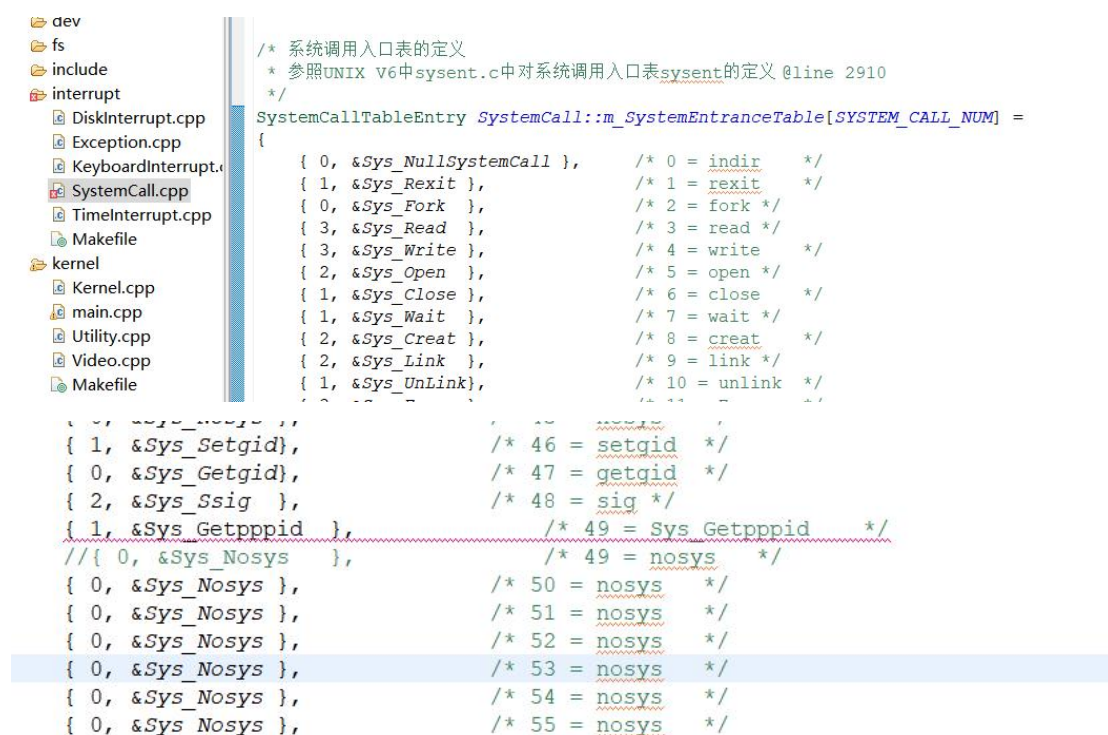


P04: UNIX V6++中添加新的系统调用

2152118 史君宝

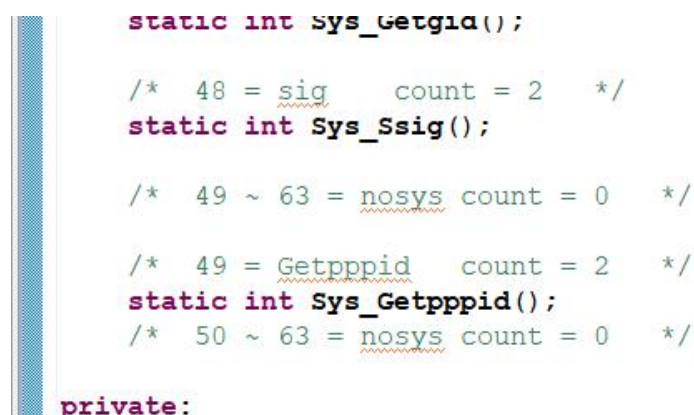
任务一、完成实验 4.1，截图说明操作过程。

(1) 在系统调用子程序入口表中添加新的入口



```
/* 系统调用入口表的定义
 * 参照UNIX V6中sysent.c中对系统调用入口表sysent的定义 @line 2910
 */
SystemCallTableEntry SystemCall::m_SystemEntranceTable[SYSTEM_CALL_NUM] =
{
    { 0, &Sys_NullSystemCall },          /* 0 = indir */
    { 1, &Sys_Rexit },                    /* 1 = rexit */
    { 0, &Sys_Fork },                      /* 2 = fork */
    { 3, &Sys_Read },                      /* 3 = read */
    { 3, &Sys_Write },                     /* 4 = write */
    { 2, &Sys_Open },                      /* 5 = open */
    { 1, &Sys_Close },                     /* 6 = close */
    { 1, &Sys_Wait },                      /* 7 = wait */
    { 2, &Sys_Creat },                     /* 8 = creat */
    { 2, &Sys_Link },                      /* 9 = link */
    { 1, &Sys_UnLink },                    /* 10 = unlink */
    { 1, &Sys_Setgid },                    /* 46 = setgid */
    { 0, &Sys_Getgid },                    /* 47 = getgid */
    { 2, &Sys_Ssig },                      /* 48 = sig */
    { 1, &Sys_Getppid },                   /* 49 = Sys Getppid */
    // { 0, &Sys_Nosys },                    /* 49 = nosys */
    { 0, &Sys_Nosys },                     /* 50 = nosys */
    { 0, &Sys_Nosys },                     /* 51 = nosys */
    { 0, &Sys_Nosys },                     /* 52 = nosys */
    { 0, &Sys_Nosys },                     /* 53 = nosys */
    { 0, &Sys_Nosys },                     /* 54 = nosys */
    { 0, &Sys_Nosys },                     /* 55 = nosys */
}
```

(2) 在 SystemCall 类中添加系统调用处理子程序的定义



```
static int Sys_Getgid();

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

/* 49 ~ 63 = nosys count = 0 */

/* 49 = Getppid count = 2 */
static int Sys_Getppid();
/* 50 ~ 63 = nosys count = 0 */

private:
```

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

```

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

    int i;
    int curpid = (int)u.u_arg[0];

    u.u_ar0[User::EAX] = -1;

    for(int 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; /* GCC likes it ! */
}

```

(4) 总结具体步骤:

添加系统调用的具体步骤:

步骤一: 在系统调用子程序入口表中添加新的入口。

步骤二: 在 SystemCall 类中添加系统调用处理子程序的定义。

步骤三: 在 SystemCall.cpp 中添加 Sys_Getppid 的具体定义。

任务二、完成实验 4.2，掌握在 UNIX V6++中添加库函数的方法。

(1) 在 sys.h 文件中添加库函数的声明

```

lib
├── include
│   ├── file.h
│   ├── malloc.h
│   ├── stddef.h
│   ├── stdio.h
│   ├── stdlib.h
│   ├── string.h
│   └── sys.h
├── objs
├── src
│   ├── ctype.h
│   ├── double.c
│   ├── ...

```

```

unsigned int getgid();
unsigned int getuid();
int setgid(short gid);
int setuid(short uid);
int getppid(int pid);
int gettimeofday(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);

```

(2) 在 sys.c 中添加库函数的定义

```
int setuid(short uid)
{
    int res;
    __asm__ volatile ( "int $0x80": "=a"(res): "a"(23), "b"(uid) );
    if ( res >= 0 )
        return res;
    return -1;
}

int getppid(int pid)
{
    int res;
    __asm__ volatile ( "int $0x80": "=a"(res): "a"(49), "b"(pid) );
    if ( res >= 0 )
        return res;
    return -1;
}
```

(3) 编译程序:

```
CDT Build Console [oos]
copy ..\targets\objs\boot.bin ..\tools\MakeImage\bin\Debug\boot.bin
已复制    1 个文件。
copy ..\targets\objs\kernel.bin ..\tools\MakeImage\bin\Debug\kernel.bin
已复制    1 个文件。
copy ..\targets\img\c.img ..\tools\MakeImage\bin\Debug\c.img
已复制    1 个文件。
cd ..\tools\MakeImage\bin\Debug && build.exe c.img boot.bin kernel.bin programs
copy ..\tools\MakeImage\bin\Debug\c.img "..\targets\UNIXV6++"\c.img
已复制    1 个文件。

**** Build Finished ****
```

成功编译。

(4) 总结具体步骤:

添加库函数的具体步骤:

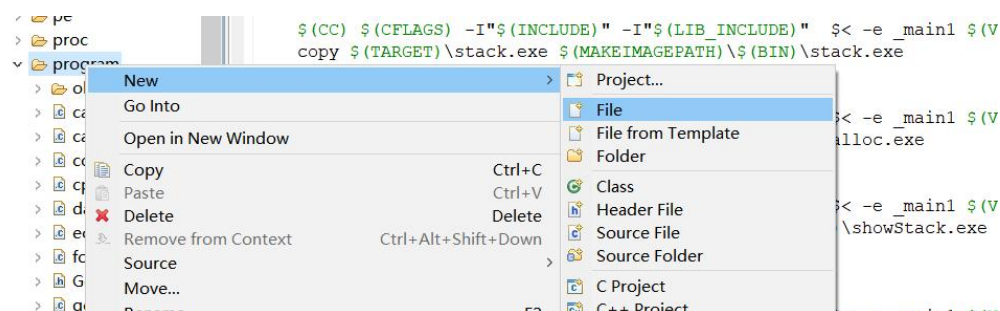
步骤一: 在 sys.h 文件中添加库函数的声明。

步骤二: 在 sys.c 中添加库函数的定义。

步骤三: 编译程序。

任务三、完成实验 4.3-4.4, 编写测试程序。

(1) 创建测试程序:



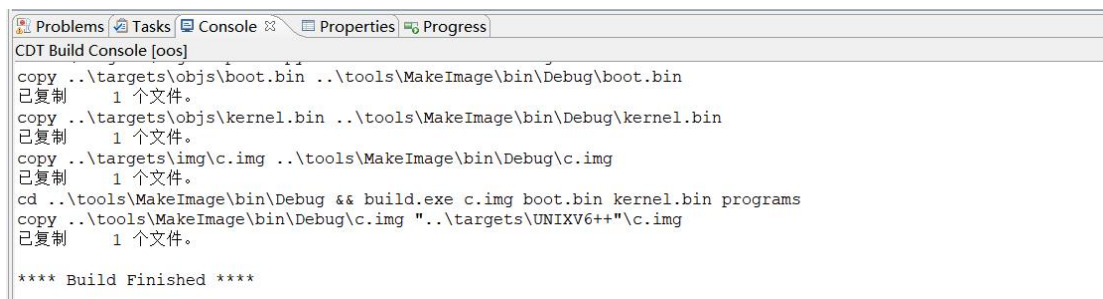
(2) 修改编译使用的 Makefile 文件:

```
$(TARGET)\sigTest.exe \
$(TARGET)\stack.exe \
$(TARGET)\malloc.exe\
$(TARGET)\showStack.exe\
$(TARGET)\getppid.exe

$(TARGET)\showStack.exe :      showStack.c
$(CC) $(CFLAGS) -I"$(INCLUDE)" -I"$(LIB_INCLUDE)" $< -e _main1 $(V6++LIB) -o $@
copy $(TARGET)\showStack.exe $(MAKEIMAGEPATH)\$(BIN)\showStack.exe

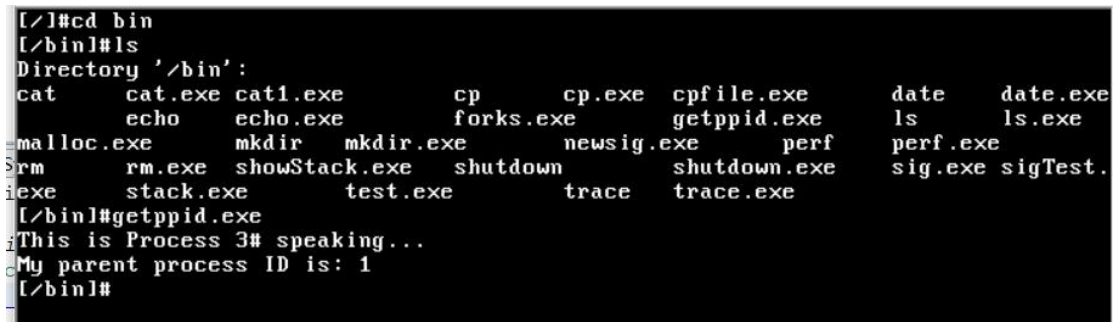
$(TARGET)\getppid.exe :      getppid.c
$(CC) $(CFLAGS) -I"$(INCLUDE)" -I"$(LIB_INCLUDE)" $< -e _main1 $(V6++LIB) -o $@
copy $(TARGET)\getppid.exe $(MAKEIMAGEPATH)\$(BIN)\getppid.exe
```

(3) 编译程序:



```
Problems Tasks Console Properties Progress
CDT Build Console [oos]
copy ..\targets\objs\boot.bin ..\tools\MakeImage\bin\Debug\boot.bin
已复制 1 个文件。
copy ..\targets\objs\kernel.bin ..\tools\MakeImage\bin\Debug\kernel.bin
已复制 1 个文件。
copy ..\targets\img\c.img ..\tools\MakeImage\bin\Debug\c.img
已复制 1 个文件。
cd ..\tools\MakeImage\bin\Debug && build.exe c.img boot.bin kernel.bin programs
copy ..\tools\MakeImage\bin\Debug\c.img "..\targets\UNIXV6++"\c.img
已复制 1 个文件。
**** Build Finished ****
```

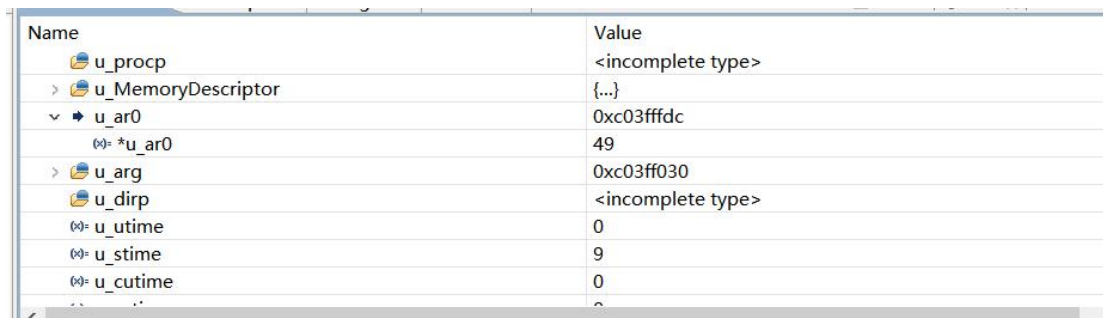
(4) 调试程序观察结果:



```
[/]\#cd bin
[/bin]#ls
Directory '/bin':
cat      cat.exe  cat1.exe      cp      cp.exe  cpfile.exe  date      date.exe
      echo      echo.exe      forks.exe  getppid.exe  ls      ls.exe
malloc.exe  mkdir  mkdir.exe  newsig.exe  perf      perf.exe
rm      rm.exe  showStack.exe  shutdown  shutdown.exe  sig.exe  sigTest.
exe      stack.exe  test.exe      trace      trace.exe
[/bin]#getppid.exe
This is Process 3# speaking...
My parent process ID is: 1
[/bin]#
```

(5) 设置断点, 具体调试观察现象。

执行到断点: `int curpid = (int)u.u_arg[0];`



Name	Value
u_procp	<incomplete type>
u_MemoryDescriptor	{...}
u_ar0	0xc03fffdc
*u_ar0	49
u_arg	0xc03ff030
u_dirp	<incomplete type>
u_utime	0
u_stime	9
u_cutime	0

0XC03FFFD0 <Hex> 0XC03FFFD0 : 0XC03FFFD0 <Hex Integer> + New Renderings...					
Address	0 - 3	4 - 7	8 - B	C - F	
C03FFFD0	000E0000	0000FFAC	C03FFFE8	00000031	
C03FFFE0	00000016	C03FFFE8	007FFFB8	0040141C	
C03FFFF0	0000001B	00000216	007FFAC	00000023	
C0400000	000E0000	0000FFAC	C03FFFE8	00000031	
C0400010	000E0000	0000FFAC	C03FFFE8	00000031	
C0400020	00000016	C03FFFE8	007FFFB8	0040141C	

执行到断点：Getppid 函数的最后一句。

name	value
u_MemoryDescriptor	{...}
u_ar0	0xc03fffdc
*u_ar0	1
u_arg	0xc03ff030
u_dirp	<incomplete type>
u_utime	0
u_stime	9
u_cutime	0
u_cstime	0

0XC03FFFD0 <Hex> 0XC03FFFD0 : 0XC03FFFD0 <Hex Integer> + New Renderings...					
Address	0 - 3	4 - 7	8 - B	C - F	
C03FFFD0	000E0000	0000FFAC	C03FFFE8	00000001	
C03FFFE0	00000016	C03FFFE8	007FFFB8	0040141C	
C03FFFF0	0000001B	00000216	007FFAC	00000023	
C0400000	000E0000	0000FFAC	C03FFFE8	00000001	
C0400010	000E0000	0000FFAC	C03FFFE8	00000001	
C0400020	00000016	C03FFFE8	007FFFB8	0040141C	

	EBP	0xC03FFFE8
0xC03FFFD0	EAX	49
		0x00000016
		0xC03FFFE8
0xC03FFFE8		0xC07FFFE8
0xC03FFFE8	EIP	0x0040141C
	CS	0x0000001B
	EFLAGS	0x00000216
	ESP	0x007FFAC
0xC03FFFFC	SS	0x00000023

可以还原出核心栈应该为上面。

任务四、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。

	EDI		
	EBP	0xC03FFFE8	
0xC03FFFD0	EAX	49	
		0x0000001D	
		0xC03FFFE0	
0xC03FFFE8		0x007FFFB8	
0xC03FFFE4	EIP	0x00401400	
	CS	0x0000001B	后两位为 11
	EFLAGS	0x00000202	
	ESP	0x007FFFB0	
0xC03FFFFC	SS	0x00000023	

图 10: 系统调用时的核心栈

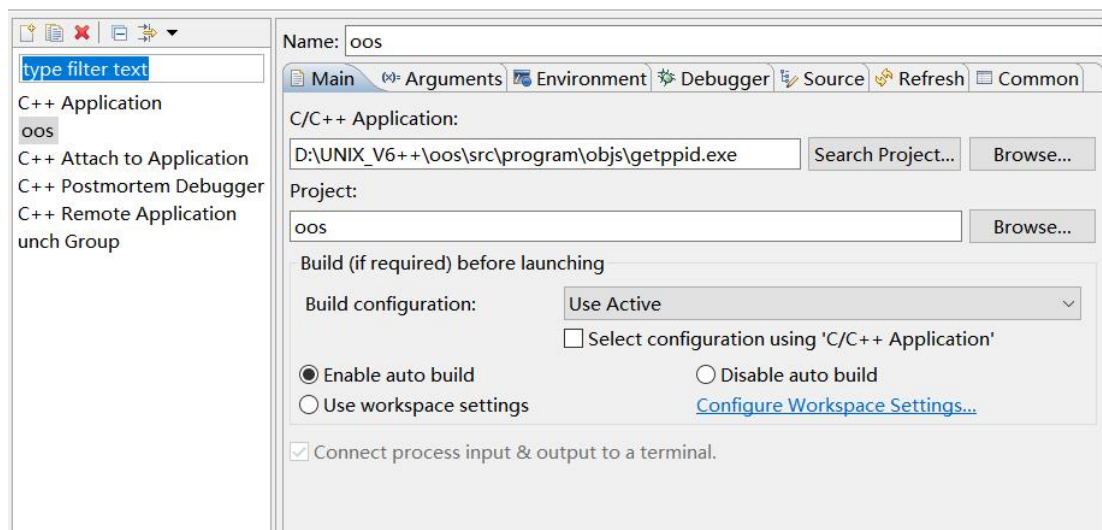
ebp 值是 getpid 函数栈帧的栈基址。

esp 为指向 getpid 函数栈帧的栈顶。

eip 寄存器的值为是下一指令的地址，执行现场保护后存入核心栈。

在 getpid 函数的

`_asm _volatile("int$0x80": "=a"(res): "a"(49), "b"(pid));` 语句处设置断点，修改调试对象和调试入口。



```
int getppid(int pid)
{
    int res;
    __asm__ volatile ( "int $0x80":"=a"(res):"a"(49),"b"(pid) );
    if ( res >= 0 )
        return res;
    return -1;
}
```

Address	Disassembly
00401412:	mov \$0x31,%eax
00401417:	mov 0x8(%ebp),%ebx
0040141a:	int \$0x80
0040141c:	mov %eax,-0x8(%ebp)
152	if (res >= 0)
0040141f:	cmpl \$0x0,-0x8(%ebp)
00401423:	js 0x40142d <getppid+34>
153	return res;
00401425:	mov -0x8(%ebp),%eax
00401428:	mov %eax,-0xc(%ebp)
0040142b:	jmp 0x401434 <getppid+41>
154	return -1;
0040142d:	movl \$0xffffffff,-0xc(%ebp)
155	}
00401434:	mov -0xc(%ebp),%eax
00401437:	add \$0x8,%esp

我们通过查看在 getppid 具体程序的执行过程时的 memory 和 disassembly 可以发现，

第一个黄色地址单元中，esp 的值为 0x007FFFAC，是执行系统调用前的用户栈的栈顶位置。

第二个黄色单元中，eip 的值为 0x0040141C，表示执行完 int \$0x80 指令后的下一条指令的地址。

第三个黄色单元中，0xC03FFFE8 是指向 getppid 函数栈帧的 ebp，即当前栈帧的基址指针。

第四个黄色单元中，ebp 的值为 0xC07FFFE8，表示指向 49 号系统调用入口程序的栈帧的地址。