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

```
文件... [ C getppid.c U G SystemCall.cpp M X
                      src > interrupt > G SystemCall.cpp > [0] m_SystemEntranceTable
                            SystemCallTableEntry SystemCall::m_SystemEntranceTable[SYSTEM_CALL_NUM]
> shell
                                 { υ, αργε_Nusys },
                                 { 1, &Sys_Nice },
                                 { 1, &Sys_Sslep },
                                 { 0, &Sys_Sync },
 > dev
                                 { 2, &Sys Kill
 > elf
                                 { 0, &Sys_Getswit},
 > fs
                                 { 1, &Sys_Pwd },
 > include
                                 { 0, &Sys_Nosys },

✓ interrupt

                                { 1, &Sys_Dup },
 DiskInterrupt.cpp
                                { 1, &Sys_Pipe
                                { 1, &Sys_Times },
 G Exception.cpp
                                { 4, &Sys_Profil},
 G KeyboardInterrupt....
                                 { 0, &Sys_Nosys },
  { 1, &Sys_Setgid},
 G TimeInterrupt.cpp
                                 { 0, &Sys_Getgid},
 > kernel
                                 { 2, &Sys_Ssig },
 > libyrosstd
                                 { 1, &Sys_Getppid},
                       66
 > machine
                                 { 0, &Sys_Nosys },
                                     &Sys_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

```
文件夹: UNIX-V6PP-TONGJI [... 🔓 📴 ひ 🗿
                                      G SystemCall.cpp 1 ● C SystemCall.h M X

✓ include

                                                 static int Sys_Profil();
 C MapNode.h
 C MemoryDescriptor.h
 C New.h
 C OpenFileManager.h
 C PageDirectory.h
                                                static int Sys_Setgid();
 C PageManager.h
 C PageTable.h
                                                static int Sys_Getgid();
 C PEParser h
 C ProcessManager.h
                                                 static int Sys_Ssig();
 C Regs.h
 C Simple.h
                                                 /*新增一个49号调用入口位置*/
 C stdint.h
                                                 static int Sys_Getppid();
 C SwapperManager.h
 C TaskStateSegment.h
 C Text.h
 C TimeInterrupt.h
                                                  static SystemCallTableEntry m_SystemEntranceTable[SYSTEM_CALL_NUM]
 C Utility.h
 C Video.h
DiskInterrupt.cpp
 Exception.cpp
G KeyboardInterrupt.cpp
```

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

```
文件... [] [] [] [] C SystemCall.cpp M X C SystemCall.h M
                     727 /*添加Sys_Getppid的定义*/
728 /* 49 = Sys Getppid count
C ProcessManager.h
                            int SystemCall::Sys_Getppid()
C Simple.h
C stdint.h
                               ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
C SwapperManager.h
C Text.h
                               int i; /* 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*/
C Video.h
                                for(i=0;i<ProcessManager::NPROC;i++)</pre>
G DiskInterrupt.cpp
                                    if(procMgr.process[i].p_pid == curpid)
                                        u.u_ar0[User::EAX] = procMgr.process[i].p_ppid;
G SystemCall.cpp M
> kernel
                                return 0;
```

【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

```
文件夹: U.... 📭 📮 ひ 🗿
                                           C sys.h M X
                       C User.h
                                                                         C Sy
> .vscode
                       lib > include > C sys.h > ...
                              int gettime(struct tms* ptms); /* 读系统时钟 */
> build
> doc
                              /* 获取进程用户态、核心态CPU时间片数 */
> img
                              int times(struct tms* ptms);
∨ lib

✓ include

                              /* 获取系统进程切换次数 */
 C file.h
                              int getswtch();
 C malloc.h
                              /* 启用屏幕底部的lines行输出调试信息 */
 C stddef.h
                              int trace(int lines);
 C stdio.h
 C stdlib.h
                         83
                              /* 添加新增库函数声明 */
 C string.h
                              int getppid(int pid);
 C time.h
                              #endif
                         87
> src
```

4.2.2. 在 SYS.C 中添加库函数的定义

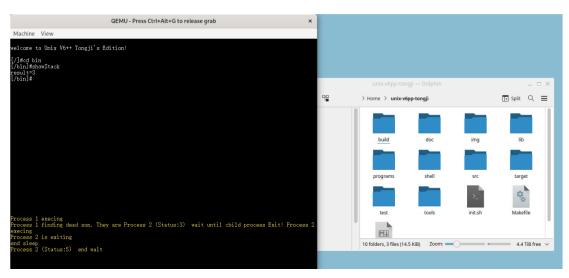
```
> doc
                      lib > src > C sys.c > ♦ getppid(int)
> img
                      184 int sbrk(int increment)
∨ lib
                                unsigned int newedata = fakeedata + increment - 1;
 > include
                                brk((void*) (((newedata >> 12) + 1) << 12));</pre>
∨ src
                                 fakeedata = newedata + 1;
 M CMakeLists.txt
                                return fakeedata;
 C ctype.h
 C double.c
 C double.h
 C file.c
                            int getppid(int pid)
 C float.c
                                 int res;
 C float.h
                                  _asm__ volatile ("int $0x80":"=a"(res):"a"(49),"b"(pid));
 C malloc.c
                                 if(res>=0)
  C print_parse.c
                                 return res;
  C print_parse.h
                      203
                                 return -1;
 C sprintf.c
```

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]$
```

```
C sys.c M X C User ∷ II 🗘 🐈 🗘 Ɗ 🔲
  ▶ V6PP - bui ∨ ∰
                       C sys.h M
〉变量
                       lib > src > \mathbf{C} sys.c > \mathbf{\mathfrak{Q}} getPath(char *)
                              int sbrk(int increment)
> 监视
                                  fakeedata = newedata + 1;
> 调用堆栈
                                  return fakeedata;
〉断点
                              int getppid(int pid)
                                    _asm__ volatile ("int $0x80":"=a"(res):"a"(49),"b"(pid));
                                  if(res>=0)
                                   调试控制台
                         =thread-group-added,id="i1"
                         This GDB was configured as "x86_64-pc-linux-gnu".
                         For bug reporting instructions, please see:
                         Find the GDB manual and other documentation resources online at:
                         Warning: Debuggee TargetArchitecture not detected, assuming x86_64.
```



4.3 编写测试程序

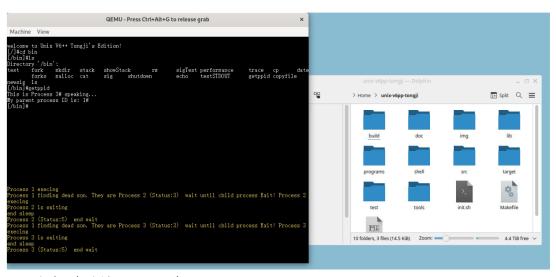
4.3.1 编写 getppid.c 文件

```
文件... 日 日 ひ 日
                       C getppid.c U X
                       programs > C getppid.c > 分 main1()

✓ programs

                               #include <stdio.h>
C cat.c
M CMakeLists.txt
 C copyfile.c
                               int main1()
 C cp.c
 C date.c
                                   int pid,ppid;
 C echo.c
                                   pid=getpid();
 C fork.c
                                   ppid=getppid(pid);
 C forks.c
 C GetOptAndPath.h
                                   printf("This is Process %d# speaking...\n",pid);
 C getppid.c
                                   printf("My parent process ID is: %d#\n",ppid);
C Is.c
 C malloc.c
                                   return 0;
 C mkdir.c
                         15
 C newsig.c
 C performance.c
C rm.c
C showStack.c U
```

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

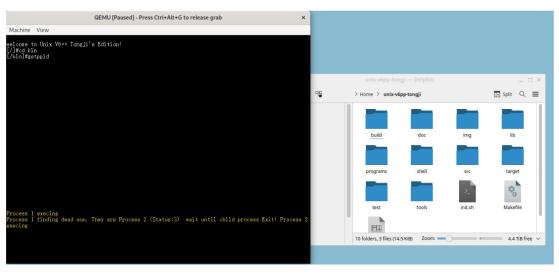
4.4.1. 观察系统调用参数和返回值的传递

- (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));"

```
文件… [4 日 ひ 日
                     {} launch.json M
                                       C sys.c M X
∨ lib
                      lib > src > C sys.c > ۞ getppid(int)
                           /* 添加新增库函数的定义 */
                             int getppid(int pid)
 C time.c
                                 int res;
                                 __asm__ volatile ("int $0x80":"=a"(res):"a"(49),"b"(pid));
 C valist.h
                                 if(res>=0)
                      201
✓ programs
                                 return res;
C cat.c
                                 return -1;
M CMakeLists.txt
C copyfile.c
```



```
□ ୯ ‡ ‡ 5 □
 V6PP - bui ∨ ∰ ··· {} launch.json M
/ 变量
                        lib > src > C sys.c > 分 getppid(int)
∨ Locals
                               int getppid(int pid)

∨ Registers

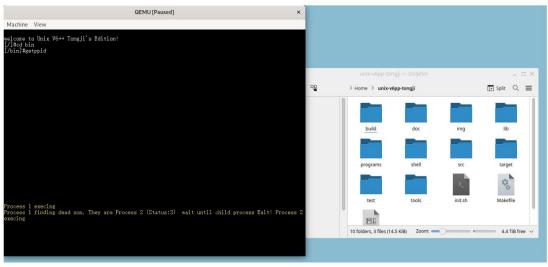
                                    __asm__ volatile ("int $0x80":"=a"(res):"a"(49),"b"(pid)); if(res>=0)
                       200
   eax = 0x31
   edx = 0x2
   ebx = 0x2
                        嵏 反汇编
                        0x00403570
                      0x004035710x004035740x004035790x0040357c
                                                                                          esp,0x10
   esi = 0x0
                                                   b8 31 00 00 00
                                                                                          eax,0x31
                                                                                          edx,DWORD PTR [ebp+0x8]
   eip = 0x40357e
                                                   89 d3
    eflags = 0x202
                      ▶ 0x0040357e
   cr0 = 0x80000011
                      0x004035800x00403583
                                                                                          DWORD PTR [ebp-0x8],eax
                                                   89 45 f8
                                                   83 7d f8 00
                                                                                   стр
                                                                                          DWORD PTR [ebp-0x8],0x0
                                                   78 05
                                                                                          0x40358e <getppid+33>
   cr3 = 0x200000
                                                                                          eax,DWORD PTR [ebp-0x8]
                        0x00403589
                                                   8b 45 f8
                         0x0040358c
                                                   eb 05
                                                                                          0x403593 <getppid+38>
                                                                                   jmp
   cr8 = 0x0
                                                   b8 ff ff ff
                                                                                          eax,0xffffffff
                         0x0040358e
                                                                                   mov
监视
                         0x00403593
                                                   8b 5d fc
                                                                                          ebx, DWORD PTR [ebp-0x4]
                                                                                   mov
调用堆栈 图 instruction breakpoi.
                         0x00403596
                         0x00403597
 0x00403598
```

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

```
✓ .vscode
                        .vscode > {} launch.json > Launch Targets > {} V6PP - build and debug kernel
 gitignore
                                   "version": "0.2.0",
{} c_cpp_properties.json
                                   "configurations": [
{} launch.json
{} settings.json
                                           "name": "V6PP - build and debug kernel",
"type": "cppdbg",
> build
                                           "request": "launch",
> doc
> img
∨ lib
                                           "program": "${workspaceFolder}/target/objs/kernel.exe",
```

- ② 修改端点位置
- 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



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。

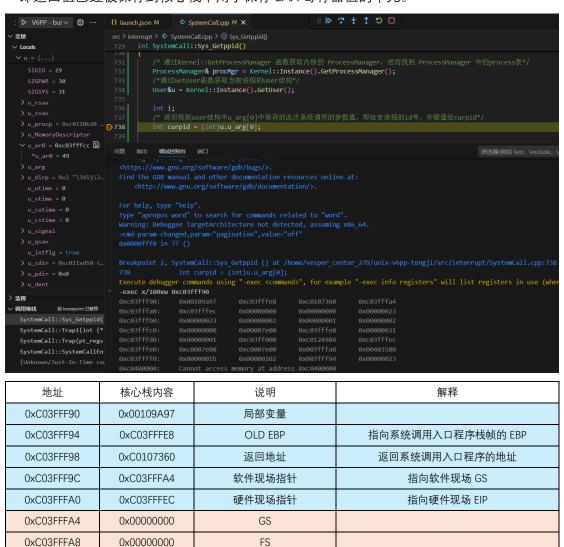
2) 在 Sys_Getppid 函数的"return 0;"返回语句处添加断点,并继续调试

```
□ € ‡ ‡ 5 □
 : ▷ V6PP - bui ∨ 🝪 ··· {} launch.json M
                                            ∨ 変量
                      f src > interrupt > € SystemCall.cpp > 分 Sys_Getppid()
                                  int SystemCall::Sys_Getppid()

∨ Locals

                                      ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
                                      User&u = Kernel::Instance().GetUser();
  > u MemoryDescriptor
  ∨ u_ar0 = 0xc03fffcc
                                      int curpid = (int)u.u_arg[0];
                                      u.u_ar0[User::EAX]=-1;
                                      /*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:,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。
                                      for(i=0;i<ProcessManager::NPROC;i++)</pre>
                                          if(procMgr.process[i].p_pid == curpid)
   > u_dirp = 0x2 "\345j\3...
                                              u.u_ar0[User::EAX] = procMgr.process[i].p_ppid;
                         > 750
                                      return 0;
```

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。



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	
0xC03FFFBC	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
0xC03FFFCC	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	
0xC03FFFDC	0xC03FFFEC	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
0xC03FFFEC	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