实验三、为 Unix V6++系统添加新的系统调用

同济大学计算机系操作系统　　国豪 2023 计算机拔尖班 2023　　2025/11/10

一、任务：

1、为 Unix V6++添加 49#系统调用 get_ppid，获得父进程的 pid 号

2、为 Unix V6++添加 50#系统调用 get_pids，获得自己 和 父进程的 pid 号

3、为 Unix V6++添加 51#系统调用 get_proc，获得进程 Process 结构和 User 结构中的多个字段，比如：

- 虚空间：代码段起始地址，代码段长度，数据段起始地址，数据段长度，堆栈段长度
- 物理空间：代码段起始地址，可交换部分起始地址

4、以最后一小题为背景，评论 CISC、RISC 架构的优势和不足

二、实验指导　Unix V6++系统中执行系统调用所需的函数

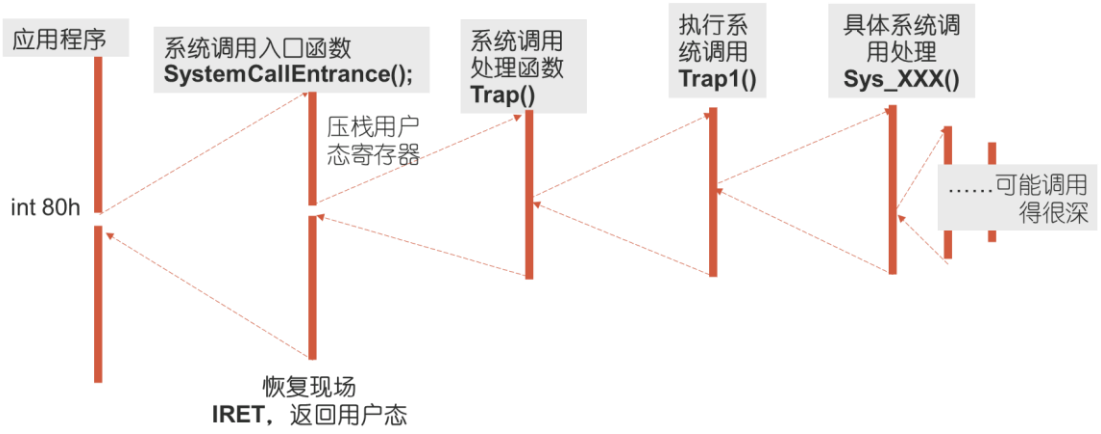


图 1

三、Unix V6++系统调用的实现　&　添加一个新的系统调用

1、分配一个中断号，用来实现系统调用　　**Unix V6++用的是 0x80h**

将系统调用入口函数的起始地址(&SystemCallEntrance)登记在 IDT 表**下标为 0x80h 的陷入门中。**

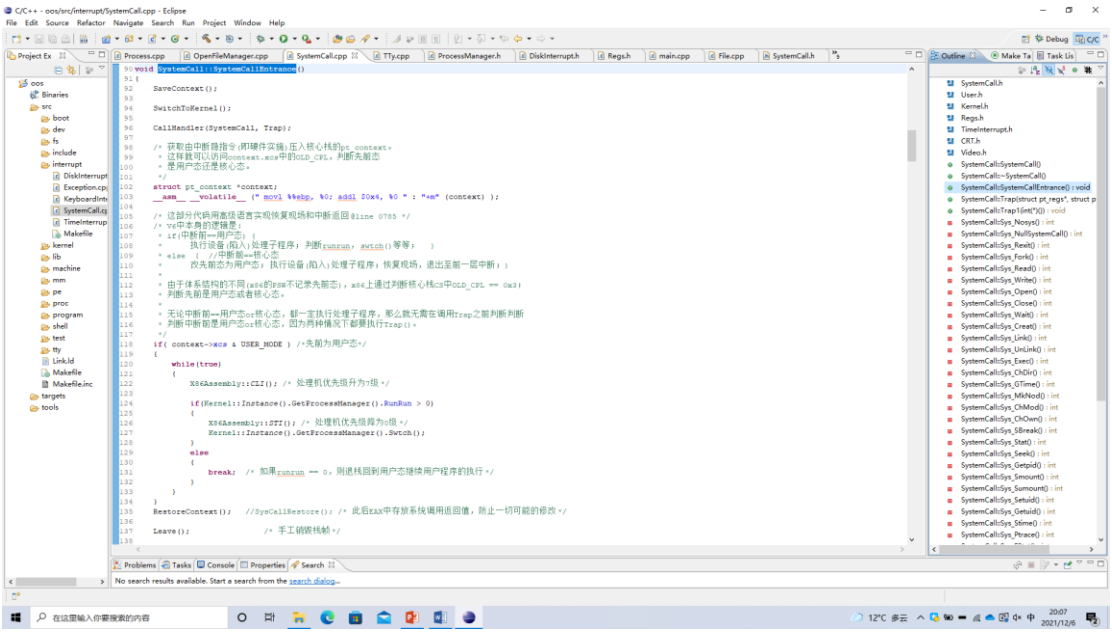


图 2

2、内核设置系统调用表 **m_SystemEntranceTable**。这是一个静态数组，内核支持多少个系统调用，这个数组就有多少项。

```
class SystemCall
{
static const unsigned int SYSTEM_CALL_NUM = 64;  // UNIX V6++支持64种系统调用
static SystemCallTableEntry m_SystemEntranceTable[SYSTEM_CALL_NUM];   //64个表项
```

```
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  */
    { 3, &Sys_Exec  },              /* 11 = Exec    */
    { 1, &Sys_ChDir },              /* 12 = chdir   */
```

系统调用号是系统调用表的下标

```
struct SystemCallTableEntry
{
    unsigned int   count;   //参数个数
    int     (*call)();      //系统调用子程序的入口地址
};
```

图 3

系统调用表中的每个表项是一个叫做 SystemCallTableEntry 的数据结构，count 是系统调用参数的个数，call 是函数指针、是系统调用处理函数 Sys_***的入口地址。

这个表现在没有用完。我们要新加一个系统调用的话，找一个空的表项。其下标就是新的系统调用的系统调用号。　　看，49~63 这 14 个系统调用号还没分配出去。可以用它们实现新系统调用。　　比如 **我们用 50，这个号**。

```
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
```

图 4、SystemCall.h 文件

回到图 3，新加一个系统调用的话，**修改 m_SystemEntranceTable 的第 50 个表项：**

原本是：{ 0, &Sys_Nosys }　　改成：{ n, &Sys_No50 }

n 是 你写的 50 号系统调用的参数的数量。Sys_No50( )是新的 50#系统调用的处理函数。比如最简单的，照抄 20# 系统调用 getpid( )，读取、返回现运行进程的 pid。n 应该是 0。

所有系统调用处理函数定义在 SystemCall.cpp 里。新加的函数 Sys_No50( ) 应该定义在 SystemCall.cpp 文件里。

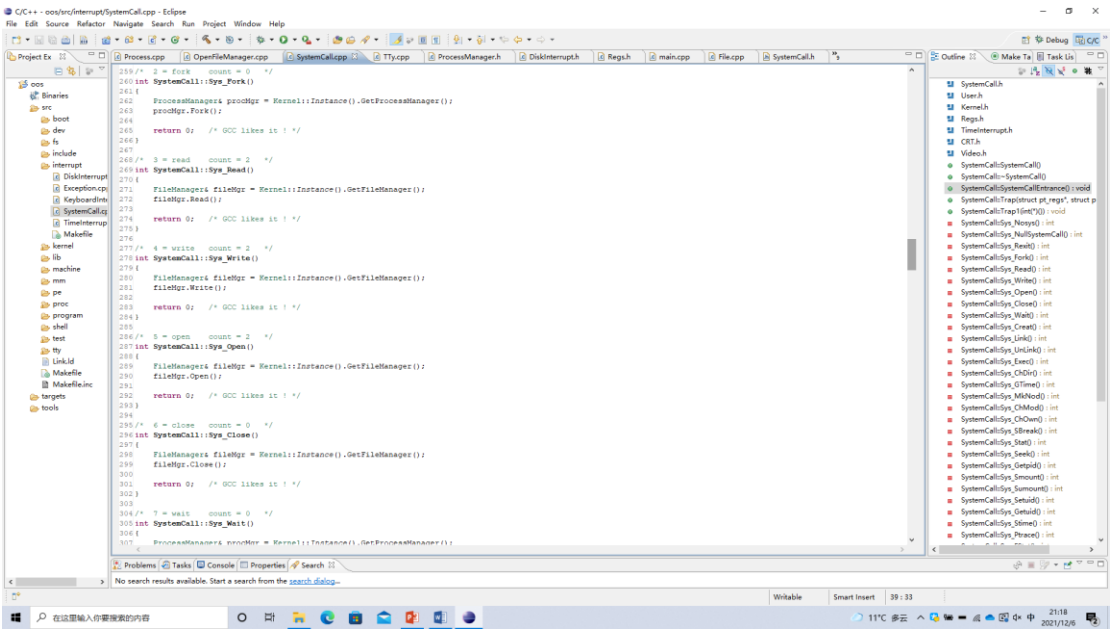


图 5、SystemCall.cpp 文件

完成步骤 1 和步骤 2，内核层面，系统调用添加完毕。

下面需要编程进程的用户空间。

3、库函数里有每个系统调用的钩子函数

每个系统调用都有一个库函数与之相对，称其为系统调用的**钩子函数（hook）**。

编译链接时GCC会将程序员写的源码和所需库函数链接成可执行程序。因此钩子函数运行在用户态。

UNIX V6++ 的钩子函数定义在： lib/src/sys.c 和 lib/src/file.c。声明在 lib/include/...h

Linux 的钩子函数定义在： unistd.h

这是 Unix V6++ 系统的函数库。编译时，引用的库函数静态链接在应用程序里。

图 6

以系统调用 open( )为例。open 是一个有 2 个参数的 5#系统调用。

执行open系统调用的应用程序

```
fd = open("fileName", mode);
if( fd < 0 ) {
    printf("Can not open file!\n");
    exit();
}
```

open系统调用
(5号系统调用)
的钩子函数

```
int open(char* pathname, unsigned int mode)
{
    int res;
    __asm__ __volatile__ ("int
            $0x80":"=a"(res):"a"(5),"b"(pathname),"c"(mode));
    if ( res >= 0 )
        return res;
    return -1;
}
```

图 7

添加一个新系统调用的话，我们应该**在 lib/src/sys.c 函数里，为它添加一个钩子函数**。

4、编写应用程序，测试新加的 50#系统调用

下图是 系统调用 getpid( )的钩子函数。再下面一张图是测试 getpid 系统调用的应用程序。Unix V6++系统添加新的应用程序的方法，大家看实验一，回顾我们编 helloWorld 程序的方法。

转到(G) 运行(R) 终端(T) 帮助(H) sys.c - oos - Visual Studio Code

fork.c | sys.c

src > lib > src > sys.c > getpid()

```
101
102   int getpid()
103   {
104       int res;
105       __asm__ volatile ( "int $0x80":"=a"(res):"a"(20) );
106       if ( res >= 0 )
107           return res;
108       return -1;
109   }
110
```

转到(G) 运行(R) 终端(T) 帮助(H) fork.c

fork.c × sys.c

src > program > fork.c > main1(int, char * [])

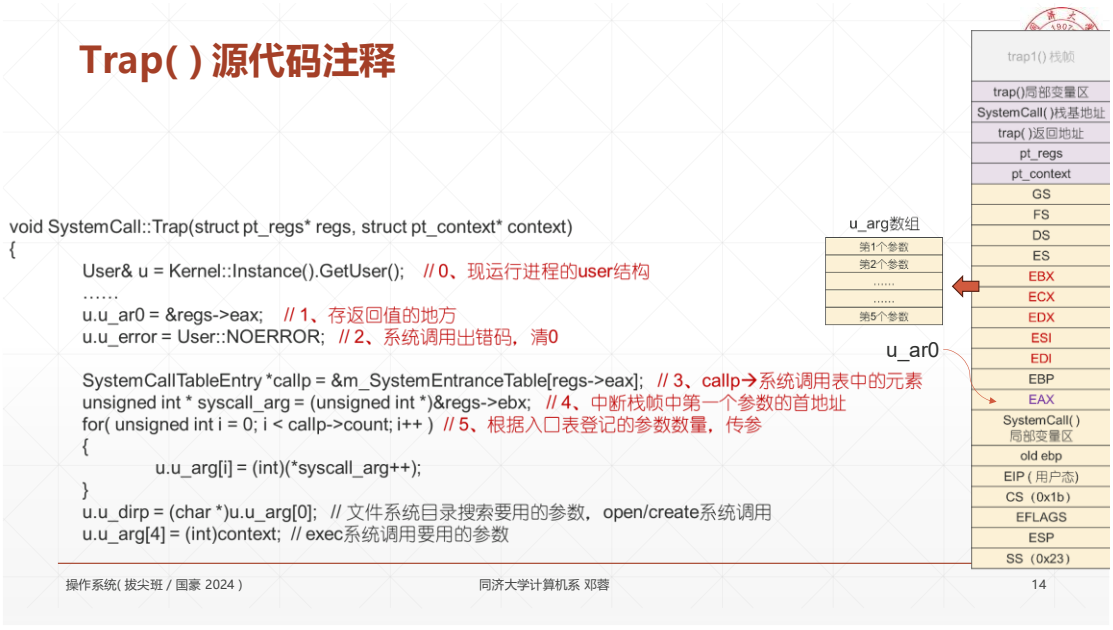
```c
#include <stdio.h>
#include <stdlib.h>

int main1(int argc, char* argv[])
{
    int pid = getpid();

    printf("My PID is %d\n",pid);

    exit(0);
}
```

四、阅读源代码，理清系统调用执行过程



Trap( ) 源代码注释

```
Trap1(callp->call);   // 6、从入口函数进，执行系统调用，read，getpid ……
 ……
                                  从u_arg数组取入口参数，
                                  将运行结果写入u.u_ar0，
                                  如果出错，错误码写u.u_error。
if( User::NOERROR != u.u_error )
{    // 7、出错嘛？User::NOERROR是0，表示没出错
     regs->eax = -u.u_error; //   出错码取负，存入中断栈帧，带回用户态
} //  错误码定义，参见PPT

……

u.u_procp->SetPri();   // 8、计算、设置执行应用程序的优先数
}
```