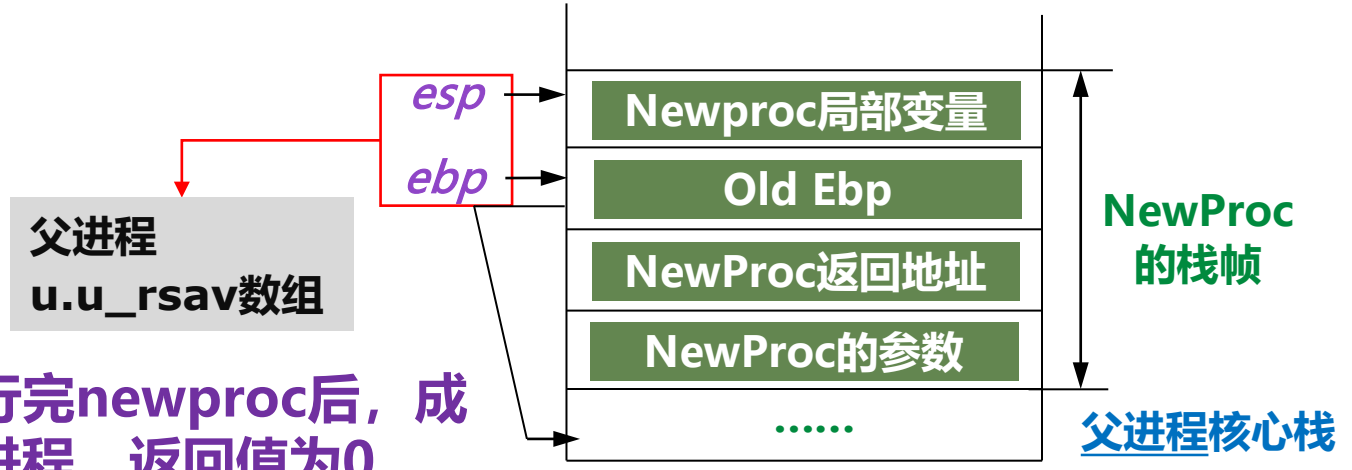
子进程：图象在内存，SRUN，p_pri=0；等待调度



进程的创建与终止

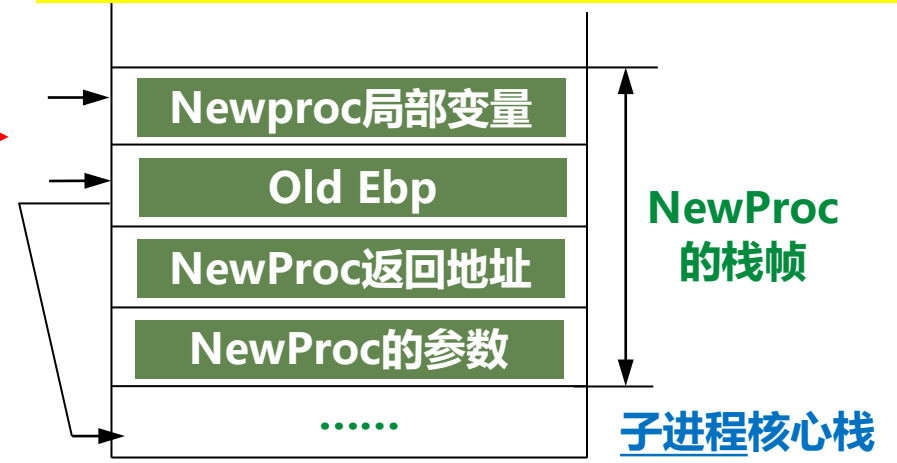


父子进程完全相同的
user结构和核心栈



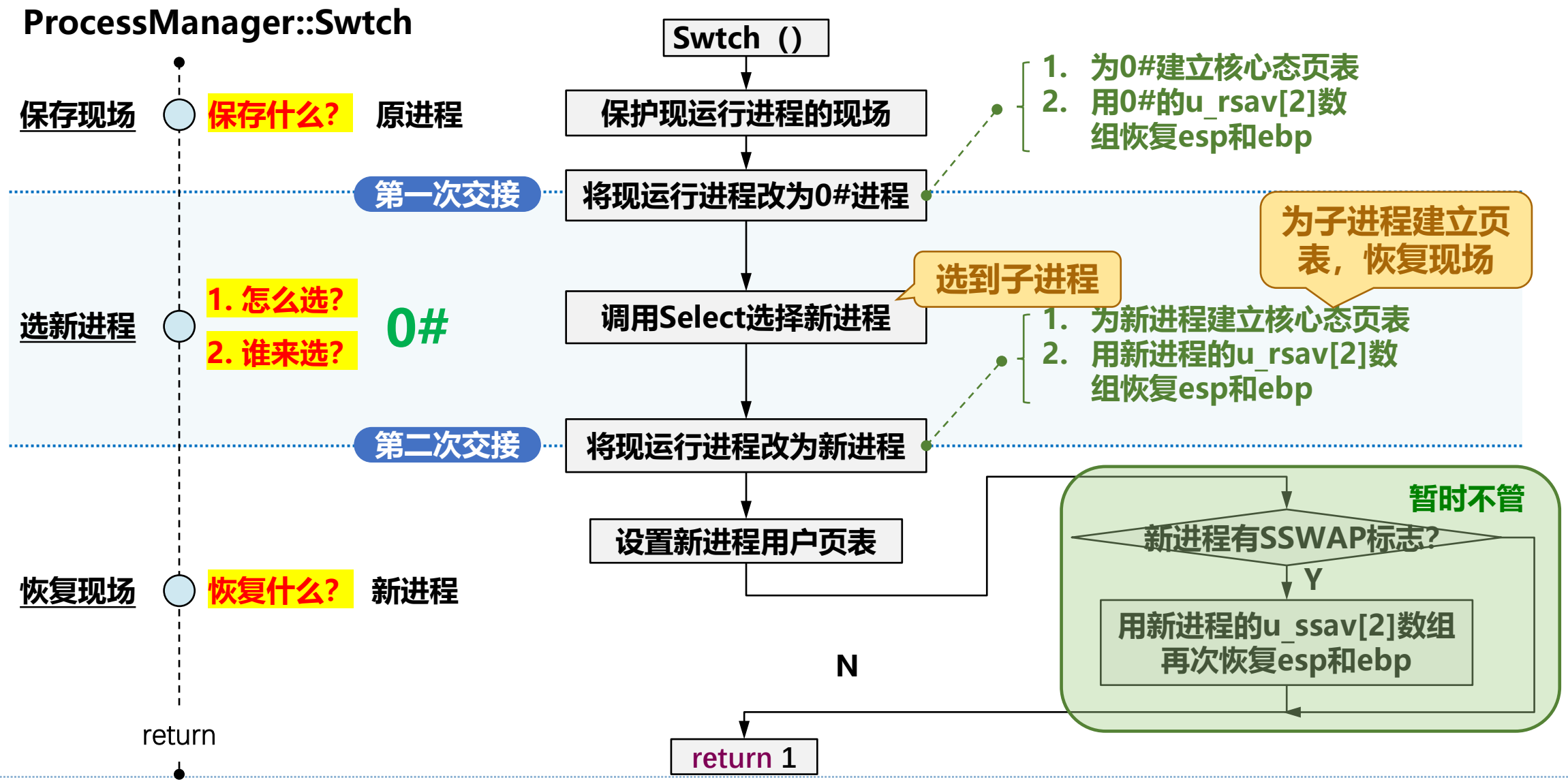
父进程执行完newproc后，成功创建子进程，返回值为0

子进程：图象在内存，SRUN，p_pri=0；等待调度



子进程未来在某次Swtch被选中，以第三棒上台执行

UNIX的进程调度控制



UNIX的进程调度控制

ProcessManager::Swtch

保存现场 — 保存什么？ — 原进程

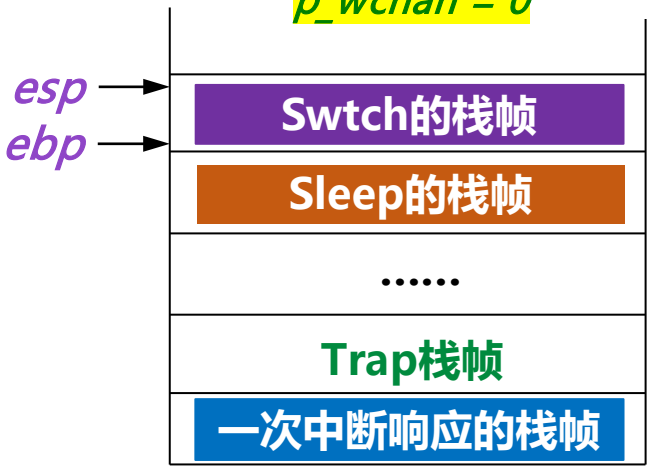
选新进程 — 1. 怎么选？ 2. 谁来选？ — 0#

恢复现场 — 恢复什么？ — 新进程

return 回哪里？

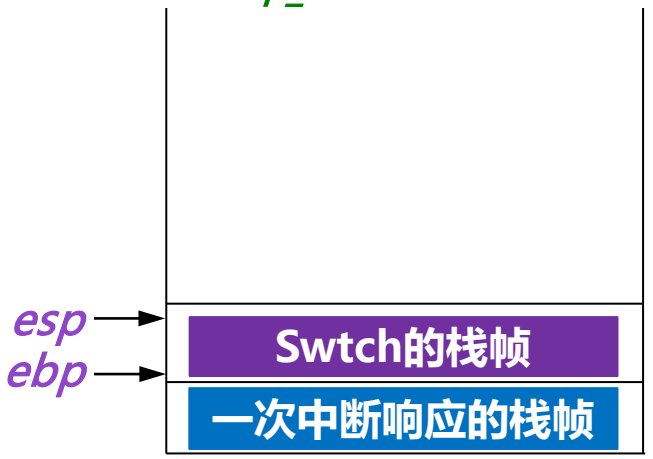
非抢占式

p_stat=SRUN　p_pri >= 100
p_wchan = 0



抢占式

p_stat=SRUN, p_pri >=100
p_wchan = 0



从未上过台的子进程不属于上述两种情况

UNIX的进程调度控制

ProcessManager::Swtch

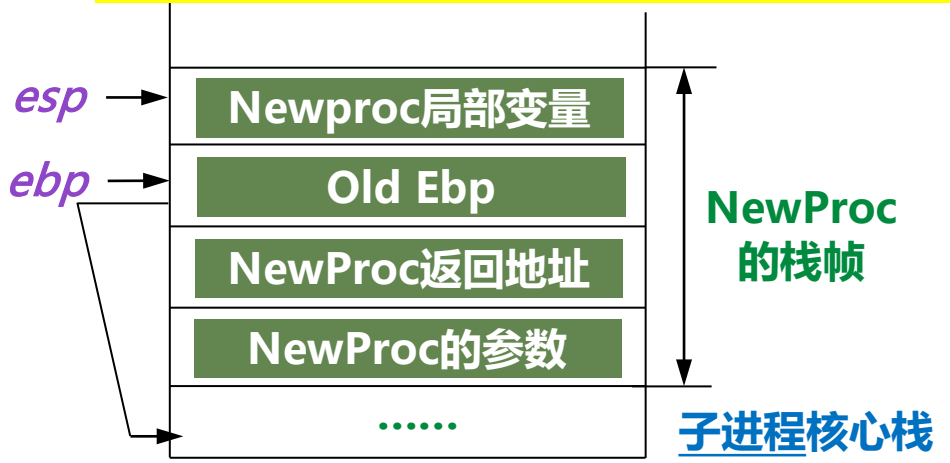
保存现场 — 保存什么？ — 原进程

选新进程 — 1. 怎么选？ 2. 谁来选？ — 0#

恢复现场 — 恢复什么？ — 新进程

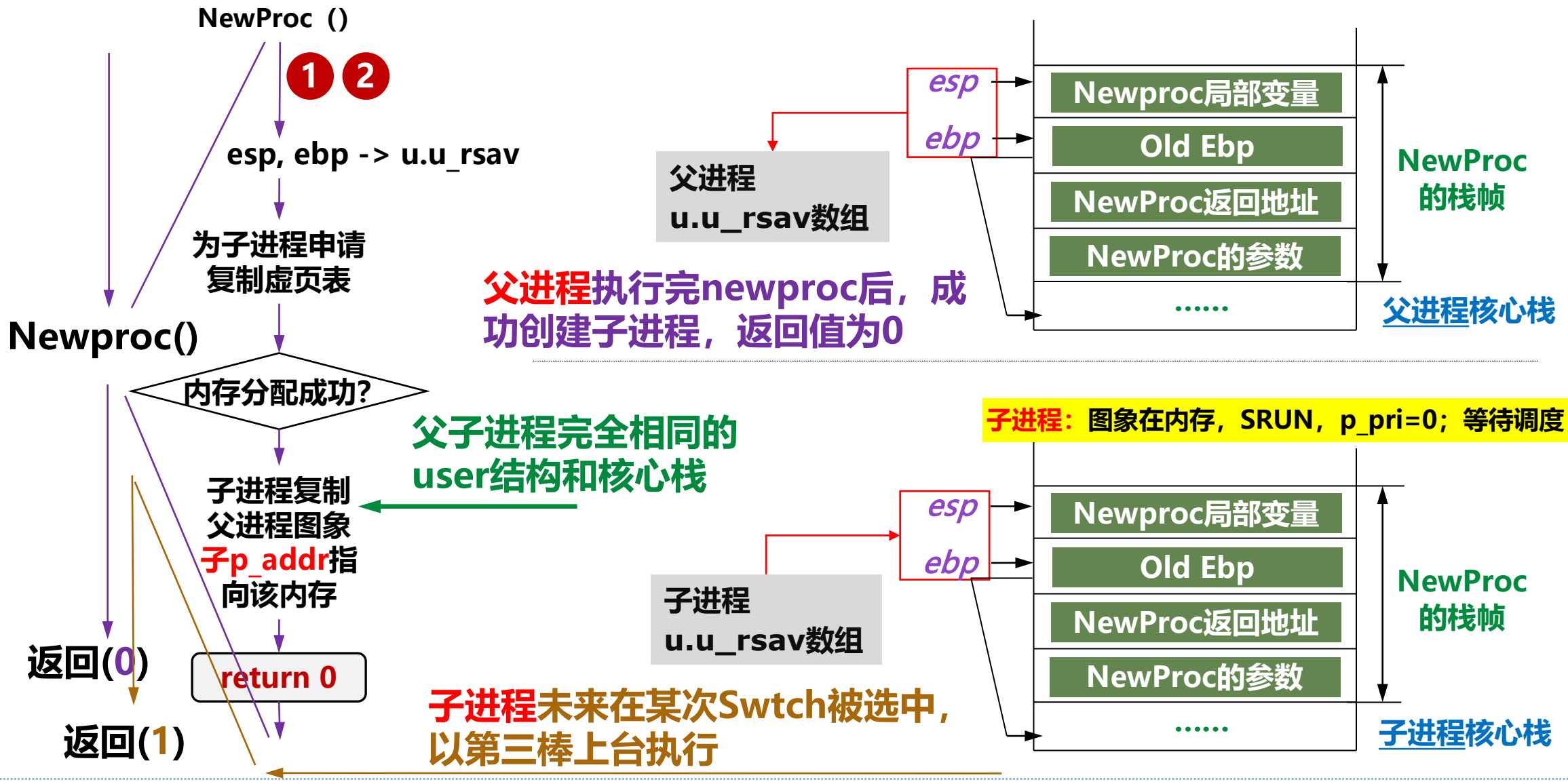
return 回哪里？

子进程：图象在内存，SRUN，p_pri=0；等待调度



Newproc的调用函数，调用位置的下一条语句
返回值为1

进程的创建与终止


NewProc ()
1
2
esp, ebp -> u.u_rsav
为子进程申请
复制虚页表
Newproc()
内存分配成功?
子进程复制
父进程图象
子p_addr指
向该内存
返回(0)
return 0
返回(1)
父子进程完全相同的
user结构和核心栈
父进程
u.u_rsav数组
esp
ebp
Newproc局部变量
Old Ebp
NewProc返回地址
NewProc的参数
.....
NewProc
的栈帧
父进程核心栈
父进程执行完newproc后，成
功创建子进程，返回值为0
子进程：图象在内存，SRUN，p_pri=0；等待调度
子进程
u.u_rsav数组
esp
ebp
Newproc局部变量
Old Ebp
NewProc返回地址
NewProc的参数
.....
NewProc
的栈帧
子进程核心栈
子进程未来在某次Swtch被选中，
以第三棒上台执行

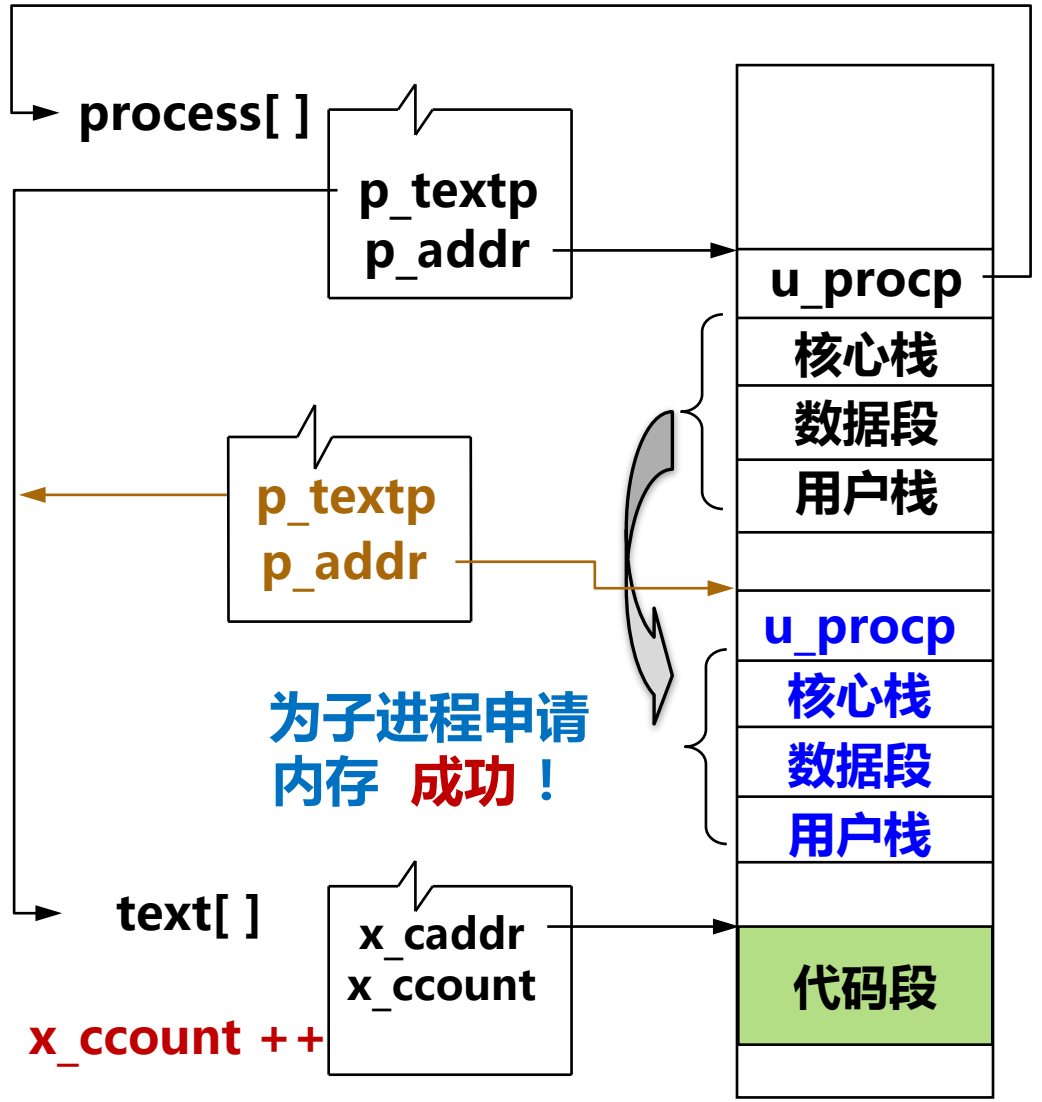
进程的创建与终止

创建子进程的关键：
为子进程构造一个能正确上台的图像

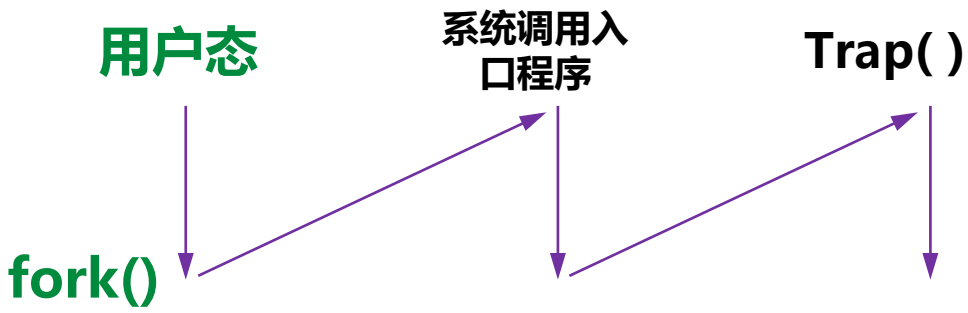
子进程得到了一张和父进程基本一样的图像

和父进程返回到同一个位置：Newproc的下一条语句

如何让父子进程完成不同的工作?



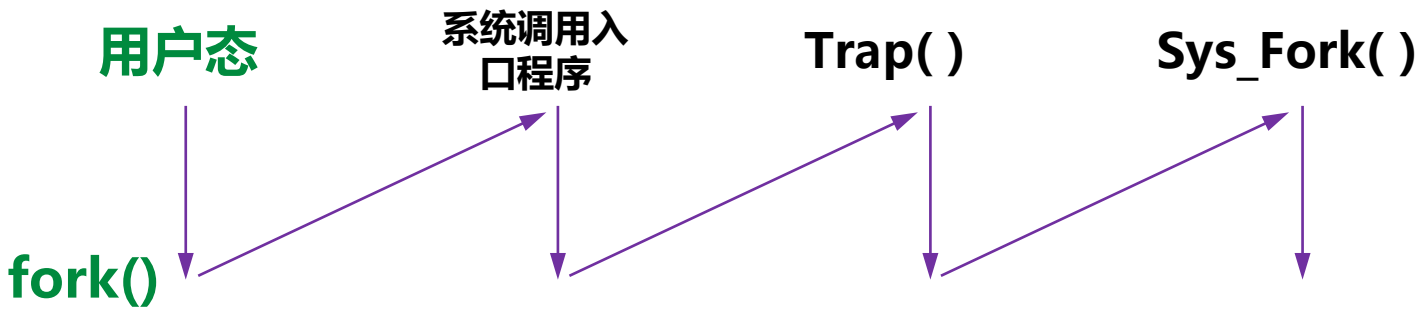
进程的创建与终止




```
int fork()
{
    int res;
    __asm__ __volatile__ ( "int $0x80":"=a"(res):"a"(2));
    if ( res >= 0 )
        return res;
    return -1;
}
```

1

进程的创建与终止



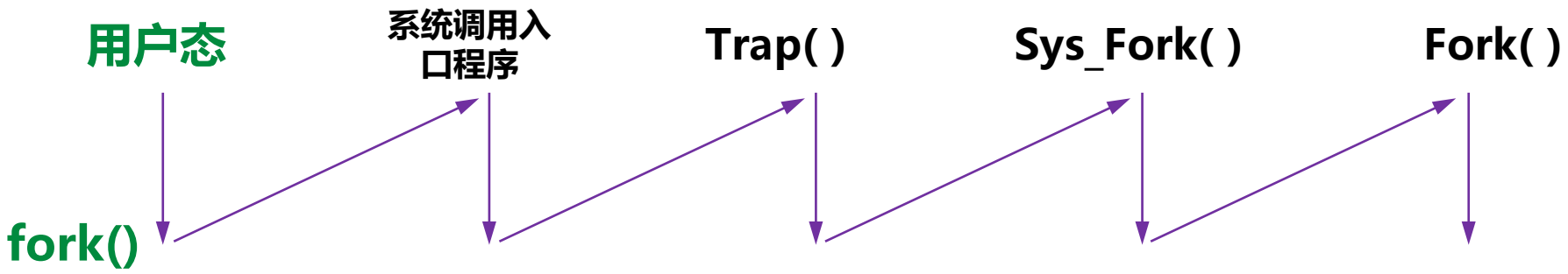

```
int fork()
{
    int res;
    __asm__ __volatile__ ( "int $0x80":"=a"(res):"a"(2));
    if ( res >= 0 )
        return res;
    return -1;
}
```

1

```
SystemCallTableEntry SystemCall::m_SystemEntranceTable[SYSTEM_CALL_NUM] = {
    { 0, &Sys_NullSystemCall },        /* 0 = indir*/
    { 1, &Sys_Rexit },                 /* 1 = rexit*/
    { 0, &Sys_Fork },                  /* 2 = fork*/
    { 3, &Sys_Read },                  /* 3 = read*/
    ...
}
```

2

进程的创建与终止




```
int fork()
{
    int res;
    __asm__ __volatile__ ( "int $0x80":"=a"(res):"a"(2));
    if ( res >= 0 )
        return res;
    return -1;
}
```

1

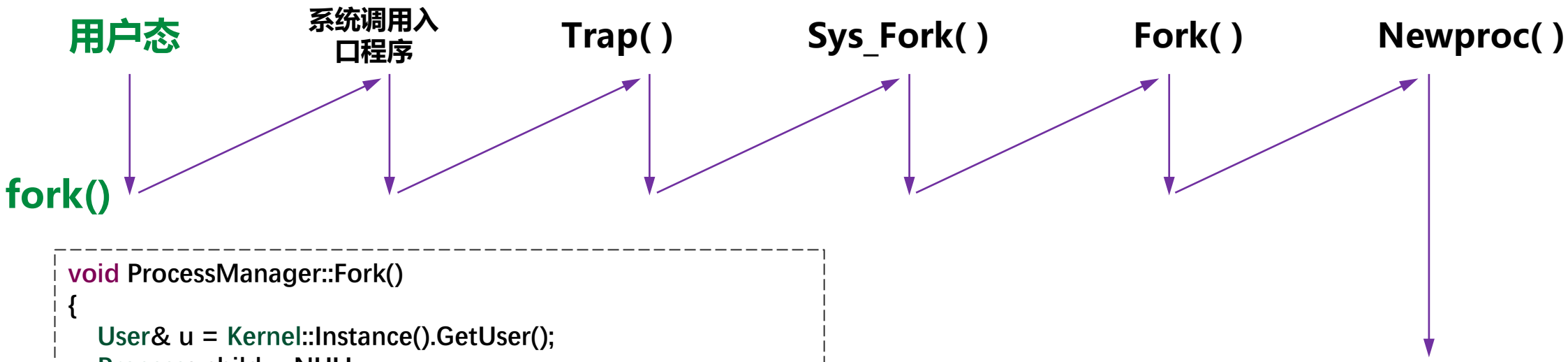
```
int SystemCall::Sys_Fork()
{
    ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
    procMgr.Fork();
    return 0;/* GCC likes it ! */
}
```

3

```
SystemCallTableEntry SystemCall::m_SystemEntranceTable[SYSTEM_CALL_NUM] = {
    { 0, &Sys_NullSystemCall },        /* 0 = indir*/
    { 1, &Sys_Rexit },                 /* 1 = rexit*/
    { 0, &Sys_Fork },                  /* 2 = fork*/
    { 3, &Sys_Read },                  /* 3 = read*/
    ...
}
```

2

进程的创建与终止

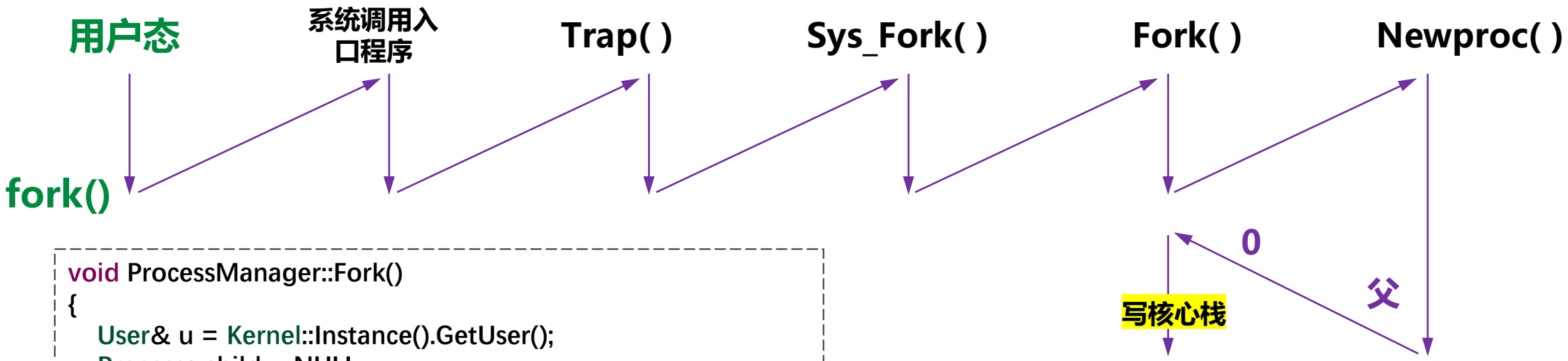



```
void ProcessManager::Fork()
{
    User& u = Kernel::Instance().GetUser();
    Process* child = NULL;;
    ...
    if ( this->NewProc() )
    {
        u.u_ar0[User::EAX] = 0;
        u.u_cstime，u.u_stime，u.u_cutime，u.u_utime等参数清0;
    }
    else
    {
        u.u_ar0[User::EAX] = child->p_pid;
    }
    return;
}
```

4

进程的创建

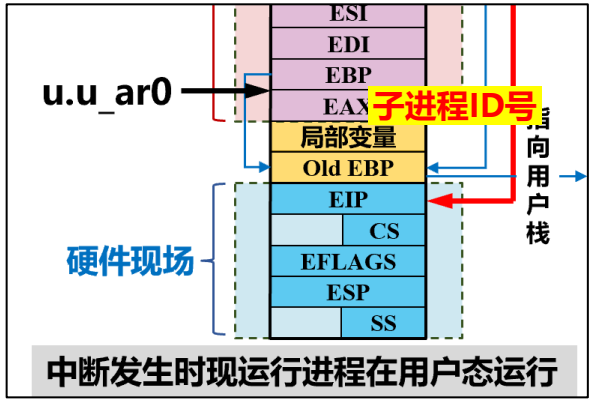
进程的创建与终止



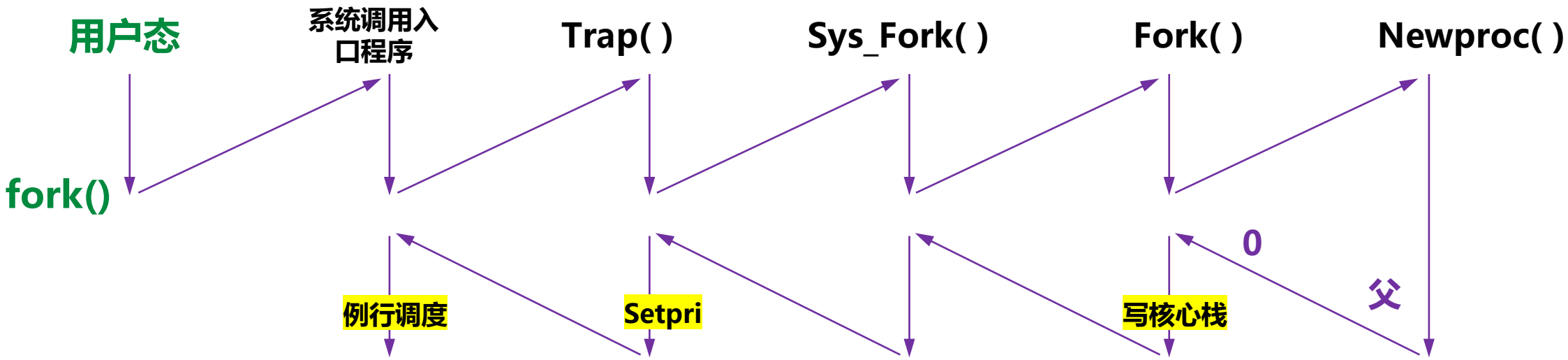

```
void ProcessManager::Fork()
{
    User& u = Kernel::Instance().GetUser();
    Process* child = NULL;;
    ...
    if ( this->NewProc() )
    {
        u.u_ar0[User::EAX] = 0;
        u.u_cstime，u.u_stime，u.u_cutime，u.u_utime等参数清0;
    }
    else
    {
        u.u_ar0[User::EAX] = child->p_pid;
    }
    return;
}
```

4

父进程核心栈



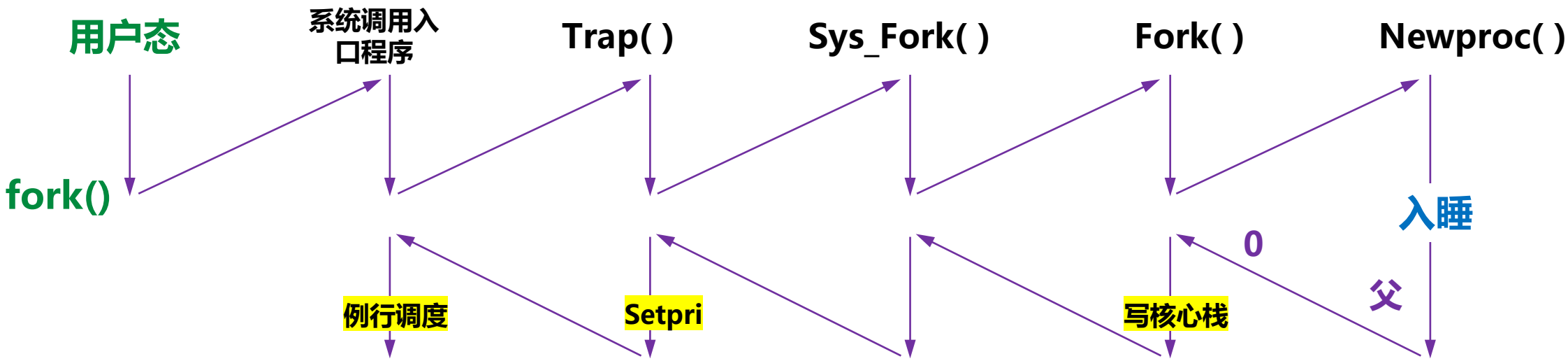
进程的创建与终止



父进程即将返回fork()调用点时，
由于例行调度，Swtch可能被调用！

Swtch被调用的原因：RunRun>0

进程的创建与终止



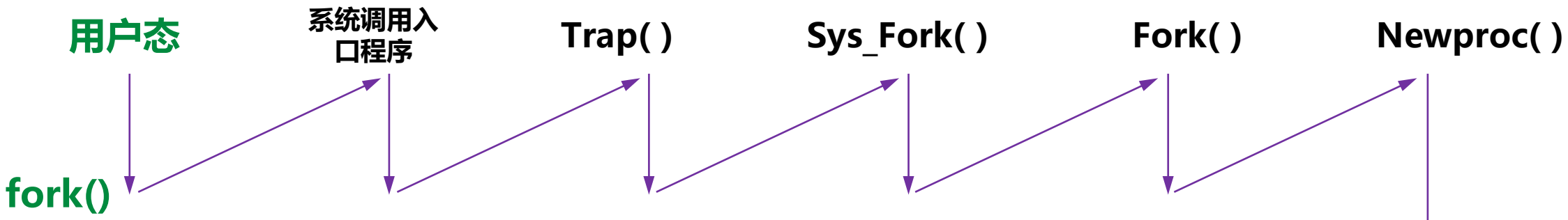
父进程即将返回fork()调用点时，
由于例行调度，Swtch可能被调用！

Swtch被调用的原因：RunRun>0

1. 父进程在台上时间足够长，重算时，p_pri↗
2. 父进程曾经入睡，重算时，p_pri↗
3. 父进程曾响应中断，唤醒更高优先级的进程

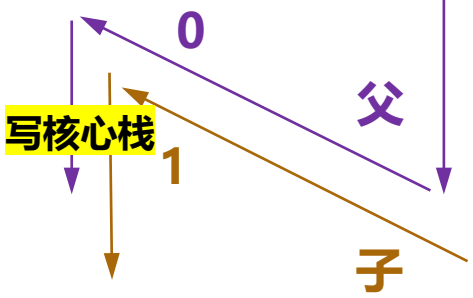
父进程：p_pri>=100，导致子进程
（p_pri =0）在父进程前先上台！

进程的创建与终止

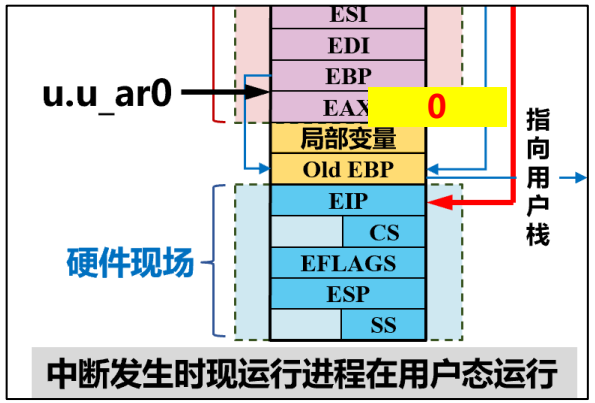



```
void ProcessManager::Fork()
{
    User& u = Kernel::Instance().GetUser();
    Process* child = NULL;;
    ...
    if ( this->NewProc() )
    {
        u.u_ar0[User::EAX] = 0;
        u.u_cstime，u.u_stime，u.u_cutime，u.u_utime等参数清0;
    }
    else
    {
        u.u_ar0[User::EAX] = child->p_pid;
    }
    return;
}
```

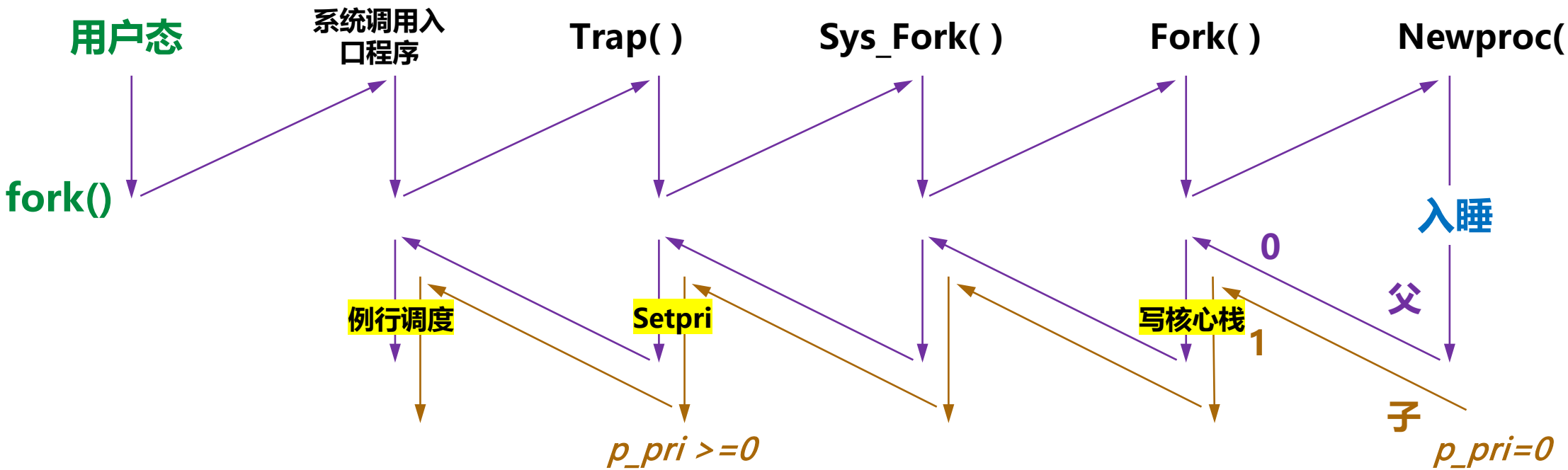
4



子进程核心栈



进程的创建与终止



父进程即将返回fork()调用点时，由于例行调度，Swtch可能被调用！

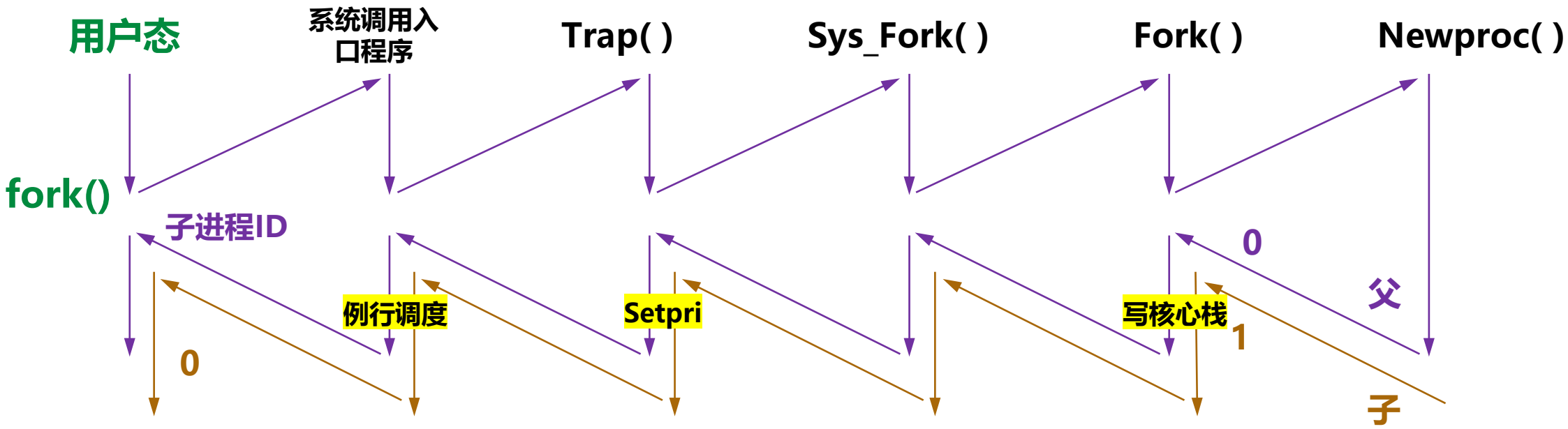
Swtch被调用的原因：RunRun>0

父进程：p_pri>=100，导致子进程（p_pri =0）在父进程前先上台！

子进程：在这里重算优先数，p_pri>=100，RunRun被设置

当子进程即将返回到fork()调用点时，因为RunRun被设置，由于例行调度的关系，Swtch执行。父子进程都有机会上台！

进程的创建与终止



总结：

1. fork系统调用的结果，父进程返回创建的子进程的ID，子进程返回0
2. 父子进程都有可能先返回用户态（父进程被抢占的条件）
3. 进程刚创建成功时p_pri=0，从fork返回后，p_pri >=100

进程的创建与终止

(1) 父子进程执行同一个应用程序

```
main( )
{
   if( fork()==0 )
      printf ("child process id is : %d\n", getpid());
   else
      printf ("parent process id is: %d\n", getpid());
}
```

先调度父进程:
parent process id is: XXX
child process id is: XXX

先调度子进程:
child process id is: XXX
parent process id is: XXX

进程的创建与终止

(1) 父子进程执行同一个应用程序

```
main()
{
   int i;
   int a=0;
   if(i=fork())
   {
      a=a+1;
      printf("parent: i= %d, a= %d\n", i, a);
   }
   else
   {
      a=a+4;
      printf("child: i= %d, a= %d\n", i,  a);
   }
}
```

父进程先执行printf:

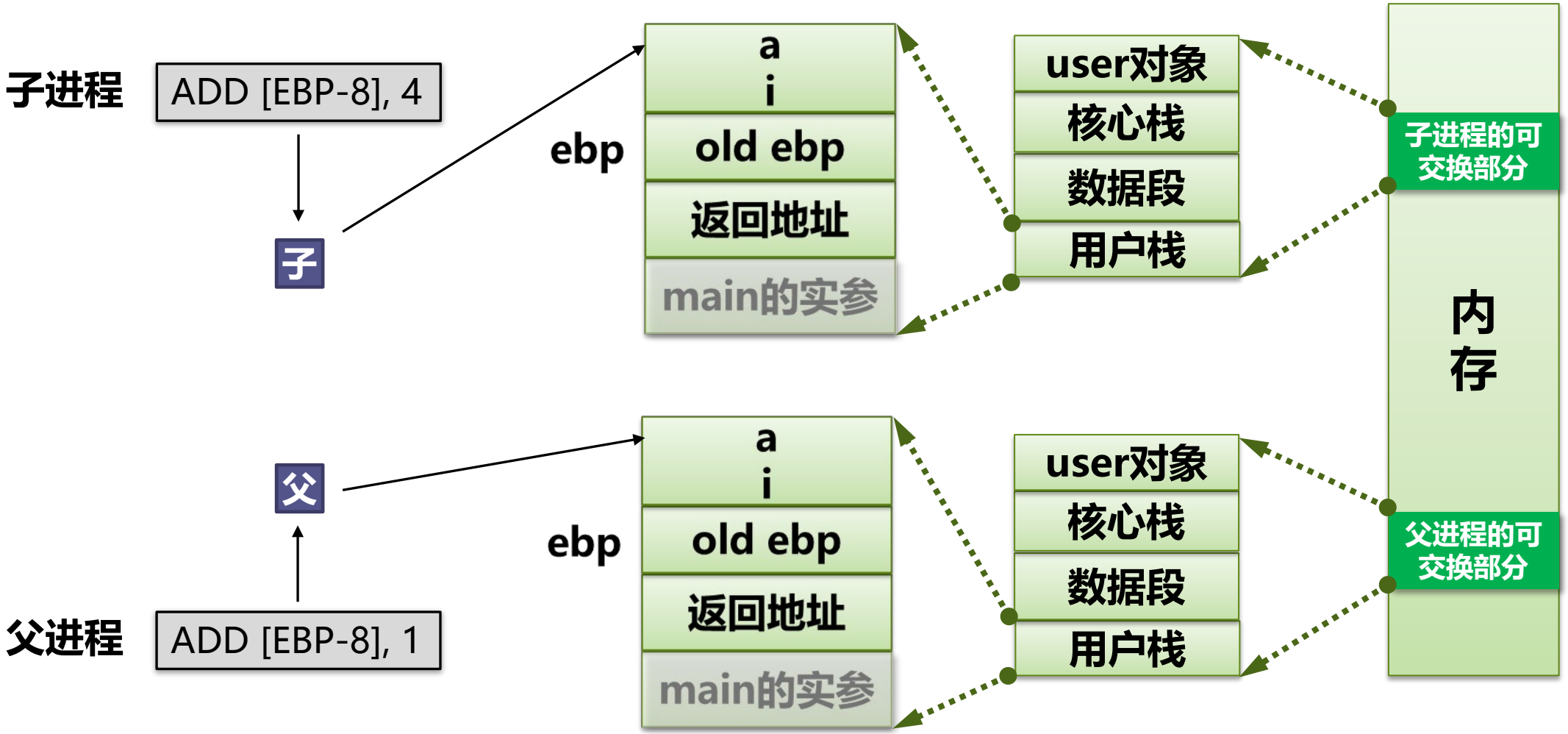
```
parent: i= 子进程ID, a= 1
child: i= 0, a= 4
```

子进程先执行printf:

```
child: i= 0, a= 4
parent: i= 子进程ID, a= 1
```

进程的创建与终止

（1）父子进程执行同一个应用程序



进程的创建与终止

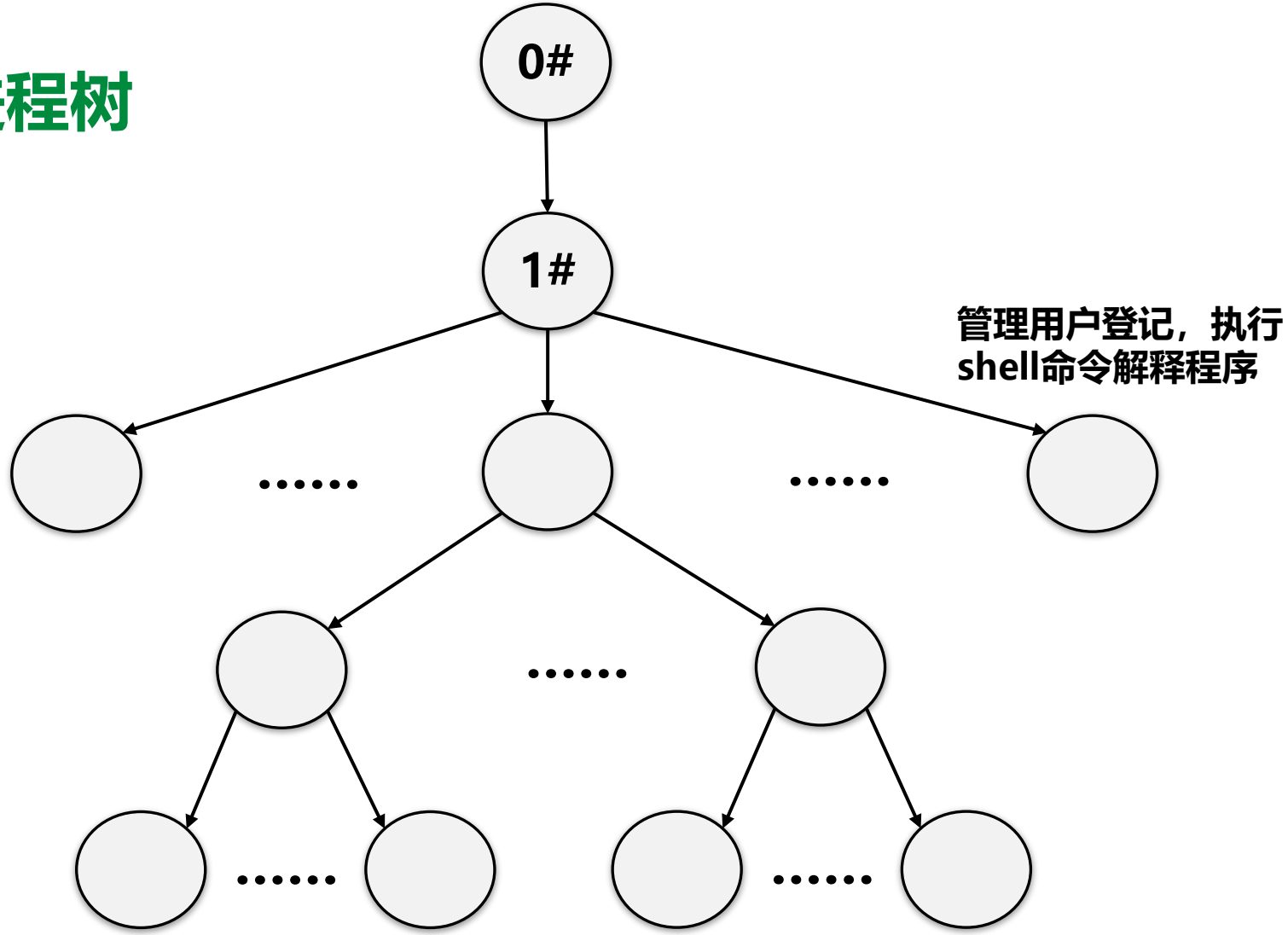
(2) 让子进程执行另一个应用程序

```
main( )
{
   ……;
   if ( !fork())
   {
        exec("/bin/date", "date", 0);
        exit();
    }
   ……;
}
```

子进程改换进程图象

进程的创建与终止

UNIX的进程树



主 要 内 容

4.1 UNIX时钟中断与异常

4.2 UNIX系统调用

4.3 UNIX的进程调度状态

4.4 UNIX的进程调度控制

- 进程切换调度
- 进程创建与终止
- 进程图像交换

子进程创建后可能先返回用户态，故有不确定性

```
main()
{
   if( fork()==0 )
   {
      printf("child process id is : %d\n",getpid());

   }
   else
   {

      printf("parent process id is: %d\n",getpid());

   }
}
```

先调度父进程：
parent process id is: XXX
child process id is: XXX

先调度子进程：
child process id is: XXX
parent process id is: XXX

子进程创建后可能先返回用户态，故有不确定性

```
main()
{
   if( fork()==0 )
   {
      printf("child process id is : %d\n",getpid());

   }
   else
   {
      sleep(1) ;
      printf("parent process id is: %d\n",getpid());

   }
}
```

如何确保子进程先终止?

现在能保证子进程先终止了?

在负载重的系统中仍然没法保证!

子进程创建后可能先返回用户态，故有不确定性

```
main()
{
   if( fork()==0 )
   {
      printf("child process id is : %d\n",getpid());
      exit();
   }
   else
   {
      int i=wait();
      printf("parent process id is: %d\n",getpid());
      printf("The child process %d is finished.\n ",i);
   }
}
```

```
child process id is: XXX
parent process id is: XXX
The child process XXX is finished.
```

子进程创建后可能先返回用户态，故有不确定性

```
main( )
{
   int i,j;
   if (fork())
   {
            i=wait(&j);     /* i 为终止的子进程的进程号 */
            printf("It is parent process. \n");
            printf("The finished child process is %d. \n", i);
            printf("The exit status is %d. \n", j);
   }
   else
   {
            printf("It is child process. \n");
            exit(0);
   }
}
```

It is child process.
It is parent process.
The finished child process is XXX.
The exit status is 0.

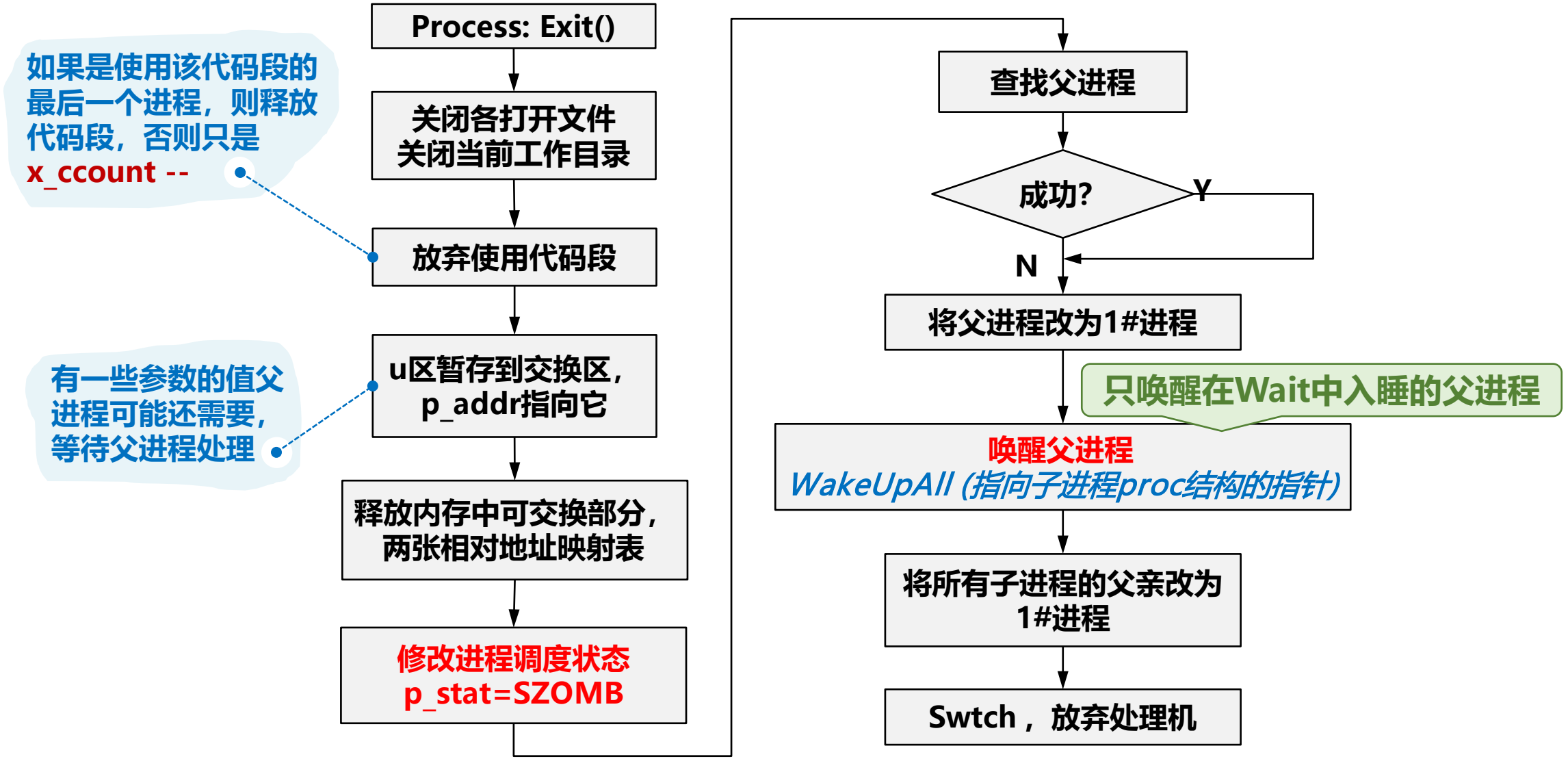
进程的创建与终止


```
int exit(int status)     /* 子进程返回给父进程的Return Code */
{
   int res;
   __asm__ __volatile__ ( "int $0x80":"=a"(res):"a"(1),"b"(status));
   if ( res >= 0 )
              return res;
   return -1;
}
```

```
SystemCallTableEntry
SystemCall::m_SystemEntranceTable[SYSTEM_CALL_NUM] =
{
    { 0, &Sys_NullSystemCall },/* 0 = indir*/
    { 1, &Sys_Rexit },     /* 1 = rexit*/
    { 0, &Sys_Fork },      /* 2 = fork*/
    { 3, &Sys_Read },      /* 3 = read*/
    ...
}
```

```
/*  1 = rexit            count = 1 */
int SystemCall::Sys_Rexit()
{
    User& u = Kernel::Instance().GetUser();
    u.u_procp->Exit();
    return 0;            /* GCC likes it ! */
}
```

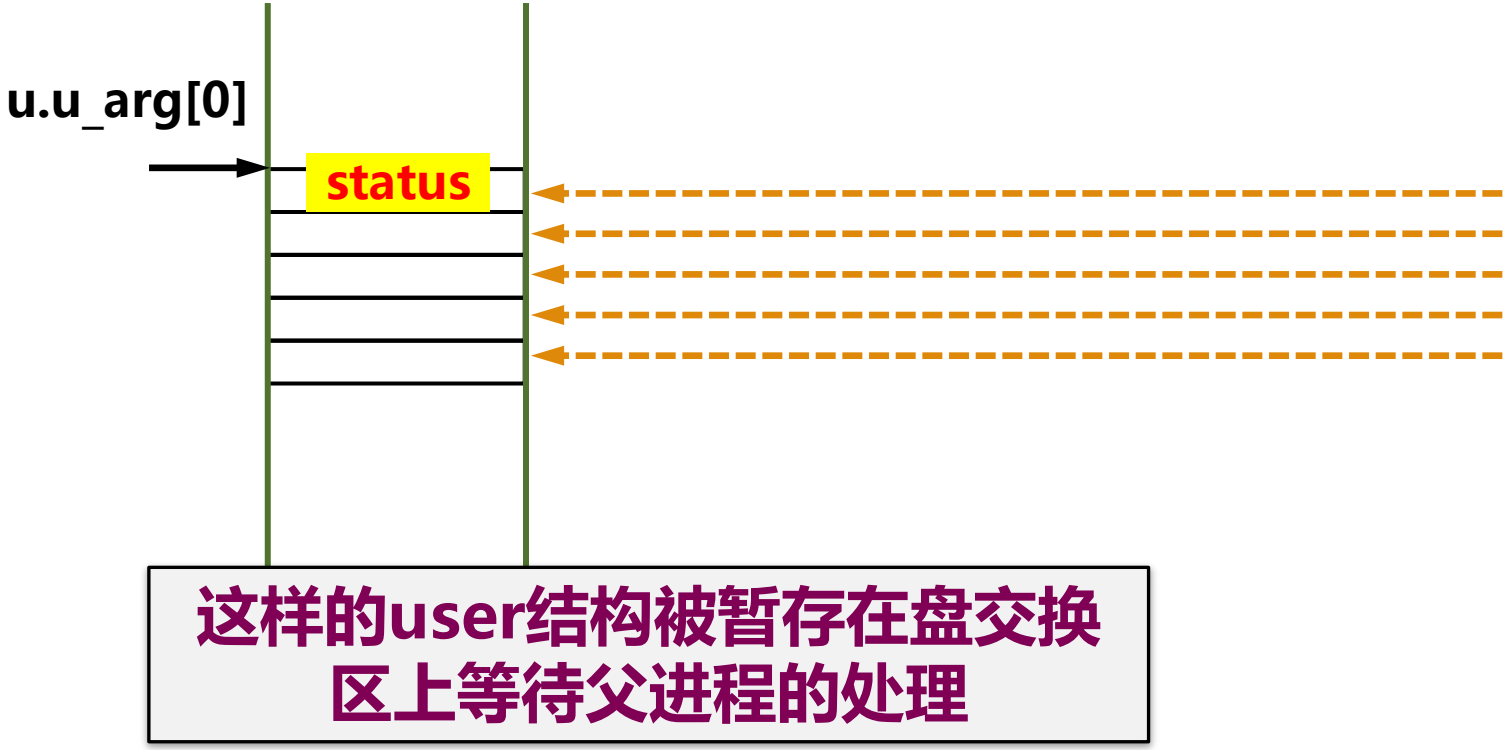
进程的创建与终止


如果是使用该代码段的最后一个进程，则释放代码段，否则只是 x_ccount --
有一些参数的值父进程可能还需要，等待父进程处理
Process: Exit()
关闭各打开文件
关闭当前工作目录
放弃使用代码段
u区暂存到交换区，p_addr指向它
释放内存中可交换部分，两张相对地址映射表
修改进程调度状态
p_stat=SZOMB
查找父进程
成功?
Y
N
将父进程改为1#进程
只唤醒在Wait中入睡的父进程
唤醒父进程
WakeUpAll (指向子进程proc结构的指针)
将所有子进程的父亲改为1#进程
Swtch，放弃处理机

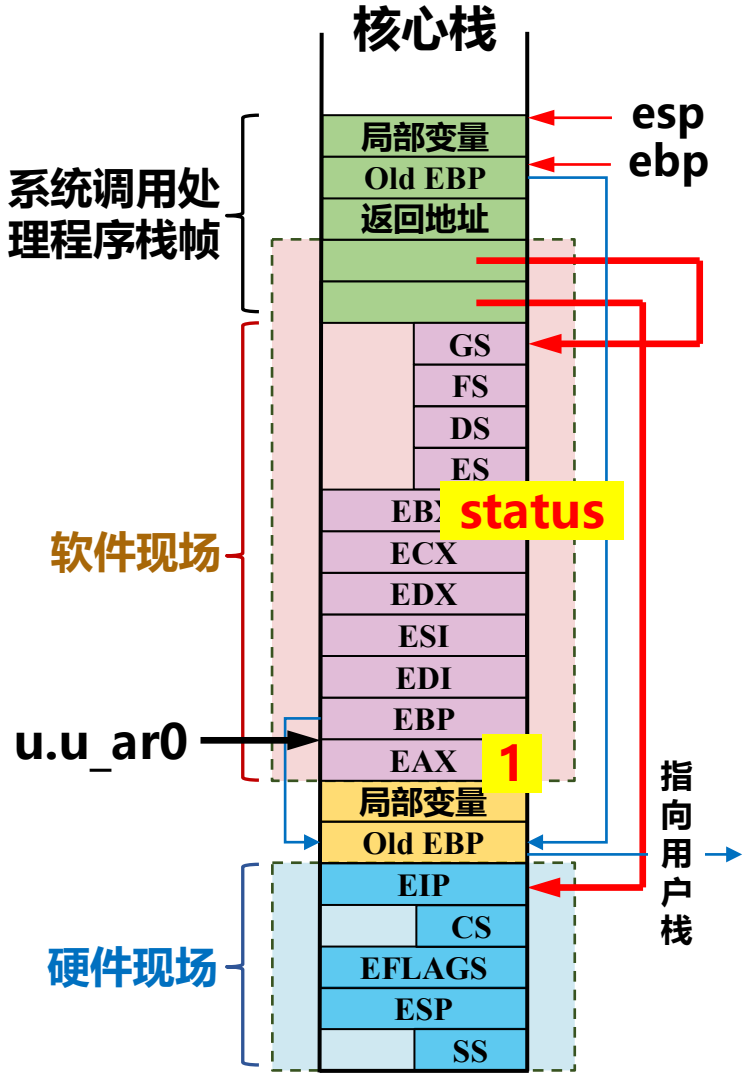
进程的创建与终止

子进程的User结构

进程的user结构



这样的user结构被暂存在盘交换区上等待父进程的处理



中断发生时现运行进程在用户态运行

进程的创建与终止


```
int wait(int* status) /* 获取子进程返回的Return Code */
{
   int res;
   __asm__ __volatile__ ( "int $0x80":"=a"(res):"a"(7),"b"(status));
   if ( res >= 0 )
            return res;
   return -1;
}
```

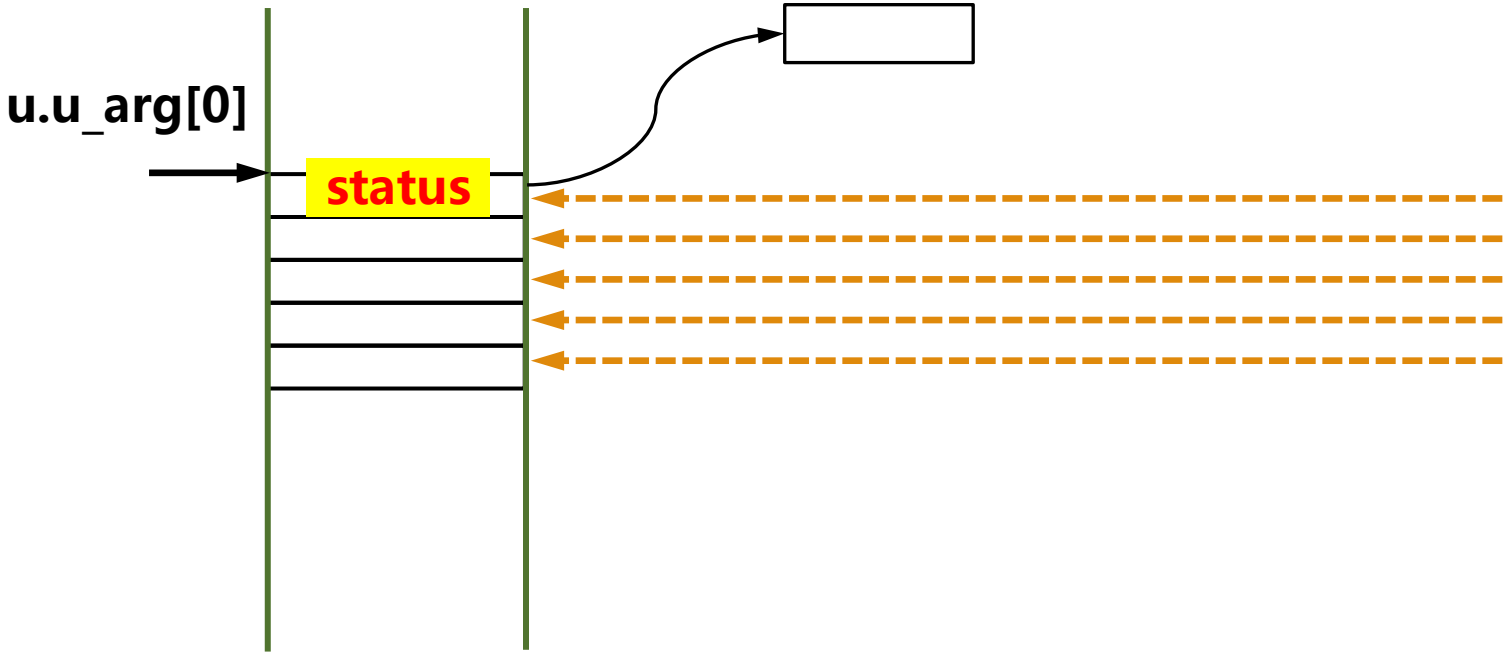
```
SystemCallTableEntry
SystemCall::m_SystemEntranceTable[SYSTEM_CALL_NUM] =
{
    { 0, &Sys_NullSystemCall },/* 0 = indir*/
    { 1, &Sys_Rexit },     /* 1 = rexit*/
    ...
    { 1, &Sys_Wait },      /* 7 = wait*/
    ...
}
```

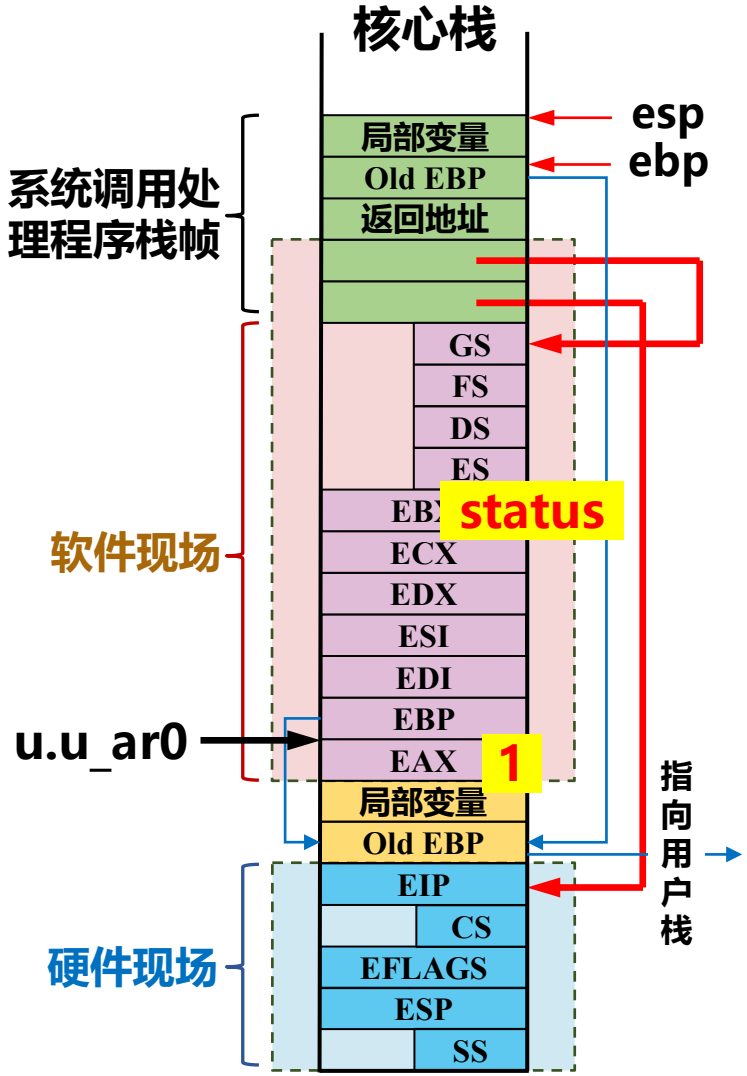
```
/*  7 = wait            count = 1 */
int SystemCall::Sys_Wait()
{
    ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
    procMgr.Wait();
    return 0;           /* GCC likes it ! */
}
```

进程的创建与终止

父进程的User结构

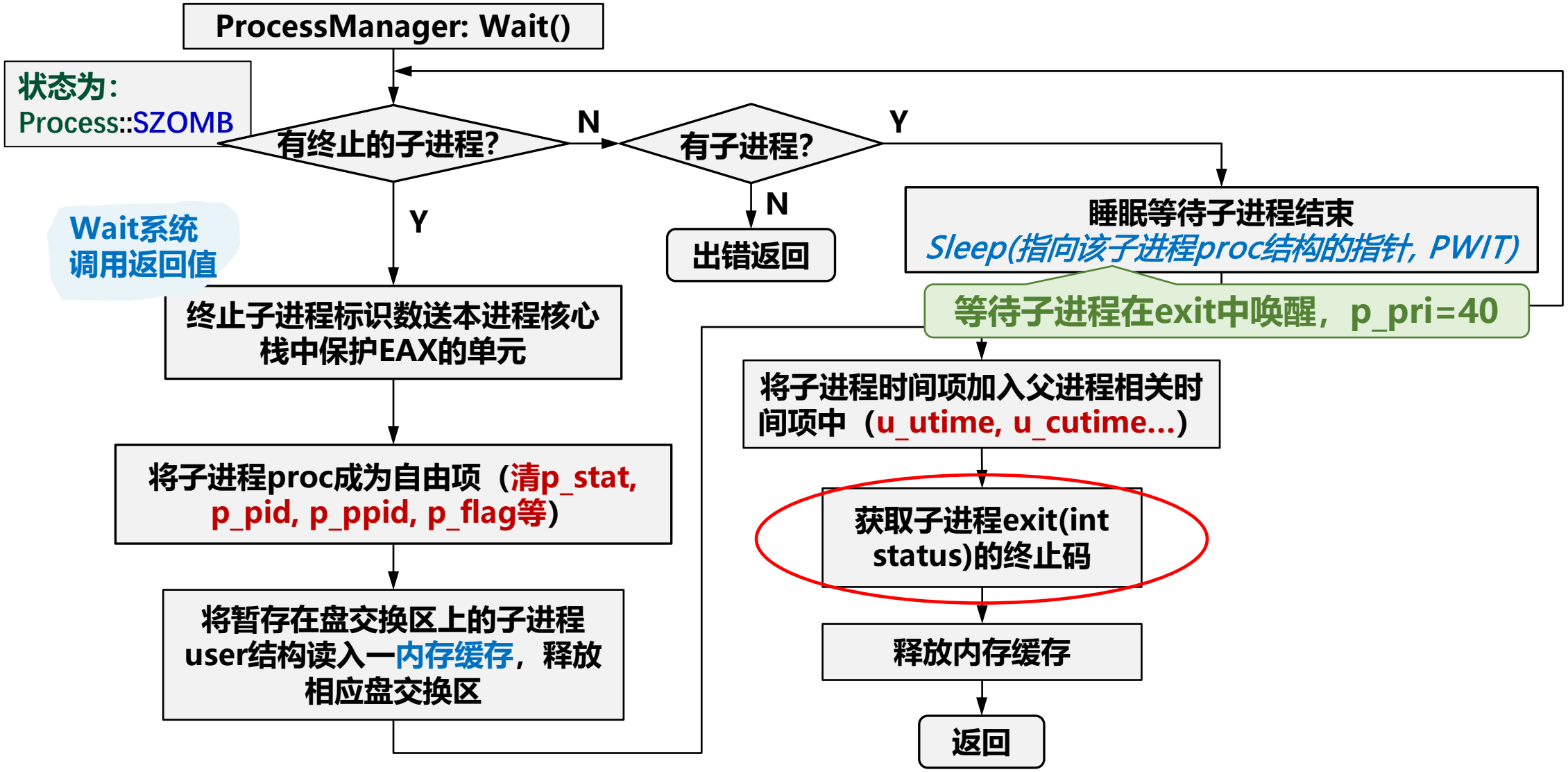
进程的user结构





中断发生时现运行进程在用户态运行

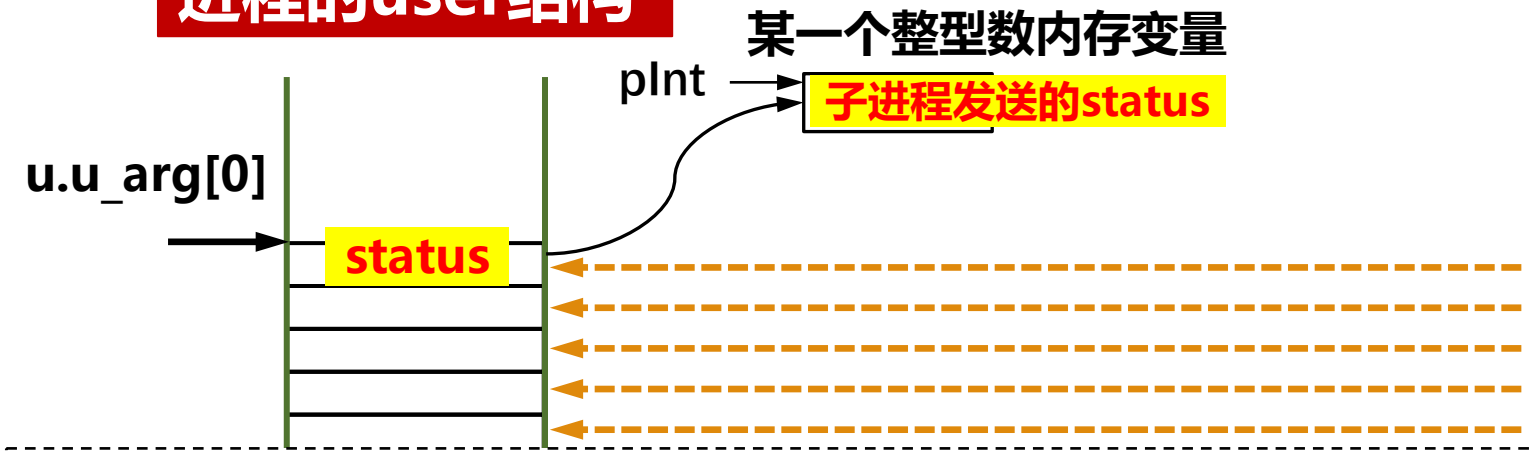
进程的创建与终止


ProcessManager: Wait()
状态为:
Process::SZOMB
有终止的子进程?
N
有子进程?
Y
N
出错返回
Y
Wait系统
调用返回值
睡眠等待子进程结束
Sleep(指向该子进程proc结构的指针, PWIT)
等待子进程在exit中唤醒，p_pri=40
终止子进程标识数送本进程核心栈中保护EAX的单元
将子进程时间项加入父进程相关时间项中（u_utime, u_cutime...）
将子进程proc成为自由项（清p_stat, p_pid, p_ppid, p_flag等）
获取子进程exit(int status)的终止码
将暂存在盘交换区上的子进程user结构读入一内存缓存，释放相应盘交换区
释放内存缓存
返回

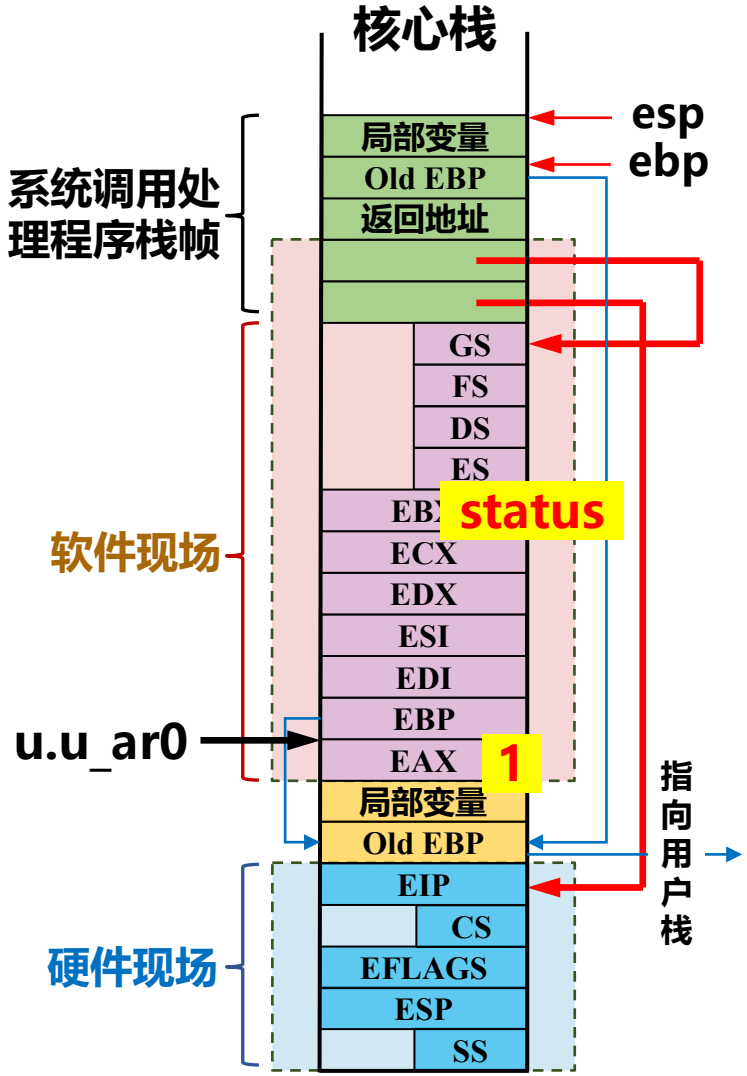
进程的创建与终止

父进程的User结构

进程的user结构




```
User* pUser = 指向读入缓存的子进程User结构的指针;
/* pInt指向通过status传递的某一个整型数内存变量 */
int* pInt = (int *)u.u_arg[0];
/* 该内存单元的值等于子进程User结构中u_arg[0]位置的值，*/
/* 即子进程执行exit时的参数stauts的值 */
*pInt = pUser->u_arg[0];
```



中断发生时现运行进程在用户态运行

子进程创建后可能先返回用户态，故有不确定性

```
main( )
{
   int i,j;
   if (fork())
   {
            i=wait(&j);     /* i 为终止的子进程的进程号 */
            printf("It is parent process. \n");
            printf("The finished child process is %d. \n", i);
            printf("The exit status is %d. \n", j);
   }
   else
   {
            printf("It is child process. \n");
            exit(0);
   }
}
```

It is child process.
It is parent process.
The finished child process is XXX.
The exit status is 0.

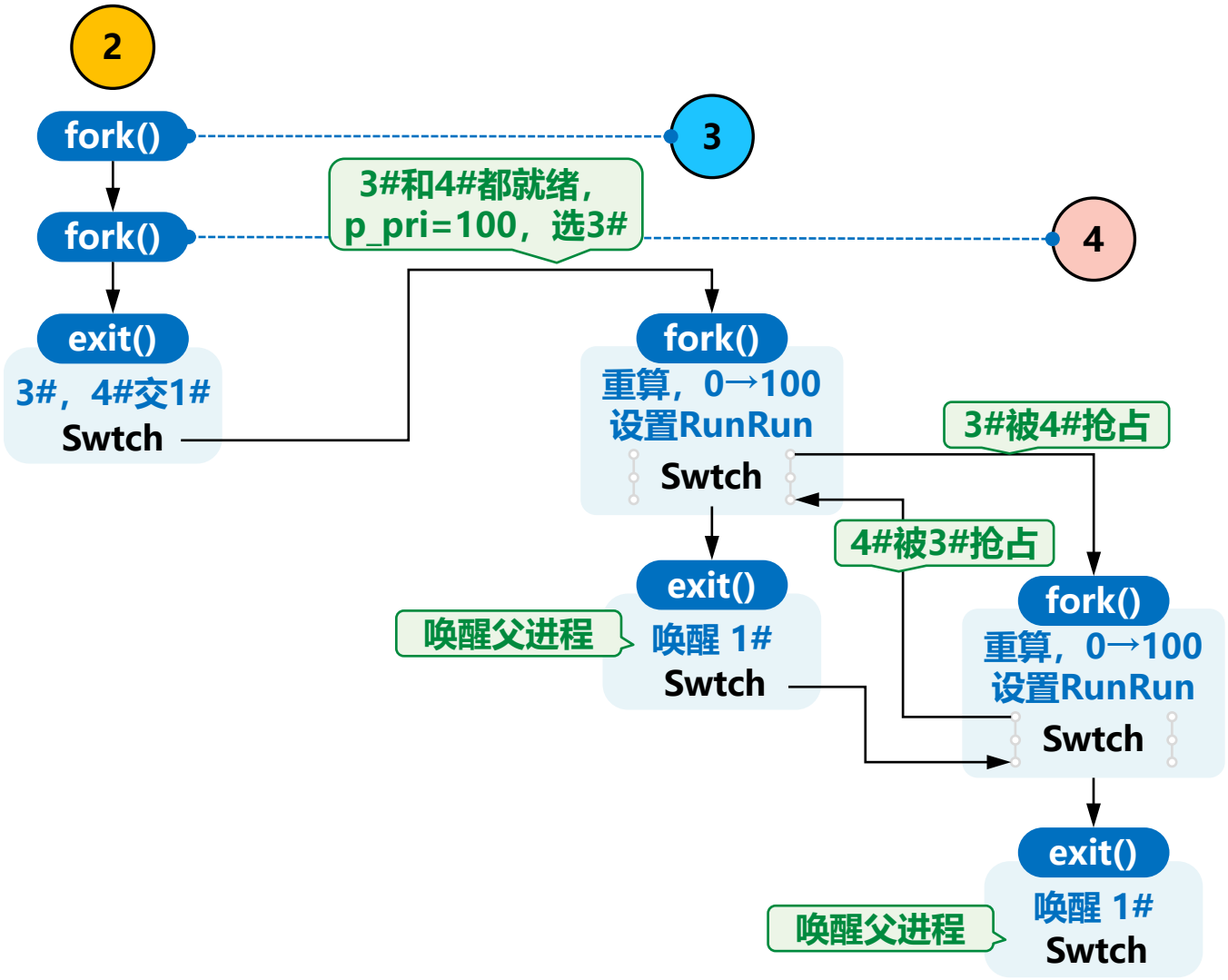
例：假设UNIX V6++中，存在如下所示的父子进程关系：

```
main()
{

  if( fork()==0 )
  {
    exit(0);
  }
  else
  {
    if( fork()==0 )
    {
      exit(0);
    }
    else
    {


    }
  }
}
```

	父进程不等待两个子进程	父进程等一个子进程	父进程等两个子进程
2#	创建两个子进程后， 执行exit终止		
3#	执行exit终止		
4#	执行exit终止		
终止顺序			
子进程回收			

进程的创建与终止


```
main()
{

  if( fork()==0 )
  {
    exit(0);
  }
  else
  {
    if( fork()==0 )
    {
      exit(0);
    }
    else
    {


    }
  }
}
```



进程的创建与终止

例：假设UNIX V6++中，存在如下所示的父子进程关系：

```
main()
{

  if( fork()==0 )
  {
    exit(0);
  }
  else
  {
    if( fork()==0 )
    {
      exit(0);
    }
    else
    {


    }
  }
}
```

	父进程不等待两个子进程	父进程等一个子进程	父进程等两个子进程
2#	创建两个子进程后， 执行exit终止		
3#	执行exit终止		
4#	执行exit终止		
终止顺序	2#，3#，4#		
子进程回收	2#终止时将3#，4#交给 1#；3#，4#终止时唤醒 1#回收		

代码如何修改使得父进程有可能被子进程抢占?

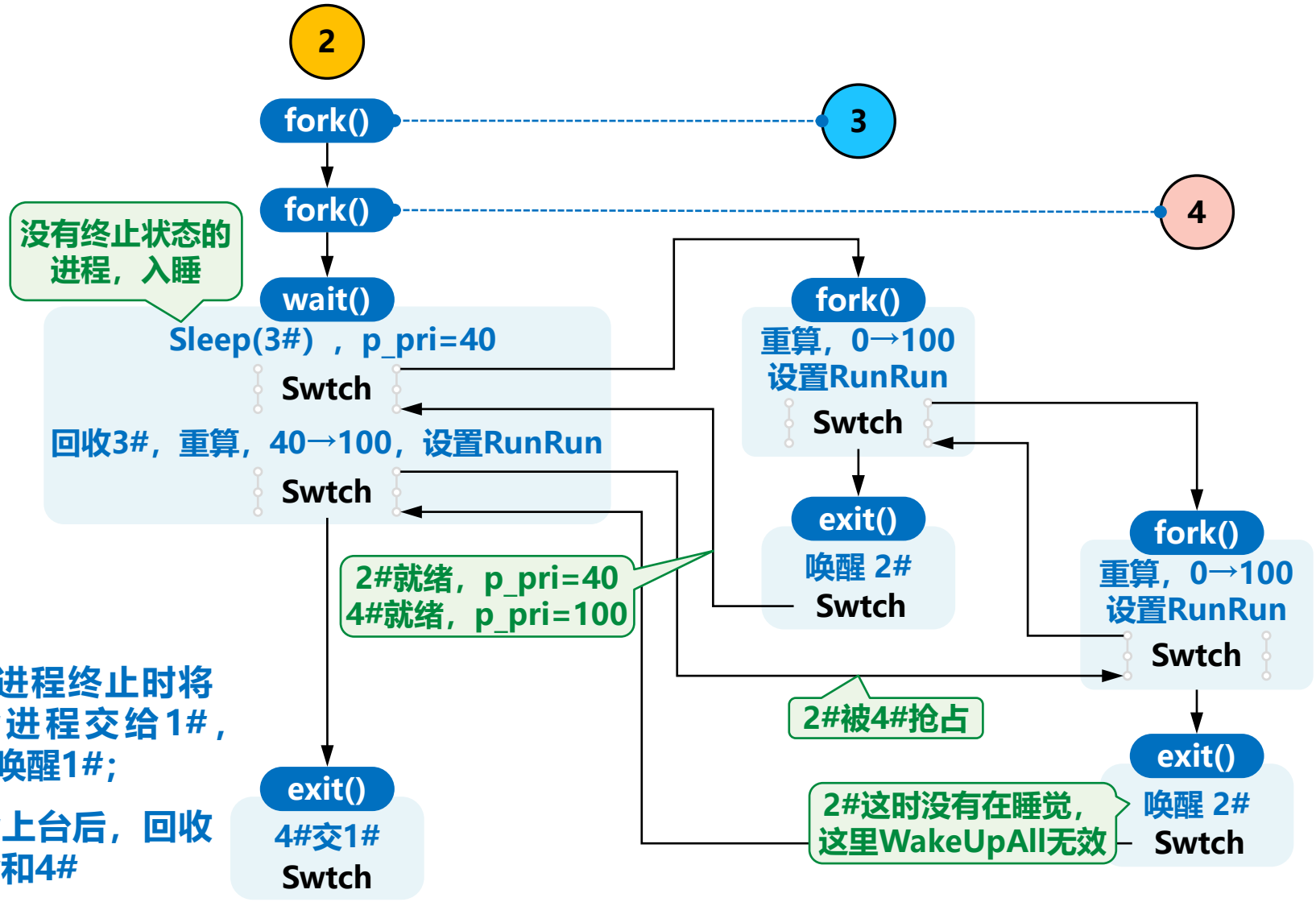
例：假设UNIX V6++中，存在如下所示的父子进程关系：

```
main()
{
   int i1, j1;
   if( fork()==0 )
   {
      exit(0);
   }
   else
   {
      if( fork()==0 )
      {
         exit(0);
      }
      else
      {
          i1=wait(&j1);

      }
   }
}
```

	父进程不等待两个子进程	父进程等一个子进程	父进程等两个子进程
2#	创建两个子进程后，执行exit终止	创建两个子进程后执行1个wait	
3#	执行exit终止	执行exit终止	
4#	执行exit终止	执行exit终止	
终止顺序	2#，3#，4#		
子进程回收	2#终止时将3#，4#交给1#；3#，4#终止时唤醒1#回收		

进程的创建与终止


```
main()
{
   int i1, j1;
   if( fork()==0 )
   {
      exit(0);
   }
   else
   {
      if( fork()==0 )
      {
         exit(0);
      }
      else
      {
          i1=wait(&j1);

      }
   }
}
```



父进程终止时将4#进程交给1#，并唤醒1#；

1#上台后，回收2#和4#

进程的创建与终止

例：假设UNIX V6++中，存在如下所示的父子进程关系：

```
main()
{
   int i1, j1;
   if( fork()==0 )
   {
      exit(0);
   }
   else
   {
      if( fork()==0 )
      {
         exit(0);
      }
      else
      {
          i1=wait(&j1);

      }
   }
}
```

	父进程不等待两个子进程	父进程等一个子进程	父进程等两个子进程
2#	创建两个子进程后，执行exit终止	创建两个子进程后执行1个wait	
3#	执行exit终止	执行exit终止	
4#	执行exit终止	执行exit终止	
终止顺序	2#，3#，4#	3#，	
子进程回收	2#终止时将3#，4#交给1#；3#，4#终止时唤醒1#回收	2#回收3#，2#终止时将4#交给1#，并唤醒1#回收2#和4#	

代码如何修改使得父进程回收4#进程而不是3#?

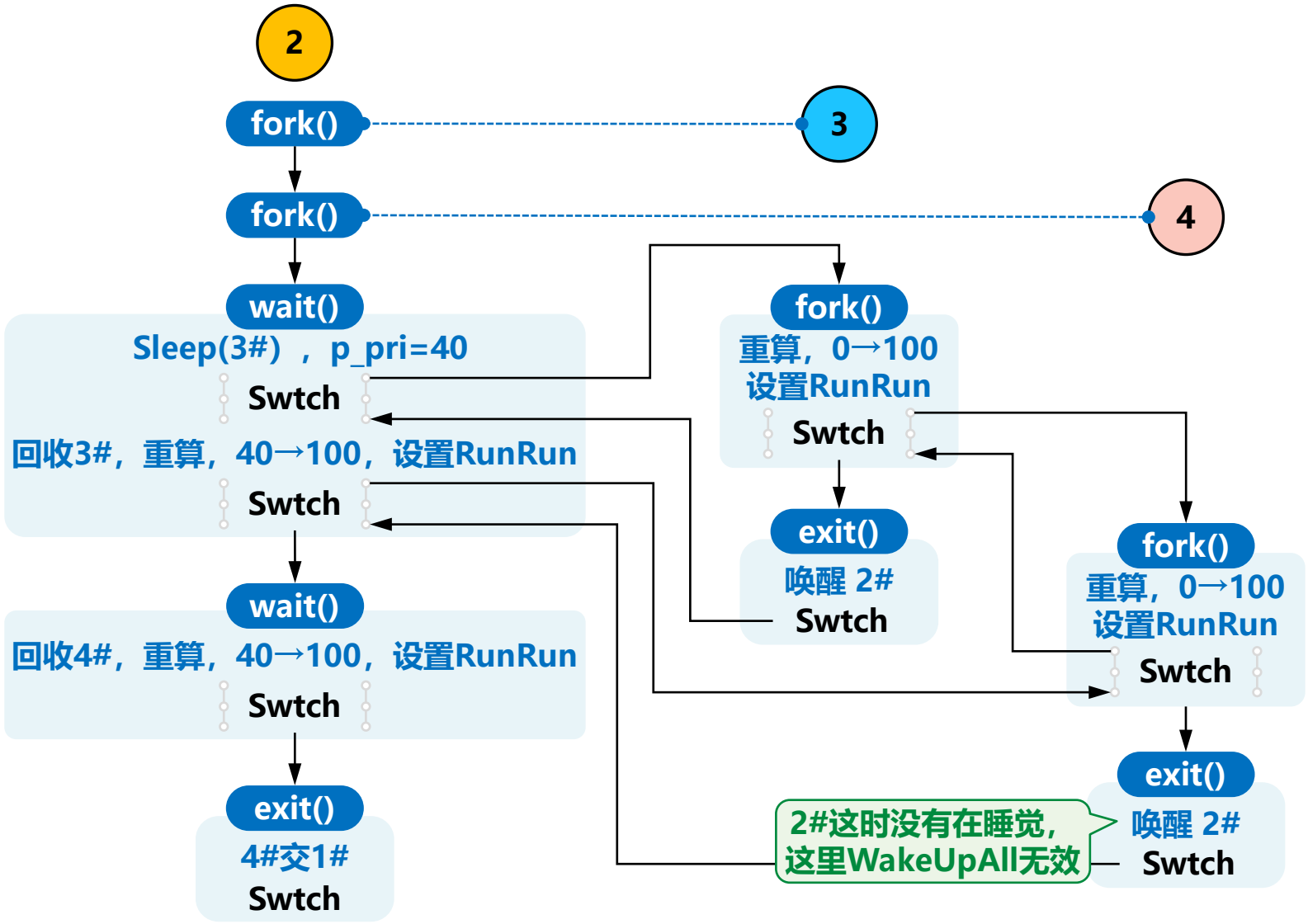
例：假设UNIX V6++中，存在如下所示的父子进程关系：

```
main()
{
   int i1, j1, i2, j2;
   if( fork()==0 )
   {
      exit(0);
   }
   else
   {
      if( fork()==0 )
      {
         exit(0);
      }
      else
      {
         i1=wait(&j1);
         i2=wait(&j2);
      }
   }
}
```

	父进程不等待两个子进程	父进程等一个子进程	父进程等两个子进程
2#	创建两个子进程后，执行exit终止	创建两个子进程后执行1个wait	创建两个子进程后执行2个wait
3#	执行exit终止	执行exit终止	执行exit终止
4#	执行exit终止	执行exit终止	执行exit终止
终止顺序	2#，3#，4#	3#，4#，2#	
子进程回收	2#终止时将3#，4#交给1#；3#，4#终止时唤醒1#回收	2#回收3#，2#终止时将4#交给1#，并唤醒1#回收2#和4#	

进程的创建与终止


```
main()
{
   int i1, j1, i2, j2;
   if( fork()==0 )
   {
      exit(0);
   }
   else
   {
      if( fork()==0 )
      {
         exit(0);
      }
      else
      {
         i1=wait(&j1);
         i2=wait(&j2);
      }
   }
}
```



进程的创建与终止

例：假设UNIX V6++中，存在如下所示的父子进程关系：

```
main()
{
   int i1, j1, i2, j2;
   if( fork()==0 )
   {
      exit(0);
   }
   else
   {
      if( fork()==0 )
      {
         exit(0);
      }
      else
      {
         i1=wait(&j1);
         i2=wait(&j2);
      }
   }
}
```

	父进程不等待两个子进程	父进程等一个子进程	父进程等两个子进程
2#	创建两个子进程后，执行exit终止	创建两个子进程后执行1个wait	创建两个子进程后执行2个wait
3#	执行exit终止	执行exit终止	执行exit终止
4#	执行exit终止	执行exit终止	执行exit终止
终止顺序	2#，3#，4#	3#，4#	
子进程回收	2#终止时将3#，4#交给1#；3#，4#终止时唤醒1#回收	2#回收3#，2#终止时将4#交给1#，并唤醒1#回收2#和4#	两个wait分别回收3#和4#

代码如何修改使得父进程等到某一个指定的子进程后就不再等了?

本节小结

1　UNIX进程的创建与终止

2　UNIX父子进程的同步

请阅读教材：187页 ~ 204页

E13：进程管理（UNIX的进程创建于父子进程同步）

P05：UNIX V6++父子进程的同步

关于P05：UNIX V6++父子进程的同步