

实验四：UNIX 中新进程创建与父子进程同步

实验目的

- 结合课程所学知识，通过在 UNIX V6++实验环境中编写与父进程创建子进程的系统调用 `fork` 有关的应用程序，并观察其调试运行，熟悉 UNIX V6++中关于进程创建的过程及多进程编程技巧。
- 结合课程所学知识，通过在 UNIX V6++实验环境中编写与进程终止及父子进程同步的系统调用 `exit` 和 `wait` 有关的应用程序，并观察其调试运行，熟悉 UNIX V6++中关于子进程终止的详细过程及父子进程之间的数据传递。

实验要求

(1) 按上述过程，分别编辑、编译并运行 `forktest` 和 `exitwaittest` 程序，截图展示程序的输出结果。

(2) 回答下列思考题：

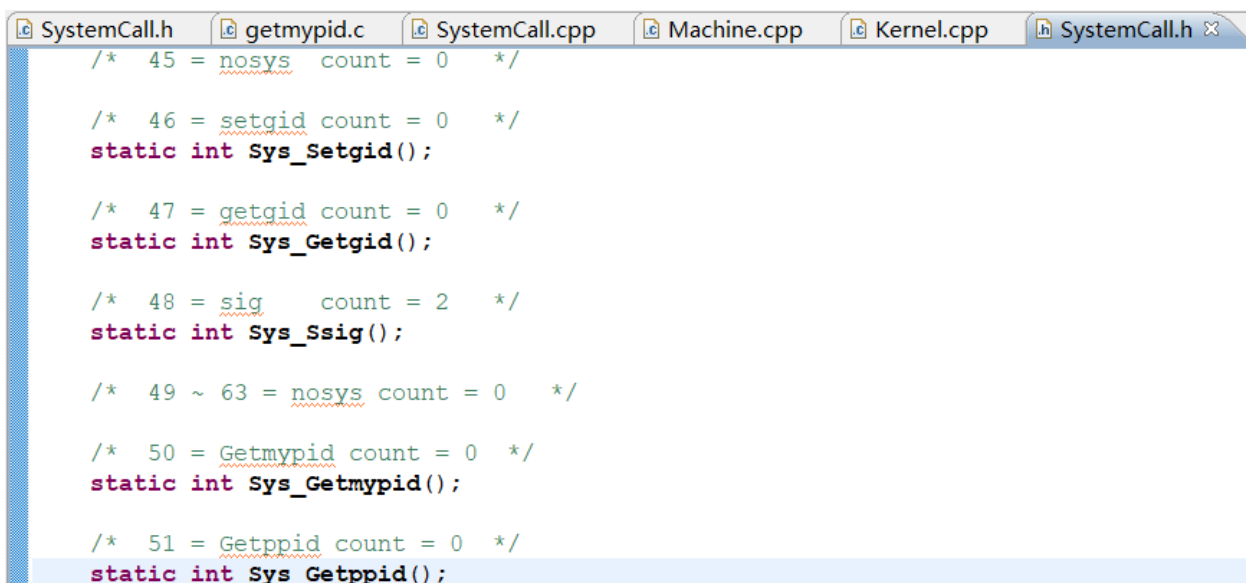
- 思考题 1 (2 分)：分析图 1 中的输出结果，指出 `i` 分别为 0, 1, 2 时，创建的是哪个进程？画出各个进程之间的父子关系。
- 思考题 2 (1 分)：分析图 3 中的输出结果，解释父进程是如何接收到子进程的终止码的（可结合调试过程说明）？
- 思考题 3*：分析图 2 中的输出结果和图 1 输出不同的原因是什么？

实验准备

添加获得某一进程的父进程 ID 号的系统调用及库函数

在 `SystemCall` 类中添加系统调用处理子程序的定义

首先在 `SystemCall.h` 文件中添加一个新的系统调用处理子程序的声明



```
SystemCall.h | getmypid.c | SystemCall.cpp | Machine.cpp | Kernel.cpp | SystemCall.h x
/* 45 = nosys count = 0 */
/* 46 = setgid count = 0 */
static int Sys_Setgid();
/* 47 = getgid count = 0 */
static int Sys_Getgid();
/* 48 = sig count = 2 */
static int Sys_Ssig();
/* 49 ~ 63 = nosys count = 0 */
/* 50 = Getmypid count = 0 */
static int Sys_Getmypid();
/* 51 = Getppid count = 0 */
static int Sys_Getppid();
```

其次，在 `SystemCall.cpp` 中添加 `Sys_Getppid` 的定义

```

/* 51 = getpid    count = 0    */
int SystemCall::Sys_Getppid()
{
    User& u = Kernel::Instance().GetUser();
    u.u_ar0[User::EAX] = u.u_procp->p_ppid;

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

```

在新的系统调用处理子程序加入系统调用子程序入口表中

在 SystemCall.cpp 中找到对系统调用子程序入口表 m_SystemEntranceTable 赋值的一段程序代码，如下：

SystemCallTableEntry SystemCall::m_SystemEntranceTable[SYSTEM_CALL_NUM] = 选择第51项，并用 { 0, &Sys_Getppid } 来替换原来的 { 0, &Sys_Nosys }，即：第51号系统调用所需参数为 0 个，系统调用处理子程序的入口地址为 &Sys_Getppid。

```

{ 0, &Sys_Nosys },          /* 49 = nosys */
{ 0, &Sys_Getmypid },      /* 50 = getmypid */
{ 0, &Sys_Getppid },      /* 51 = getpid */
{ 0, &Sys_Nosys },          /* 52 = nosys */
{ 0, &Sys_Nosys },          /* 53 = nosys */

```

在 sys.h 文件中添加库函数的声明

找到 sys.h 文件，在其中加入名为 getpid 的库函数的声明。

```

int getmypid();

int getpid();

unsigned int getgid();

```

在 sys.c 中添加库函数的定义

在 sys.c 文件中添加库函数 getpid 的定义需要根据自己定义的系统调用在子程序入口表中的实际位置，填入正确的系统调用号。

```

int getpid()
{
    int res;
    __asm__ volatile ("int $0x80": "=a"(res): "a"(51));
    if (res >= 0)
        return res;
    return -1;
}

```

实验内容

一、按上述过程，分别编辑、编译并运行 forktest 和 exitwaittest 程序，截图展示程序的输出结果。

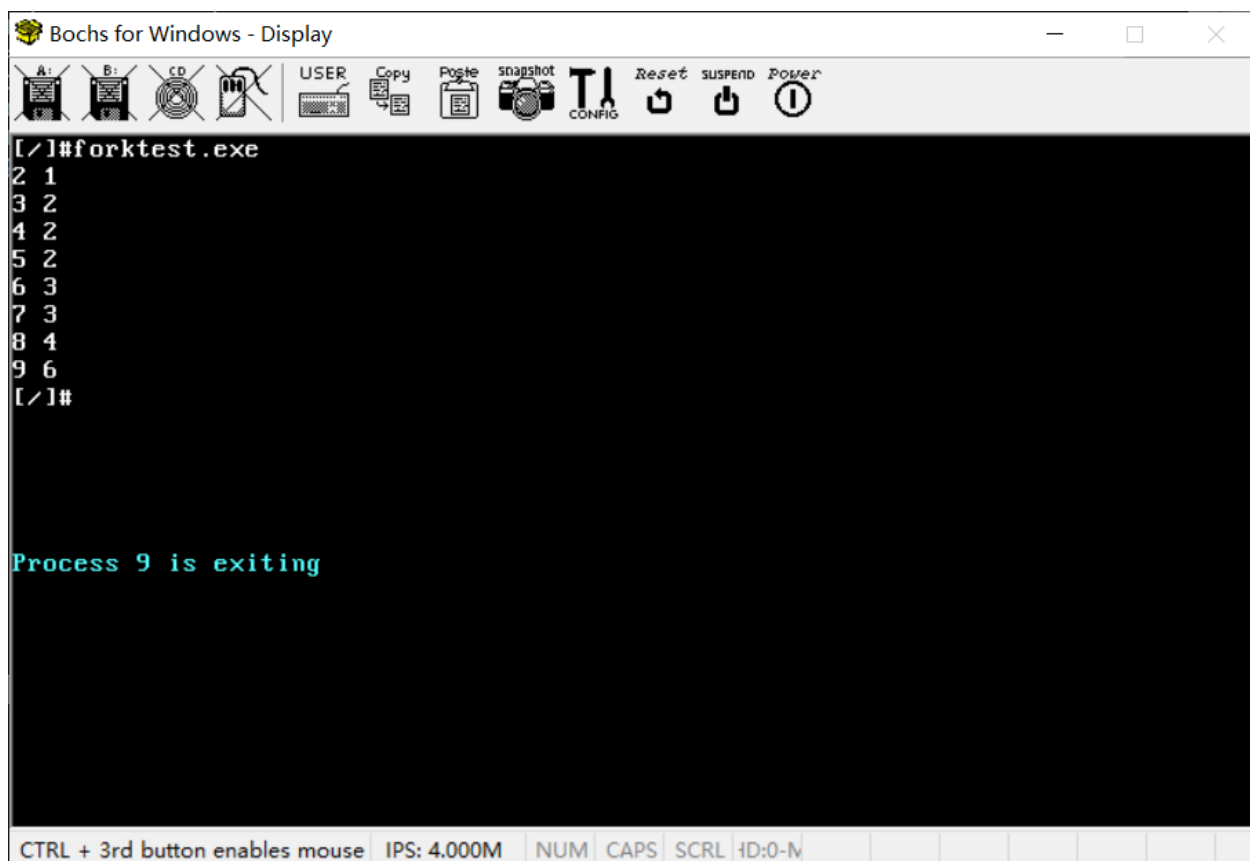
关于 fork 系统调用

在 UNIX V6++ 的 src/program 文件夹下添加一个 forktest.c 文件，按照实验二的方法编译后形成 UNIX V6++ 内核/bin 目录下的可执行文件 forktest。

```
getmypid.c SystemCall.cpp Machine.cpp Kernel.cpp SystemCall.h sys.h sys.c test.c
#include <stdio.h>
#include <sys.h>

int main1()
{
    int i;
    printf("%d %d \n", getpid(),getppid());
    for(i = 0; i < 3; ++i)
        if(fork()==0)
            printf("%d %d \n", getpid(),getppid());
    sleep(2);
    return 1;
}
```

启动 UNIX V6++, 并运行 forktest 程序, 得到下图所示的输出。



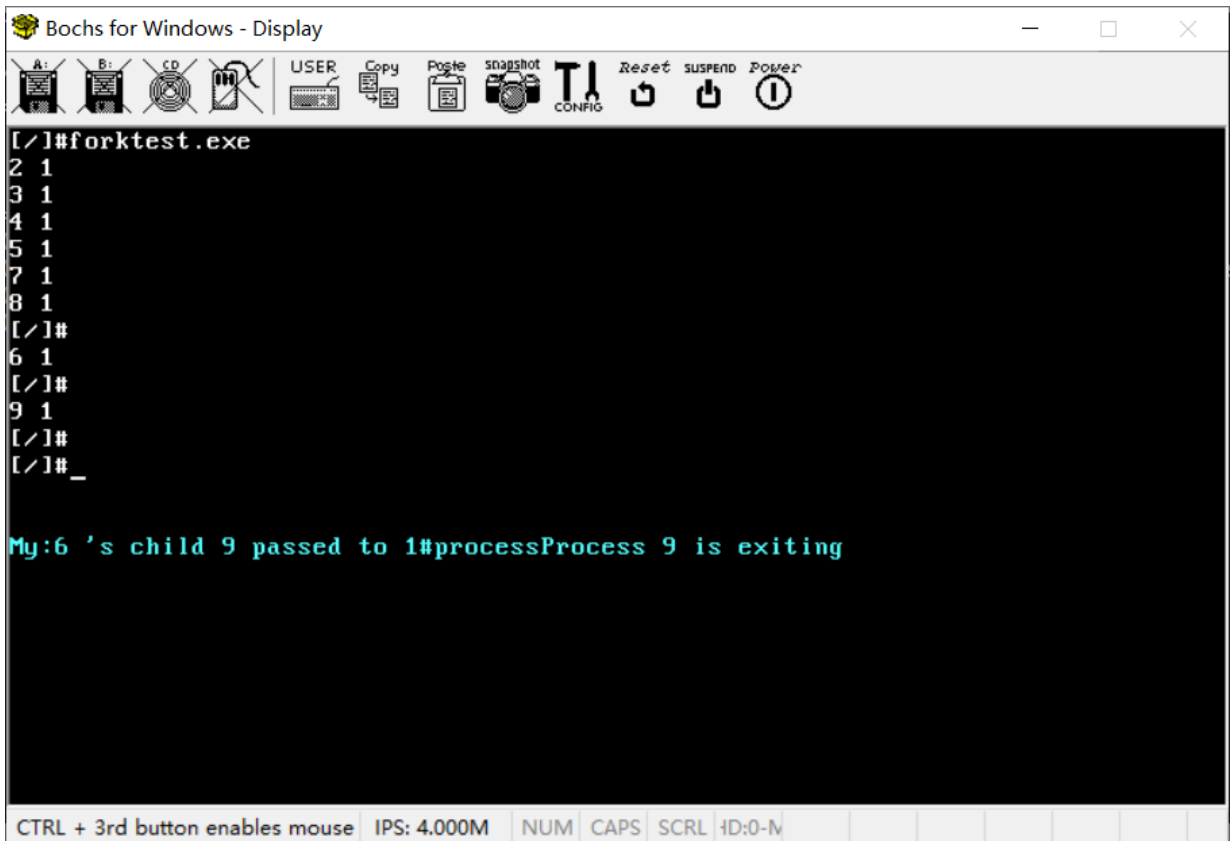
```
[/]#forktest.exe
2 1
3 2
4 2
5 2
6 3
7 3
8 4
9 6
[/]#

Process 9 is exiting
```

将 sleep(2) 语句删除, 重复上述的步骤后, 重新运行 forktest 程序, 获得下图的输出。这时按回车键, 观察屏幕输出会有什么变化。

```
getmypid.c SystemCall.cpp Machine.cpp Kernel.cpp SystemCall.h sys.h sys.c test.c
#include <stdio.h>
#include <sys.h>

int main1()
{
    int i;
    printf("%d %d \n", getpid(),getppid());
    for(i = 0; i < 3; ++i)
        if(fork()==0)
            printf("%d %d \n", getpid(),getppid());
    //sleep(2);
    return 1;
}
```



```
[/]#forktest.exe
2 1
3 1
4 1
5 1
7 1
8 1
[/]#
6 1
[/]#
9 1
[/]#
[/]#
[/]#_

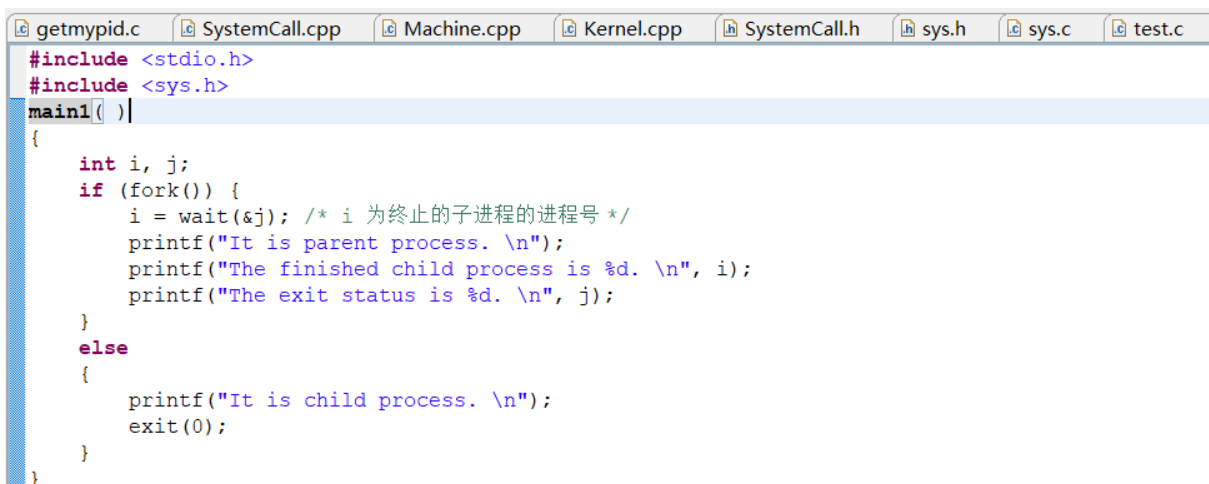
My:6 's child 9 passed to 1#processProcess 9 is exiting
```

可以看到，没按回车键之前只剩下 6 行输出结果，并且它们的父进程号都是 1。这是因为没有 `sleep`，进程运行完成后就销毁，因此回收进程的不是父进程，而是 1 号进程。

追加回车键，出现了之前本该出现的两个输出。

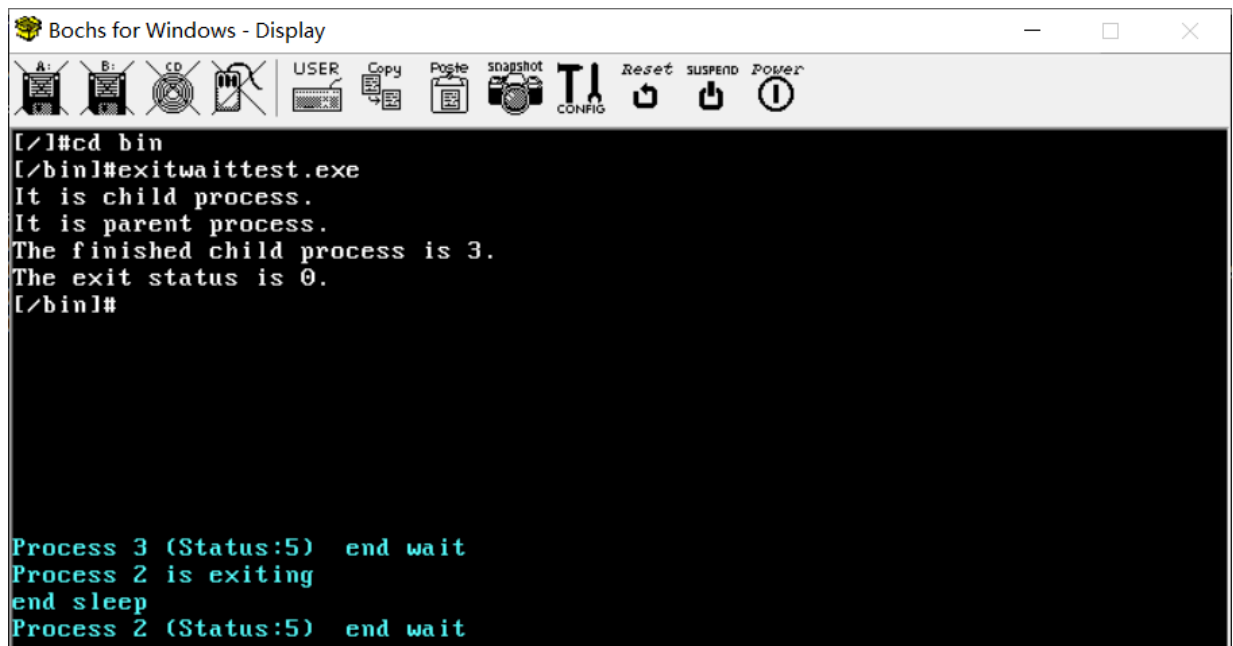
关于 `wait` 和 `exit` 系统调用

在 UNIX V6++ 的 `src/program` 文件夹下添加一个 `exitwaittest.c` 文件。按照实验三的方法编译后形成 UNIX V6++ 内核 `/bin` 目录下的可执行文件 `exitwaittest`。



```
getmypid.c SystemCall.cpp Machine.cpp Kernel.cpp SystemCall.h sys.h sys.c test.c
#include <stdio.h>
#include <sys.h>
main1( )
{
    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);
    }
}
```

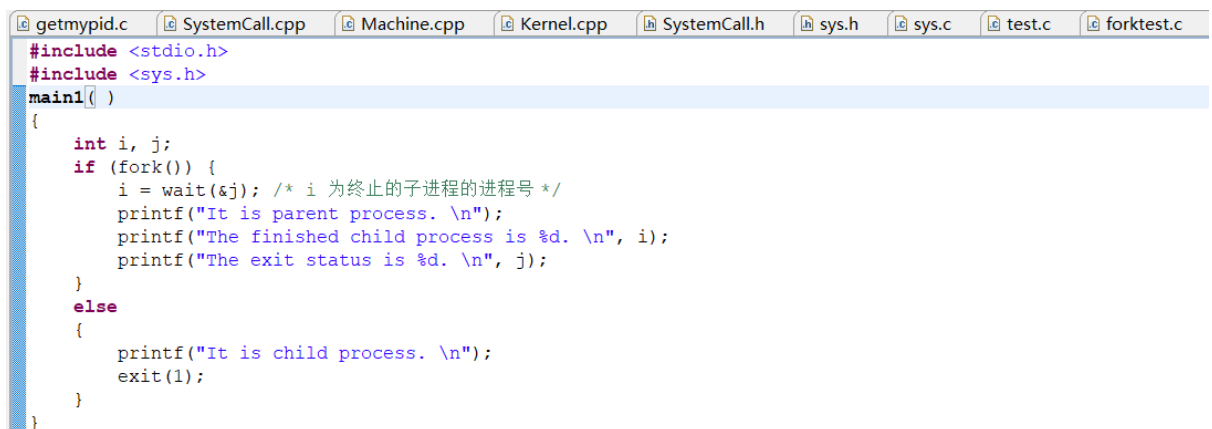
启动 UNIX V6++，并运行 `exitwaittest` 程序，得到输出如下。



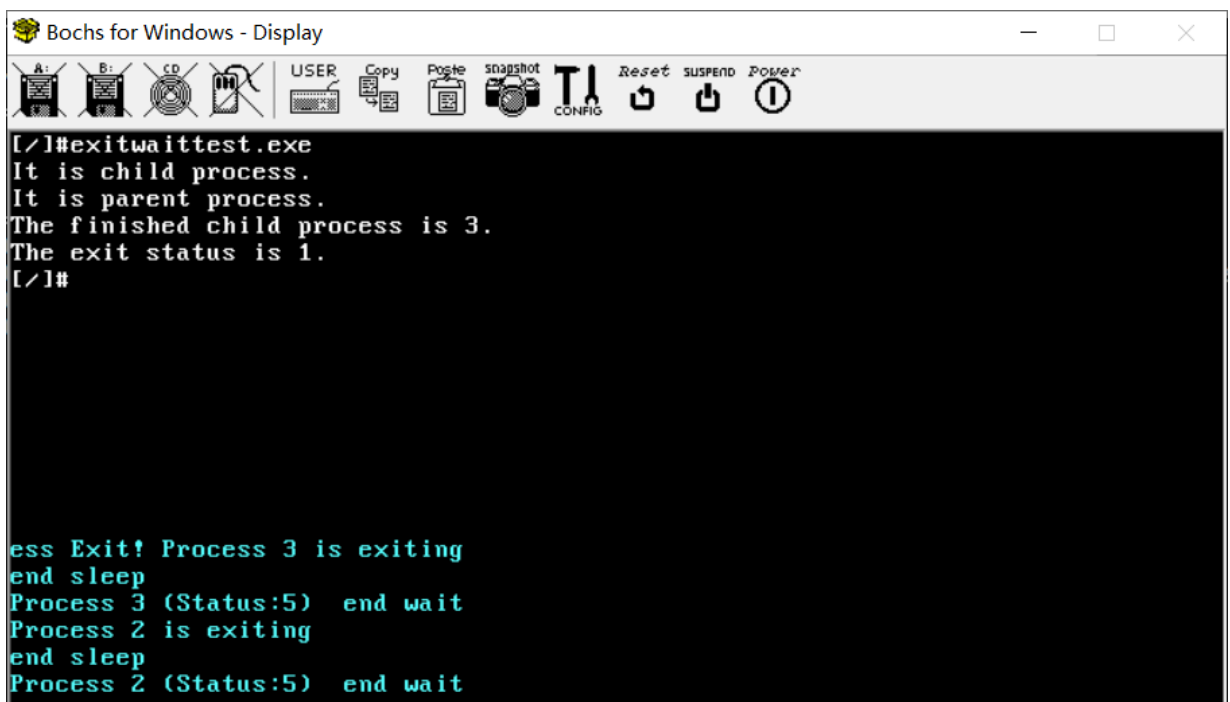
```
[/]#cd bin
[/bin]#exitwaittest.exe
It is child process.
It is parent process.
The finished child process is 3.
The exit status is 0.
[/bin]#

Process 3 (Status:5) end wait
Process 2 is exiting
end sleep
Process 2 (Status:5) end wait
```

将语句 `exit(0)` 修改为 `exit(1)`，观察编译运行后的输出情况。



```
getmypid.c SystemCall.cpp Machine.cpp Kernel.cpp SystemCall.h sys.h sys.c test.c forktest.c
#include <stdio.h>
#include <sys.h>
main1( )
{
    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(1);
    }
}
```

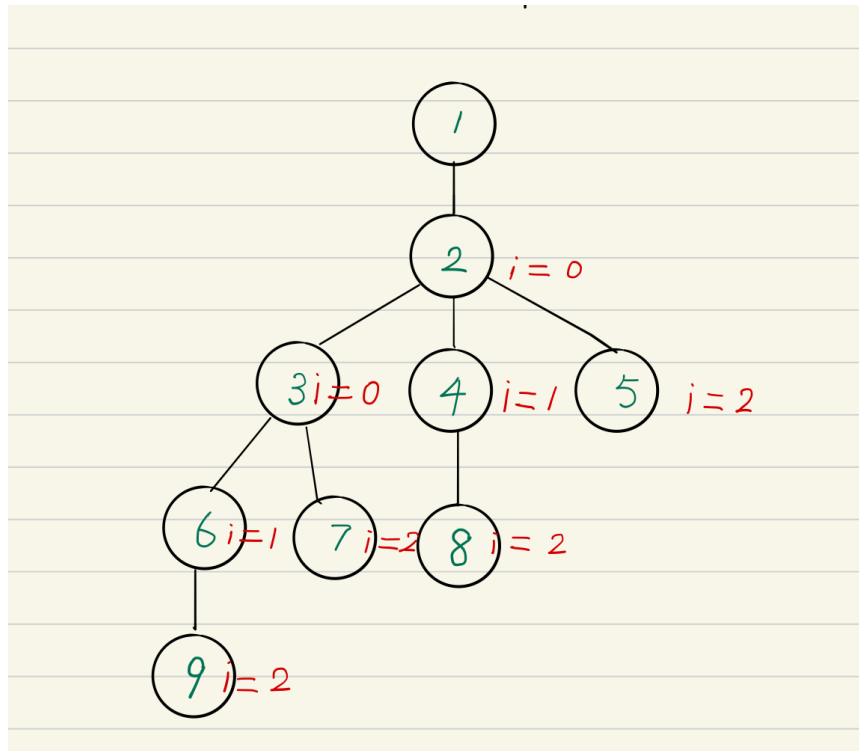


```
[/]#exitwaittest.exe
It is child process.
It is parent process.
The finished child process is 3.
The exit status is 1.
[/]#

Process 3 (Status:5) end wait
Process 2 is exiting
end sleep
Process 2 (Status:5) end wait
```

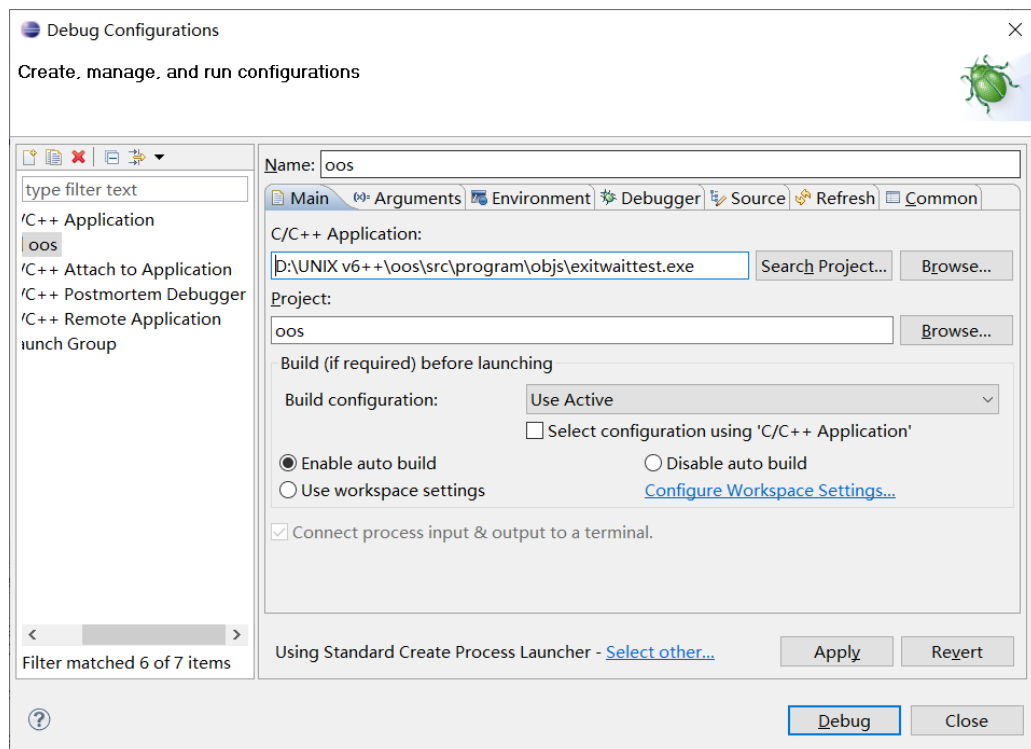
二、分析图 1 中的输出结果，指出 `i` 分别为 0，1，2 时，创建的是哪个进程？画出各个进程之间的父子关系。

当 $i = 0$ 时，创建的是进程 3；当 $i = 1$ 时，创建的是 4、6；当 $i = 2$ 时，创建的是 5、7、8、9；



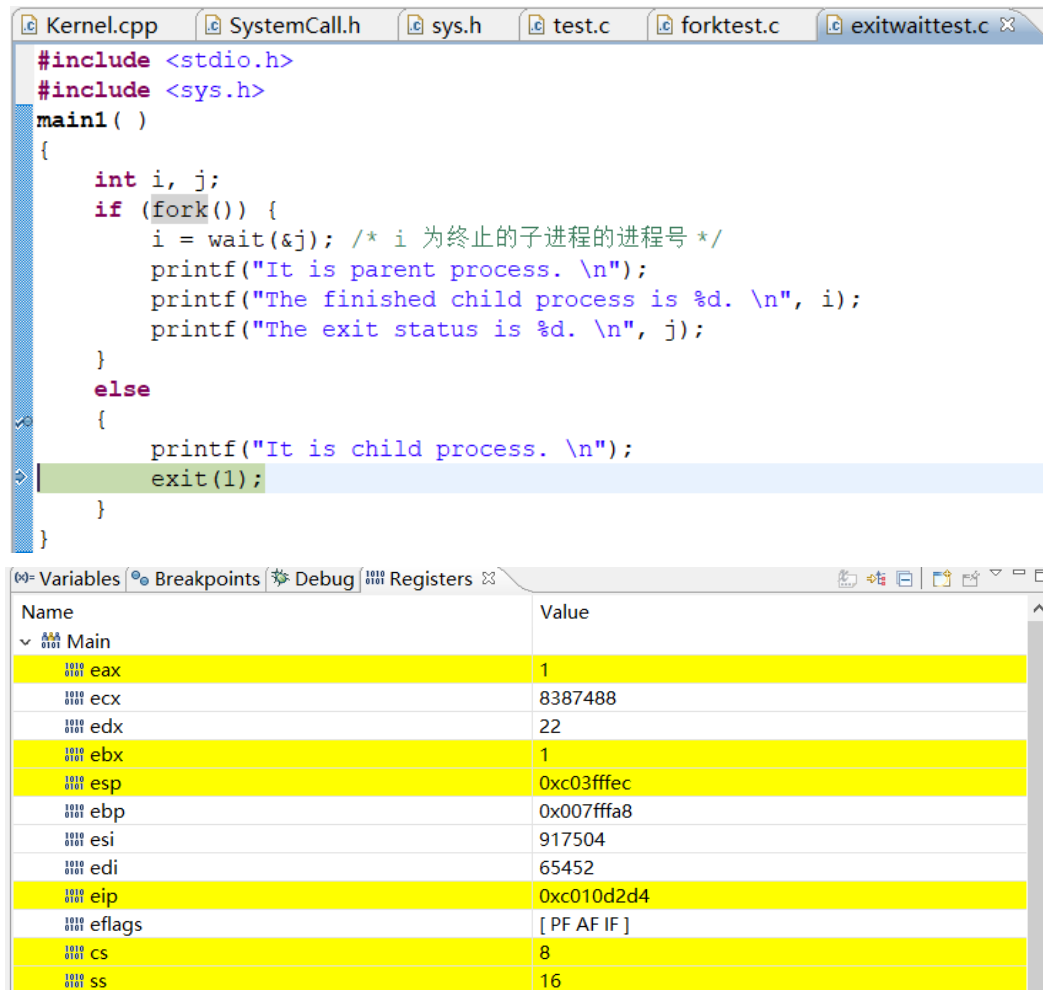
三、分析图 3 中的输出结果，解释父进程是如何接收到子进程的终止码的（可结合调试过程说明）？

对 `exitwaittest` 程序进行调试：



终止码的详细传递过程如下：

- (1) 子进程将在执行系统调用 `exit` 的过程中，借助现场保护，将终止码 1 压入其核心栈中保护 `EBX` 单元；



```
#include <stdio.h>
#include <sys.h>
main1( )
{
    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(1);
    }
}
```

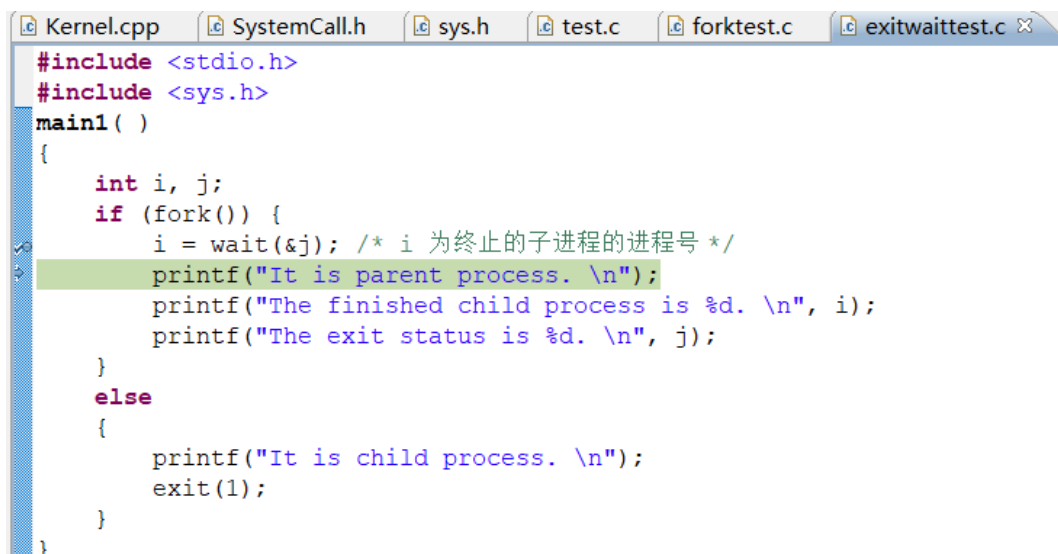
Name	Value
eax	1
ecx	8387488
edx	22
ebx	1
esp	0xc03ffec
ebp	0x007ffa8
esi	917504
edi	65452
eip	0xc010d2d4
eflags	[PF AF IF]
cs	8
ss	16

可以看到，终止码 1 已经被压入子进程核心栈中保护 EBX 单元。

(2)通过系统调用的参数传递，终止码 1 由核心栈中保护 EBX 单元送入子进程的 user 结构中的 u_arg[0];

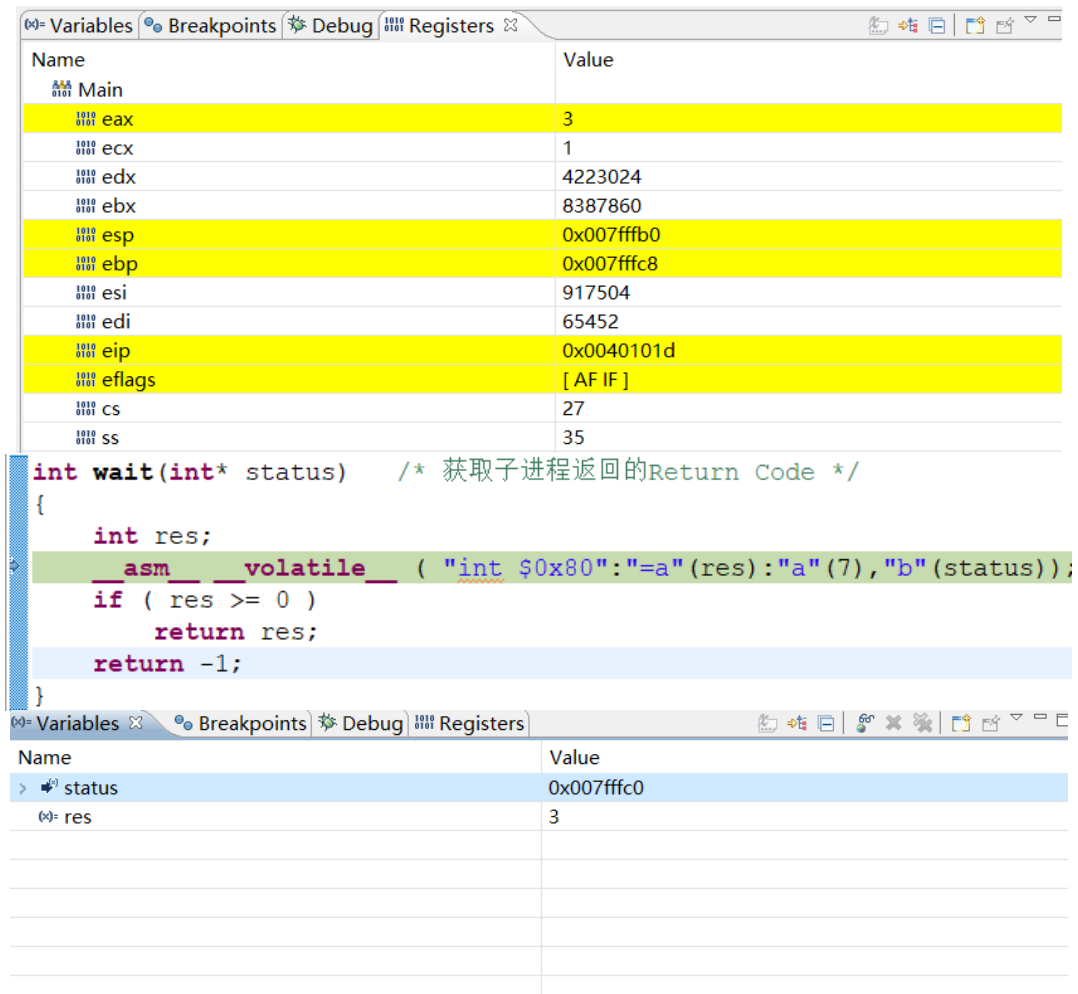
(3)在执行内核函数 Process::Exit 的过程中，user 结构被暂存在盘交换区上;

(4)父进程在执行系统调用 wait 的过程中，借助现场保护，将变量 j 的地址压入其核心栈中保护 EBX 单元;



```
#include <stdio.h>
#include <sys.h>
main1( )
{
    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(1);
    }
}
```

Name	Value
eax	1
ecx	8387488
edx	22
ebx	1
esp	0xc03ffec
ebp	0x007ffa8
esi	917504
edi	65452
eip	0xc010d2d4
eflags	[PF AF IF]
cs	8
ss	16



可以看到，wait 输入参数是变量 `j` 的地址 `0x007fffc0`，将被压入父进程核心栈中保护 EBX 单元。

(5)通过系统调用的参数传递，变量 `j` 的地址由核心栈中保护 EBX 单元送入父进程的 `user` 结构中的 `u_arg[0]`;

(6)在执行内核函数 `ProcessManager::Wait` 的过程中，从磁盘将子进程的 `user` 结构读入一个内存缓存，并将其中子进程 `u_arg[0]` 单元中保存的终止码 1 写入父进程 `u_arg[0]` 指向的内存单元即变量 `j`。

四、分析图 2 中的输出结果和图 1 输出不同的原因是什么？

`Sleep(2)` 的目的是让父进程等一等子进程，如果父进程先上台，那么父进程执行完就结束了，将来子进程结束的时候找不到父进程来协助结束进程，只能将 1 号进程作为自己的父进程，所以去掉 `Sleep(2)` 后输出的父进程都是 1 号进程。

至于去掉 `Sleep(2)` 后少了两行，而继续按回车键，又会重新输出，原因是屏幕输出需要调用外设的字符设备，而设备倾向于积累满一行再全部进行输出，因此输出具有一定的延迟，导致两条输出在进程已经结束的情况下也没有进行，此时按下回车，设备收到输出命令，把存留在缓存中的两条结果继续输出。