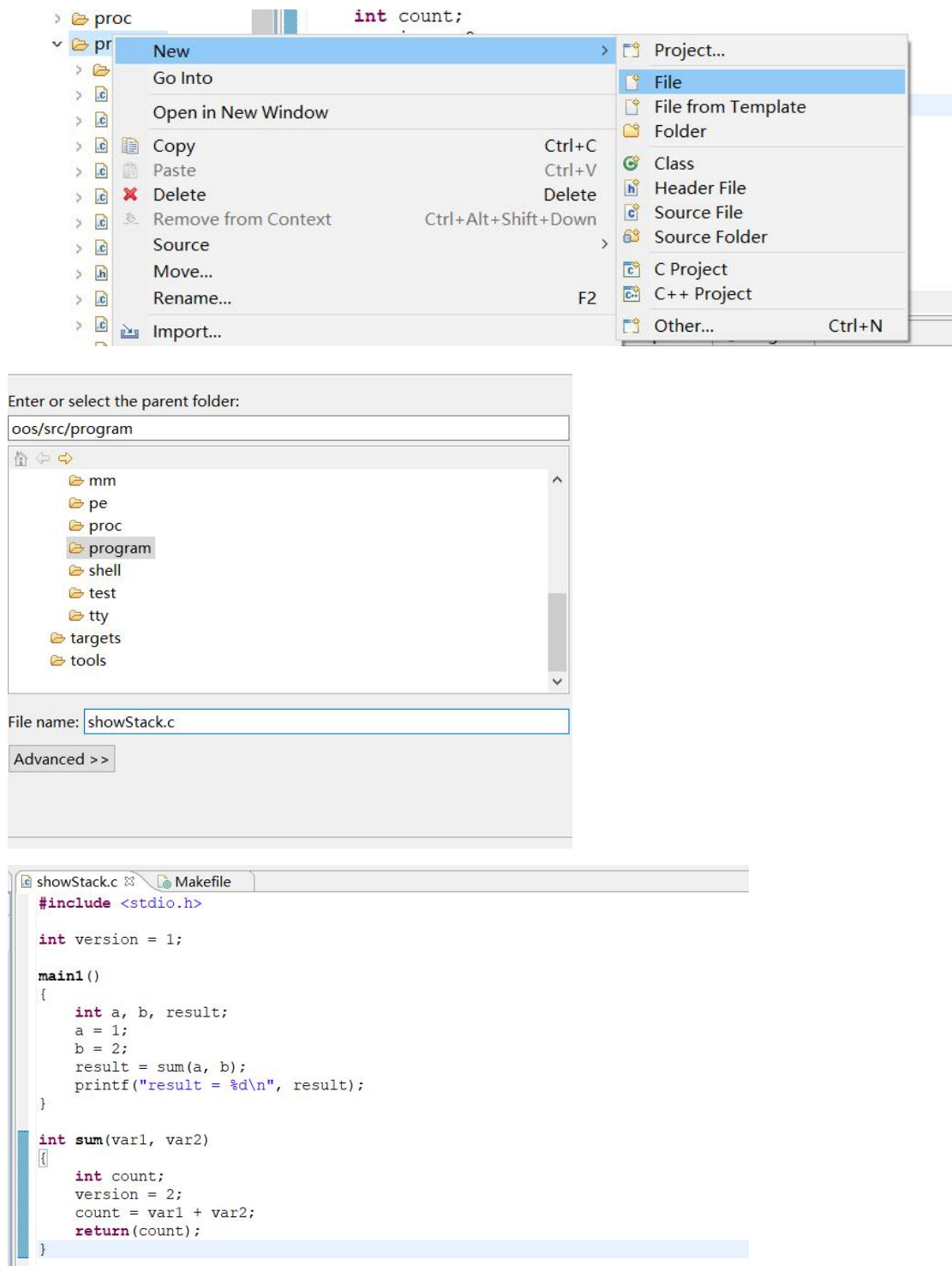


P02: UNIX V6++进程的栈帧

2152118 史君宝

一、UNIX V6++中编译链接运行一个 C 语言程序

(1) 在 program 文件加入一个新的 c 语言文件

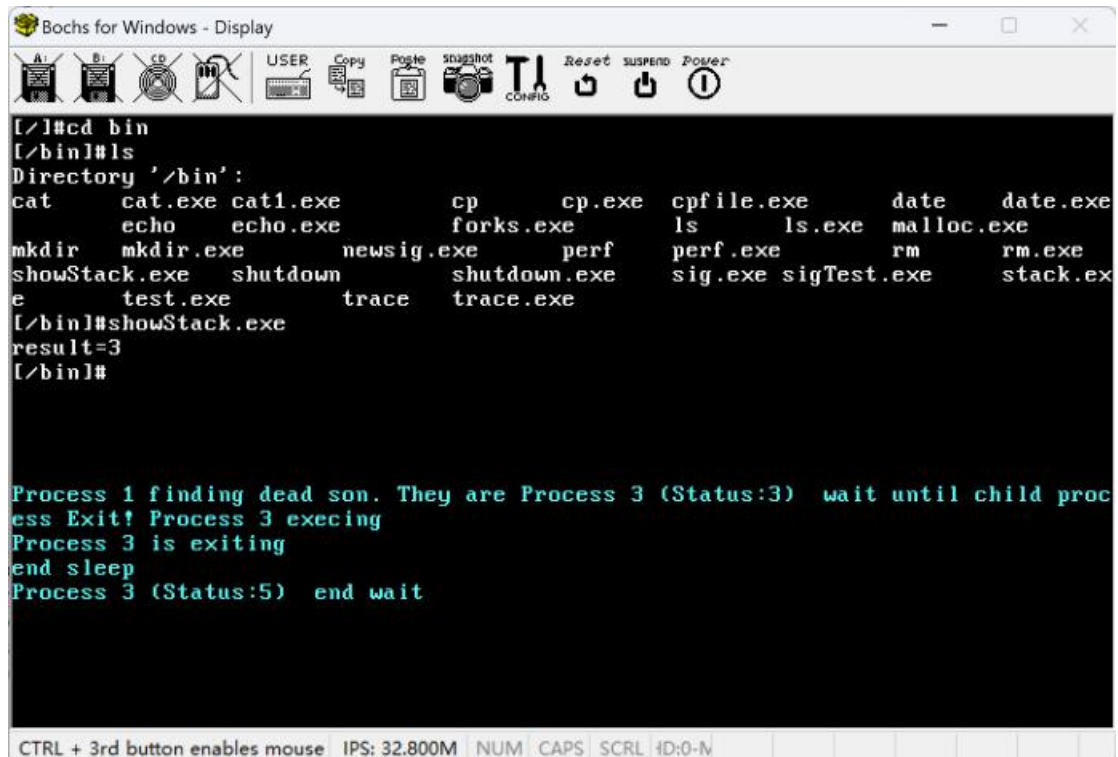
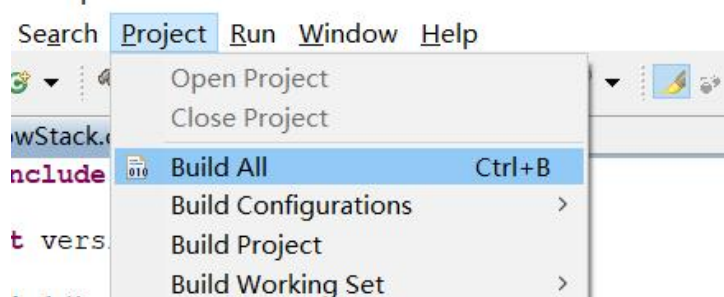


(2) 修改编译需要使用的 Makefile 文件

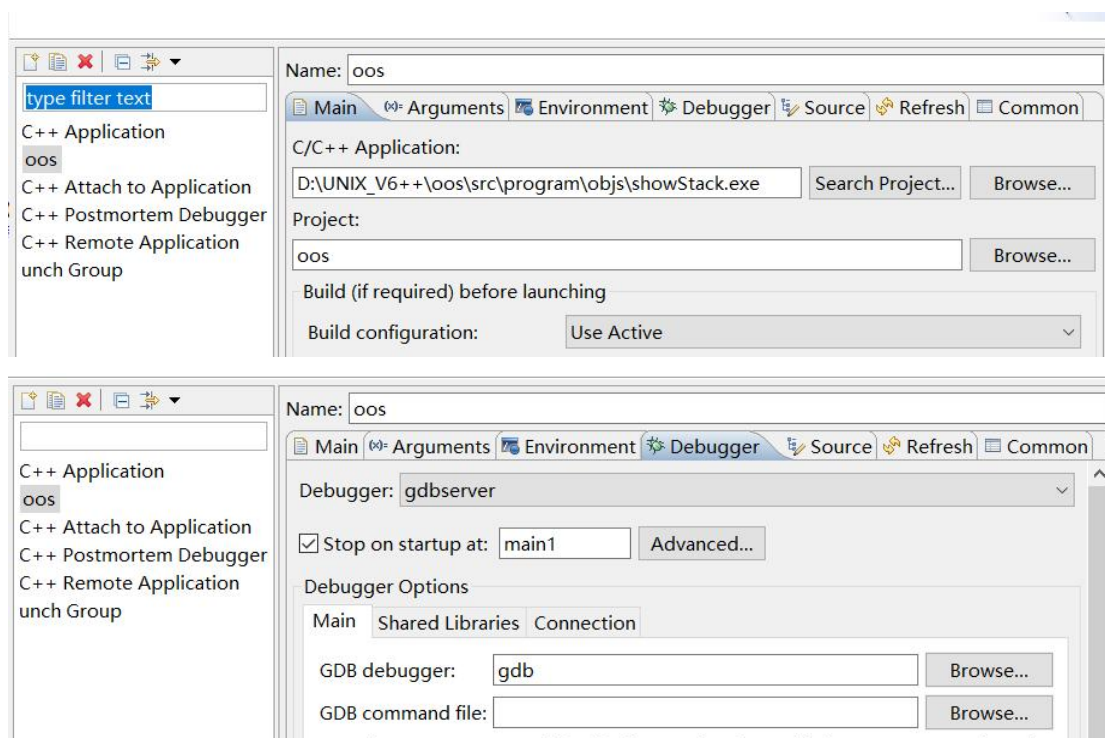
```
$(TARGET)\stack.exe \  
$(TARGET)\malloc.exe\  
$(TARGET)\showStack.exe
```

```
$(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
```

(3) 重新编译运行 UNIX V6++代码



(4) 设置调试对象和调试起点



(5) 设置并开始调试

```
[/l]#cd bin
[/bin]#showStack.exe
result = 3
[/bin]#showStack.exe
```

二、复现 main1 函数核心栈的变化

(1) 未进入 sum 中的核心栈

	0x007fffd0	2	main 的局部变量 b, 0040100d 语句后变为 2
	0x007fffd4	1	main 的局部变量 a, 00401006 语句后变为 1
ebp:	0x007fffd8	007FFFE0	上一栈帧基址, 00401000 语句压栈; 00401001 语句将 esp 修改到此处
	0x007fffdc	00000008	main 的返回地址

007FFFD0	00000000	00000000	00000000	00000000
007FFFD0	02000000	01000000	E0FF7F00	08000000

(2) 进入 sum 函数的核心栈

esp: (call 之后)	0x007fffb4	00401026	sum 的返回地址, 00401021 (call) 语句压栈, 并将 esp 修改到此处
esp: (call 之前)	0x007fffc0	1	00401003 语句将 esp 修改到此处, 空出 main 的局部变量位置和 sum 的参数位置; 此处为 sum 的参数 var1, 0040101e 语句执行完后修改为 1
	0x007fffc4	2	此处为 sum 的参数 var2, 00401017 语句执行完后修改为 2

007FFFA0	00000000	00000000	00000000	00000000
007FFFB0	00000000	00000000	▲ D8FF7F00	▲ 26104000
007FFFC0	▲ 01000000	▲ 02000000	00000000	00000000
007FFFD0	00000000	01000000	E0FF7F00	00000000

(3) Sum 函数中的 count = var1 + var2

007FFFB0	00000000	▲ 03000000	D8FF7F00	26104000
007FFFC0	01000000	02000000	00000000	00000000
007FFFD0	02000000	01000000	E0FF7F00	08000000

其中 0X007FFFD8 是上一栈帧的地址, 即对应 EBP 的上一地址
而 0X00000003 是 count 的值

(4) 返回 main1 函数

007FFFA0	00000000	00000000	00000000	00000000
007FFFB0	00000000	03000000	D8FF7F00	26104000
007FFFC0	01000000	02000000	00000000	▲ 03000000
007FFFD0	02000000	01000000	E0FF7F00	08000000

返回之后会将栈帧地址以上的第三个参数 result 赋值为 0X00000003

综上核心栈为:

地址	数值	说明
0X007fffb4	3	局部变量 count
0X007fffb8	0X007fffd8	上一栈帧 main1 的基址
0X007fffb4	00401026	sum 函数的返回地址
0X007fffc0	1	调用 sum 的参数 var1
0X007fffc4	2	调用 sum 的参数 var2
0X007fffc8	0	
0X007fffcc	3	局部变量 result
0X007fffd0	2	局部变量 b
0X007fffd4	1	局部变量 a
0X007fffd8	0X007fffe0	上一栈帧基址
0X007fffdc	00000008	main1 的返回地址

三、分析 sum 的汇编代码，完善栈帧

(1) sum 函数的汇编代码

```
sum:
0040103e: push %ebp
0040103f: mov %esp,%ebp
00401041: sub $0x4,%esp
17      version = 2;
> 00401044: movl $0x2,0x404000
18      count = var1 + var2;
0040104e: mov 0xc(%ebp),%eax
00401051: add 0x8(%ebp),%eax
00401054: mov %eax,-0x4(%ebp)
19      return(count);
00401057: mov -0x4(%ebp),%eax
20      }
0040105a: leave
0040105b: ret
printf:
```

我们依次分析：

0040103e: push %ebp :

将上一栈帧基址入栈，将 main1 栈帧的基址存放到 %esp 的位置。

0040103f: mov %esp, %ebp :

将 esp 寄存器值赋值给 ebp，这样就将 ebp 指向新栈帧的基址。

00401041: sub &0X4, %esp :

将 esp 指针上移，为 sum 函数中的局部变量申请空间。

00401044: movl &0X2, 0X404000 :

将 0X2 值赋值给地址为 0X404000 位置的变量，即为 version 变量。

0040104e: mov 0Xc(%ebp), %eax :

将 ebp 指针下面三个位置的变量，即为变量 var1 移到寄存器 eax。

00401051: add 0X8(%ebp), %eax :

将 ebp 指针下面两个位置的变量，即变量 var2 与寄存器 eax 相加。

00401054: mov %eax, -0X4(%ebp) :

将寄存器 eax 赋值给 ebp 指针上面一个位置的变量，即变量 count。

00401057: mov -0X4(%ebp), %eax :

将 ebp 指针上面一个位置的变量，即变量 count 赋值给寄存器 eax。

0040105a: leave :

0040105b: ret :

离开并退出函数，返回函数地址。

(2) 完善的栈帧

下面是完整的栈帧，在原图上直接进行修改。

其中上面的第一格应为 sum 函数中的局部变量 count，刚开始申请的时候值为 0，在经过 count = var1 + var2 计算之后，会变为 3。
第二格应为上一栈帧的基址，也就是 main1 函数的栈帧基址，即为 0X007fffd8

esp: (call 之后)	0x007fffb4	00401026	sum 的返回地址, 00401021 (call) 语句压栈, 并将 esp 修改到此处
esp: (call 之前)	0x007fffc0	1	00401003 语句将 esp 修改到此处, 空出 main 的局部变量位置和 sum 的参数位置; 此处为 sum 的参数 var1, 0040101e 语句执行完后修改为 1
	0x007fffc4	2	此处为 sum 的参数 var2, 00401017 语句执行完后修改为 2
ebp:	0x007fffc8	0	main 的局部变量 result, 00401026 语句后变为 3 main 的局部变量 b, 0040100d 语句后变为 2 main 的局部变量 a, 00401006 语句后变为 1 上一栈帧基址, 00401000 语句压栈; 00401001 语句将 esp 修改到此处
	0x007ffcc	3	
	0x007ffcd0	2	
	0x007ffcd4	1	
	0x007fffd8	007FFFE0	
	0x007fffdc	00000008	
			main 的返回地址

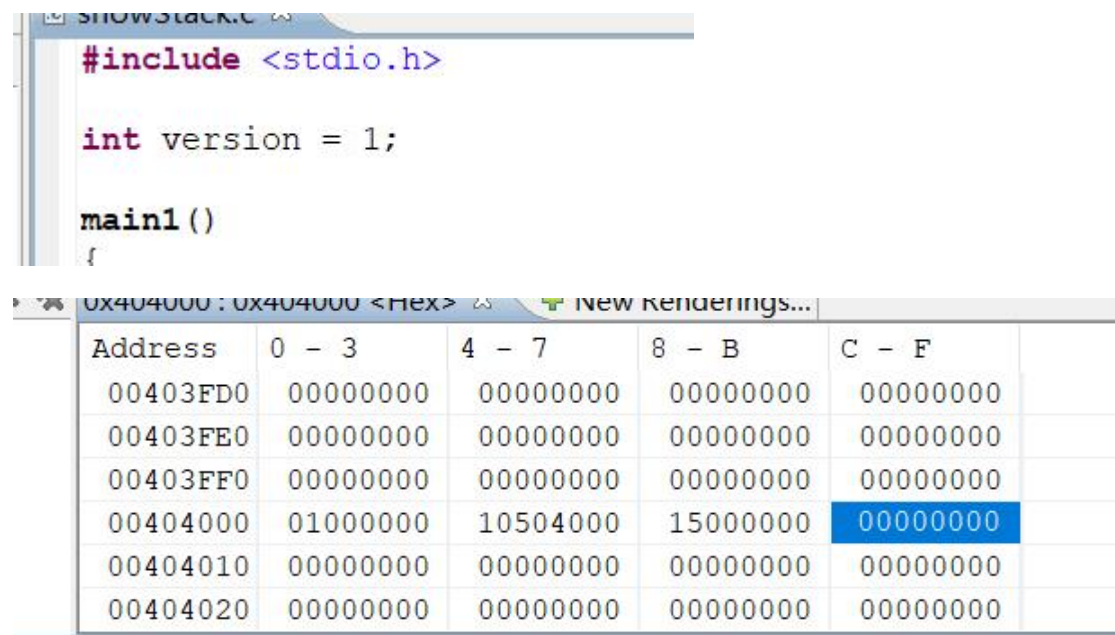
地址	数值	说明
0X007fffb4	3	局部变量 count
0X007fffb8	0X007fffd8	上一栈帧 main1 的基址
0X007fffb4	00401026	sum 函数的返回地址
0X007fffc0	1	调用 sum 的参数 var1
0X007fffc4	2	调用 sum 的参数 var2
0X007fffc8	0	
0X007ffcc	3	局部变量 result
0X007ffcd0	2	局部变量 b
0X007ffcd4	1	局部变量 a
0X007fffd8	0X007fffe0	上一栈帧基址
0X007fffdc	00000008	main1 的返回地址

四、回答问题。

在 `sum` 的汇编代码中出现了 `0x404000` 这个地址，请先通过分析代码回答这个地址是什么，然后尝试通过调试验证。

解答：我们通过分析代码可以知道，这个 `0x404000` 应该是在刚开始申请的全局变量 `version` 的地址，我们可以通过 Memory 找到具体的位置，观察其在进入 `sum` 函数前后的变化，来验证。

进入 `sum` 函数前：



The screenshot shows a debugger interface. The top pane displays the source code of a C program:

```
#include <stdio.h>

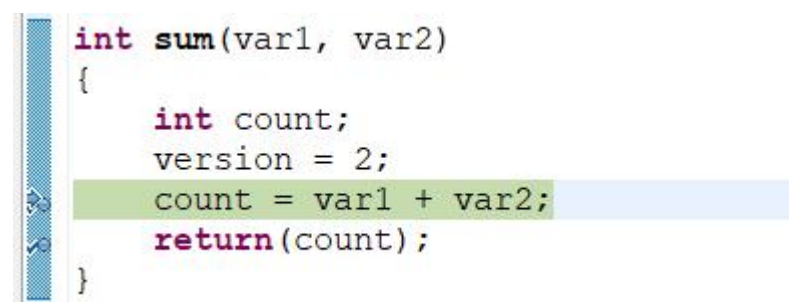
int version = 1;

main1()
{
```

The bottom pane shows a memory dump for the address `0x404000`. The dump is organized into columns for different byte ranges (0-3, 4-7, 8-B, C-F) and a column for the full address. The row for address `00404000` is highlighted, showing the value `00000000` in the C-F column.

Address	0 - 3	4 - 7	8 - B	C - F
00403FD0	00000000	00000000	00000000	00000000
00403FE0	00000000	00000000	00000000	00000000
00403FF0	00000000	00000000	00000000	00000000
00404000	01000000	10504000	15000000	00000000
00404010	00000000	00000000	00000000	00000000
00404020	00000000	00000000	00000000	00000000

进入 `sum` 函数后：



The screenshot shows the source code of the `sum` function in a debugger. The code is as follows:

```
int sum(var1, var2)
{
    int count;
    version = 2;
    count = var1 + var2;
    return(count);
}
```

Address	0 - 3	4 - 7	8 - B	C - F
00403FD0	00000000	00000000	00000000	00000000
00403FE0	00000000	00000000	00000000	00000000
00403FF0	00000000	00000000	00000000	00000000
00404000	02000000	10504000	15000000	00000000
00404010	00000000	00000000	00000000	00000000
00404020	00000000	00000000	00000000	00000000

(2) 设置断点二：

```

//init page protection
machine.InitPageDirectory();
machine.EnablePageProtection();
/*
 * InitPageDirectory() 中将线性地址0-4M映射到物理内存
 * 0-4M是为保证此注释以下至本函数结尾的代码正确执行!
 */

//使用0x10段寄存器
asm volatile

```

变量值：

Name	Value
machine	{...}
KERNEL_CODE_SEGMENT_SELECTOR	0
KERNEL_DATA_SEGMENT_SELECTOR	0
USER_CODE_SEGMENT_SELECTOR	0
USER_DATA_SEGMENT_SELECTOR	0
TASK_STATE_SEGMENT_SELECTOR	0
TASK_STATE_SEGMENT_IDX	0
PAGE_DIRECTORY_BASE_ADDRESS	
KERNEL_PAGE_TABLE_BASE_ADDRESS	
USER_PAGE_TABLE_BASE_ADDRESS	
USER_PAGE_TABLE_CNT	
KERNEL_SPACE_SIZE	0
KERNEL_SPACE_START_ADDRESS	
instance	{...}
m_IDT	<incomplete type>
m_GDT	<incomplete type>
m_PageDirectory	0x00ff53f0
m_KernelPageTable	16733168
m_UserPageTable	16733168
m_TaskStateSegment	16733168

寄存器值：

Name	Value
▼ Main	
eax	-1071640576
ecx	0
edx	-1072573280
ebx	1140736
esp	0xc00ffca
ebp	0xc00ffd2
esi	917504
edi	65452
eip	0xc0109061
eflags	[AF]
cs	24
ss	32
ds	32
es	32
fs	0
gs	0
st0	0
st1	0
st2	0

三、UNIX V6++源代码目录结构

boot/：应该是包含引导加载程序的源代码和配置文件。

dev/：应该是包含设备驱动程序的源代码，用于与硬件设备进行交互。

fs/：应该是包含文件系统相关的源代码，包括文件系统的实现和文件操作。

include/：包含系统的头文件，就像我们在 C++ 中使用的 include <iostream> 一样定义了各种常量、数据结构和函数原型等。

kernel/：包含内核的核心代码，整个操作系统最重要的部分，负责处理系统调用、进程管理、内存管理等核心功能。

lib/：包含通用的库函数和工具函数的源代码，就像我们在使用 eclipse 中导入的库一样。

mm/：应该与存储相关，包含内存管理相关的源代码，比如内存分配、虚拟内存管理等等。

sys/：应该包含有关系统调用的相关信息，比如包含系统调用相关的源代码，定义了系统调用的接口和实现。

tools/：包含一些辅助工具和实用程序的源代码，用于构建和调试系统。