Part 2、进程终止 与 exit、wait 父子进程同步

- 进程的正常终止 和 异常终止
- exit系统调用（1#系统调用）
- wait系统调用（7#系统调用）

一、进程的正常终止 和 异常终止

- 应用程序执行 exit(n) 系统调用，进程正常终止。n是进程的退出码。
  - main( )返回，正常终止，Unix V6++退出码是 0。

- 应用程序无法继续执行或没必要继续执行时，系统会给进程发信号，促使其异常终止。
  - 执行非法指令
  - 非法内存访问
  - 运算错误，比如浮点运算结果溢出
  - ctrl+c，用户终止正在执行的应用程序，进程异常终止
  - kill -9 pid，用户杀死目标进程，进程异常终止

```
extern "C" void runtime()
{
    __asm__ __volatile__("      leave;                  \
                                movl %%esp, %%ebp;      \
                                call *%%eax;            \
                          L:    movl $1, %%eax;         \
                                movl $0, %%ebx;         \   exit(0)
                                int $0x80"::);
}
```

- 无论正常终止，还是异常终止，进程执行内核函数Exit( )终止自己。
- 每个终止的进程，有一个终止码供父进程 或 系统采集
  - 正常终止的进程，终止码是 n<<8，n是exit系统调用的参数
  - 异常终止的进程，终止码是收到的信号

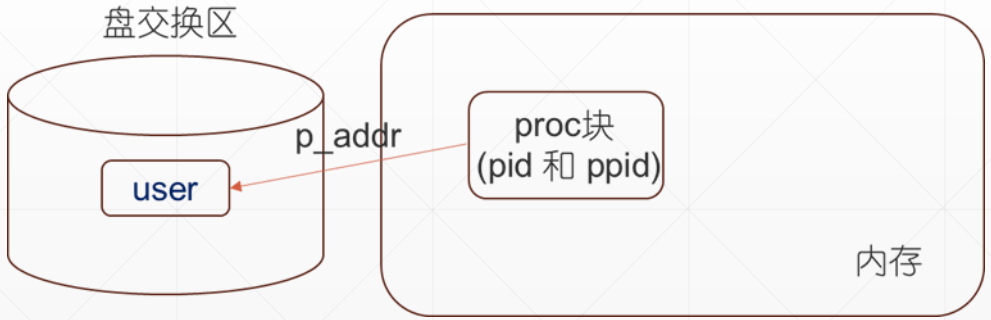
二、Unix V6++ 的 exit 系统调用

```
int exit(int status)    // 用户空间的钩子函数，  status是 退出码 exit status
{
        int res;
        __asm__ __volatile__ ( "int $0x80":"=a"(res):"a"(1),"b"(status));
        if ( res >= 0 )
                return res;
        return -1;
}

int SystemCall::Sys_Rexit()    // exit系统调用的入口
{
        User& u = Kernel::Instance().GetUser();

        // u.u_arg[0] =  u.u_arg[0] << 8;
        u.u_procp->Exit();

        return 0; /* GCC likes it ! */
}
```

u_arg[0]是进程的终止码

Exit( )函数 ★

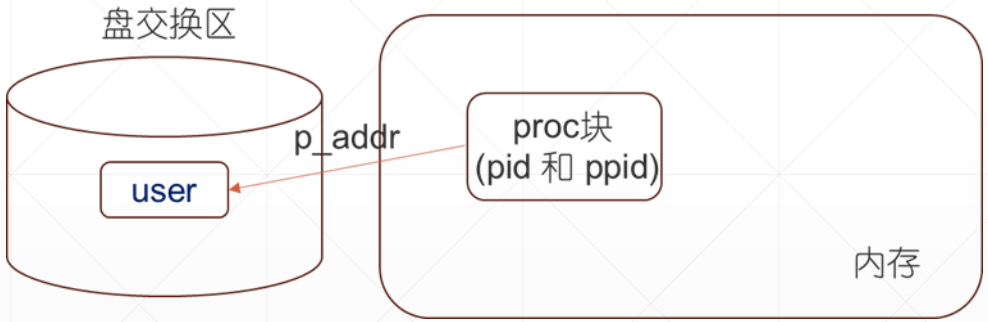
- 进程执行内核函数 Exit( )终止，完成 运行→终止 的状态变迁。需要执行的主要操作：
  - 释放进程正在使用的所有资源
    - 关闭所有打开文件
    - 当前工作目录，引用计数--
    - 释放相对虚实地址映射表
    - 释放可交换部分
    - 释放代码段（代码段引用计数减 1，减至0，释放其占用的内存和盘交换区空间）
  - 在盘交换区上申请一个扇区（512字节），启动 IO，将 user 结构复制到该扇区。



Exit( )函数 ★

父进程执行 wait 系统调用回收子进程PCB。
所以，进程终止，放弃CPU之前，需要：

- 唤醒父进程，回收自己的PCB
- 将子进程的ppid改为1#进程
- 唤醒 1#进程，回收已经终止的子进程的PCB
- p_stat = SZOMB，Swtch( )放弃CPU。



三、 Unix V6++ 的 wait 系统调用

- 父进程执行 wait 系统调用回收子进程PCB。
- wait 系统调用执行时，如果子进程已终止，立即回收其PCB。否则，父进程入睡等待，直至子进程终止。
- wait系统调用的返回值是子进程的pid号。

wait系统调用的钩子函数

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

系统调用的第一个参数 **status** 是父进程用户空间的一根指针，这个单元是一个整数，用来存放子进程的终止码。

四、wait、exit系统调用的使用方法，例1

```
#include <stdio.h>
#include <sys.h>
int main1(int argc, char* argv[])
{
  int i, j;
  if(fork())
  {
    i = wait(&j);
    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( 2 );
  }
}
```

已知， Unix V6++系统，只有这一个程序在执行。父进程pid==2 ， 子进程pid==3。

问，（1）父进程会睡吗？
（2）这个程序的输出是什么？
（3）删除sleep(2)，父进程有可能不睡吗？
（4）无论有没有sleep(2)，这个程序的输出是确定的，断言正确否？为什么？

四、wait、exit系统调用的使用方法

```
#include <stdio.h>
#include <sys.h>
int main1(int argc, char* argv[])
{
  int i, j;
  if(fork())
  {
    sleep( 2 );
    i = wait(&j);
    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");
  }
}
```

已知， Unix V6++系统，只有这一个程序在执行。父进程pid==2 ， 子进程pid==3。

问， （1）父进程会睡吗？
（2）这个程序的输出是什么？
（3）系统里有几个进程？

可以用wait、exit系统调用确保任务之间的前驱后继关系

```
int main1(int argc, char* argv[])
{
   int i, j;
   if(fork())
   {
     i = wait(&j);
     后继任务。。。
   }
   else
   {
     前驱任务。。。
   }
}
```

五、源代码 Exit( )

```
void Process::Exit()
{ ……
    /* 1、信号处理方式设置为1，不再响应任何信号 */
    for ( i = 0; i < User::NSIG; i++ )
    {
        u.u_signal[i] = 1;
    }

    /* 2、关闭所有的打开文件 */
    for ( i = 0; i < OpenFiles::NOFILES; i++ )
    {
        File* pFile = NULL;
        if ( (pFile = u.u_ofiles.GetF(i)) != NULL )
        {
            fileTable.CloseF(pFile);    //释放File结构（引用计数--）
            u.u_ofiles.SetF(i, NULL);   //打开文件表相关元素（fd）置null
        }
    }
```

```
/*  访问不存在的fd会产生error code，清除u.u_error避免影响后续程序执行流程 */
u.u_error = User::NOERROR;

/* 3、当前工作目录的引用计数-- */
inodeTable.IPut(u.u_cdir);

/* 4、释放代码段（引用计数减 1，减至 0 释放占用的内存空间和盘交换区空间） */
if ( u.u_procp->p_textp != NULL )
{
    u.u_procp->p_textp->XFree();
    u.u_procp->p_textp = NULL;
}
/* 5、user结构写入盘交换区 */
/* 1、盘交换区申请一个扇区（512字节），扇区号是blkno*/
int blkno = swapperMgr.AllocSwap(BufferManager::BUFFER_SIZE);
    ……
/* 2.1 内存申请一个缓存块（512字节），用来同步磁盘扇区blkno。磁盘IO要用到。起始地址pBuf->b_addr */
Buf* pBuf = bufMgr.GetBlk(DeviceManager::ROOTDEV, blkno);
/* 2.2  把user结构写进这个缓存块    */
Utility::DWordCopy((int *)&u, (int *)pBuf->b_addr, BufferManager::BUFFER_SIZE / sizeof(int));
/* 3 把这个缓存块同步写入磁盘。进程睡眠等待IO完成
bufMgr.Bwrite(pBuf);
```

```
/* 6、释放相对虚实地址映射表 */
u.u_MemoryDescriptor.Release();

/* 7、释放可交换部分 */
Process* current = u.u_procp;  // 现运行进程的Process结构
UserPageManager& userPageMgr = Kernel::Instance().GetUserPageManager();
userPageMgr.FreeMemory(current->p_size, current->p_addr);

/* 8、p_addr指向磁盘上的user结构。置终止状态 */
current->p_addr = blkno;
current->p_stat = Process::SZOMB;
```

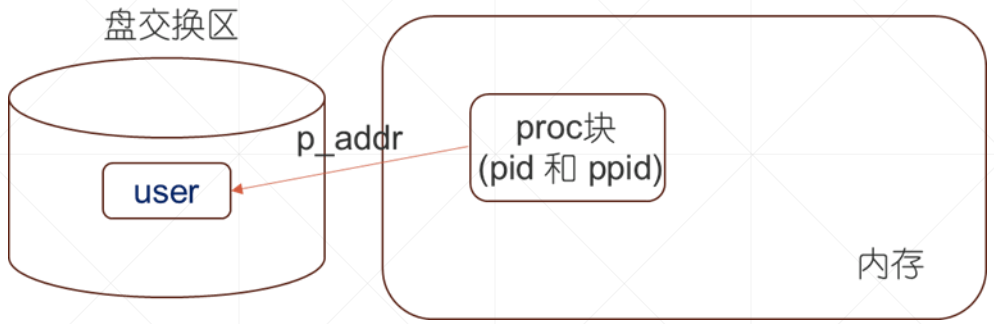
```
/* 9、唤醒父进程 */
for ( i = 0; i < ProcessManager::NPROC; i++ )
{
        if ( procMgr.process[i].p_pid == current->p_ppid )  // 父进程PID
        {
                procMgr.WakeUpAll((unsigned long)&procMgr.process[i]);
                break;
        }
}

/* 10、没找到父进程（父进程已经终止） */
if ( ProcessManager::NPROC == i )
        current->p_ppid = 1;

/* 11、无论是否找到父进程，唤醒1#进程 */
procMgr.WakeUpAll((unsigned long)&procMgr.process[1]);
```

```
    /* 12、将子进程传给1#进程 */
    for ( i = 0; i < ProcessManager::NPROC; i++ )
    {
            if ( current->p_pid == procMgr.process[i].p_ppid )     // 找到子进程
            {
                    Diagnose::Write("……",……);   // 内核使用的格式化输出函数，相当于应用程序用的printf
                    procMgr.process[i].p_ppid = 1;     //  把它们的父进程改成1#进程
                    if ( procMgr.process[i].p_stat == Process::SSTOP )
                    {
                            procMgr.process[i].SetRun();
                    }
            }
    }

    procMgr.Swtch( );     // 13、放弃CPU
}
```

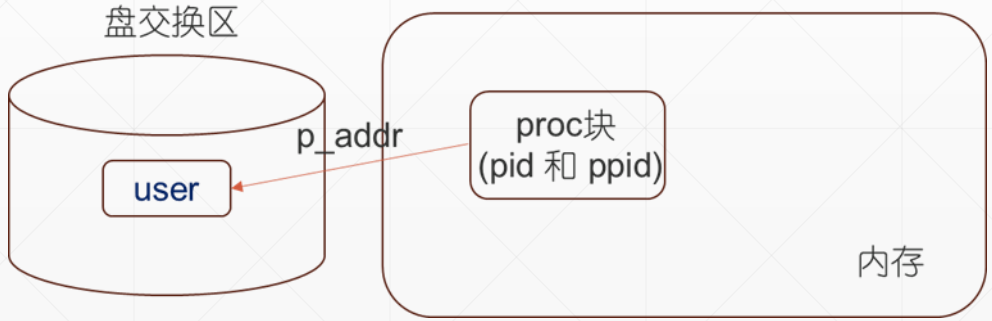
六、源代码　Wait( )

```
void ProcessManager::Wait()
{
    while(true)
    {
        for ( i = 0; i < NPROC; i++ )
        {
            if ( u.u_procp->p_pid == process[i].p_ppid )   /* 找所有的子进程 */
            {
                hasChild = true;
                if( Process::SZOMB == process[i].p_stat )     /* 回收一个终止子进程的PCB */
                {     处理终止的子进程;        return;      }
            }
        }
        if (true == hasChild)   // 存在尚未终止的子进程
        {
            Diagnose::Write("wait until child process Exit! ");
            u.u_procp->Sleep((unsigned long)u.u_procp, ProcessManager::PWAIT);    // 入睡，等待子进程终止
            continue;     /* 唤醒后，再进for循环，找终止子进程 */
        }
        else
        {
            u.u_error = User::ECHILD;
            break;      /* Get out of while loop */
        }
    }
}
```

```
if( Process::SZOMB == process[i].p_stat )      // 处理终止的子进程
{
    /* 准备wait()系统调用的返回值：子进程的pid */
    u.u_ar0[User::EAX] = process[i].p_pid;
    /* 释放 Process结构 */
    process[i].p_stat = Process::SNULL;
    process[i].p_pid = 0;
    process[i].p_ppid = -1;
    process[i].p_sig = 0;
    process[i].p_flag = 0;

    /* 启动IO，读入swap分区上的user结构，放在pBuf 管理的内存块 */
    Buf* pBuf = bufMgr.Bread(DeviceManager::ROOTDEV, process[i].p_addr);
    /* 释放盘交换区上user结构占用的空间 */
    swapperMgr.FreeSwap(BufferManager::BUFFER_SIZE, process[i].p_addr);

    User* pUser = (User *)pBuf->b_addr;    // 内存块的首地址，就是读入的子进程user结构的首地址 pUser
```



```
                /* 终止子进程的时间累加到父进程上，在字段u_c*time里 */
                u.u_cstime += pUser->u_cstime + pUser->u_stime;
                u.u_cutime += pUser->u_cutime + pUser->u_utime;

               /* pInt指向父进程wait系统调用传入的用户指针，所指变量 j，用来存放子进程的终止码 status */
                int* pInt = (int *)u.u_arg[0];     /* pInt是父进程 wait系统调用的参数，指向其用户空间单元 j，这里
                                                      放子进程 终止码 */
i=wait(&j);
                *pInt = pUser->u_arg[0];     /* 子进程的终止码 直接写 父进程的 j 变量*/

                /* 释放pBuf 管理的内存块 */
                bufMgr.Brelse(pBuf);
                return;
        }
```



wait、exit系统调用的使用方法，例2

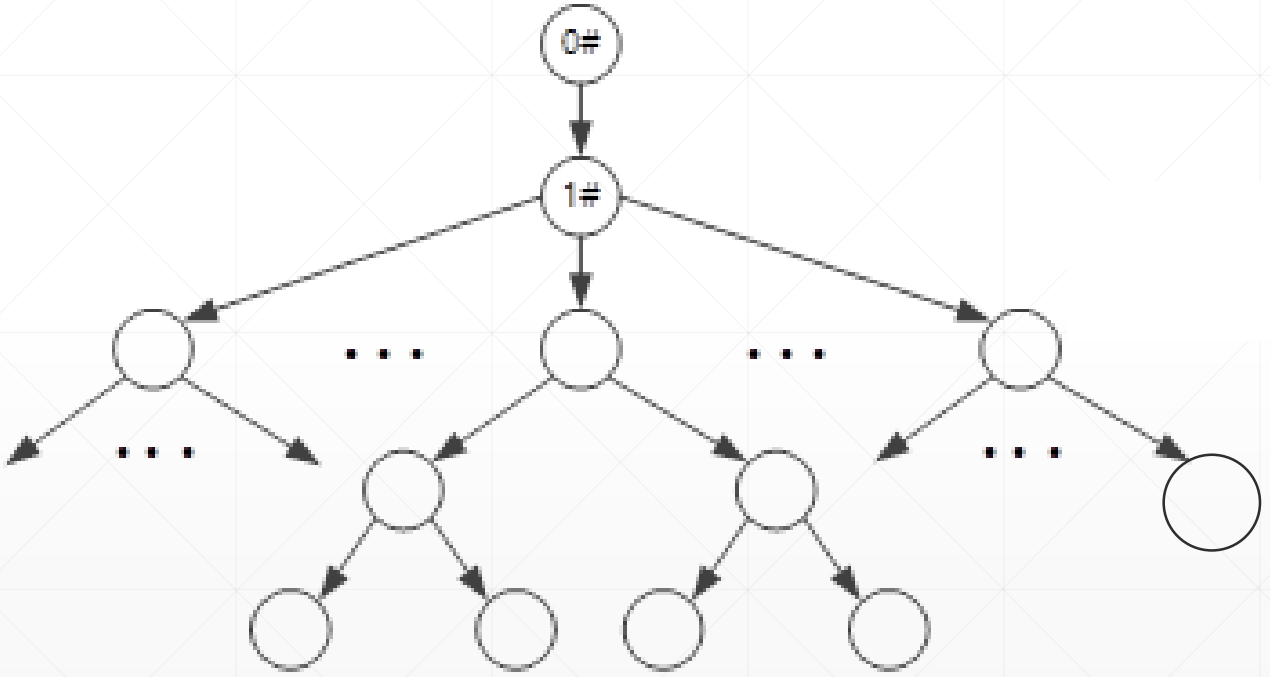
```
#include <stdio.h>
#include <sys.h>
int main1(int argc, char* argv[])
{
   int i, j;
   if(fork())
   {
       printf("father. \n");
       if(fork())
           i = wait(&j);
       else
       {  printf("second child. \n");   exit(3);   }
   }
   else
   {
       sleep( 2 );
       printf("first child. \n");
   }
}
```

已知， Unix V6++系统，只有这一个程序在执行。父进程pid==2，子进程1 pid==3，子进程2 pid==4。

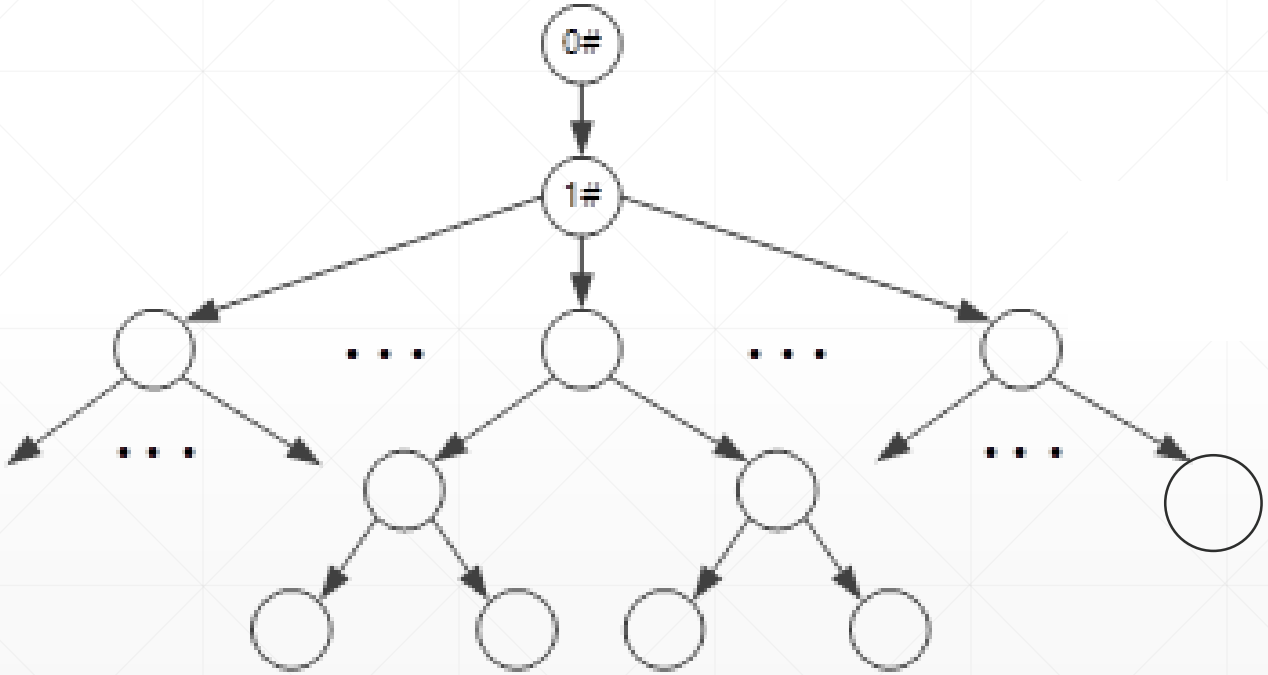
问，（1）这个程序的输出是什么？
（2）子进程1 和 子进程2 的PCB分别是哪个进程回收的？
（3）父进程终止后，PCB是哪个进程回收的？

Part 3　Unix 进程树 和 shell

- 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。



简化的 shell进程 代码框架

```
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 和 它的子进程的 PCB。