

同济大学计算机系

操作系统课程实验报告



学 号 2251557

姓 名 代文波

专 业 计算机科学与技术

授课老师 方钰

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, 进一步理解和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

已配置好的 UNIX V6++ 运行和调试环境。

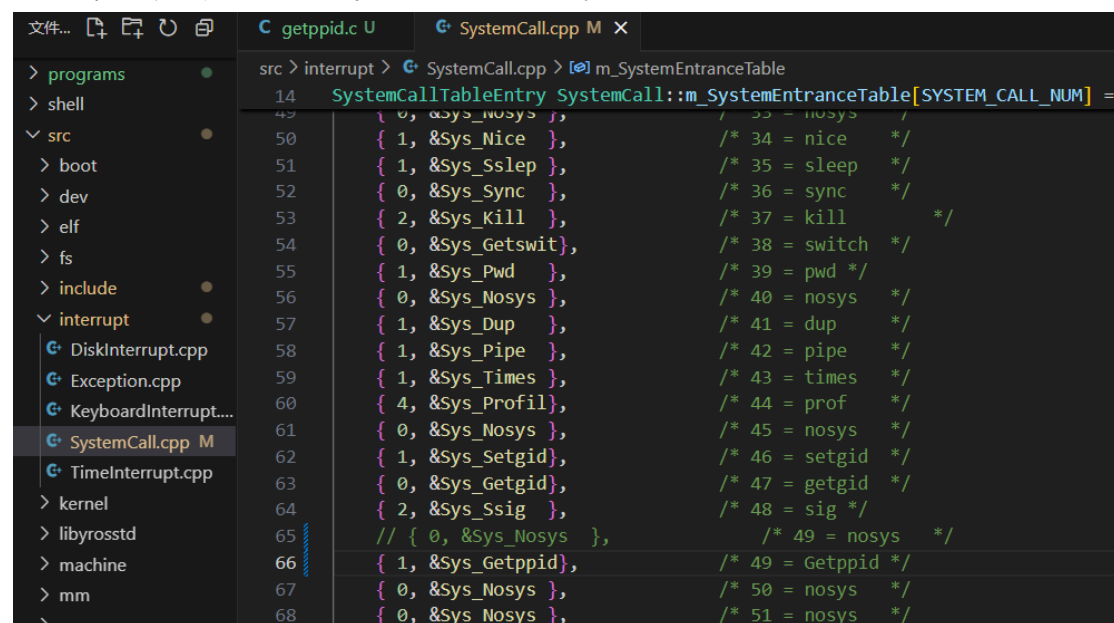
3. 预备知识

- (1) UNIX V6++ 中系统调用的执行过程;
- (2) UNIX V6++ 中所有和系统调用相关的代码模块。

4. 实验内容

4.1 在 UNIX V6++ 中添加一个新的系统调用接口

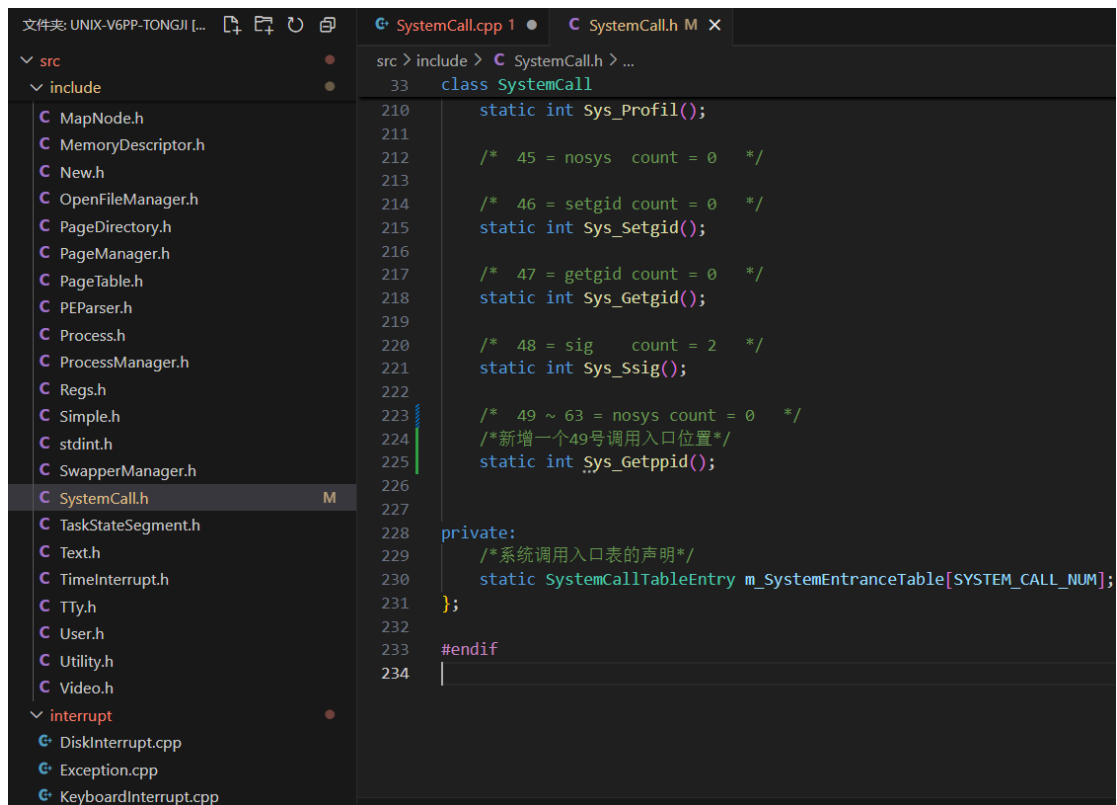
4.1.1. 在系统调用处理子程序入口表中添加新的入口



```
src > interrupt > SystemCall.cpp > [m_SystemEntranceTable
14 SystemCallTableEntry SystemCall::m_SystemEntranceTable[SYSTEM_CALL_NUM] =
49 { 0, &Sys_Nosys }, /* 33 = nosys */
50 { 1, &Sys_Nice }, /* 34 = nice */
51 { 1, &Sys_Ssleep }, /* 35 = sleep */
52 { 0, &Sys_Sync }, /* 36 = sync */
53 { 2, &Sys_Kill }, /* 37 = kill */
54 { 0, &Sys_Getswtch }, /* 38 = switch */
55 { 1, &Sys_Pwd }, /* 39 = pwd */
56 { 0, &Sys_Nosys }, /* 40 = nosys */
57 { 1, &Sys_Dup }, /* 41 = dup */
58 { 1, &Sys_Pipe }, /* 42 = pipe */
59 { 1, &Sys_Times }, /* 43 = times */
60 { 4, &Sys_Profil }, /* 44 = prof */
61 { 0, &Sys_Nosys }, /* 45 = nosys */
62 { 1, &Sys_Setgid }, /* 46 = setgid */
63 { 0, &Sys_Getgid }, /* 47 = getgid */
64 { 2, &Sys_Ssig }, /* 48 = sig */
65 // { 0, &Sys_Nosys }, /* 49 = nosys */
66 { 1, &Sys_Getppid }, /* 49 = Getppid */
67 { 0, &Sys_Nosys }, /* 50 = nosys */
68 { 0, &Sys_Nosys }, /* 51 = nosys */
```

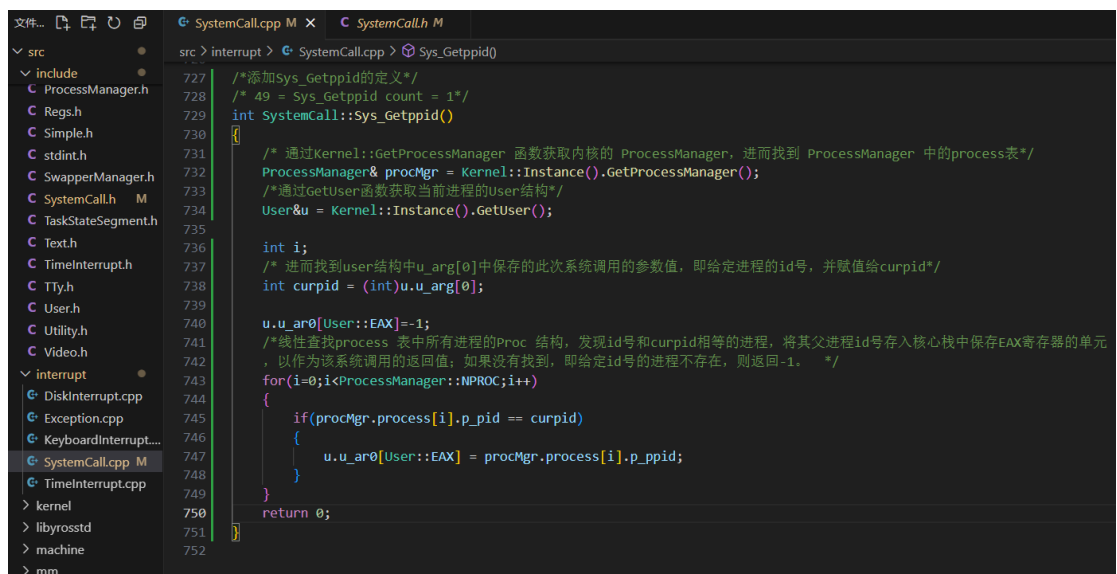
4.1.2 在 SYSTEMCALL 类中添加新的系统调用处理子程序

- (1) 在 SystemCall.h 文件中添加该系统调用处理子程序 Sys_Getppid 的声明



```
src > include > C SystemCall.h > ...
33 class SystemCall
210 static int Sys_Profil();
211
212 /* 45 = nosys count = 0 */
213
214 /* 46 = setgid count = 0 */
215 static int Sys_Setgid();
216
217 /* 47 = getgid count = 0 */
218 static int Sys_Getgid();
219
220 /* 48 = sig count = 2 */
221 static int Sys_Ssig();
222
223 /* 49 ~ 63 = nosys count = 0 */
224 /*新增一个49号调用入口位置*/
225 static int Sys_Getppid();
226
227
228 private:
229 /*系统调用入口表的声明*/
230 static SystemCallTableEntry m_SystemEntranceTable[SYSTEM_CALL_NUM];
231 };
232
233 #endif
234
```

(2) 在 SystemCall.cpp 中添加 Sys_Getppid 的定义



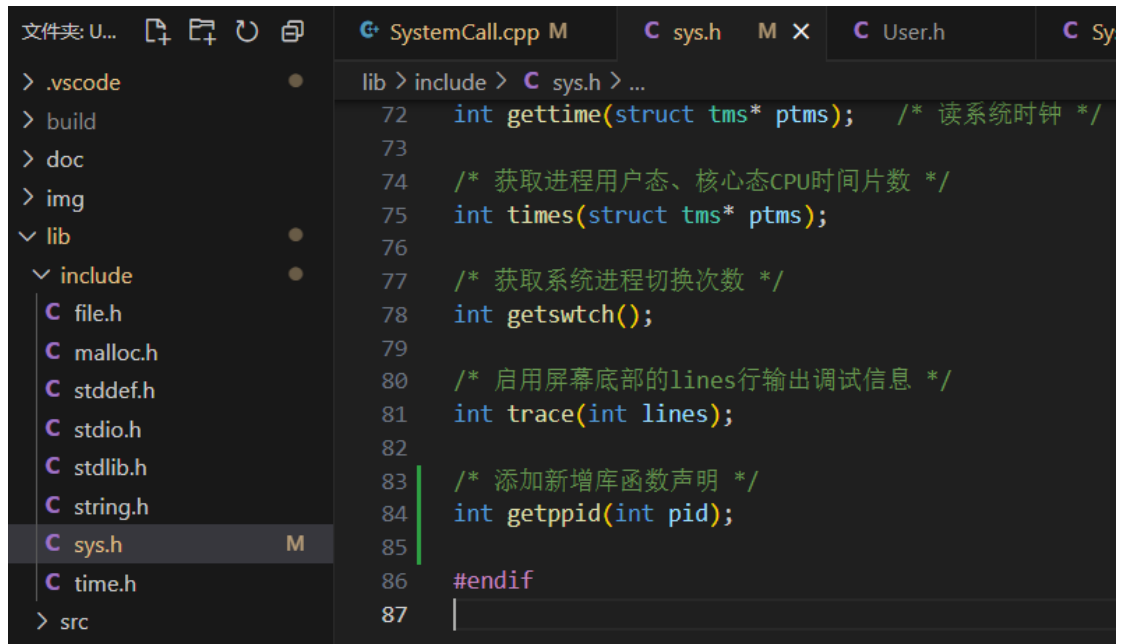
```
src > interrupt > C SystemCall.cpp > Sys_Getppid()
727 /*添加Sys_Getppid的定义*/
728 /* 49 = Sys_Getppid count = 1*/
729 int SystemCall::Sys_Getppid()
730 {
731 /* 通过Kernel::GetProcessManager 函数获取内核的 ProcessManager, 进而找到 ProcessManager 中的process表*/
732 ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
733 /*通过GetUser函数获取当前进程的User结构*/
734 User&u = Kernel::Instance().GetUser();
735
736 int i;
737 /* 进而找到user结构中u_arg[0]中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*/
738 int curpid = (int)u.u_arg[0];
739
740 u.u_arg[User::EAX]=-1;
741 /*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. */
742 for(i=0;i<ProcessManager::NPROC;i++)
743 {
744 if(procMgr.process[i].p_pid == curpid)
745 {
746 u.u_arg[User::EAX] = procMgr.process[i].p_ppid;
747 }
748 }
749 return 0;
750 }
751
752
```

【总结】在 UNIX V6++中添加一个新的系统调用的步骤

- ① 在系统调用处理子程序入口表中添加新的入口，具体操作为：在 SystemCall.cpp 中系统调用子程序入口表 m_SystemEntranceTable 中选择一个空项（赋值为{ 0, &Sys_Nosys }的项），将其修改为{所需参数个数，系统调用处理子程序的入口地址}；
- ② 在 SystemCall.h 文件中添加该系统调用处理子程序的声明
- ③ 在 SystemCall.cpp 中添加 该系统调用子程序的定义

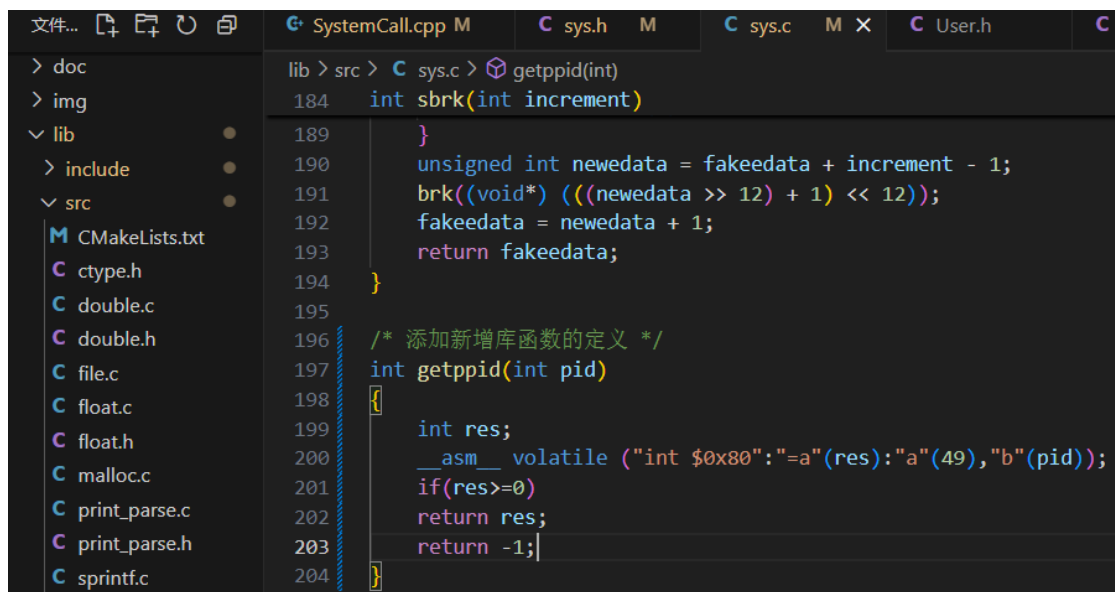
4.2. 为新的系统调用添加对应的库函数

4.2.1 在 SYS.H 文件中添加库函数的声明



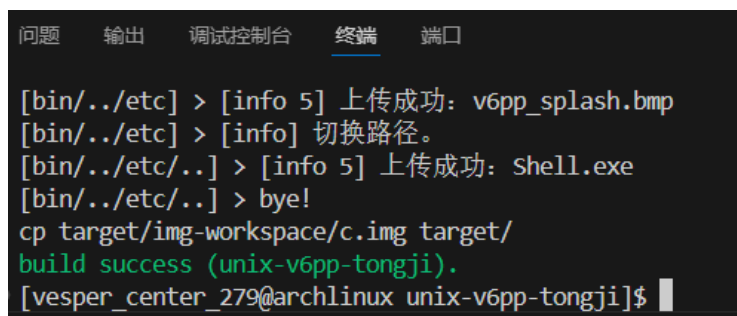
```
lib > include > C sys.h > ...
72 int gettime(struct tms* ptms); /* 读系统时钟 */
73
74 /* 获取进程用户态、核心态CPU时间片数 */
75 int times(struct tms* ptms);
76
77 /* 获取系统进程切换次数 */
78 int getswitch();
79
80 /* 启用屏幕底部的lines行输出调试信息 */
81 int trace(int lines);
82
83 /* 添加新增库函数声明 */
84 int getppid(int pid);
85
86 #endif
87
```

4.2.2. 在 SYS.C 中添加库函数的定义

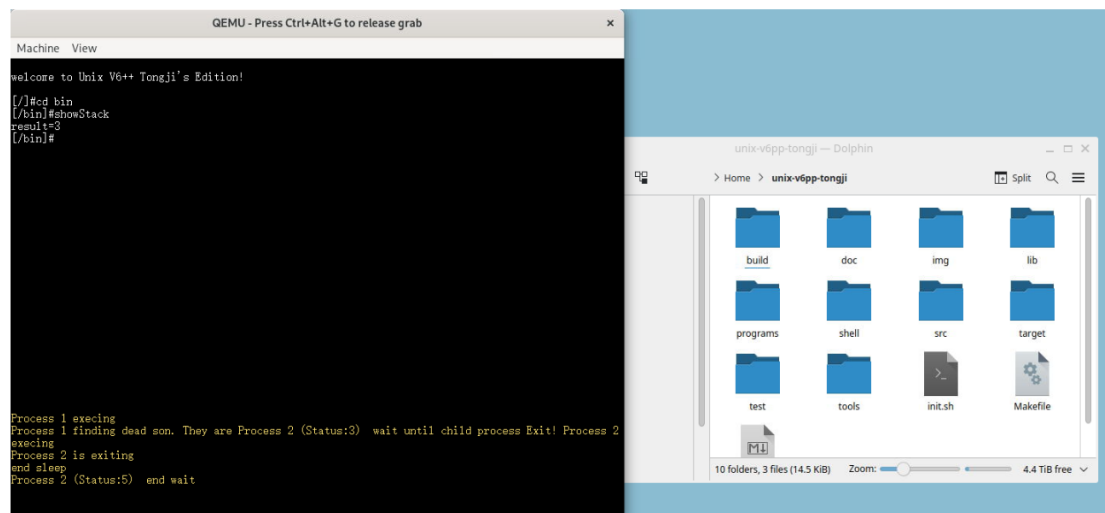
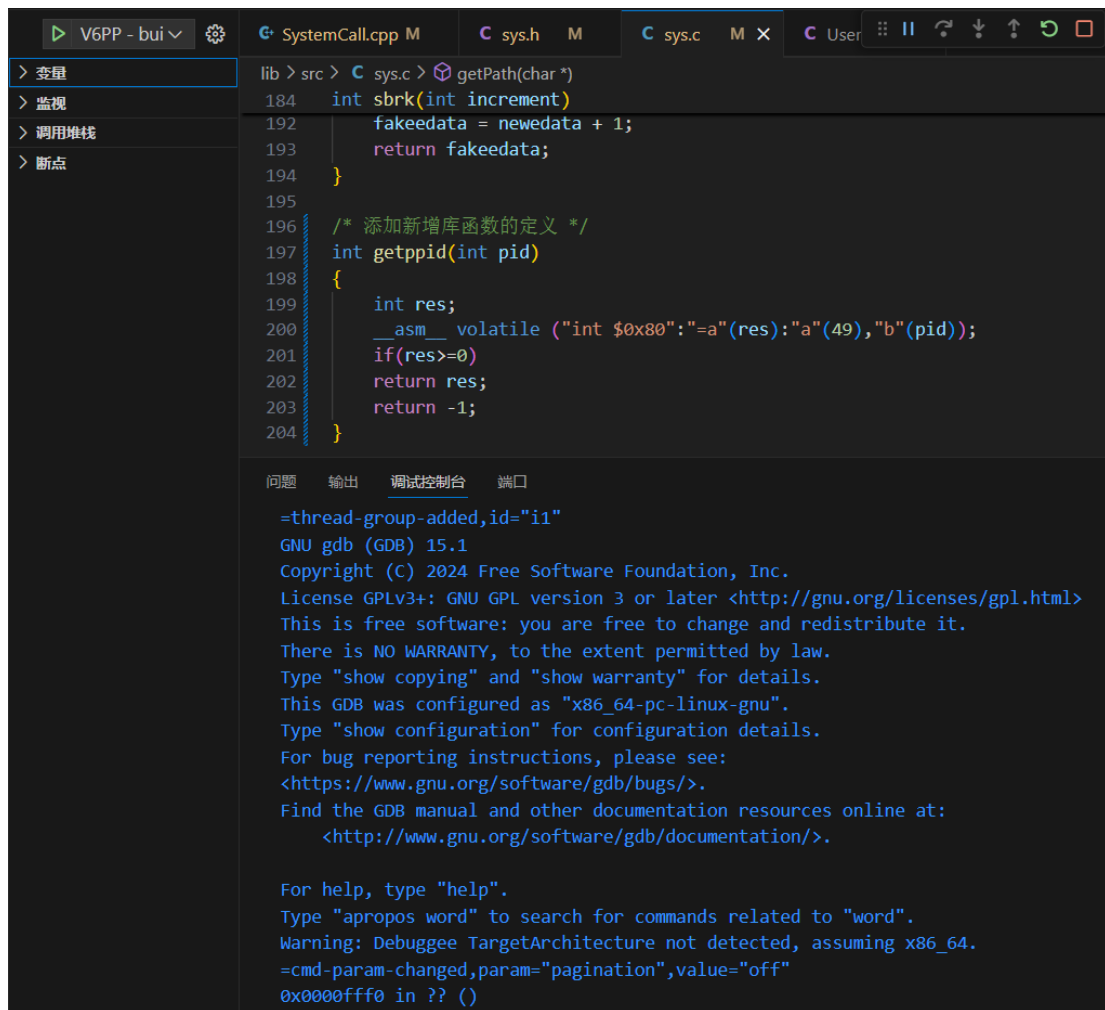


```
lib > src > C sys.c > getppid(int)
184 int sbrk(int increment)
189 {
190     unsigned int newedata = fakeedata + increment - 1;
191     brk((void*) (((newedata >> 12) + 1) << 12));
192     fakeedata = newedata + 1;
193     return fakeedata;
194 }
195
196 /* 添加新增库函数的定义 */
197 int getppid(int pid)
198 {
199     int res;
200     __asm__ volatile ("int $0x80":"=a"(res):"a"(49),"b"(pid));
201     if(res>=0)
202         return res;
203     return -1;
204 }
```

4.2.3 重新编译 UNIX V6++, 输入 make all 指令, 没有问题

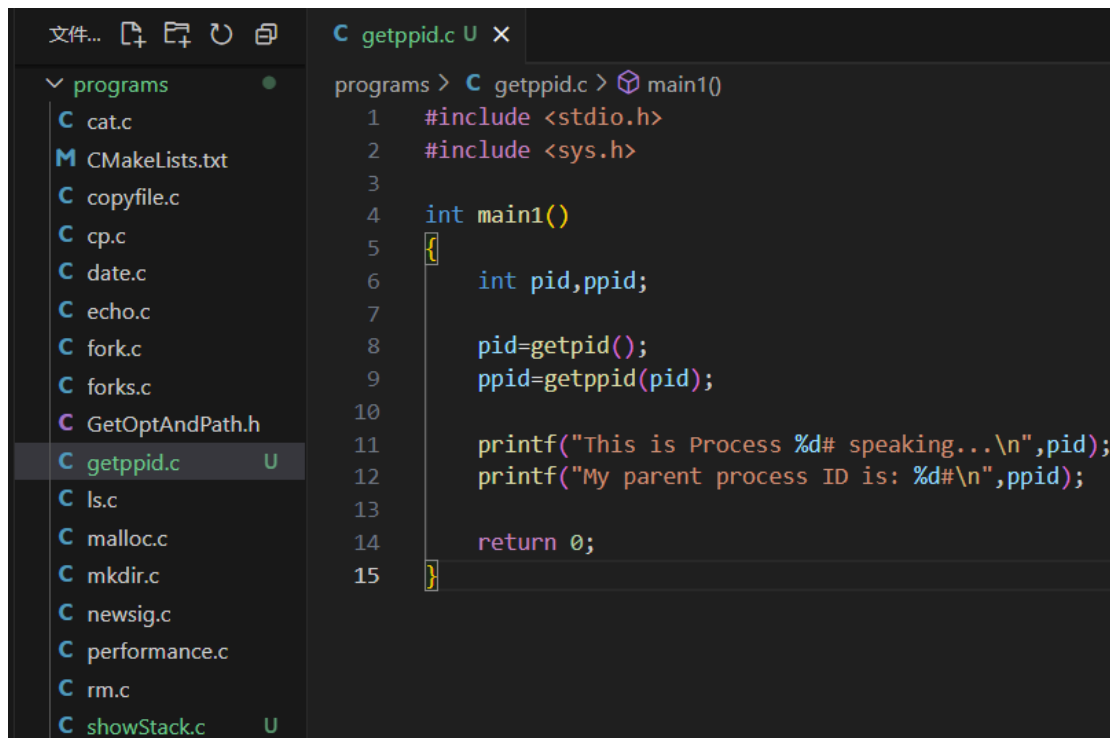


```
问题 输出 调试控制台 终端 端口
[bin/./etc] > [info 5] 上传成功: v6pp_splash.bmp
[bin/./etc] > [info] 切换路径。
[bin/./etc/..] > [info 5] 上传成功: Shell.exe
[bin/./etc/..] > bye!
cp target/img-workspace/c.img target/
build success (unix-v6pp-tongji).
[vesper_center_279@archlinux unix-v6pp-tongji]$
```

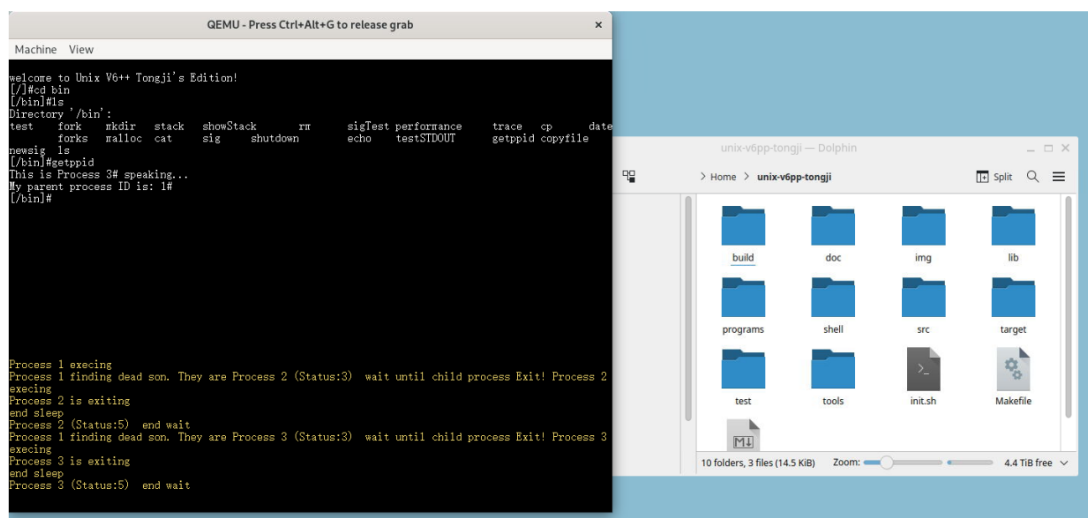


4.3 编写测试程序

4.3.1 编写 getppid.c 文件



4.3.2 在运行模式下启动 UNIX V6++，观察程序的输出结果



可以看到，实验结果显示正确。

【总结】：

- 1、在 UNIX V6++中添加库函数的步骤：
 - ① 在 SYS.H 文件中添加库函数的声明
 - ② 在 sys.c 中添加库函数的定义
- 2、重新编译的步骤：

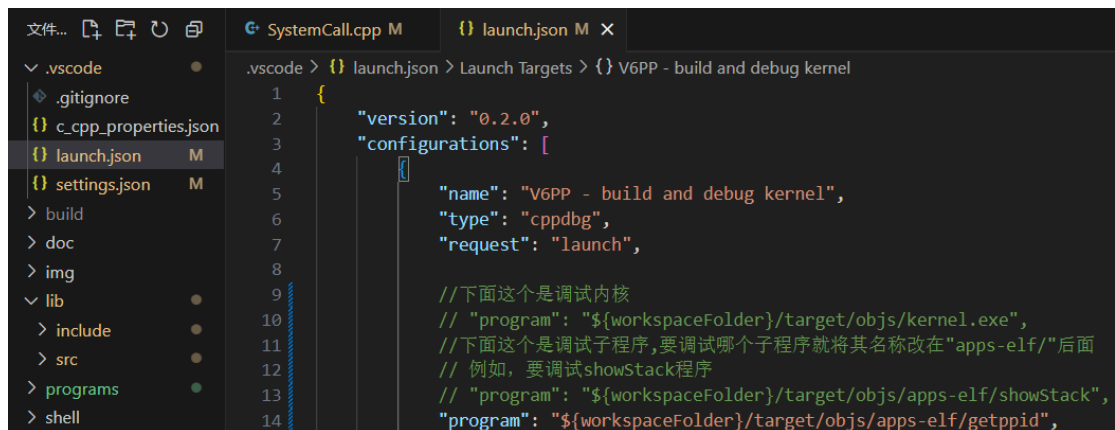
在 UNIX V6 的终端窗口输入“make all”指令

4.4 调试程序

4.4.1. 观察系统调用参数和返回值的传递

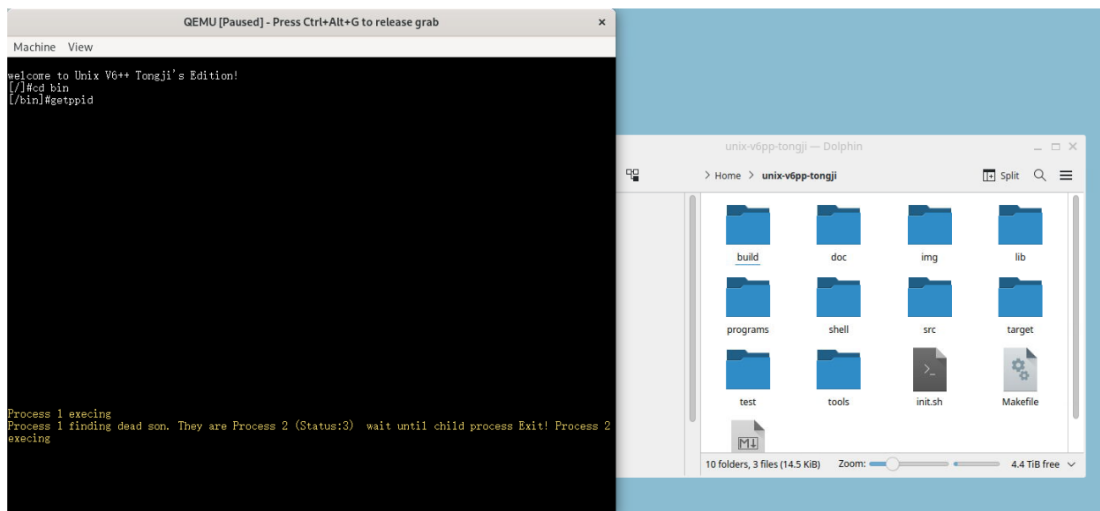
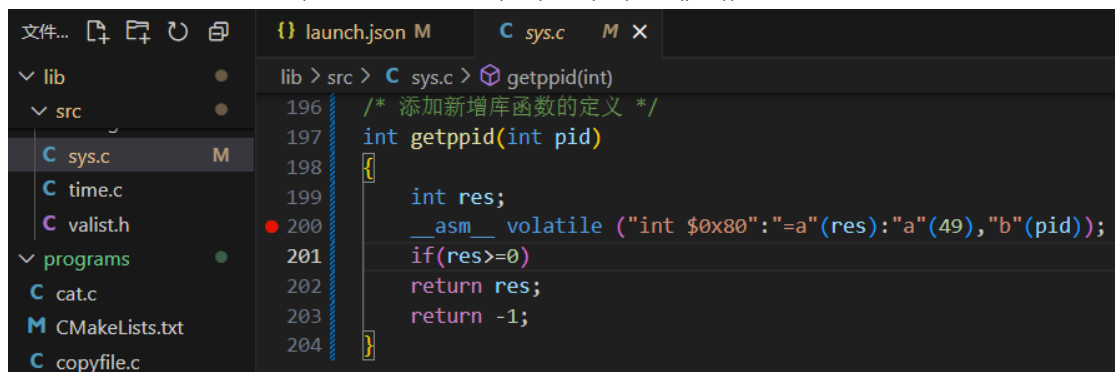
(1) 观察用户态到核心态参数的传递

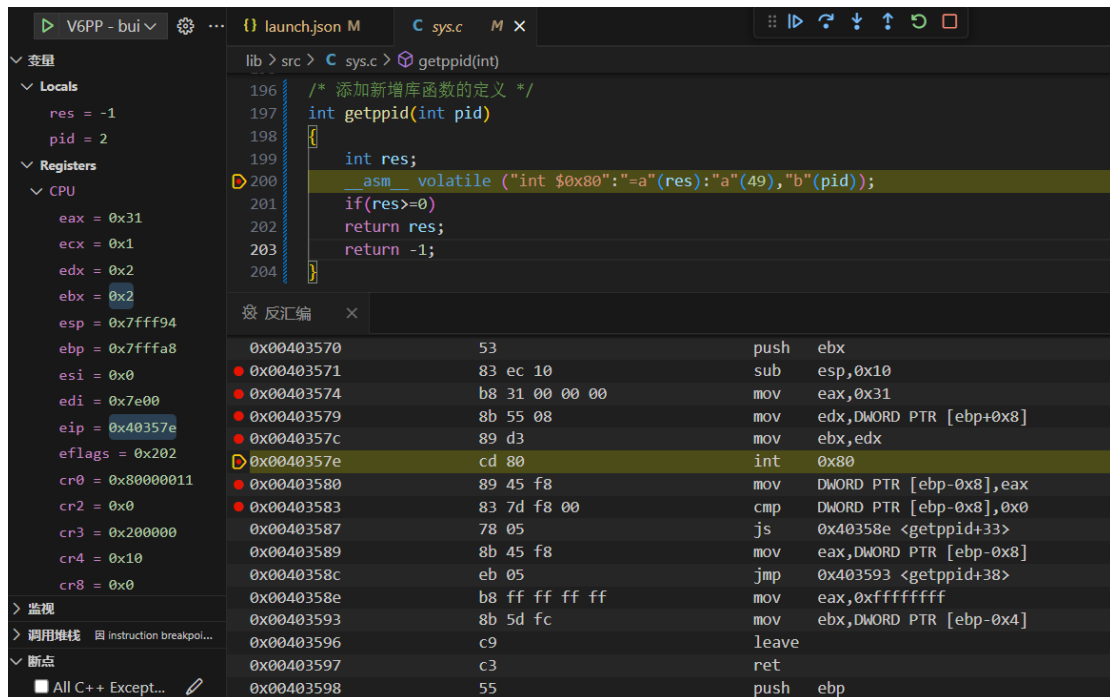
① 将调试目标设为应用程序 getppid



② 将断点设置在库函数 `getppid` 中的语句:

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

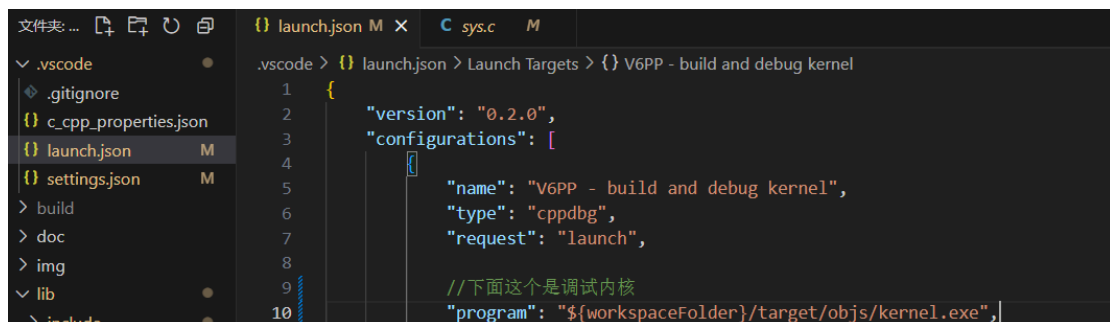




从上面的图片，我们可以看到，当将要执行 int 80 指令时，eax 中为系统调用号 49（16 进制 0x31），ebx 中为参数值 2（现运行进程的 ID 号），eip 的值正好是“int \$0x80”的地址。

（2）观察核心态到用户态的返回值的传递

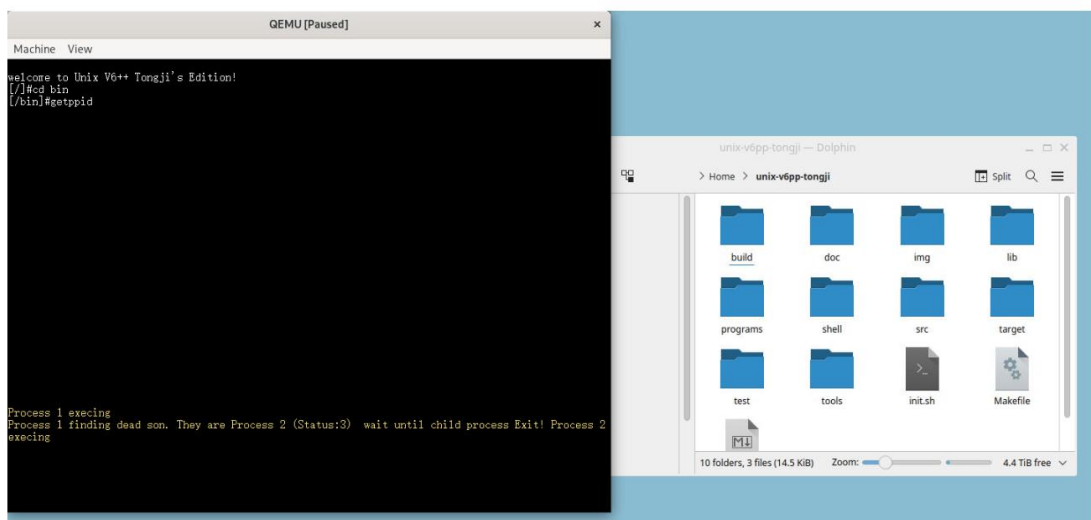
① 将调试目标修改为内核调试



② 修改断点位置

1) 在 Sys_Getppid 函数的“int curpid=(int)u.u_arg[0]”赋值语句处添加断点，并进行调试


```
src > interrupt > SystemCall.cpp > Sys_Getppid()
727  /*添加Sys_Getppid的定义*/
728  /* 49 = Sys_Getppid count = 1*/
729  int SystemCall::Sys_Getppid()
730  {
731
732      /* 通过Kernel::GetProcessManager 函数获取内核的 ProcessManager, 进而找到 ProcessManager 中的process表*/
733      ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
734      /*通过GetUser函数获取当前进程的User结构*/
735      User&u = Kernel::Instance().GetUser();
736
737      int i;
738      /* 进而找到user结构中u_arg[0]中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*/
739      int curpid = (int)u.u_arg[0];
740
741      u.u_ar0[User::EAX]=-1;
742      /*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. */
743      for(i=0;i<ProcessManager::NPROC;i++)
744      {
745          if(procMgr.process[i].p_pid == curpid)
746          {
747              u.u_ar0[User::EAX] = procMgr.process[i].p_ppid;
748          }
749      }
750      return 0;
751  }
```



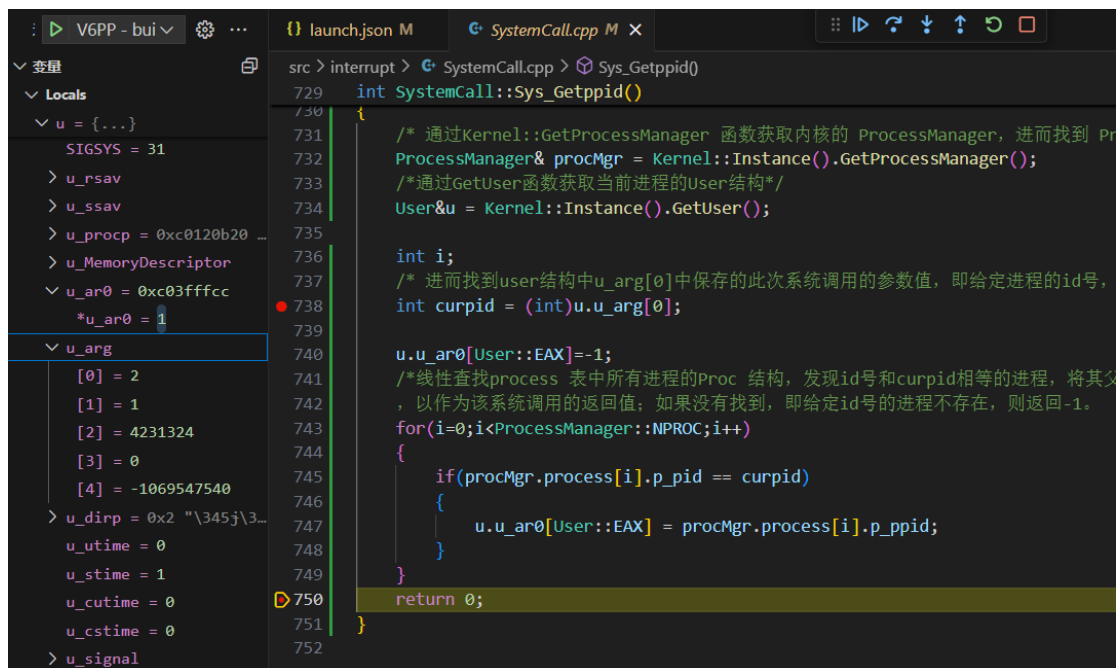
```
src > interrupt > SystemCall.cpp > Sys_Getppid()
729  int SystemCall::Sys_Getppid()
730  {
731
732      /* 通过Kernel::GetProcessManager 函数获取内核的 ProcessManager, 进而找到 ProcessManager 中的process表*/
733      ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
734      /*通过GetUser函数获取当前进程的User结构*/
735      User&u = Kernel::Instance().GetUser();
736
737      int i;
738      /* 进而找到user结构中u_arg[0]中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*/
739      int curpid = (int)u.u_arg[0];
740
741      u.u_ar0[User::EAX]=-1;
742      /*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. */
743      for(i=0;i<ProcessManager::NPROC;i++)
744      {
745          if(procMgr.process[i].p_pid == curpid)
746          {
747              u.u_ar0[User::EAX] = procMgr.process[i].p_ppid;
748          }
749      }
750      return 0;
751  }
```

Debugger Variables:

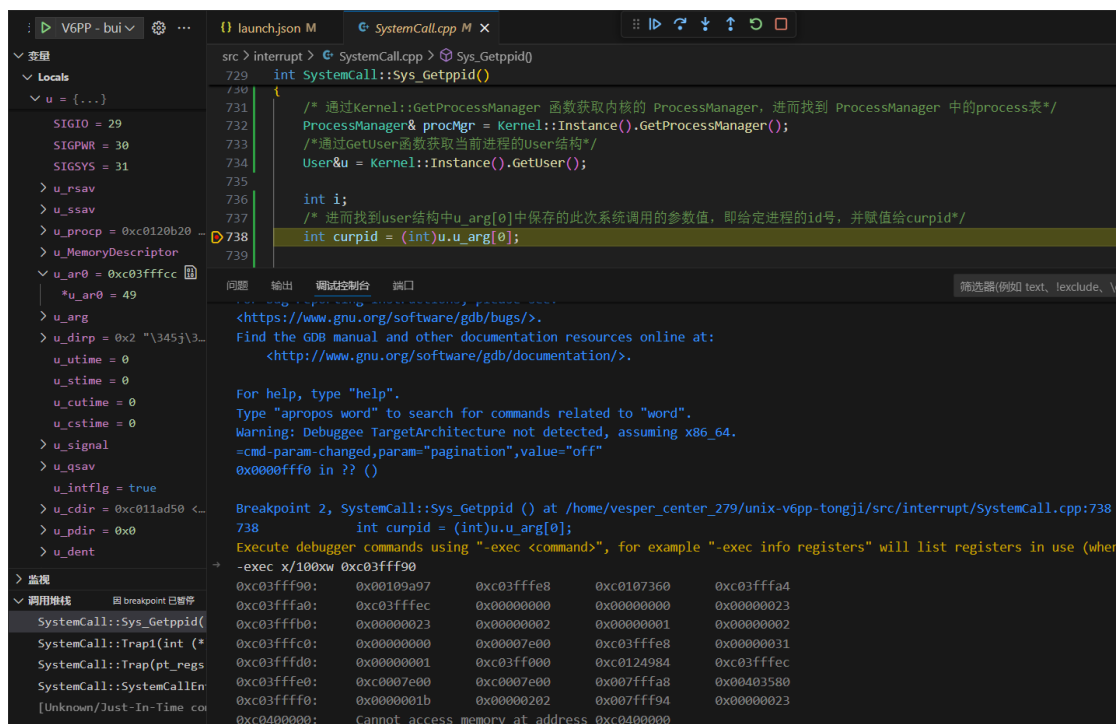
- u = {...}
- u_ar0 = 0xc03ffcc, *u_ar0 = 49
- u_arg
 - [0] = 2
 - [1] = 1
 - [2] = 4231324
 - [3] = 0
 - [4] = -1069547540
- u_dirp = 0x2 "\345j\3...
- u_etime = 0
- u_stime = 0
- u_cutime = 0
- u_cstime = 0
- u_signal
- u_qsav
- u_intflg = true
- u_cdir = 0xc01lad50 <...
- u_pdir = 0x0

当程序停在该断点处时, 系统调用已经开始执行。可以看到, 此时 `u_ar0` 指向的核心栈中保存 `EAX` 单元的值为 49, 说明系统调用号 49 已经通过系统调用的压栈操作由 `EAX` 寄存器带入到进程核心栈, `u_arg[0]`处的值为 2, 说明参数已进入进程的 `User` 结构。

2) 在 `Sys_Getppid` 函数的“`return 0;`”返回语句处添加断点, 并继续调试



当程序执行到该断点时, 可以看到, u_ar0 指向核心栈中保存 EAX 的位置的值变为 1, 即返回值已经被保存到核心栈中用于保存 EAX 寄存器值的单元。



地址	核心栈内容	说明	解释
0xC03FFF90	0x00109A97	局部变量	
0xC03FFF94	0xC03FFFE8	OLD EBP	指向系统调用入口程序栈帧的 EBP
0xC03FFF98	0xC0107360	返回地址	返回系统调用入口程序的地址
0xC03FFF9C	0xC03FFFA4	软件现场指针	指向软件现场 GS
0xC03FFFA0	0xC03FFFE8	硬件现场指针	指向硬件现场 EIP
0xC03FFFA4	0x00000000	GS	
0xC03FFFA8	0x00000000	FS	

0xC03FFFAC	0x00000023	DS	
0xC03FFFB0	0x00000023	ES	
0xC03FFFB4	0x00000002	EBX	系统调用处理程序所需要的参数 pid
0xC03FFFB8	0x00000001	ECX	
0xC03FFFB8	0x00000002	EDX	
0xC03FFFC0	0x00000000	ESI	
0xC03FFFC4	0x00007E00	EDI	
0xC03FFFC8	0xC03FFFE8	EBP	指向系统调用入口程序栈帧的 EBP
0xC03FFFC8	0x00000031	EAX	保存 EAX 单元，值为系统调用号
0xC03FFFD0	0x00000001	局部变量	
0xC03FFFD4	0xC03FF000	局部变量	
0xC03FFFD8	0xC0124984	局部变量	
0xC03FFFD8	0xC03FFFE0	局部变量	
0xC03FFFE0	0xC0007E00	局部变量	
0xC03FFFE4	0xC0007E00	局部变量	
0xC03FFFE8	0x007FFFA8	OLD EBP	指向进行系统调用的用户态进程的 EBP
0xC03FFFE8	0x00403580	EIP	从 INT 0x80 返回后将要执行的下一条指令
0xC03FFFF0	0x0000001B	CS	末 2 位为 1，之前为用户态
0xC03FFFF4	0x00000202	EFLAGS	
0xC03FFFF8	0x007FFF94	ESP	用户栈栈顶指针
0xC03FFFFC	0x00000023	SS	末 2 位为 1，之前为用户态

经检验，该表格与图 8 对比正确