

同济大学计算机系

操作系统课程实验报告



学 号 2154312

姓 名 郑博远

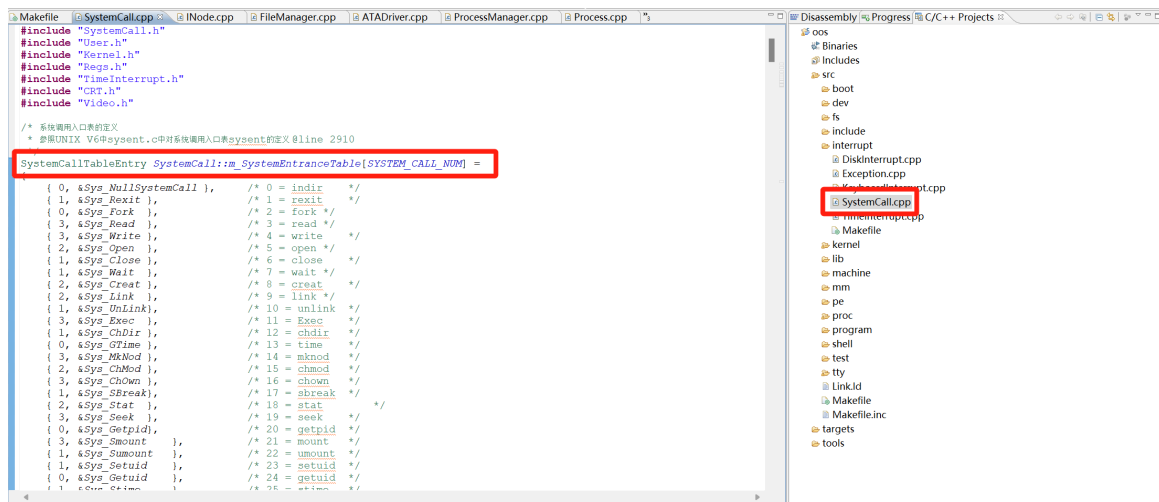
专 业 计算机科学与技术

授课老师 邓蓉老师

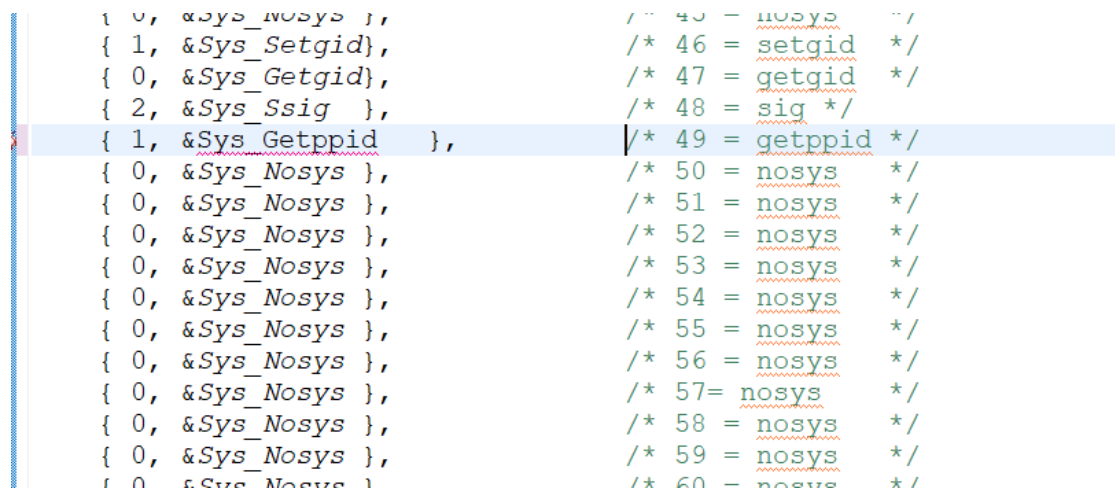
P04: UNIX V6++中添加新的系统调用

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 SystemCall.cpp 中找到对系统调用子程序入口表 m_SystemEntranceTable 赋值的一段程序代码：



2. 选择第 49 项，并用 { 1, &Sys_Getppid } 来替换原来的 { 0, &Sys_Nosys }, 即：第 49 号系统调用所需参数为 1 个，系统调用处理子程序的入口地址为 &Sys_Getppid。



3. 在 SystemCall.h 文件中添加该系统调用处理子程序 Sys_Getppid 的声明:

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

/* 49 = getppid count = 1 */
static int Sys_Getppid();

/* 50 ~ 63 = nosys count = 0 */

private:
/*系统调用入口表的声明*/
static SystemCallTableEntry m_SystemEntranceTable[SYSTEM_CALL_NUM];
};
```

- > ProcessManager.h
- > Regs.h
- > Simple.h
- > SwapperManager.h
- > SystemCall.h
- > TaskStateSegment.h
- > Text.h
- > TimeInterrupt.h
- > Tty.h
- > User.h
- > Utility.h

4. 在 SystemCall.cpp 中添加 Sys_Getppid 的定义:

```
u.u_procp->>sig();

return 0; /* GCC likes it ! */
}

/* 49 = getppid count = 1 */
int SystemCall::Sys_Getppid()
{
    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(i = 0; i < ProcessManager::NPROC; i++){
        if (curpid == procMgr.process[i].p_pid){
            u.u_ar0[User::EAX] = procMgr.process[i].p_ppid;
            break;
        }
    }

    return 0; /* GCC likes it ! */
}
```

- > Exception.cpp
- > KeyboardInterrupt.cpp
- > SystemCall.cpp
- > TimeInterrupt.cpp
- > Makefile
- > kernel
- > lib
- > machine
- > mm
- > pe
- > proc
- > program
- > shell
- > test
- > tty
- > Link.ld
- > Makefile
- > Makefile.inc
- > targets
- > tools

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. 找到 sys.h 文件, 在其中加入名为 getppid 的库函数的声明:

```
int getPath(char* path);

int getpid();

int getppid(int pid);

unsigned int getgid();

unsigned int getuid();

int setgid(short gid);

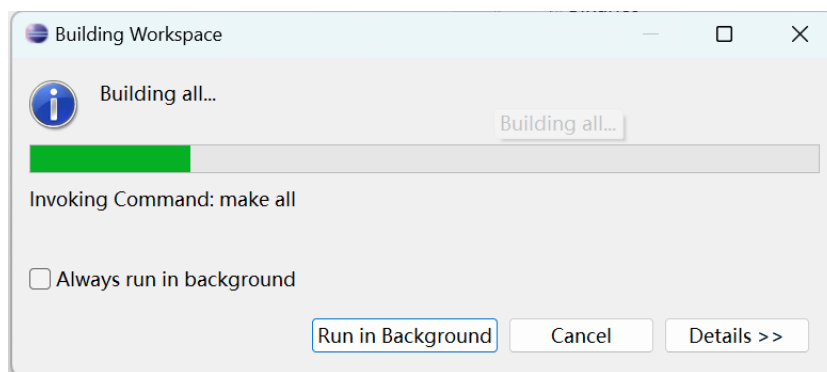
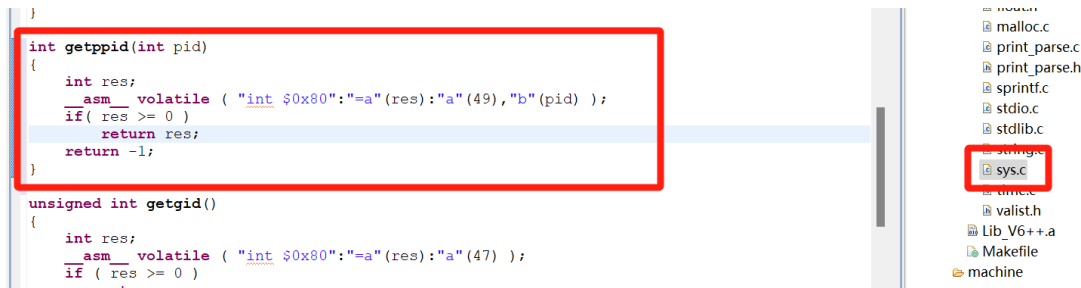
int setuid(short uid);

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);
```

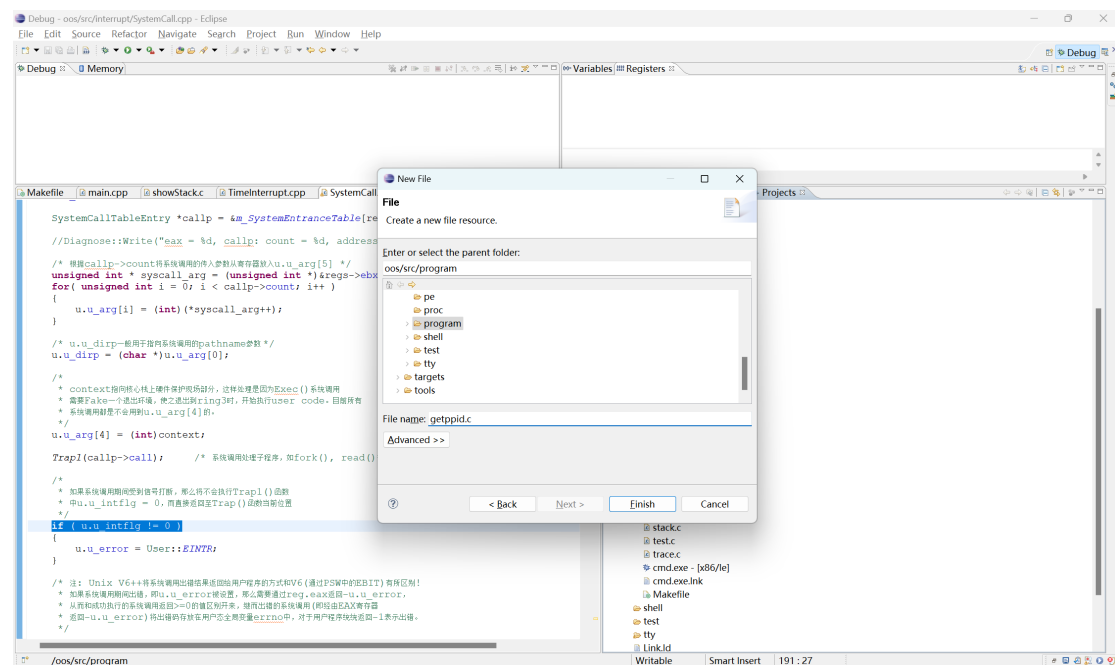
- lib
- include
- file.h
- malloc.h
- stddef.h
- stdio.h
- stdlib.h
- string.h
- sys.h
- time.h
- objs
- src
- Lib_V6++.a
- Makefile
- machine
- mm
- pe

2. 在 sys.c 文件中添加库函数 getppid 的定义，并选择“Build All”重新编译 UNIX V6++:



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. 在 program 文件夹下新建 getppid.c:



2. 编写 getppid.c 的代码如下:

```
getppid.c x
#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;
}
```

3. 修改 Makefile 文件用于编译:

```
SHELL_OBJS = $(TARGET)\cat.exe \
$(TARGET)\cat1.exe \
$(TARGET)\cp.exe \
$(TARGET)\ls.exe \
$(TARGET)\mkdir.exe \
$(TARGET)\rm.exe \
$(TARGET)\perf.exe \
$(TARGET)\sig.exe \
$(TARGET)\copyfile.exe \
$(TARGET)\shutdown.exe \
$(TARGET)\test.exe \
$(TARGET)\forks.exe \
$(TARGET)\trace.exe \
$(TARGET)\echo.exe \
$(TARGET)\date.exe \
$(TARGET)\newsig.exe \
$(TARGET)\sigTest.exe \
$(TARGET)\stack.exe \
$(TARGET)\malloc.exe \
$(TARGET)\showStack.exe \
$(TARGET)\getppid.exe

#$(TARGET)\performance.exe

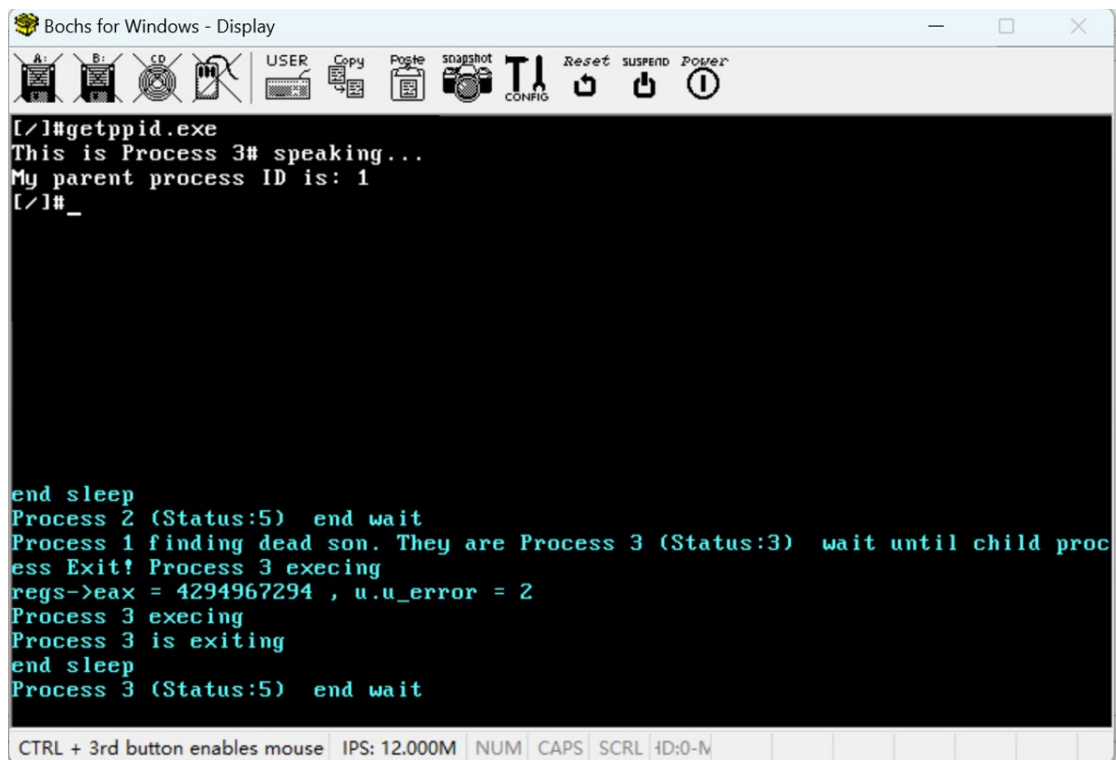
$(TARGET)\malloc.exe :      malloc.c
$(CC) $(CFLAGS) -I"$(INCLUDE)" -I"$(LIB_INCLUDE)" $< -e _main1 $(V6++LIB) -o $@
copy $(TARGET)\malloc.exe $(MAKEIMAGEPATH)\$(BIN)\malloc.exe

$(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

clean:
del $(TARGET)\*.o
del $(TARGET)\*.exe
del $(TARGET)\*.asm
```

4. 在运行模式下启动 UNIX V6++, 观察程序的输出正确:



```
[/]#getppid.exe
This is Process 3# speaking...
My parent process ID is: 1
[/]#_

end sleep
Process 2 (Status:5) end wait
Process 1 finding dead son. They are Process 3 (Status:3) wait until child process Exit! Process 3 execing
regs->eax = 4294967294, u.u_error = 2
Process 3 execing
Process 3 is exiting
end sleep
Process 3 (Status:5) end wait
```

5. 在 Sys_Getppid 函数的 “int curpid=(int)u.u_arg[0]” 赋值语句处添加断点:

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

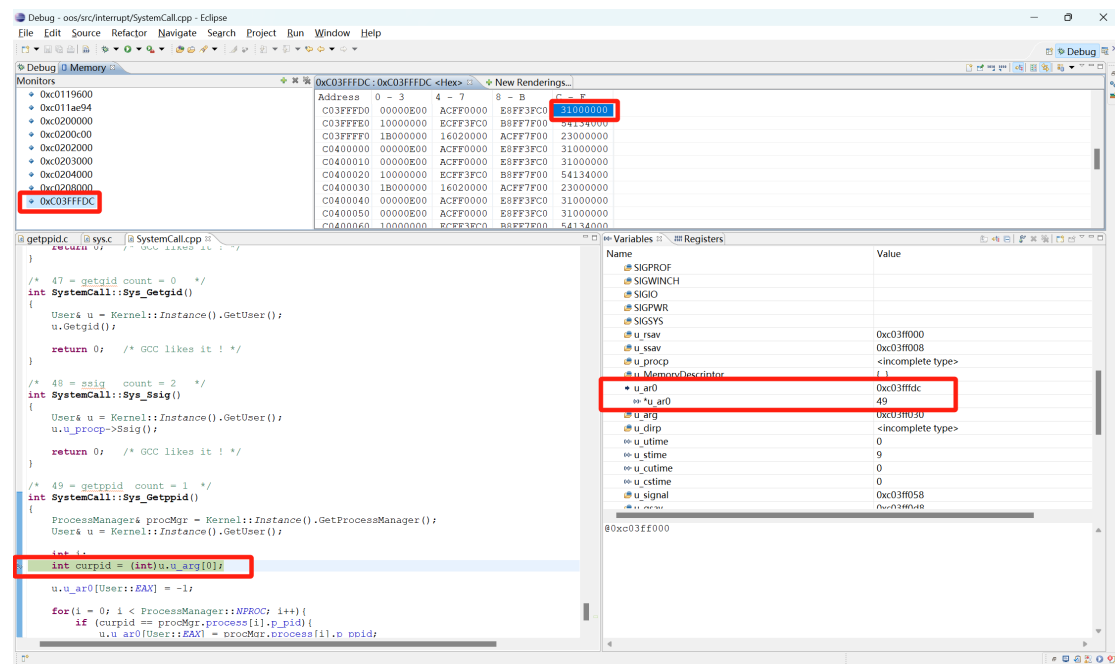
    int i;
    int curpid = (int)u.u_arg[0];

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

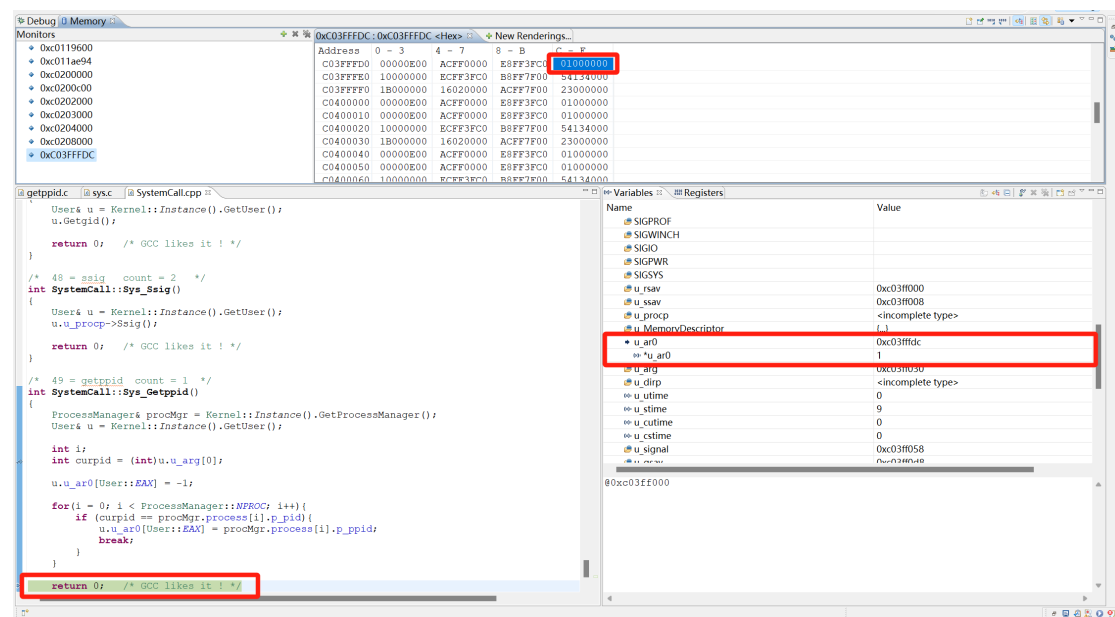
    for(i = 0; i < ProcessManager::NPROC; i++){
        if (curpid == procMgr.process[i].p_pid){
            u.u_ar0[User::EAX] = procMgr.process[i].p_ppid;
            break;
        }
    }

    return 0; /* GCC likes it ! */
}
```

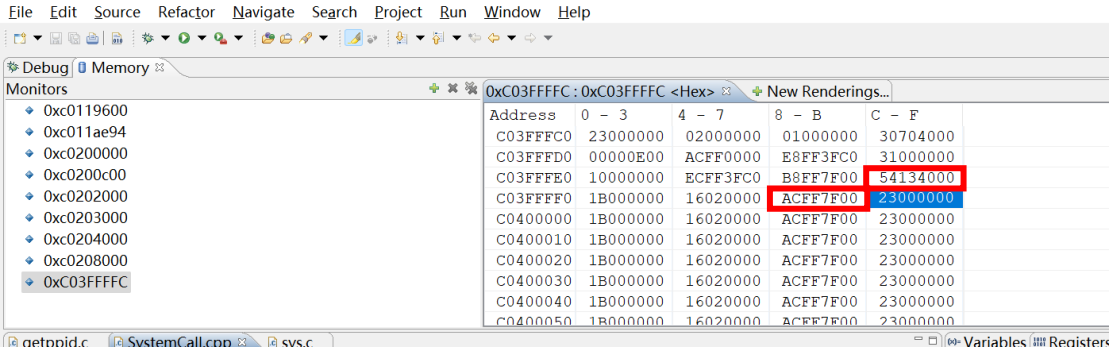
6. 程序停在该断点处时，可以看到，u_ar0 指向的核心栈中保存 EAX 单元的值为 49，在 Memory 窗口的对应地址也可以验证：



7. 程序运行到 Sys_Getppid()末尾时，观察到到 0xC03FFDC 地址处对应的值变为 1（对应返回的 ppid 值）：



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（由于运行结果略有不同，下表根据 Memory 用红色修改）。



地址	内容	值	说明
0xC03FFFA0			
	OID EBP	0xC03FFFE8	
	返回地址	0xC010D302	返回系统调用入口程序的地址
		0xC03FFFB4	指向软件现场 gs
0xC03FFFB0		0xC03FFFE4	指向软件现场 EIP
0xC03FFFB4:	GS		
	FS		
	DS		
	ES		
	EBX		
	ECX		
	EDX		
	ESI		
	EDI		
	EBP	0xC03FFFE8	
0xC03FFDC	EAX	49	
		0x0000001D	
		0xC03FFFE4	
0xC03FFFE8		0x007FFFB8	
0xC03FFFE4	EIP	0x00401354	
	CS	0x0000001B	后两位为 11
	EFLAGS	0x00000202	
	ESP	0x007FFFA0	
0xC03FFFC	SS	0x00000023	

系统调用入口程序的
栈帧底部 EBP 值，即
该表格下方的
0xC03FFFE8 单元。

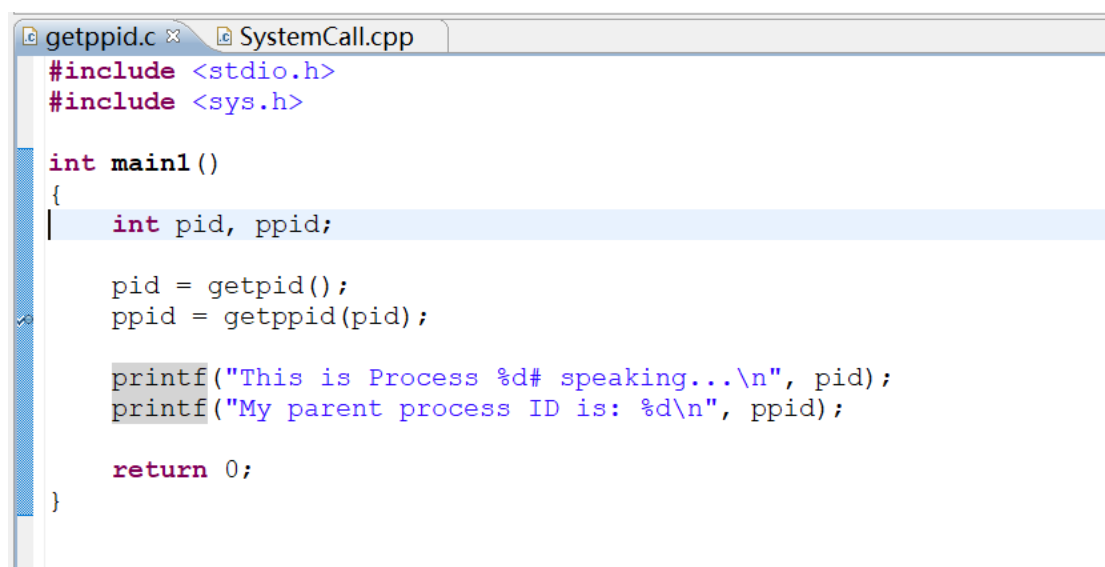
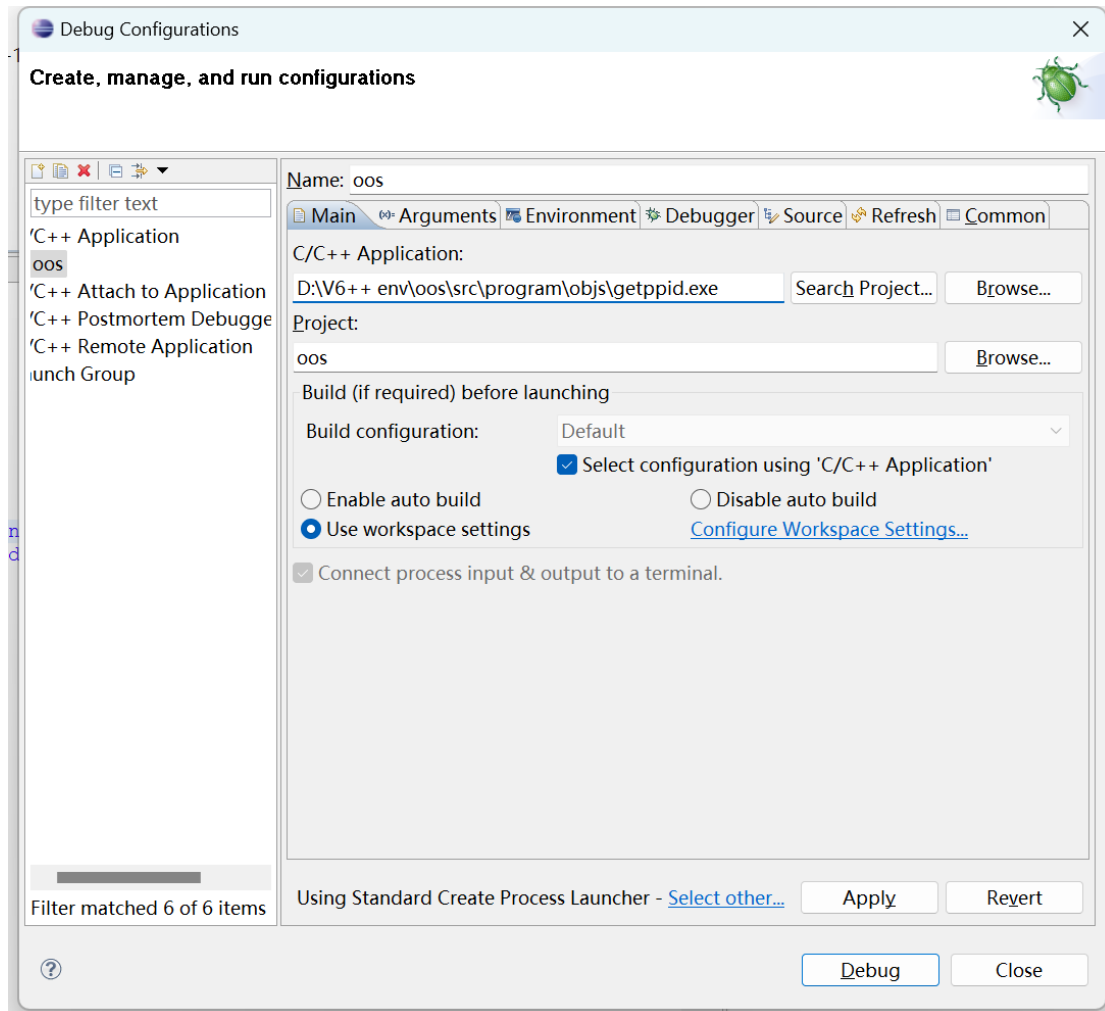
响应系统调用前的
ebp，对应用户
栈当前栈帧底。

getppid() 中内联汇
编语句的返回部分，
即%eax 赋值 res。

响应系统调用前的
esp，对应用户
栈当前栈帧顶。

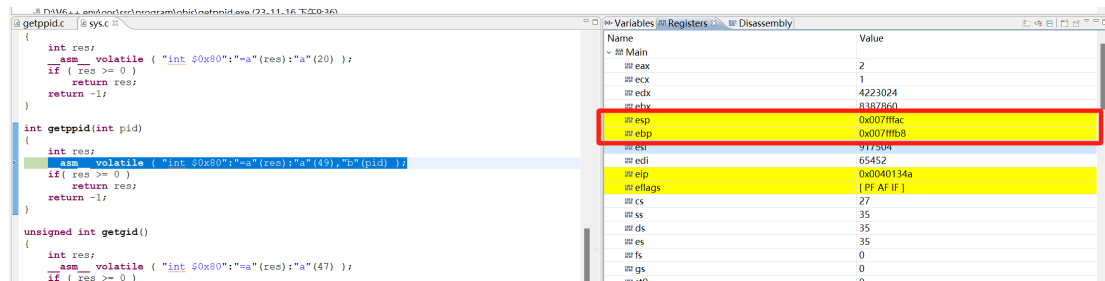
1. 最上方的黄色标注的地址单元即对应下方的 0xC03FFFE8 处，根据中断压栈的常识，以及其所存的地址数值与下方单元的关系即可得出是系统调用入口程序的栈帧底部，即对应当时的 EBP 值；

2. 修改调试对象，打断点：



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

语句执行之前，观察到寄存器的值如下图。因此可以佐证第二个黄色区域与第四个黄色区域
的值分别为响应系统调用前的 `ebp` 以及 `esp`：



4. 查看汇编代码可以发现，地址 `0x401354` 对应汇编代码：`mov %eax,-0x8(%ebp)`。即内联汇编代码抛出系统处理请求后，系统处理完毕后唤醒原进程后，调度上台时所执行的第一条指令：将 `EAX` 寄存器中得到的 `ppid` 值赋予局部变量 `rex`。

