

# 同济大学计算机系

## 操作系统 P04 实验报告



学 号 2151769

姓 名 吕博文

专 业 计算机科学与技术

授课老师 邓蓉

## 实验四：UNIX V6++中添加新的系统调用

一、完成实验 4.1，截图说明操作过程，掌握在 UNIX V6++中添加一个新的系统调用的方法，并总结出主要步骤

### (1) 在系统调用子程序入口表中添加新的入口

如实验文档所示，我们在 `SystemCall.cpp` 中找到对系统调用子程序入口表 `m_SystemEntranceTable` 赋值的一段代码并选择第 49 项设置为 `getppid` 项

```
{ 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 */
```

这里加入的子程序名位 Sys\_Getppid,后续实现也相应的实现了得到当前进程父进程 id 号的功能

## (2) 在 SystemCall 类中添加系统调用子程序的定义

在 SystemCall.h 文件中添加该系统调用处理子程序的声明如下:

```
int setuid(short uid);  
  
int getppid(int pid); //获取当前进程的父进程id号  
  
int gettimeofday(struct tms* ptms); /* 读系统时钟 */
```

其次，在 `SystemCall.cpp` 中添加对应的实现

```

/* 49 = getppid count = 1*/
int SystemCall::Sys_Getppid()
{
    //获取当前的User结构和proc表
    ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
    User& u = Kernel::Instance().GetUser();

    int i;
    //根据User结构中的内容获取cur_pid
    int curpid = (int)u.u_arg[0];
    //循环整个proc表，寻找当前进程的fa进程id

    u.u_ar0[User::EAX] = -1;
    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) 总结: 在 UNIX V6++ 中添加一个新的系统调用步骤为: (1) 在系统

调用子程序入口表中添加新的入口；（2）在 SystemCall 类中添加系统调用子程序的定义

## 二、完成实验 4.2，掌握在 UNIX V6++中添加库函数的方法，截图说明主要操作步骤。

（1）在 src/lib/src/sys.h 中添加库函数的声明：

```
int setuid(short uid);

int getppid(int pid); //获取当前进程的父进程id号

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

（2）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）总结，在 UNIX V6++中添加库函数的方法步骤为：在 src/lib/src/sys.h 中添加库函数的声明；在 sys.c 中添加库函数的定义

## 三、完成实验 4.3 ~ 4.4，编写测试程序，通过调试运行说明添加的系统调用的正确，截图说明主要的调试过程和关键结果。

（1）首先编写简单测试程序，这里直接用之前的 showStack.exe 作为测试程序，改动了程序中的代码实现：

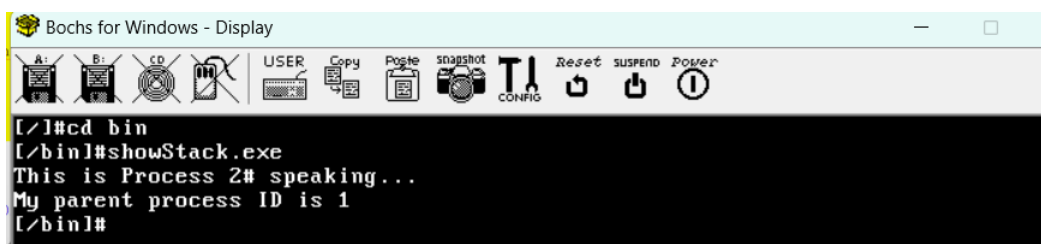
```
#include<stdio.h>
#include<sys.h>

int main1()
{
    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;
}
```

(2) 在运行模式下进入 UNIX V6++运行测试程序，观察测试结果如下：

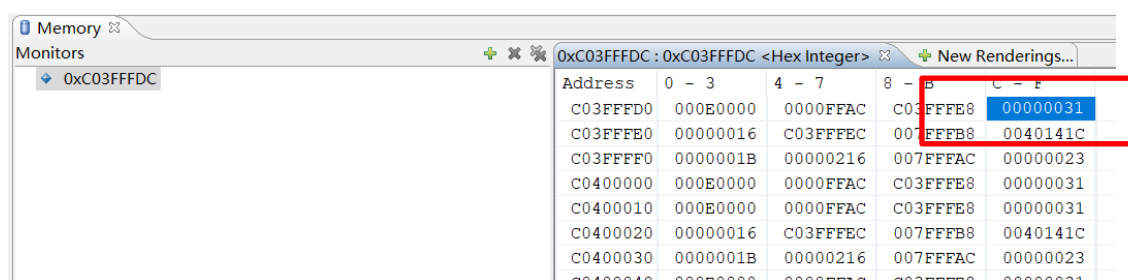


```
[/]#cd bin
[/bin]#showStack.exe
This is Process 2# speaking...
My parent process ID is 1
[/bin]#
```

(3) 因为涉及到到核心态函数的运行调试，我们设置调试对象为 Kernel.exe，在调试模式下进行调试如下：

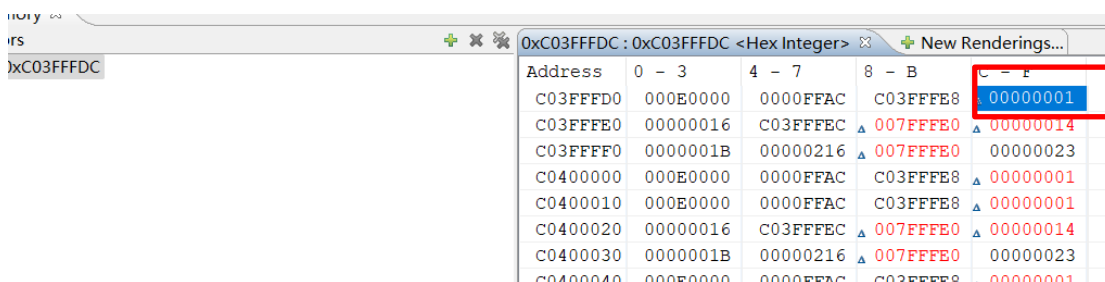
Name	Value
> u_rsav	0xc03ff000
> u_ssav	0xc03ff008
> u_procp	<incomplete type>
> u_MemoryDescriptor	
u_ar0	0xc03fffdc
*u_ar0	49
> u_arg	0xc03ff030

观察到 u\_ar0 的值为 49



Address	0 - 3	4 - 7	8 - B	C - F
C03FFFD0	000E0000	0000FFAC	C03FFE8	00000031
C03FFFE0	00000016	C03FFFE0	007FFFB8	0040141C
C03FFFF0	0000001B	00000216	007FFAC	00000023
C0400000	000E0000	0000FFAC	C03FFE8	00000031
C0400010	000E0000	0000FFAC	C03FFE8	00000031
C0400020	00000016	C03FFFE0	007FFFB8	0040141C
C0400030	0000001B	00000216	007FFAC	00000023
C0400040	000E0000	0000FFAC	C03FFE8	00000031

当前内存中观察也是十六进制的 0x31 为 10 进制的 49



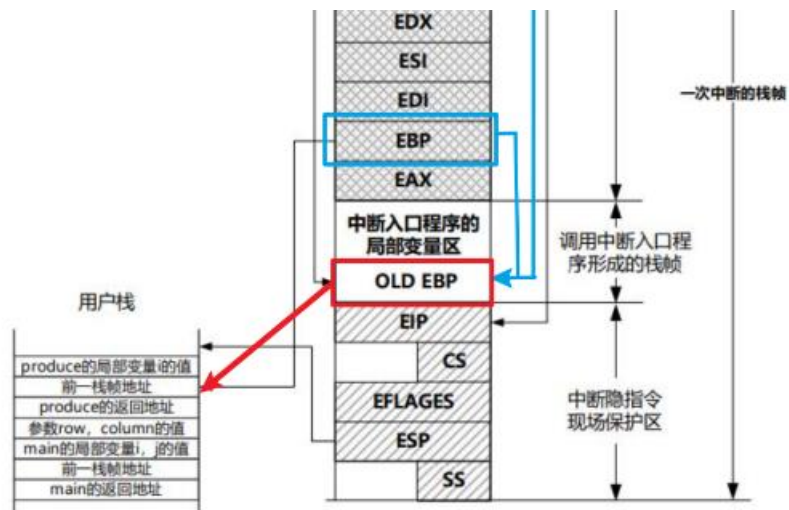
Address	0 - 3	4 - 7	8 - B	C - F
C03FFFD0	000E0000	0000FFAC	C03FFE8	00000001
C03FFFE0	00000016	C03FFFE0	007FFFB8	00000014
C03FFFF0	0000001B	00000216	007FFAC	00000023
C0400000	000E0000	0000FFAC	C03FFE8	00000001
C0400010	000E0000	0000FFAC	C03FFE8	00000001
C0400020	00000016	C03FFFE0	007FFFB8	00000014
C0400030	0000001B	00000216	007FFAC	00000023
C0400040	000E0000	0000FFAC	C03FFE8	00000001

调试运行结束后，我们发现，u\_ar0 存储的值变为 1，也和最后屏幕上输出的结果当前进程的父进程为 1 号进程相吻合。

四、在完成 4.4 的基础上，设计调试方案，确定图 10 中黄色标注的几个地址单元分别是什么。

我们将断点设置在 getppid 函数的汇编语言执行前，首先，我们在本地运行出的 ESP 值为 0x007ffac, EIP 值为 0x0040141c, EIP 上方值为 0x007fffb8, 最上方的 EBP 值为 0xC03FFFE8;

我们可以考虑使用以下图来理解四个地址的含义：



最下方的 ESP 表示原来用户栈中的 ESP 值，之后的 EIP 表示 getpid 函数接下来要执行的语句地址，在本例中及是 `if(res<=0)return -1` 语句，之后的 `0x007fffb8` 是 OLD EBP 保存用户栈的 EBP，最上方的 EBP 保存当前 getpid 函数的 EBP 的值。