P04实验报告-在UNIXV6++中增加新的系统调用

1. 实验目的

（1）结合课程所学知识，通过在 UNIX V6++源代码中实践操作添加一个新的系统调用，熟悉 UNIX V6++中系统调用相关部分的程序结构。
（2）通过调试观察一次系统调用的全过程，进一步掌握系统调用响应与处理的流程，特别是其中用户态到核心态的切换和栈帧的变化。
（3）通过实践，进一步掌握 UNIX V6++重新编译及运行调试的方法。

2.实验设备及工具

已配置好UNIX V6++运行和调试环境的PC 机一台。

3.实验内容

3.1在UNIX V6++中添加一个新的系统调用接口

先在SystemCall 类中找一个空余的系统调用号添加系统调用处理子程序的定义，此处为49，加入的子程序的名字是 Sys_Getppid，系统调用的功能为返
回指定进程的父进程的 ID 号。

```
    { 1, &Sys_Pipe  },              /* 42 = pipe    */
    { 1, &Sys_Times },              /* 43 = times   */
    { 4, &Sys_Profil},              /* 44 = prof    */
    { 0, &Sys_Nosys },              /* 45 = nosys   */
    { 1, &Sys_Setgid},              /* 46 = setgid  */
    { 0, &Sys_Getgid},              /* 47 = getgid  */
    { 2, &Sys_Ssig  },              /* 48 = sig */
    { 1, &Sys_Getppid },            /* 49 = Getppid */
    { 0, &Sys_Nosys },              /* 50 = nosys   */
    { 0, &Sys_Nosys },              /* 51 = nosys   */
    { 0, &Sys_Nosys },              /* 52 = nosys   */
```

然后在SystemCall 类中添加系统调用处理子程序的定义，先添加声明，再写好函数内容，

```
    static int Sys_Setgid();

    /*  47 = getgid count = 0   */
    static int Sys_Getgid();

    /*  48 = sig    count = 2   */
    static int Sys_Ssig();

    /*49=getppid count=1*/
    static int Sys_Getppid();
    /*  50 ~ 63 = nosys count = 0   */
```

Sys_Getppid 函数完成的功能是根据给定的进程id 的值，返回该进程
的父进程，具体实现步骤包括：

（1）通过 Kernel::User 函数获取当前进程的User 结构（详见实验三），进而找到User结构中u_arg[0]保存的此次系统调用的参数值，即给定进程的id 号，并赋值给 curpid；
（2）通过Kernel::GetProcessManager 函数获取内核的ProcessManager，进而找到ProcessManager中的 process 表；
（3）线性查找process 表中所有进程的Proc 结构，发现id 号和curpid 相等的进程，将其父进程id 号存入核心栈中保存EAX 寄存器的单元，以作为该系统调用的返回值；如果没有找到，即给定id 号的进程不存在，则返回-1。

```
/*49=getppid    count=1*/
int SystemCall::Sys_Getppid()
{
    ProcessManager& procMgr=Kernel::Instance().GetProcessManager();
    User& u=Kernel::Instance().GetUser();

    int i;
    int curpid=(int)u.u_arg[0];
    for(i=0;i<ProcessManager::NPROC;i++)
    {
        if(procMgr.process[i].p_pid==curpid)
        {
            u.u_ar0[User::EAX]=procMgr.process[i].p_ppid;
        }
    }
    return 0;
}
```

3.2为新的系统调用添加对应的库函数

UNIX V6++中，所有的库函数的声明在文件src/lib/include/sys.h 中，而所有库函数的定义在文件src/lib/src/sys.c 中，我们完成与Sys_Getppid 系统调用对应的库函数，首先在sys.h 文件中添加库函数的声明

```
int gettime(struct tms* ptms);    /* 读系统时钟 */

/* 获取进程用户态、核心态CPU时间片数 */
int times(struct tms* ptms);

/* 获取系统进程切换次数 */
int getswtch();

/* 启用屏幕底部的lines行输出调试信息 */
int trace(int lines);

int getppid(int pid);

#endif
```

然后在sys.c 中添加库函数的定义

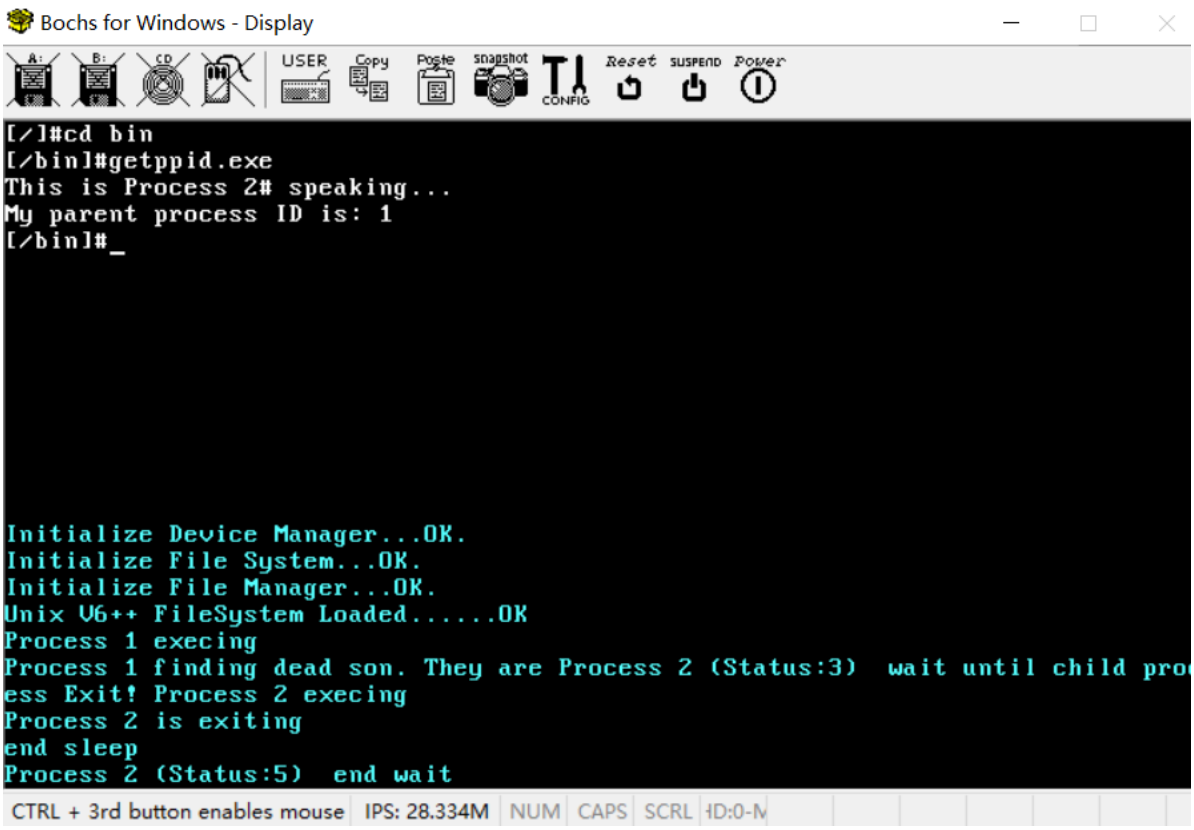
```
int getppid(int pid)
{
    int res;
    __asm__ volatile("int $0x80":"=a"(res):"a"(49),"b"(pid) );

    if(res>=0)
    {
        return res;
    }
    return -1;
}
```

3.3编写测试程序

我们编写一个简单的测试程序来测试我们添加的新的系统调用的有效性

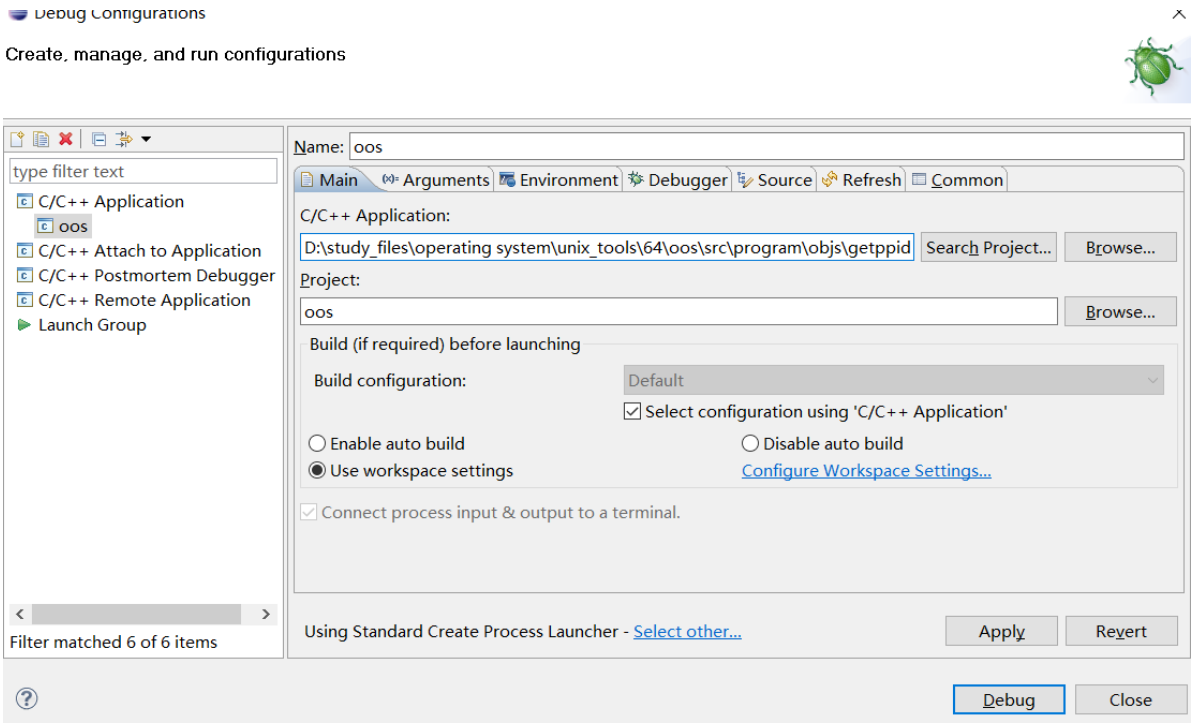
```
#include<stdio.h>
#include<sys.h>

int main()
{
    int pid,ppid;

    pid=getpid();
    ppid=getppid(pid);

    printf("This is Process %d# speaking...\n",pid);
    printf("My parent process ID is: %d#\n",ppid);
    return 0;
}
```

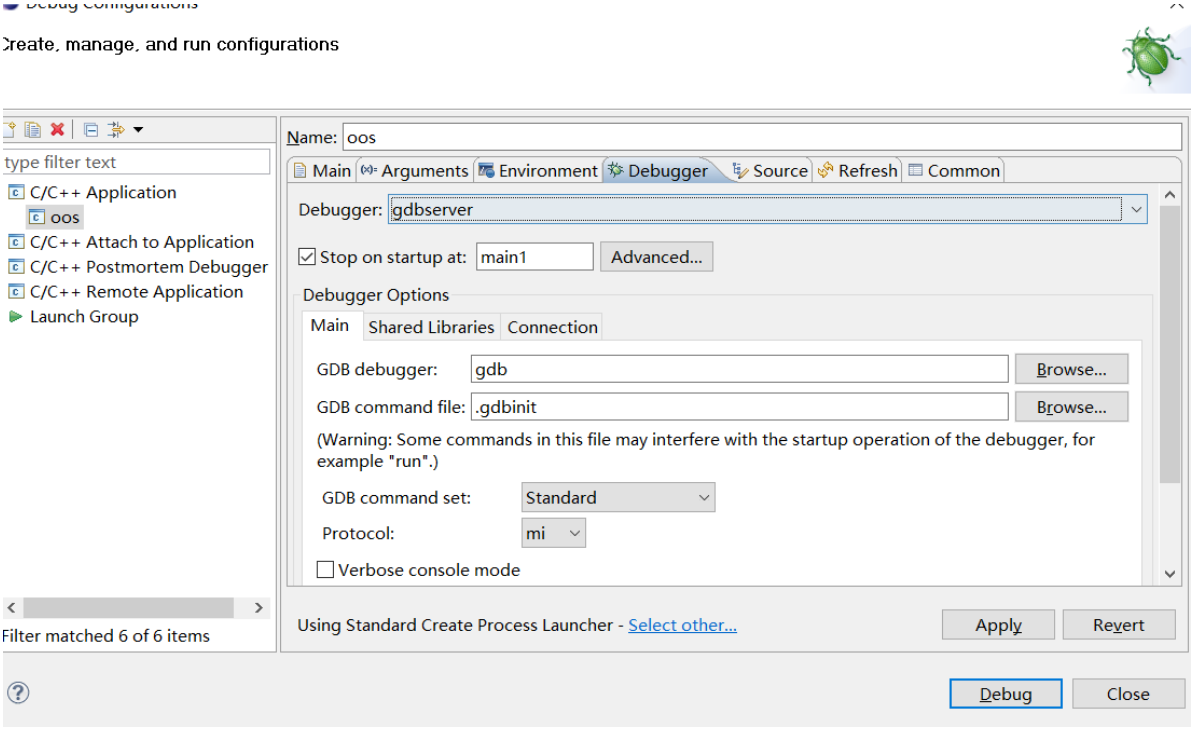
运行模式下启动 UNIX V6++，可以看到程序输出正确



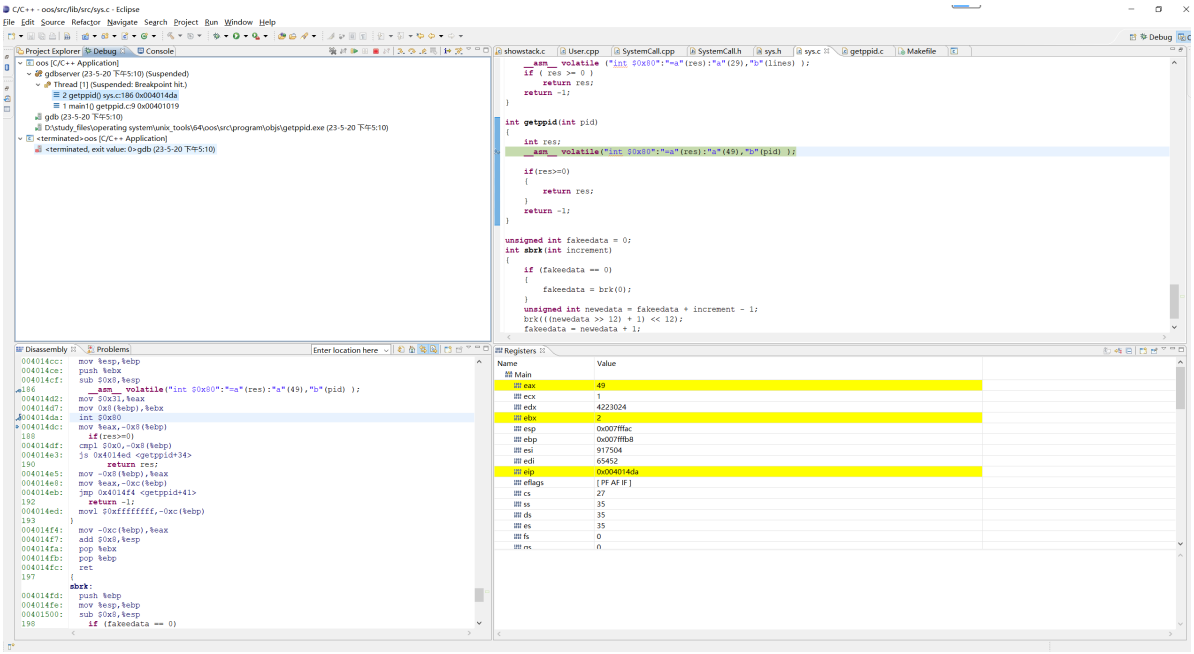
3.4调试程序

首先改成调试模式，然后更改调试入口设置，先在用户态调试

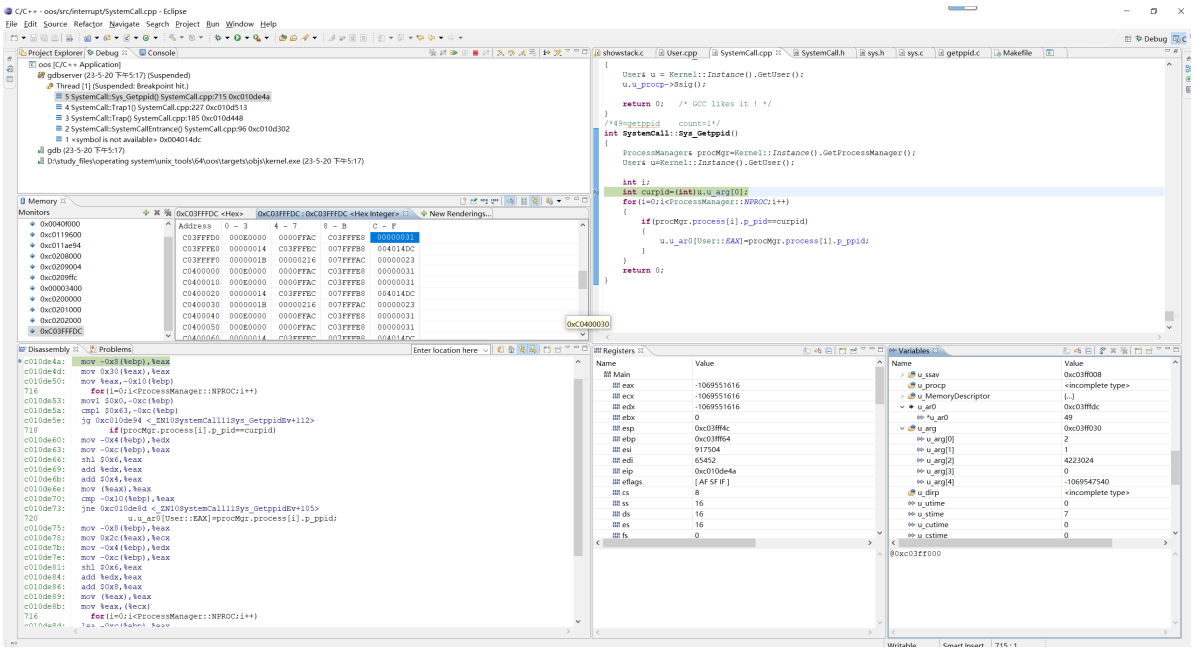




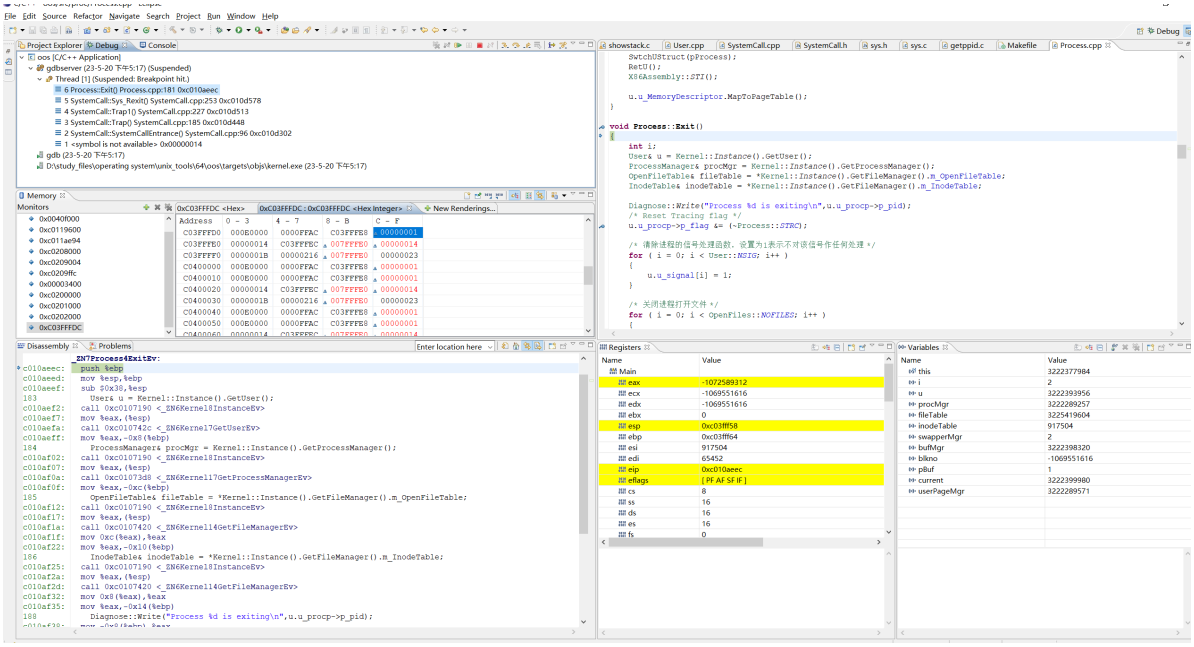
我们将断点设置在库函数getppid 中的语句，待程序停在此处时，在汇编指令“int 0x80”处增加一个断点并让程序运行到这里。可以看到此时，eax 中为系统调用号49，ebx 中为参数值2（现运行进程的ID号）。eip 的值正好是“int 0x80”的地址。这是系统调用发生前在用户态下执行的代码，接下来将调试模式改为核心态调试



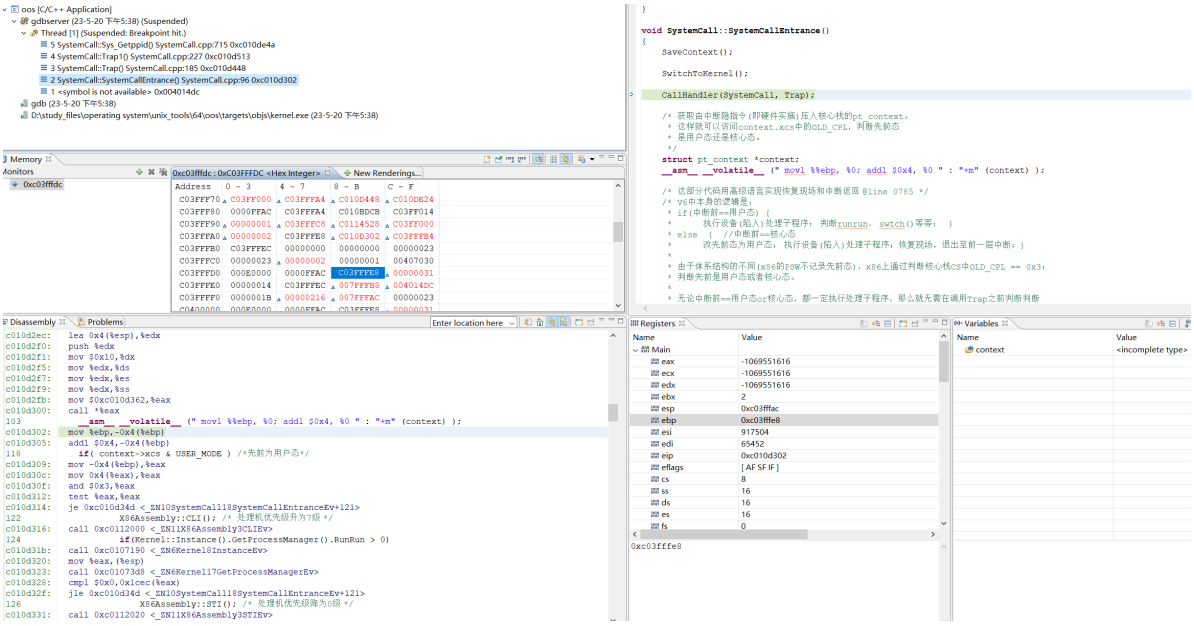
内核态下，在 Sys_Getppid 函数的“int curpid=(int)u.u_arg[0]”赋值语句处添加断点。可以看到。u_arg[0]处的值为2，说明参数已进入进程的User结构。



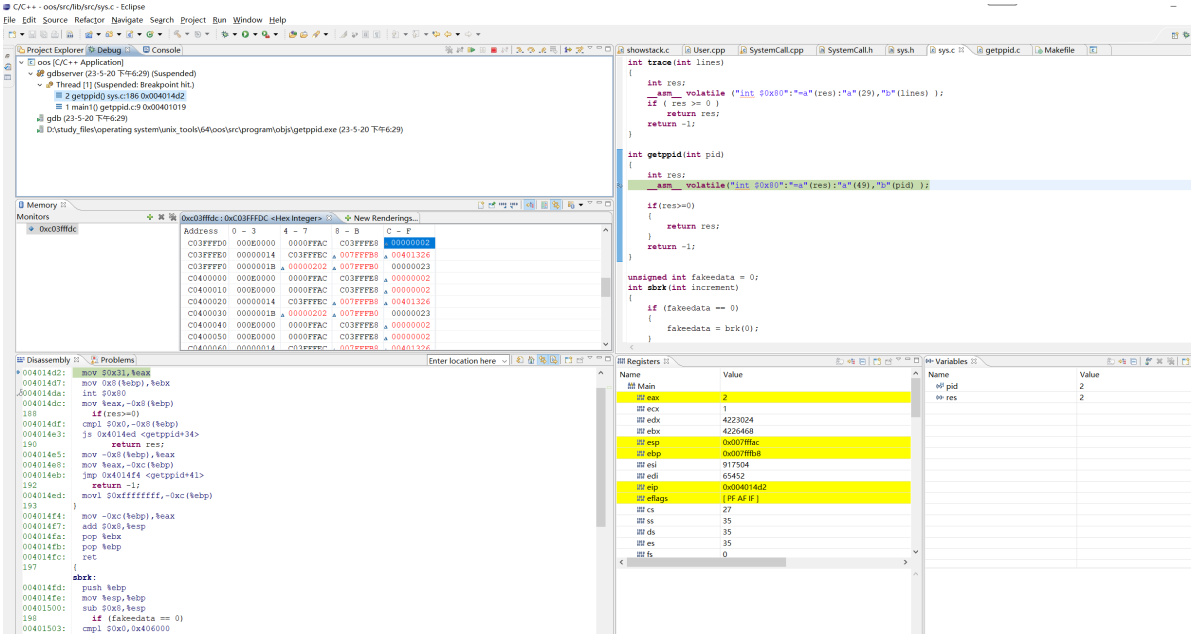
当执行到 Sys_Getppid 的最后一条语句时，可以看到 0xC03FFFDC 地址处对应的值变为1，即返回值被存入核心栈中保存EAX 的单元。



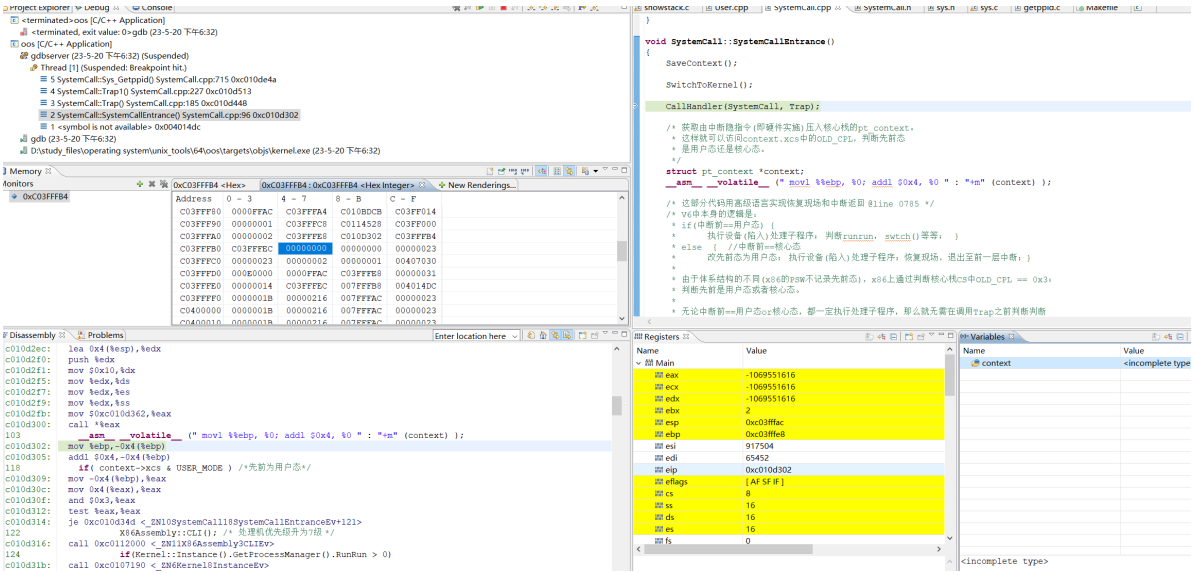
从u.u_ar0 的值 0xC03FFFDC 入手，我们可以在Memory 窗口中恢复整个核心栈系统调用栈帧的全部内容，可以看到0xC03FFFD8处存放0xC03FFFE8为系统调用子程序的入口地址，0xC03FFFDC存放系统调用号0xC03FFFC4存u_arg数组起始地址，后面五项依次为系统调用参数



让程序重新在用户态执行，查看窗口，找到0x007FFFB8和0x007FFFAC分别为中断前用户态下的EBP和ESP，0x00000023即35为SS寄存器中的值，0x0000001B即27位CS寄存器中值



在核心态下执行，可以观察到地址0xC03FFFB4以上的核心栈，0xC03FFFB4和0xC03FFFEC是中断保护现场后的esp，然后0xC010D302为系统调用入口程序地址，栈顶的0xC03FFFE8则为系统调用处理程序局部变量区



补全图中如下

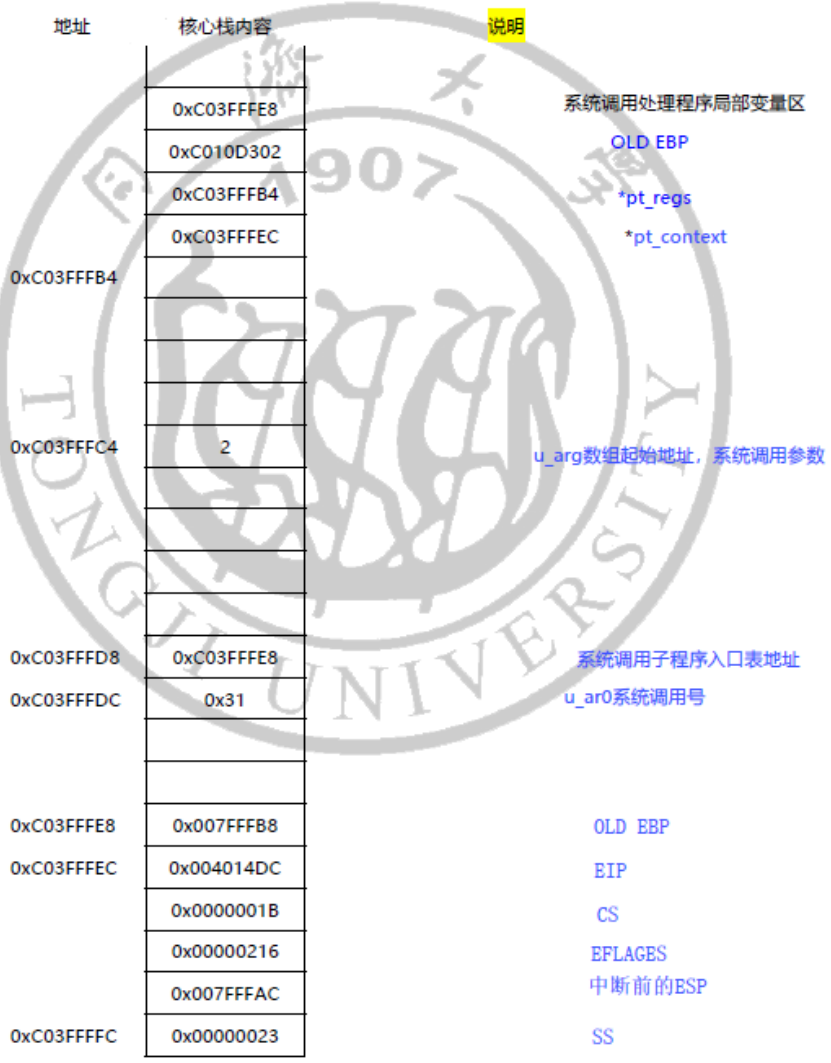


图 10：系统调用号的核心栈