

操作系统 第四章 进程管理

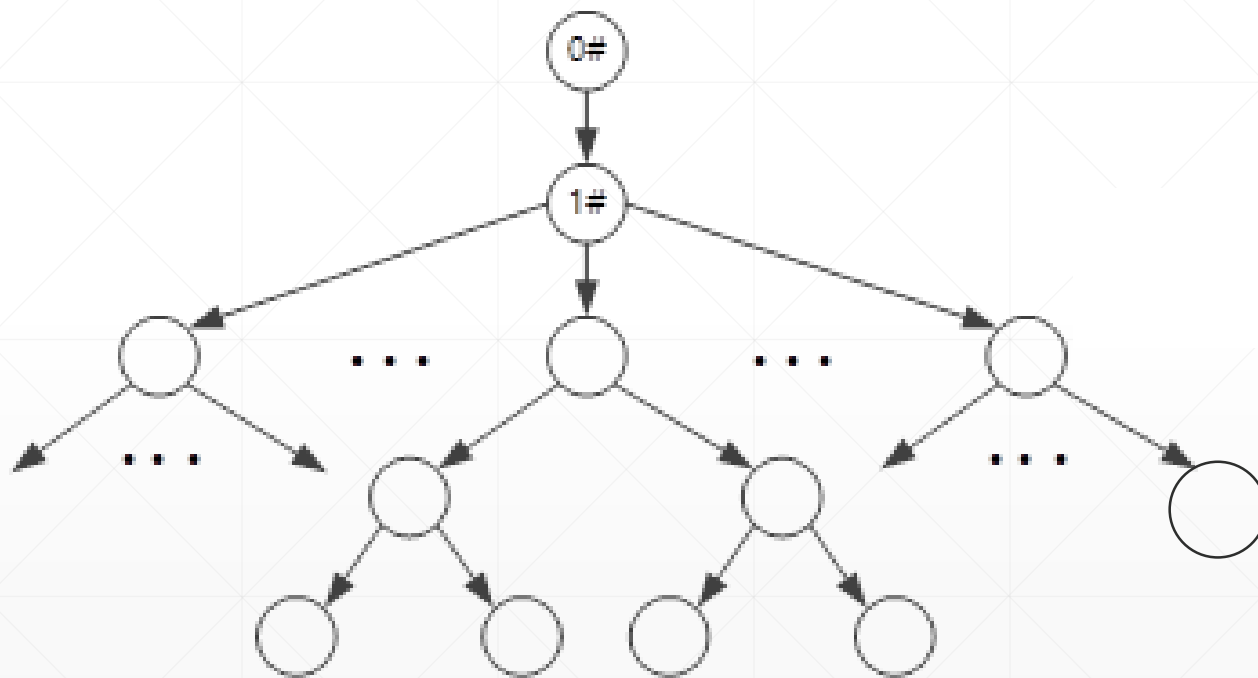
4.6 进程树，应用程序加载 和 进程终止



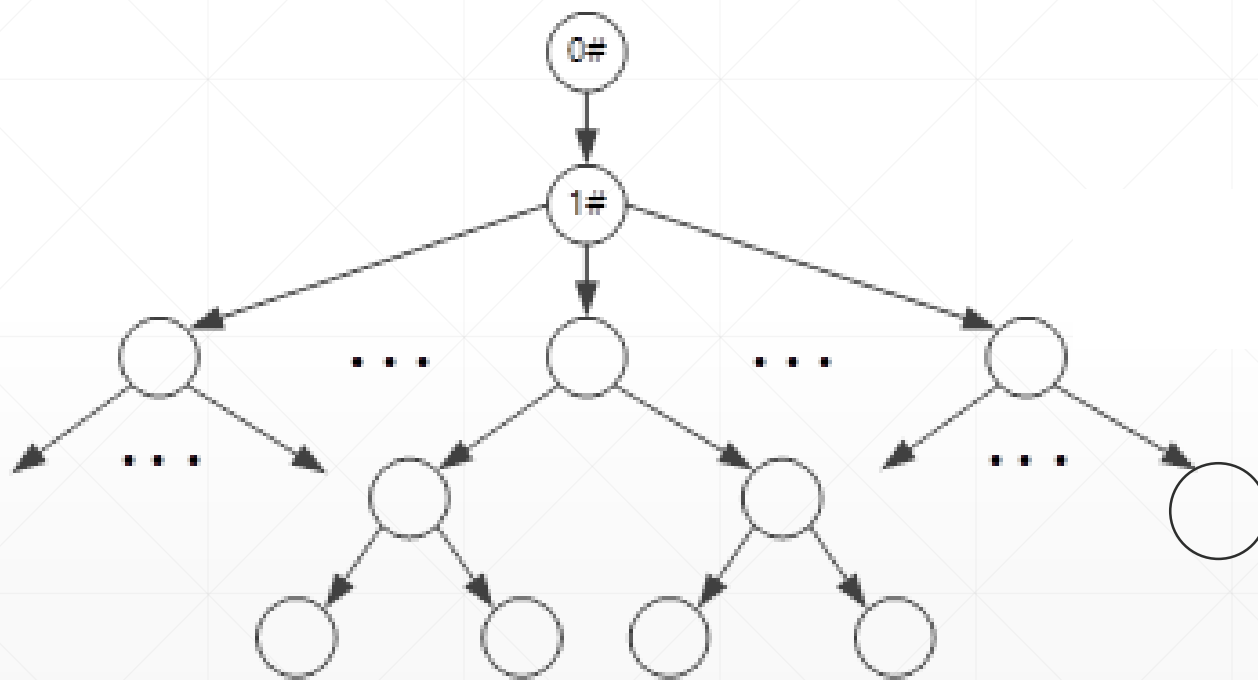
Part 1: 进程树 和 可执行程序加载

所有进程都是fork创建出来的

- 在Unix系统中，除0#进程外，所有进程都是fork创建出来的。
- 执行fork的是父进程，被创建的是子进程。
 - 0#进程创建1#进程。1#进程，是整个Unix系统的监控进程。
 - 1#进程为每个加电的tty创建一个进程，这个进程先初始化终端；然后执行login程序，接受用户输入的用户名和口令字；最终会执行shell程序，变成shell进程，为使用这个终端的用户提供命令行界面服务。
 - shell进程解析命令行，为命令行中出现的每个应用程序创建一个进程。这个进程负责执行这个应用程序。
 - 进程之间的父子关系，绘成进程树。



- 每个进程承担一项任务。任务完成，进程终止。
- 0#进程是内核的服务器进程，永不终止。
- 1#进程是系统监控进程。永不终止。
 - 监控终端运行状态。shell进程终止后，创建新进程等待新用户。
 - 回收孤儿进程的PCB
- shell进程为用户提供命令行界面服务，用户logout，shell进程终止。
- 其余进程，应用程序执行完毕，进程终止。



exec, 加载、执行磁盘上的可执行程序

需要执行应用程序时, Unix系统 (shell进程)

- fork 一个新进程
- 让新进程 exec 装入目标程序的图像, 承担执行应用程序的任务

例:

```
$ gcc -o testStack testStack.c
```

新建gcc进程。gcc进程是shell进程的子进程。

- 子进程 fork 返回时, 执行shell程序
- 随后, 子进程执行exec系统调用
 - 清除线性地址空间中的shell进程图像
 - 装入目标程序gcc, 承担执行gcc程序的任务

main() // shell程序代码片段

```
{
    .....
    int i;
    if( i = fork( ))
        .....
    else
        exec ( "gcc" , arg1, arg2.....);
}
```

子进程要执行的程序

程序的命令行参数



1、exec("commd" , arg1, arg2.....) 概貌

- 目录搜索可执行程序commd 没找到, 出错返回
- 有 执行权限X 吗? No, 出错返回
- Yes, 释放代码段

释放数据段和堆栈段 (存命令行参数arg1, arg2...)

- 读commd文件的程序头, 得程序代码段、数据段和堆栈段.....的段头 (section header)
- 写 MemoryDescriptor
- 清空、重写相对虚实地址映射表
- Text数组有commd程序吗? 有, 置可复用标识
无, 为代码段分配Text结构, 分配物理内存, 清0
- 为可交换部分分配物理内存 尺寸: 数据段 + bss + 初始栈 + 2页
- MaptoUserPageTable, 建立用户空间的地址映射关系
- 从磁盘可执行文件中读入代码段 (无可复用标识) 和 数据段

- 得 entry Point
- 得section headers
 - virtualBegin、virtualSize
 - fileOffset, fileBegin
 - mode (只读还是可写)

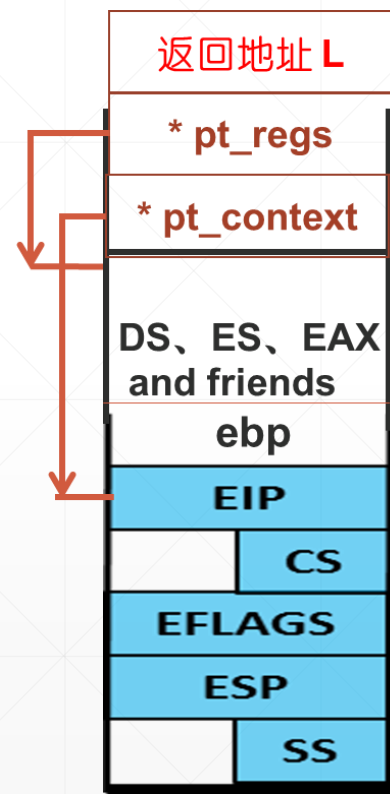
```
class MemoryDescriptor
{
public:
    /* 用户空间大小 8M 0x0 - 0x800000 2 PageTable */
    static const unsigned int USER_SPACE_SIZE = 0x800000;
    static const unsigned int USER_SPACE_PAGE_TABLE_CNT = 0x2;
    static const unsigned long USER_SPACE_START_ADDRESS = 0x0;

    ...

public:
    PageTable* m_UserPageTableArray;

    /* 以下数据都是线性地址 */
    unsigned long m_TextStartAddress; /* 代码段起始地址 */ 0x401000
    unsigned long m_TextSize; /* 代码段长度 */ 0x2000
    unsigned long m_DataStartAddress; /* 数据段起始地址 */ 0x402000
    unsigned long m_DataSize; /* 数据段长度 */ 0x2000
    unsigned long m_StackSize; /* 栈段长度 */ 0x1000
};
```

- 栈底创建 main 栈帧，压入 可执行程序名，arg1，arg2.....
 - **esp = 8M**
 - 用push的方法依次压入命令行参数
 - 完成后，esp是用户栈的栈顶
- 清0 u_signal 数组 //设置默认的信号处理方式。收到信号后，进程被杀死。
- 清0 saveContext 压入的所有用户态寄存器（ebp不能碰）
- `pContext->eip = 0x00000000; /* 退出到ring3特权级下从线性地址0x00000000处 runtime()开始执行 */`
- `//pContext->eip = parser.EntryPointAddress;`
- `pContext->xcs = Machine::USER_CODE_SEGMENT_SELECTOR;`
- `pContext->eflags = 0x200; // 开中断`
- `pContext->esp = esp;`
- `pContext->xss = Machine::USER_DATA_SEGMENT_SELECTOR;`

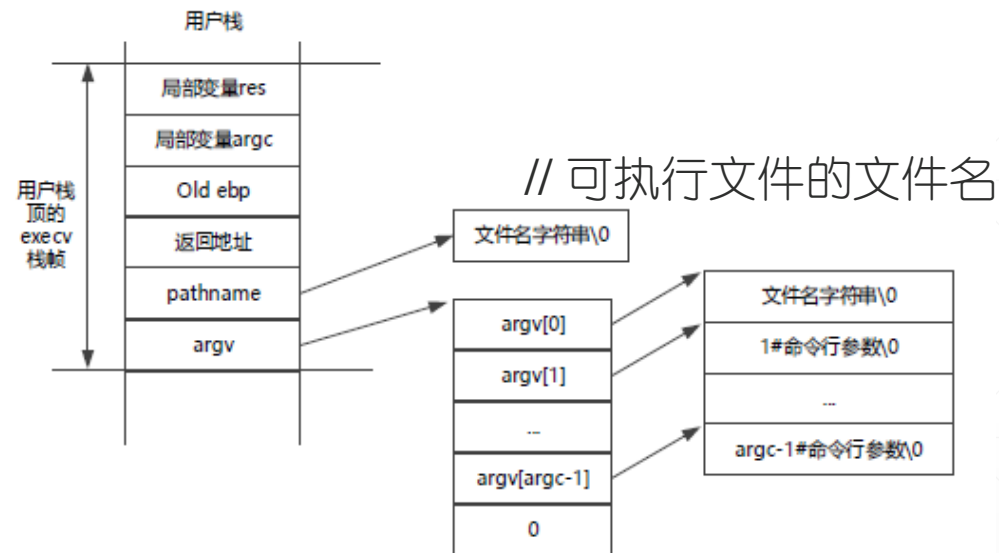


2、代码 —— exec 的钩子函数

```

4 int execl(char *pathname, char *argv[])
5 {
6     int res;
7     int argc = 0;
8     while(argv[argc] != 0) // 清点命令行参数的数量: argc
9         argc++;
10    __asm__ volatile ( "int $0x80": "=a"(res): "a"(11), "b"(pathname), "c"(argc), "d"(argv));
11    if ( res >= 0 )
12        return res;
13    return -1;
14 }

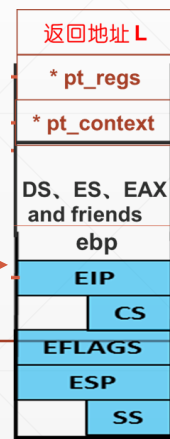
```



陷入内核后，核心栈底用户寄存器区：EBX、ECX 和 EDX

u_arg[0] u_arg[1] u_arg[2]

u.u_dirp = (char *)u.u_arg[0]; u.u_arg[4] = (int)context;



用户空间的其它位置


```
ProcessManager.cpp  main.cpp  MemoryDescriptor.cpp
457 void ProcessManager::Exec()
458 {
459     Inode* pInode;
460     Text* pText;
461     User& u = Kernel::Instance().GetUser();
462     FileManager& fileMgr = Kernel::Instance().GetFileManager();
463     UserPageManager& userPgMgr = Kernel::Instance().GetUserPageManager();
464     KernelPageManager& kernelPgMgr = Kernel::Instance().GetKernelPageManager();
465     BufferManager& bufMgr = Kernel::Instance().GetBufferManager();
466
467     Diagnose::Write("Process %d execing\n", u.u_procp->p_pid);
468     pInode = fileMgr.NameI(FileManager::NextChar, FileManager::OPEN);
469     if ( NULL == pInode )    //搜索目录失败
470     {
471         return;
472     }
473
474     /* 如果同时进行图像改换的进程数超出限制，则先进入睡眠 */
475     while( this->ExeCnt >= NEXEC )
476     {
477         u.u_procp->Sleep((unsigned long)&ExeCnt, ProcessManager::EXPRI);
478     }
479     this->ExeCnt++;
480
481     /* 进程必需拥有可执行文件的执行权限，且被执行的只能是一般文件。 */
482     if ( fileMgr.Access(pInode, Inode::IEXEC) || (pInode->i_mode & Inode::IFMT) != 0 )
483     {
484         fileMgr.m_InodeTable->IPut(pInode);
485         if ( this->ExeCnt >= NEXEC )
486         {
487             WakeUpAll((unsigned long)&ExeCnt);
488         }
489         this->ExeCnt--;
490         return;
491     }
492 }
```

1、文件系统搜索可执行程序。 `pInode`是可执行程序的文件控制块

2、检查进程有没有这个程序的可执行权限

```

493  PEParser parser;
494
495  if ( parser.HeaderLoad(pInode)==false )
496  {
497      fileMgr.m_InodeTable->IPut(pInode);
498      return;
499  }
500
501  /* 获取分析PE头结构得到正文段的起始地址、长度 */
502  u.u_MemoryDescriptor.m_TextStartAddress = parser.TextAddress;
503  u.u_MemoryDescriptor.m_TextSize = parser.TextSize;
504
505  /* 数据段的起始地址、长度 */
506  u.u_MemoryDescriptor.m_DataStartAddress = parser.DataAddress;
507  u.u_MemoryDescriptor.m_DataSize = parser.DataSize;
508
509  /* 堆栈段初始化长度 */
510  u.u_MemoryDescriptor.m_StackSize = parser.StackSize;
511
512  if ( parser.TextSize + parser.DataSize + parser.StackSize + PageManager::PAGE_SIZE > MemoryDescriptor::USER_SPACE_SIZE - parser.TextAddress )
513  {
514      fileMgr.m_InodeTable->IPut(pInode);
515      u.u_error = User::ENOMEM;
516      return;
517  }
518

```

1、借用2张页表，读入文件头（主要是section headers 和 entryptoint）

2、写进程的内存描述符

数据段和堆栈段之间一定要空一个逻辑页，这页PTE是null。
(否则堆栈涨到擦除了全局变量，系统都不会报错)

8M
(用户空间总长)

4M+4K (代码段起始地址)

3、可执行文件需要的线性空间 > 8M，系统无法执行该进程，OOM

总长
[代码+数据 (含bss) +堆栈]



```
ProcessManager.cpp | main.cpp | MemoryDescriptor.cpp | sys.c
525 int allocLength = (parser.StackSize + PageManager::PAGE_SIZE * 2 - 1) >> 13 << 13;
526 unsigned long fakeStack = kernelPgMgr.AllocMemory(allocLength);
527
528 int argc = u.u_arg[1];
529 char** argv = (char**)u.u_arg[2];
530
531 /* esp定位到栈底 */
532 unsigned int esp = MemoryDescriptor::USER_SPACE_SIZE;
533 /* 使用核心态页表映射，所以在物理地址上加0xC0000000构成线性地址 */
534 unsigned long desAddress = fakeStack + allocLength + 0xC0000000;
535 //unsigned long desAddress = fakeStack + parser.StackSize + 0xC0000000;
536 int length;
537
538 /* 复制argv[]指针数组指向的命令行参数字符串 */
539 for (int i = 0; i < argc; i++)
540 {
541     length = 0;
542     /* 计算参数字符串长度，length不含'\0' */
543     while( NULL != argv[i][length] )
544     {
545         length++;
546     }
547     desAddress = desAddress - (length + 1);
548     /* 拷贝时将'\0'一起拷贝过去 */
549     Utility::MemCopy((unsigned long)argv[i], desAddress, length + 1);
550     /* 将参数字符串在新进程图像用户栈中的起始位置存入argv[i]，用户栈位于进程逻辑地址空间0x800000的底部 */
551     esp = esp - (length + 1);
552     argv[i] = (char*)esp;
553 }
554
555 /* 后续存放的是int型数值，这里以16字节边界对齐 */
556 desAddress = desAddress & 0xFFFFFFF0;
557 esp = esp & 0xFFFFFFF0;
558
559 /* 复制argc和argv[] */
560 int endValue = 0;
561 desAddress -= sizeof(endValue);
562 esp -= sizeof(endValue);
563 /* 向用户栈中写入endValue作为argv[]的结束 */
564 Utility::MemCopy((unsigned long)&endValue, desAddress, sizeof(endValue));
565
566 desAddress -= argc * sizeof(int);
567 esp -= argc * sizeof(int);
568 /* 写入argv[]的内容 */
569 Utility::MemCopy((unsigned long)argv, desAddress, argc * sizeof(int));
570
```

1、在页表区分配2个连续物理页框，放命令行参数

2、exec的入口参数是程序的命令行参数：个数argc 和 参数字符串数组 argv

3、在借来的页框里构造main栈帧，这就留下了命令行参数

```
570
571 /* 令endValue指向当前栈中argv[]的起始地址，即argv[]入栈完毕后当前栈顶地址 */
572 endValue = esp;
573 desAddress -= sizeof(int);
574 esp -= sizeof(int);
575 Utility::MemCopy((unsigned long)&endValue, desAddress, sizeof(int));
576
577 /* 最后入栈argc */
578 desAddress -= sizeof(int);
579 esp -= sizeof(int);
580 Utility::MemCopy((unsigned long)&argc, desAddress, sizeof(int)); /* Done! */
581
```



2、代码 —— exec 的钩子函数

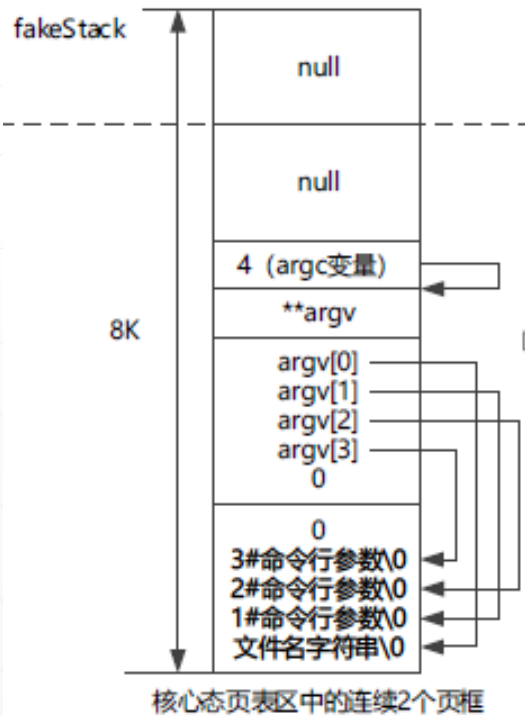
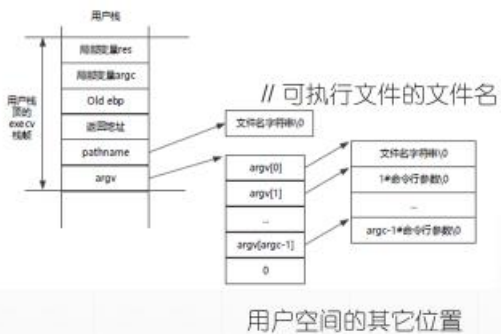
```

1 int exeov(char *pathname, char *argv[])
2 {
3     int res;
4     int argc = 0;
5     while(argv[argc] != 0) // 清点命令行参数的数量: argc
6         argc++;
7     __asm__ volatile ( "int $0x50":"=a"(res):"a"({1}),"b"(pathname),"c"(argc),"d"(argv));
8     if ( res >= 0 )
9         return res;
10    return -1;
11 }

```

陷入内核后，核心栈底用户寄存器区：EBX、ECX 和 EDX

u_arg[0] u_arg[1] u_arg[2]



```

582
583 /* 释放原进程图像的共享正文段，数据段，堆栈段 */
584 if ( u.u_procp->p_textp != NULL )
585 {
586     u.u_procp->p_textp->XFree();
587     u.u_procp->p_textp = NULL;
588 }
589 u.u_procp->Expand(ProcessManager::USIZE);
590
591 pText = NULL;
592 /* 分配一个空闲Text结构，或者和其它进程共享同一正文段 */
593 for ( int i = 0; i < ProcessManager::NTEXT; i++ )
594 {
595     if ( NULL == this->text[i].x_iptr )      /* 记下找到的第一个空闲text结构 */
596     {
597         if ( NULL == pText )
598         {
599             pText = &(this->text[i]);
600         }
601     }
602     else if ( pInode == this->text[i].x_iptr ) /* 如果，这不是一个空闲text结构，看一下text结构指向的可执行文件是exec系统调用要执行的应用程序吗？ */
603     {
604         this->text[i].x_count++;
605         this->text[i].x_ccount++;
606         u.u_procp->p_textp = &(this->text[i]);
607         pText = NULL; /* 与其它进程共享同一正文段，则pText重新清零，否则指向一空闲Text结构 */
608         break;
609     }
610 }

```

1、可交换部分缩小至只有PPDA区，这就释放了数据段和堆栈段

3、分配一个空闲 text 结构

2、有其它进程正在执行该程序。复用 text结构 和 代码段。
不需要读可执行文件中的代码。

```
613 int sharedText = 0;
614
615 /* 没有可共享的现成Text结构, 进行相应初始化 */
616 if ( NULL != pText )
617 {
618     /*
619     * 此处i_count++用于平衡XFree()函数中的IPut(x_iptr); 倘若只有Exec()开始处
620     * 调用NameI()函数中IGet(), 以及Exec()结尾处IPut()释放exe文件的Inode回到空闲Inode表,
621     * 极端情况下: 若后续进程很快也Exec(), 获取空闲Inode恰好是之前加载的exe文件释放的Inode,
622     * 则会错误地判断: pInode (当前exe对应Inode) == this->text[i].x_iptr(之前exe文件Inode),
623     * 导致和之前进程共享同一Text结构, 即同一正文段, 而实际上本该是两个独立的程序。
624     */
625     pInode->i_count++;
626
627     pText->x_ccount = 1;
628     pText->x_count = 1;
629     pText->x_iptr = pInode;
630     pText->x_size = u.u_MemoryDescriptor.m_TextSize;
631     /* 为正文段分配内存, 而具体正文段内容的读入需要等到建立页表映射之后, 再从mapAddress地址起始的exe文件中读入 */
632     pText->x_caddr = userPgMgr.AllocMemory(pText->x_size);
633     pText->x_daddr = Kernel::Instance().GetSwapperManager().AllocSwap(pText->x_size);
634     /* 建立u区和Text结构的勾连关系 */
635     u.u_procp->p_textp = pText;
636 }
637 else
638 {
639     pText = u.u_procp->p_textp;
640     sharedText = 1;
641 }
```

初始化 text 结构, x_size是代码段长度

为代码段分配内存 和 盘交换区空间

text结构连上 Process结构



```
ProcessManager.cpp | main.cpp | MemoryDescriptor.cpp | sys.c | FileManager.cpp
643 unsigned int newSize = ProcessManager::USIZE + u.u_MemoryDescriptor.m_DataSize + u.u_MemoryDescriptor.m_StackSize;
644 /* 将进程图像由USIZE扩充为USIZE + dataSize + stackSize */
645 u.u_procp->Expand(newSize); // 为可交换部分分配内存空间, 修改 p_addr 和 p_size
646
647 /* 根据正文段、数据段、堆栈段长度建立相对地址映照表, 并加载到页表中 */
648 u.u_MemoryDescriptor.EstablishUserPageTable(parser.TextAddress, parser.TextSize, parser.DataAddress, parser.DataSize, parser.StackSize);
649 // 写相对虚实地址映射表, 刷系统页表, 建立地址映射关系
650 /* 从exe文件中依次读入.text段、.data段、.rdata段、.bss段 */
651 parser.Relocate(pInode, sharedText);
652 // 清0分配给代码段, bss段和数据段的内存空间。
653 /* .text段在swap分区上留复本 */ // 从可执行文件读入代码 和 全局变量初值
654 if(sharedText == 0)
655 {
656     u.u_procp->p_flag |= Process::SLOCK;
657     bufMgr.Swap(pText->x_daddr, pText->x_caddr, pText->x_size, Buf::B_WRITE);
658     u.u_procp->p_flag &= ~Process::SLOCK;
659 }
```



```
ProcessManager.cpp  main.cpp  MemoryDescriptor.cpp  sys.c  FileManager.cpp  Process.cpp
20
21 unsigned int PEParse::Relocate(Inode* p_inode, int sharedText)
22 {
23     User& u = Kernel::Instance().GetUser();
24     unsigned long srcAddress, desAddress;
25     unsigned cnt = 0;
26     unsigned int i = 0;
27     unsigned int i0 = 0;
28
29     /* 如果可以和其它进程共享正文段，无需文件中读入正文段 */
30     PageTable* pUserPageTable = Machine::Instance().GetUserPageTableArray(); // 用户页表。起始于0xC0202000 的PTE数组，2048项
31     unsigned int textBegin = this->TextAddress >> 12, textLength = this->TextSize >> 12;
32     PageTableEntry* pointer = (PageTableEntry *)pUserPageTable;
33
34     /*如果与其它进程共享正文段，共享正文段切不可清0*/
35     if(sharedText == 1)
36         i = 1; // 从数据段开始写
37     else
38     {
39         i = 0; // 从代码段开始写
40         // 修改正文段的读写标志，为内核写代码段做准备
41         for (i0 = textBegin; i0 < textBegin + textLength; i0++)
42             pointer[i0].m_ReadWriter = 1;
43
44         FlushPageDirectory();
45     }
46
47     /* 对所有页面执行清0操作，这样bss变量的初值就是0 */
48     for (; i <= this->BSS_SECTION_IDX; i++)
49     {
50         ImageSectionHeader* sectionHeader = &(this->sectionHeaders[i]);
51         int beginVM = sectionHeader->VirtualAddress + ntHeader.OptionalHeader.ImageBase;
52         int size = ((sectionHeader->Misc.VirtualSize + PageManager::PAGE_SIZE - 1)>>12)<<12;
53         int j;
54
55         if(sharedText == 0 || i != 0)
56         {
57             for (j=0; j<size; j++)
58             {
59                 unsigned char* b = (unsigned char*) (j + beginVM);
60                 *b = 0;
61             }
62         }
63     }
```

用户页表。起始于0xC0202000 的PTE数组，2048项

// 逐段清0。两个作用：

- BSS段中全局变量赋初值0
- 分配给用户的存储空间清0，可以保证系统安全



```
ProcessManager.cpp  main.cpp  MemoryDescriptor.cpp  sys.c  FileManager.cpp  Process.cpp  *PEParser.cpp  Machine.cp
64
65  /* 从文件中读入正文段 (optional), 只读数据 和 全局变量的初值 */
66  if(sharedText == 1)
67      i = 1;
68  else
69      i = 0;
70
71  for ( ; i < this->BSS_SECTION_IDX; i++ )
72  {
73      ImageSectionHeader* sectionHeader = &(this->sectionHeaders[i]);
74      srcAddress = sectionHeader->PointerToRawData;
75      desAddress =
76          this->ntHeader.OptionalHeader.ImageBase + sectionHeader->VirtualAddress;
77
78      u.u_IOParam.m_Base = (unsigned char*)desAddress;
79      u.u_IOParam.m_Offset = srcAddress;
80      u.u_IOParam.m_Count = sectionHeader->Misc.VirtualSize;
81
82      p_inode->ReadI();
83
84      cnt += sectionHeader->Misc.VirtualSize;
85  }
86
87  if(sharedText == 0)
88  { //将正文段页面改回只读
89      for (i0 = 0; i0 < textLength; i0++)
90          pointer[i0].m_ReadWriter = 0;
91
92      FlushPageDirectory();
93  }
94
95  KernelPageManager& kpm = Kernel::Instance().GetKernelPageManager();
96  kpm.FreeMemory(PageManager::PAGE_SIZE * 2, (unsigned long)this->sectionHeaders - 0xC0000000 );
97 // kpm.FreeMemory(section_size * ntHeader.FileHeader.NumberOfSections, (unsigned long)this->sectionHeaders - 0xC0000000 );
98 // delete this->sectionHeaders;
99  return cnt;
100 }
```

// 进程会在这里睡很多次

// 释放用来临时保存 section header 的物理页框

接 PPT 14

```

660
661 /* 将fakeStack中备份的用户栈参数复制到新进程图像的用户栈中 */
662 //Utility::MemCopy(fakeStack | 0xC0000000, MemoryDescriptor::USER_SPACE_SIZE - parser.StackSize, parser.StackSize);
663 Utility::MemCopy(fakeStack + allocLength - parser.StackSize | 0xC0000000, MemoryDescriptor::USER_SPACE_SIZE - parser.StackSize, parser.StackSize);
664 /* 释放用于读入exe文件和备份用户栈参数的内存: MapAddress和fakeStack */
665 kernelPgMgr.FreeMemory(allocLength, fakeStack);
666
667 /*
668  * 将runtime()、SignalHandler()函数拷贝到进程用户态地址空间0x00000000线性地址处, runtime()
669  * 用于ring0退出到ring3特权级之后执行的代码, SignalHandler()为进程的信号处理函数入口, 负责
670  * 调用具体信号的Handler。每一个进程0x00000000线性地址处都应该有一份独立的runtime()及SignalHandler()
671  * 函数副本!
672  */
673 // unsigned char* runtimeSrc = (unsigned char*)runtime;
674 // unsigned char* runtimeDst = 0x00000000;
675 // for (unsigned int i = 0; i < (unsigned long)ExecShell - (unsigned long)runtime; i++)
676 // {
677 //     *runtimeDst++ = *runtimeSrc++;
678 // }
679
680 /* 释放Inode, 减少ExeCnt计数值 */
681 fileMgr.m_InodeTable->IPut(pInode);
682 if ( this->ExeCnt >= NEXEC )
683 {
684     WakeUpAll((unsigned long)&ExeCnt); // 唤醒等待执行exec系统调用的其它进程
685 }
686 this->ExeCnt--;
687

```

// 解锁 可执行程序 的文件控制块

// 唤醒等待执行exec系统调用的其它进程

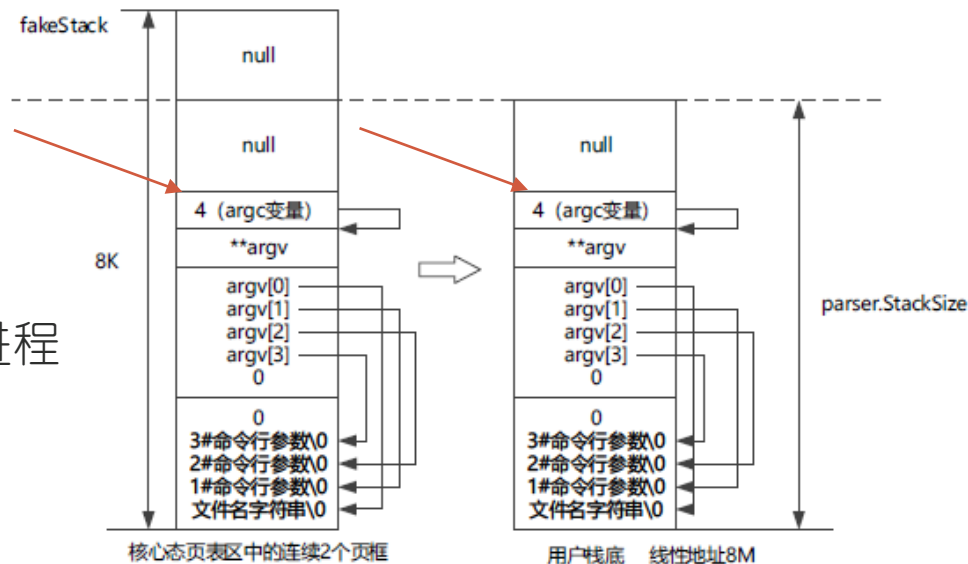


图 8.3: fakeStack 和转换图像后的用户栈

```
687
688 /* 用默认的方式处理信号 */
689 for (int i = 0; i < u.NSIG ; i++)
690 {
691     u.u_signal[i] = 0;
692 }
693
694 /* 清0所有通用寄存器 */
695 for (int i = User::EAX - 4; i < User::EAX - 4*7 ; i = i - 4)
696 {
697     u.u_ar0[i] = 0;      /* 下标写成 User::EAX + i 可读性要强一些，但是运算速度慢了。就小抠，追求速度吧 */
698 }
699
700 /* 将exe程序的入口地址放入核心栈现场保护区中的EAX作为系统调用返回值，这个是runtime要用 */
701 u.u_ar0[User::EAX] = parser.EntryPointAddress;
702
703 /* 构造出Exec()系统调用的退出环境，使之退出到ring3时，开始执行user code */
704 struct pt_context* pContext = (struct pt_context *)u.u_arg[4];
705 pContext->eip = 0x00000000; /* 退出到ring3特权级下从线性地址0x00000000处runtime()开始执行 */
706 //pContext->eip = parser.EntryPointAddress;
707 pContext->xcs = Machine::USER_CODE_SEGMENT_SELECTOR;
708 pContext->eflags = 0x200; /* 此项是否篡改无关 */
709 pContext->esp = esp;
710 pContext->xss = Machine::USER_DATA_SEGMENT_SELECTOR;
711 }
```

只有 IF 标识是1，其余全是0。

exec系统调用成功，返回用户态

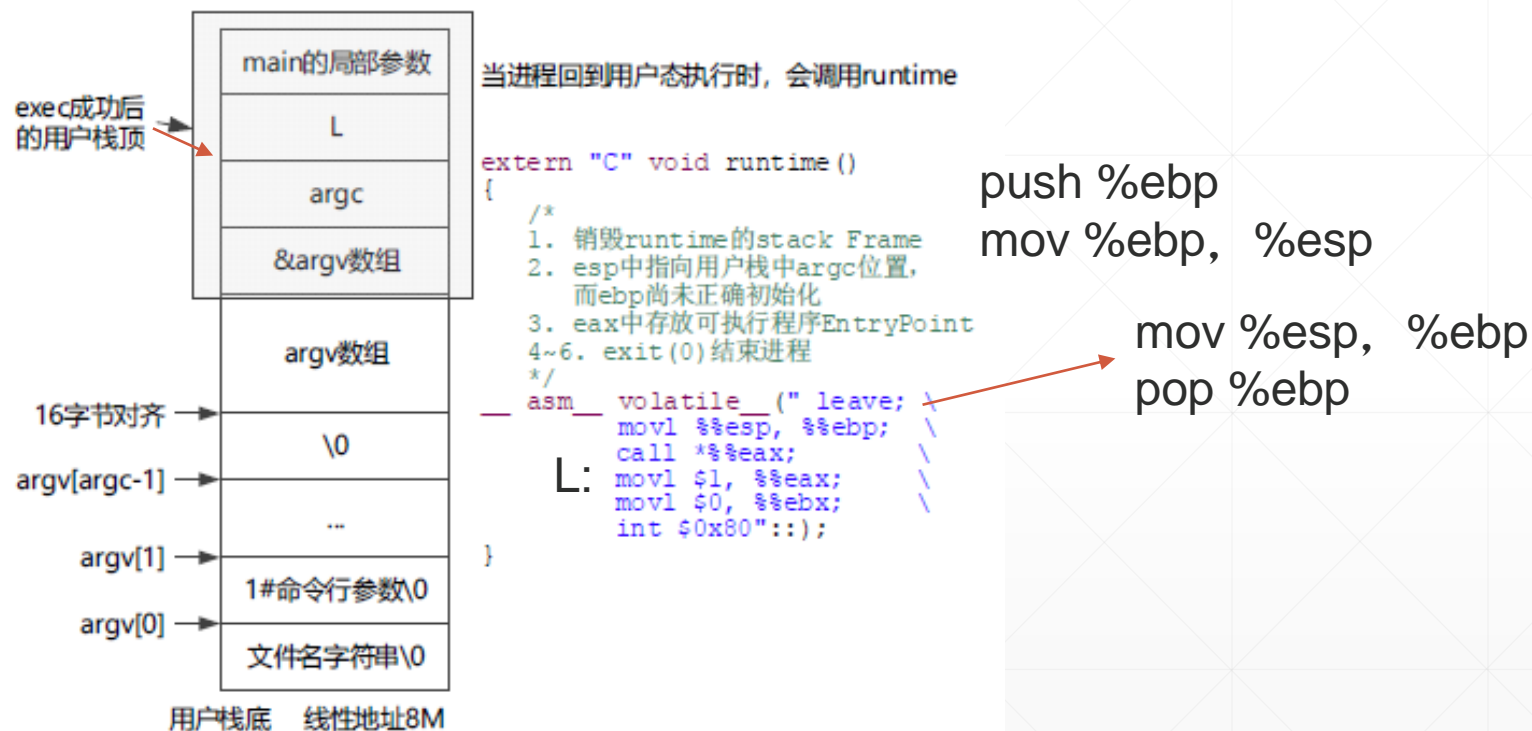


图 8.4：进程的堆栈段和 runtime()

Part 2、进程终止 exit

- 进程终止后，除PCB外的所有的图像均释放。
 - PCB中有系统需要采集的信息：PID、终止码和运行时间
- 子进程的PCB由父进程回收。读出上述3条信息后，释放子进程PCB。
- 子进程终止后，
 - 不能立即释放PCB
 - 应尽快回收PCB。保证系统有足够的PCB可供创建新进程。
 - 这个叫做**子进程终止的 dilemma**。

终止码

应用程序执行完毕，正常终止（Normal termination）

- 执行**exit (n)** 系统调用
- 从**main**返回 **return n**，相当于**exit (n)** **n 是exit status**

进程收到了无法**catch**的信号，异常终止（Abnormal termination）

- 执行**abort**系统调用，进程向自己发信号，终止自己
- 用户可以用**ctrl+c**，向前台作业发**SIGINT**信号，终止负责执行该作业的所有进程。
- 应用程序访问了非法内存单元，就会收到段错误信号**SIGSEGV**
- 应用程序执行除数为**0**的除法，就会收到信号**SIGFPE**（浮点运算异常）

进程收到的信号是**termination status**

无论进程因何原因而终止，进程总是执行 **exit (termination status)** 结束自己

若正常终止， **n < 8** → **termination status** → **exit**内核函数

wait系统调用和exit系统调用

父进程执行wait系统调用等待子进程终止 (exit) 。

子进程终止后，唤醒父进程，父进程 (wait系统调用)

- 1、读子进程PCB中的终止码 → j
- 2、释放子进程的PCB。wait系统调用返回，返回值是子进程的PID号 → i

```
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);
    }
}
```




shell进程执行wait等待子进程终止

```
main( )
```

```
{
```

```
.....
```

```
while( )
```

```
{
```

输出 \$ (“#”)，睡眠等待用户输入命令行：command arg1 arg2

如果输入的是“cd”，shell进程就改变u_cdir 和 u_curdir字符串

其余内部命令.....

如果输入的是“logout”，shell进程就exit

```
while((i=fork( ))== -1);
```

```
if( ! i)
```

```
    exec(“command”, arg1, arg2, .....); // ls,echo,date,gcc .....所有
```

```
else {
```

```
    child = wait(&terminationStatus);
```

```
    if (terminationStatus & 0xFF != 0) //是进程收到的信号的值
```

按需，根据terminationStatus & 0xFF 的值

printf出来，段错误之类的信息

```
}
```

```
}
```

```
}
```

User.h

```
59
60 static const int NSIG = 32; /* 信号个数 */
61
62 /* p_sig中接受到的信号定义 */
63 static const int SIGNUL = 0; /* No Signal Received */
64 static const int SIGHUP = 1; /* Hangup (kill controlling
65 static const int SIGINT = 2; /* Interrupt from keyboard
66 static const int SIGQUIT = 3; /* Quit from keyboard */
67 static const int SIGILL = 4; /* Illegal instruction */
68 static const int SIGTRAP = 5; /* Trace trap */
69 static const int SIGABRT = 6; /* use abort() API */
70 static const int SIGBUS = 7; /* Bus error */
71 static const int SIGFPE = 8; /* Floating point exception
72 static const int SIGKILL = 9; /* Kill (can't be caught or
73 static const int SIGUSR1 = 10; /* User defined signal 1 */
74 static const int SIGSEGV = 11; /* Invalid memory segment
75 static const int SIGUSR2 = 12; /* User defined signal 2 */
76 static const int SIGPIPE = 13; /* Write on a pipe with no
77 static const int SIGALRM = 14; /* Alarm clock */
78 static const int SIGTERM = 15; /* Termination */
79 static const int SIGSTKFLT = 16; /* Stack fault */
80 static const int SIGCHLD = 17; /* Child process has stopped
81 static const int SIGCONT = 18; /* Continue executing, if st
82 static const int SIGSTOP = 19; /* Stop executing */
83 static const int SIGTSTP = 20; /* Terminal stop signal */
84 static const int SIGTTIN = 21; /* Background process trying
85 static const int SIGTTOU = 22; /* Background process trying
86 static const int SIGURG = 23; /* Urgent condition on socke
87 static const int SIGXCPU = 24; /* CPU limit exceeded */
88 static const int SIGXFSZ = 25; /* File size limit exceeded
89 static const int SIGVTALRM = 26; /* Virtual alarm clock */
90 static const int SIGPROF = 27; /* Profiling alarm clock */
91 static const int SIGWINCH = 28; /* Window size change */
92 static const int SIGIO = 29; /* I/O now possible */
93 static const int SIGPWR = 30; /* Power failure restart */
94 static const int SIGSYS = 31; /* invalid sys call */
95
```


负责执行exit系统调用的库函数

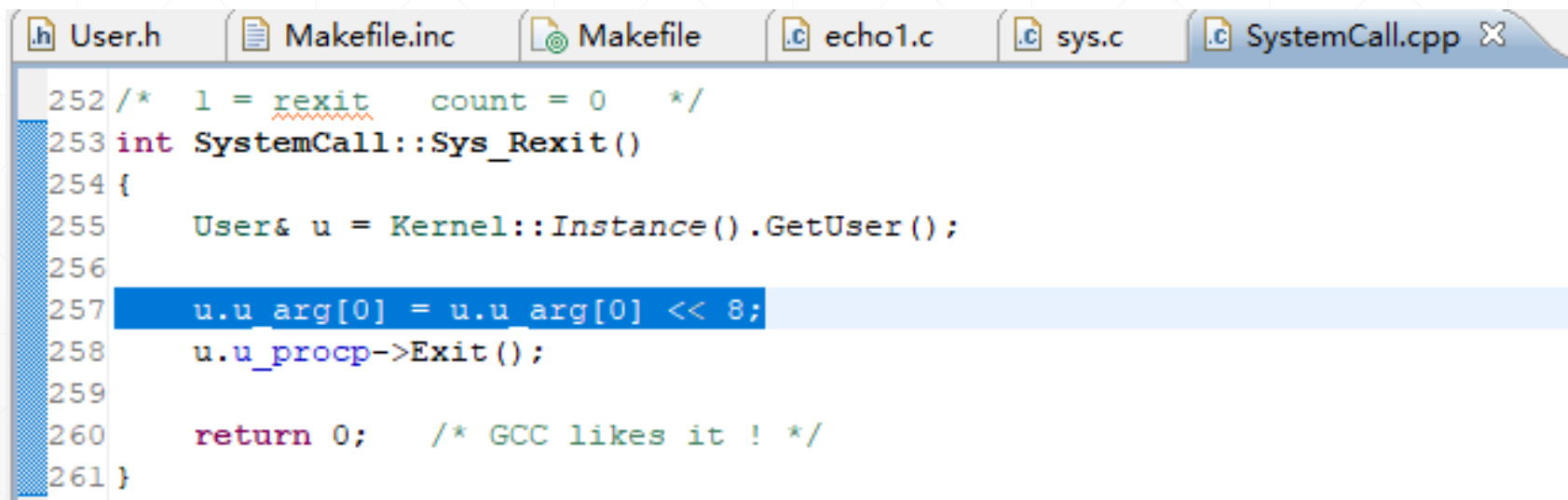
exit(n)

```
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;
}
```

u.u_arg[0]

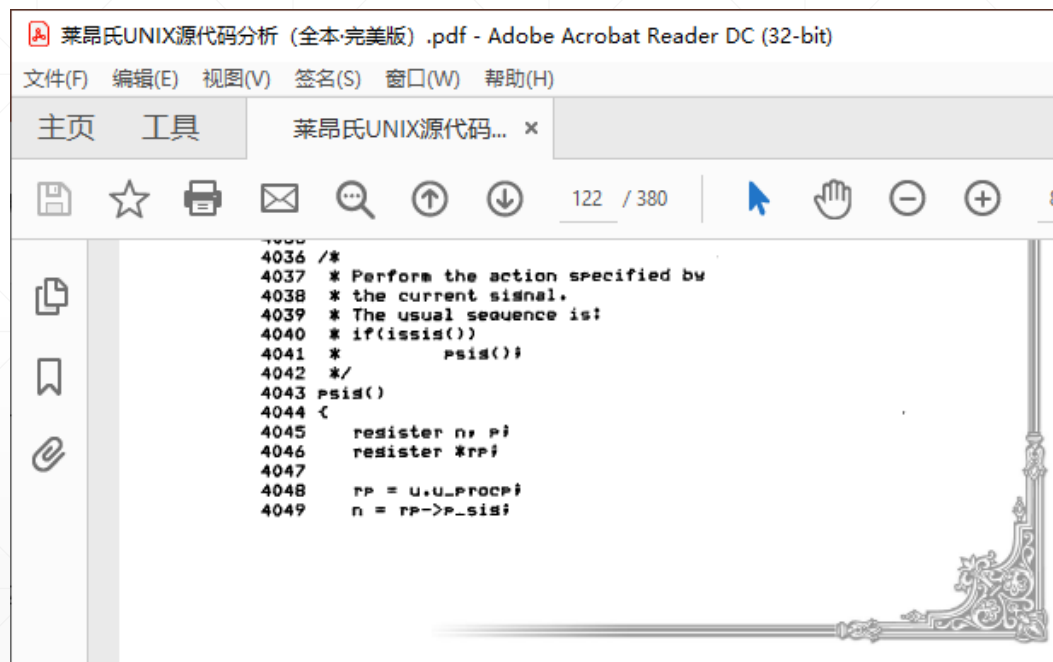
进程正常运行结束：执行exit系统调用

exit () 系统调用的入口函数Rexit ()

A screenshot of a code editor showing the implementation of the Sys_Rexit system call. The editor has tabs for User.h, Makefile.inc, Makefile, echo1.c, sys.c, and SystemCall.cpp. The code is in C++ and shows the function Sys_Rexit() which calls the kernel's Exit() function.

```
252 /* 1 = rexit count = 0 */
253 int SystemCall::Sys_Rexit()
254 {
255     User& u = Kernel::Instance().GetUser();
256
257     u.u_arg[0] = u.u_arg[0] << 8;
258     u.u_procp->Exit();
259
260     return 0; /* GCC likes it ! */
261 }
```

进程被信号杀死，直接调用 Exit() 函数



```

4050     rpi->p_sig = 0;
4051     if((p=u.u_signal[n]) != 0) {
4052         u.u_error = 0;
4053         if(n != SIGINS && n != SIGTRC)
4054             u.u_signal[n] = 0;
4055         n = u.u_ar0[R6] - 4;
4056         grow(n);
4057         suword(n+2, u.u_ar0[RPS]);
4058         suword(n, u.u_ar0[R7]);
4059         u.u_ar0[R6] = n;
4060         u.u_ar0[RPS] = & ~TBIT;
4061         u.u_ar0[R7] = p;
4062         return;
4063     }
4064     switch(n) {
4065     case SIGQUIT:
4066     case SIGINS:
4067     case SIGTRC:
4068     case SIGIOT:
4069     case SIGEMT:
4070     case SIGFPT:
4071     case SIGBUS:
4072     case SIGSEGV:
4073     case SIGSYS:
4074         u.u_ar0[R0] = n;
4075         if(core())
4076             n += 0200;
4077     }
4078     u.u_ar0[R0] = (u.u_ar0[R0] << 8) | n;
4079     exit();
4080 }
4081 /* ----- */
4082 */

```

Process::Exit ()

清除进程的信号处理函数

关闭所有进程打开的文件

递减当前工作目录的引用
计数

释放该进程对代码段的共
享

在盘交换区上申请一个空
闲扇区，将User结构复制
其中，p_addr中记录User
结构在盘交换区中的位置

释放该进程的在内核页表区
中的两张虚实地址映射表

修改进程调度状态为
SZOMB状态

process[i]

p_addr

user: u_arg[0]
u_utime
u_stime

子进程的终止码

process[i]

p_addr

user: u_arg[0]
u_utime
u_stime

```

241 /* 唤醒父进程进行善后处理 */
242 for ( i = 0; i < ProcessManager::NPROC; i++ )
243 {
244     if ( procMgr.process[i].p_pid == current->p_ppid )
245     {
246         procMgr.WakeUpAll( (unsigned long) &procMgr.process[i]);
247         break;
248     }
249 }
250 /* 没找到父进程 */
251 if ( ProcessManager::NPROC == i )
252 {
253     current->p_ppid = 1;
254     procMgr.WakeUpAll( (unsigned long) &procMgr.process[1]);
255 }
256 /* 将自己的子进程传给自己的父进程 */
257 for ( i = 0; i < ProcessManager::NPROC; i++ )
258 {
259     if ( current->p_pid == procMgr.process[i].p_ppid )
260     {
261         Diagnose::Write("My:%d 's child %d passed to l#process",current->p_pid,procMgr.process[i].p_pid);
262         procMgr.process[i].p_ppid = 1;
263         if ( procMgr.process[i].p_stat == Process::SSTOP )
264         {
265             procMgr.process[i].SetRun();
266         }
267     }
268 }
269 procMgr.Swtch();
270 }
271 }
272 }
273 }

```

```

810 void ProcessManager::WakeUpAll(unsigned long chan)
811 {
812     /* 唤醒系统中所有因chan而进入睡眠的进程 */
813     for(int i = 0; i < ProcessManager::NPROC; i++)
814     {
815         if( this->process[i].IsSleepOn(chan) )
816         {
817             this->process[i].SetRun();
818         }
819     }
820 }

```

判断:
p_wchan == chan?

父进程 执行wait 系统调用等待子进程终止

```
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);
    }
}
```

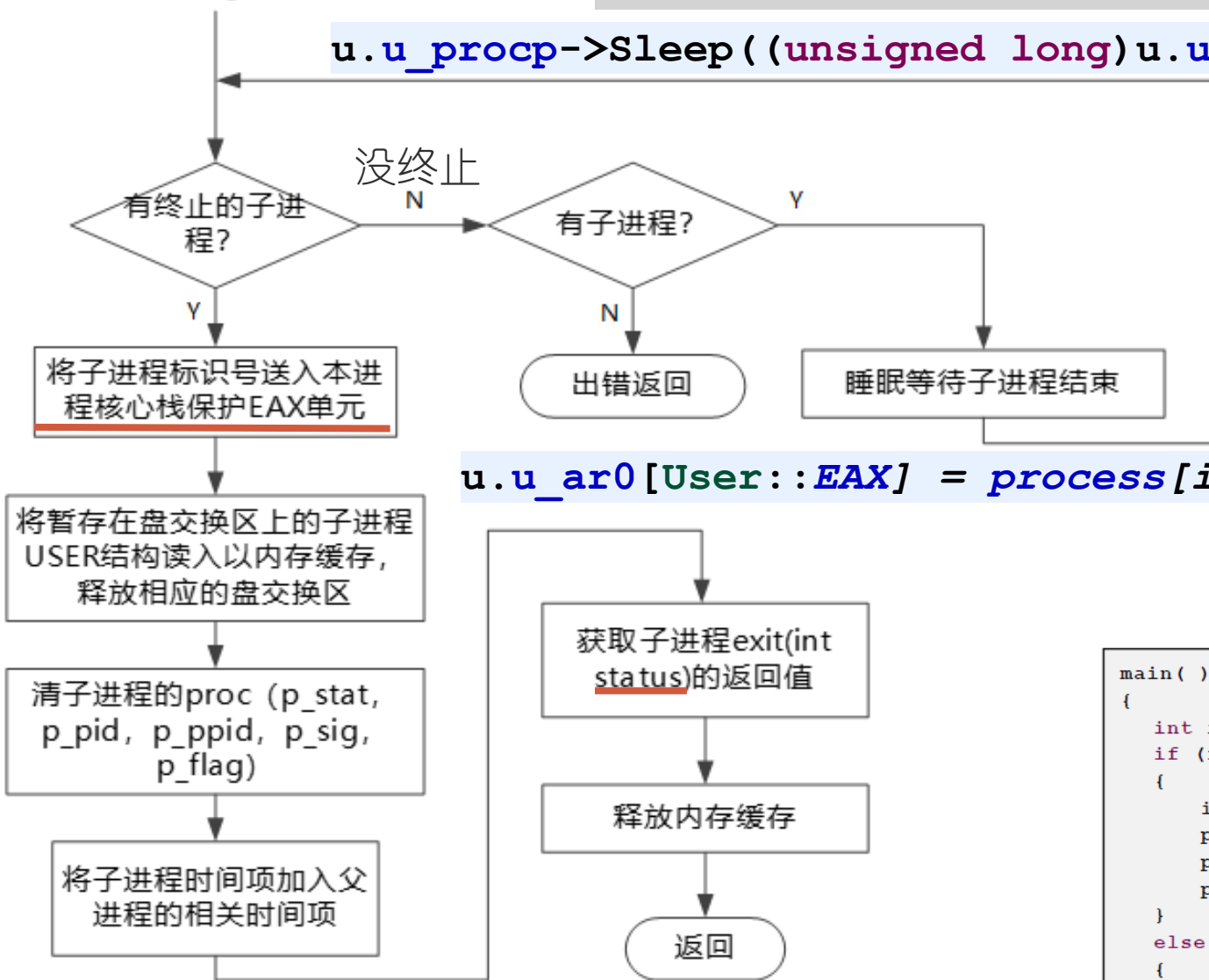
```
int wait(int* status)    /* status == &j */
{
    int res;
    __asm__ __volatile__ ( "int $0x80":"=a"(res):"a"(7),"b"(status));
    if ( res >= 0 )
        return res;
    return -1;
}
```

父进程的 u_arg[0]

ProcessManager::Wait ()

p_wchan = 父进程process对象的起始地址

`u.u_procp->Sleep((unsigned long)u.u_procp, ProcessManager::PWAIT);`

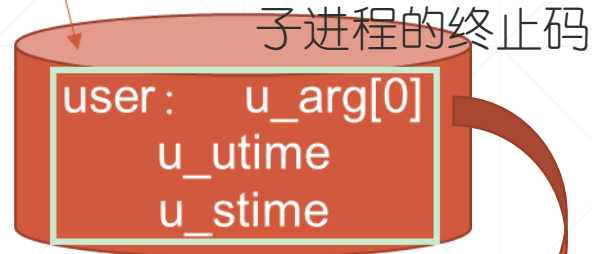


`u.u_ar0[User::EAX] = process[i].p_pid; //子进程`

process[i]



p_addr



父进程u_arg[0]指向的用户区单元 (j变量)

```

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);
    }
}
  
```



何时回收子进程的 PCB ?

```
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);
    }
}
```




如果父进程不执行wait系统调用

1、父进程终止时，1# 进程回收子进程的PCB。

```
main( )
{
    int i,j;
    if (fork())
    {
        sleep (100)
        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);
    }
}
```

如果父进程不执行wait系统调用

2: 父进程早于子进程终止

这种子进程是孤儿进程，终止时由1#进程回收其PCB。

```
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");    sleep (100) ;
        exit(0);
    }
}
```

何时回收子进程的 PCB ?

```
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);
    }
}
```

```
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
    {
        sleep (100) ;
        i = wait(&j);
        printf("It is child process. \n");
        exit(0);
    }
}
```

shell进程执行wait等待子进程终止

```
main( )  
{  
    .....  
    while( )  
    {  
  
    }  
}
```

输出 \$, 睡眠等待用户输入命令行: **command arg1 arg2**

如果输入的是 “logout”, shell进程就**exit**

while((i=fork())== -1);

if(! i)

exec(“command”, arg1, arg2,);

else if(命令行中没有后台命令符号&) {

child = wait(&terminationStatus);

if (terminationStatus & 0xFF != 0)

按需, 根据**terminationStatus & 0xFF** 的值

printf出来, 段错误核心转储之类的信息

} //shell进程不会睡眠等待负责执行后台作业的进程终止

shell进程终止后, 后台进程会继续运行。它们的父进程是1# 进程。后台进程终止后, 1# 进程回收其PCB