

实验二：UNIX V6++ 进程的栈帧

1. 实验目的

结合课程所学知识，通过编写一个简单的 C++ 代码，并在 UNIX V6++ 中编译和运行调试，观察程序运行时栈帧的变化。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) C/C++ 编译器对函数调用的处理和栈帧的构成。
- (2) UNIX V6++ 的运行和调试方法。

4. 实验内容

4.1. 在 UNIX V6++ 中编译链接运行一个 C 语言程序

这一节，我们将学习如何在 UNIX V6++ 中添加一个自己编写的 C 语言程序，并让它能够运行起来。这个方法在后续实验中会反复用到，请读者熟练掌握。

在 UNIX V6++ 中添加一个可执行程序，需要在 `src/program` 文件夹下添加一个源程序文件，并编译通过之后，才可以运行。具体过程如下。

(1) 在 program 文件加入一个新的 c 语言文件

如图 1 所示，右键点击 `program` 文件夹后，“New”一个新的名为 `showStack.c` 的源文件，并键入如图 2 所示的代码。

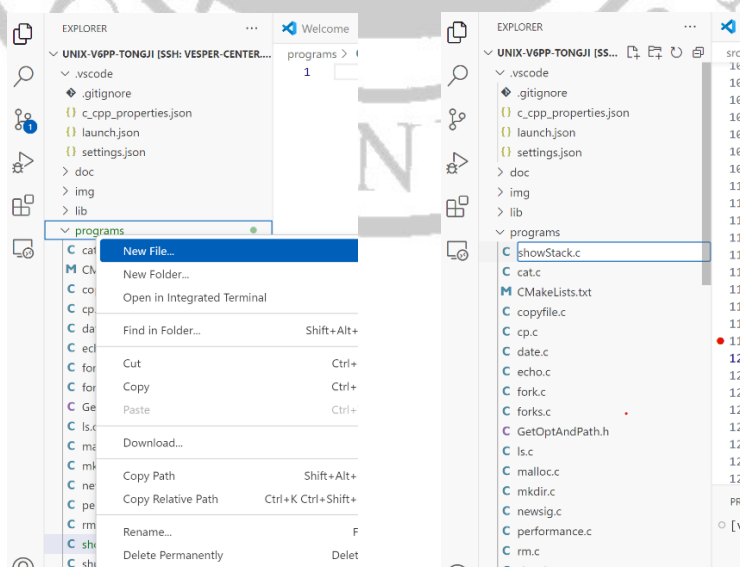


图 1：在 `program` 下新建一个文件

```
programs > C showStack.c > ...
1  #include <stdio.h>
2  int version =1;
3  int sum(int var1, int var2)
4  {
5      int count;
6      version=2;
7      count=var1+var2;
8      return(count);
9  }
10 void main1()
11 {
12     int a,b,result;
13     a=1;
14     b=2;
15     result=sum(a,b);
16     printf("result=%d\n",result);
17 }
```

图 2：示例代码

代码完成的功能很简单，只是为了后续观察堆栈的变化，编写了一个加法计算的函数调用，这里不再详细解释。需要说明的是，由于 UNIX V6++环境的一些特殊性，主程序的入口请使用 `main1`，不要使用 `main`，以免编译器报错。

(2) 重新编译运行 UNIX V6++代码



图 3：重新编译 UNIX V6++的源代码

如图 3 所示，在 vscode 的 Terminal 窗口中直接输入 `make all` 命令并回车（注意当前所在路径），将完成 UNIX V6++代码的重新编译，并给出编译成功的提示。如果编译成功，则在运行（非调试，运行和调试模式的改变见实验一）模式下，启动 UNIX V6++之后，进入 `bin` 文件夹，可以看到该文件夹下有刚编译通过形成的可执行文件 `showStack`（后续实验中，所有通过上述方法添加的可执行程序都在该文件夹下）。此时，键入 `showStack`，可以看到程序的运行结果（如图 4 所示）。

(3) 关于 UNIX V6++的调试目标

在实验一中，我们曾经提到过如何修改调试目标。本实验中，我们希望调试的是自己编写的程序而非内核，所以需要修改调试目标。如图 5 所示，将第 9 行调试内核的设置注释掉，第 10 行调试应用程序处，设置成 `showStack`（后续多个实验会在调试内核和调试应用程序之间切换，请读者牢记此处的相关设置）。

```

QEMU - Press Ctrl+Alt+G to release grab
Machine View
welcome to Unix V6++ Tongji's Edition!
[/]#ls
Directory '/' :
dev bin etc Shell.exe
[/]#cd bin
[/bin]#ls
Directory '/bin':
test fork mkdir stack showStack rm sigTest performance trace cp date
ls forks malloc cat sig shutdown echo testSTDOUT copyfile news
[/bin]#showStack
result=3
[/bin]#

Process 1 finding dead son. They are Process 2 (Status:3) wait until child process Exit! Process 2
execing
regs->eax = -4294967294 , u.u_error = 2
Process 2 execing
Process 2 is exiting
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
Process 3 is exiting
end sleep
Process 3 (Status:5) end wait
Process 1 finding dead son. They are Process 4 (Status:3) wait until child process Exit! Process 4
execing
Process 4 is exiting
end sleep
Process 4 (Status:5) end wait

```

图 4: showStack.exe 程序运行结果

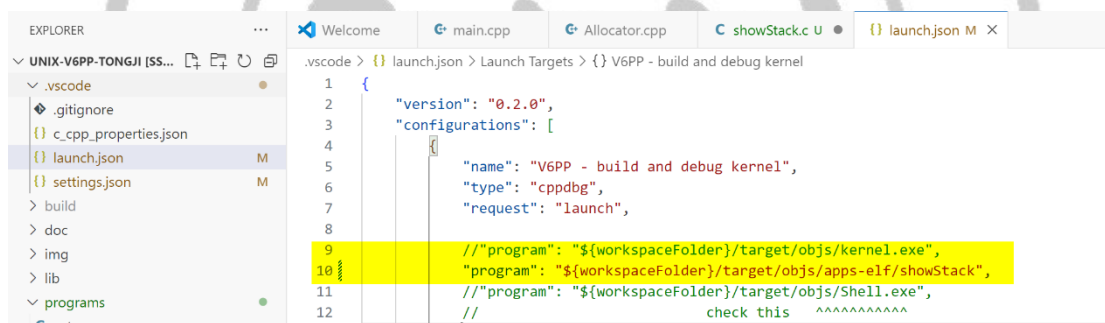


图 5: 修改调试目标

4.2. 开始程序的调试运行

以调试模式启动 UNIX V6++, 看到 UNIX V6++启动成功, 等待调试指令时, 如图 6 所示位置添加断点, 点击 vscode 中的启动调试 (如图 7 所示)。

```

programs > C showStack.c > ...
1 #include <stdio.h>
2 int version =1;
3 int sum(int var1, int var2)
4 {
5     int count;
6     version=2;
7     count=var1+var2;
8     return(count);
9 }
10 void main1()
11 {
12     int a,b,result;
13     a=1;
14     b=2;
15     result=sum(a,b);
16     printf("result=%d\n",result);
17 }
18

```

图 6: 设置断点

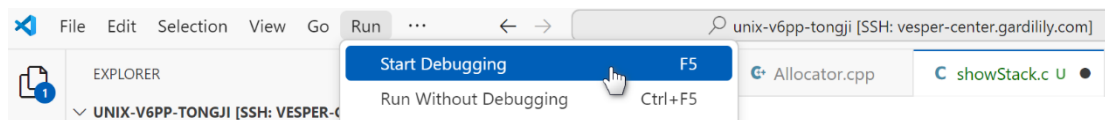




图 7：启动调试

在 UNIX V6++环境下完成“cd bin”和“showStack”的执行，直到程序在断点处停下来，如图 8 所示。如果在“showStack”之前，虚拟机就停止运行，不要紧，只需点击  上的  继续运行，直到停下来为止，即可。

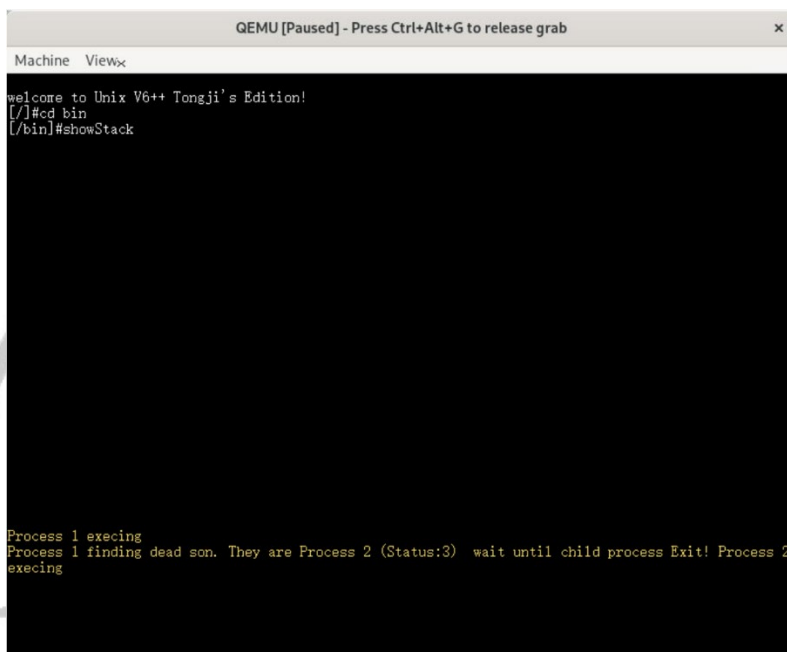


图 8：UNIX V6++调试界面

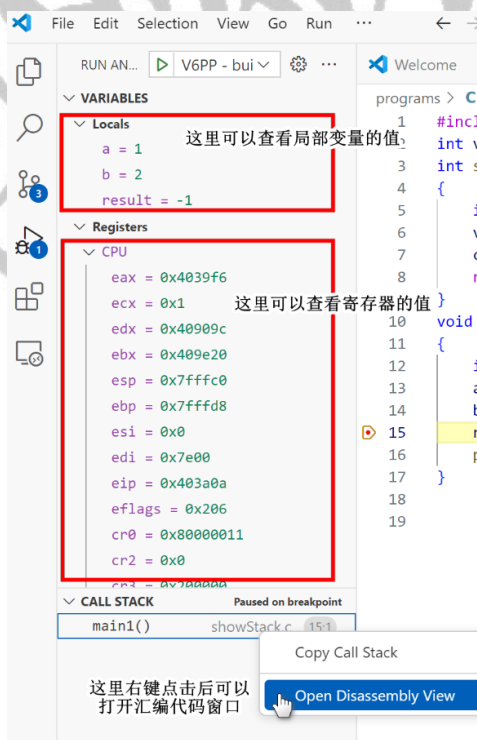


图 9：vscode 调试界面

图 9 中显示了 vscode 调试过程中的几个重要窗口，建议全部打开，以便随时观察程序执行过程中各方面的变化情况。

此时，可以采用实验一中提到的方法查看函数的汇编指令，也可以在终端命令行方式下执行如下命令：

objdump -d showStack --disassembler-options=intel

以 intel 汇编风格查看汇编代码。如图 10 所示，配合 EIP 寄存器的值，可以将当前断点位置和其汇编代码对应起来。

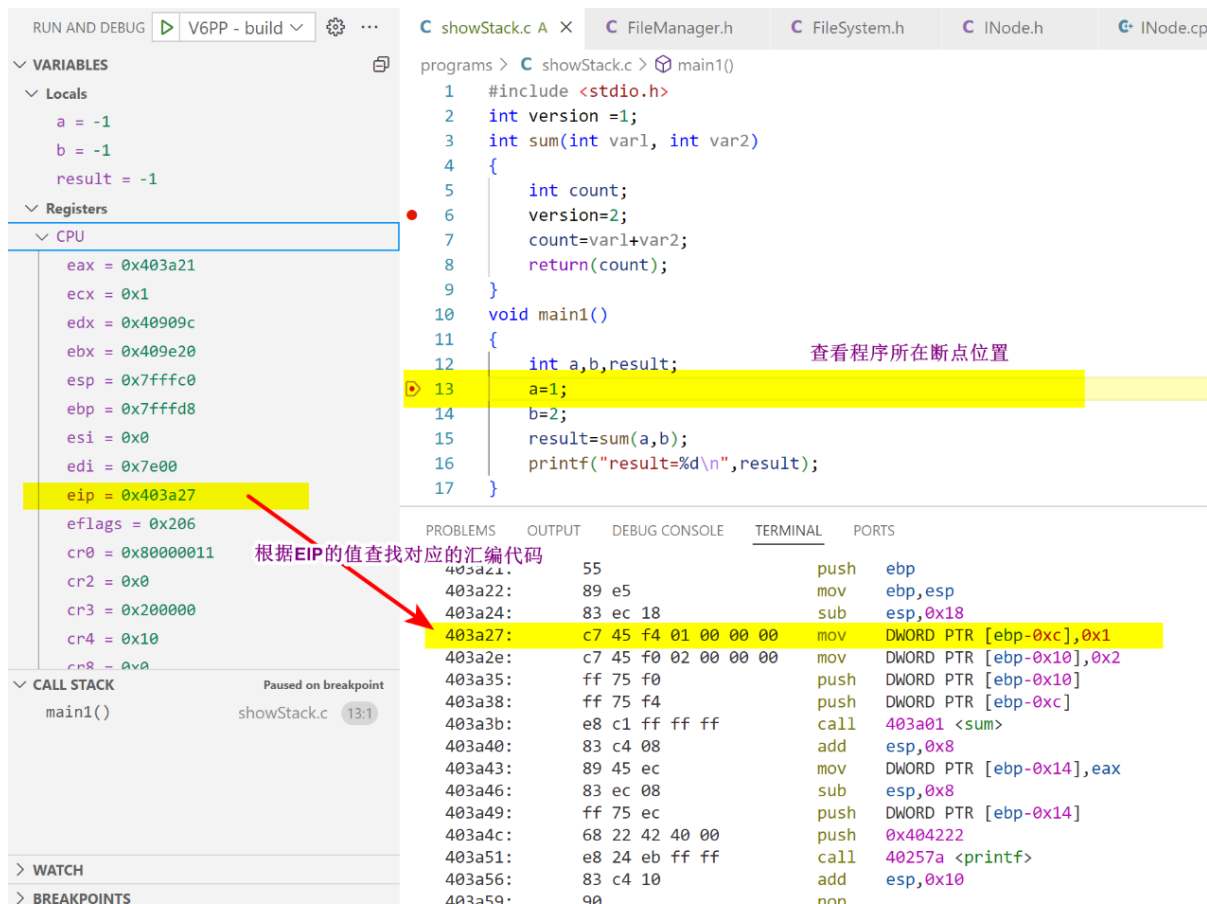


图 10：查看断点和汇编代码

此外，在 vscode 中，可以通过在调试窗口中输入下面这个命令实现对内存单元的查看：

-exec x /nfx addr

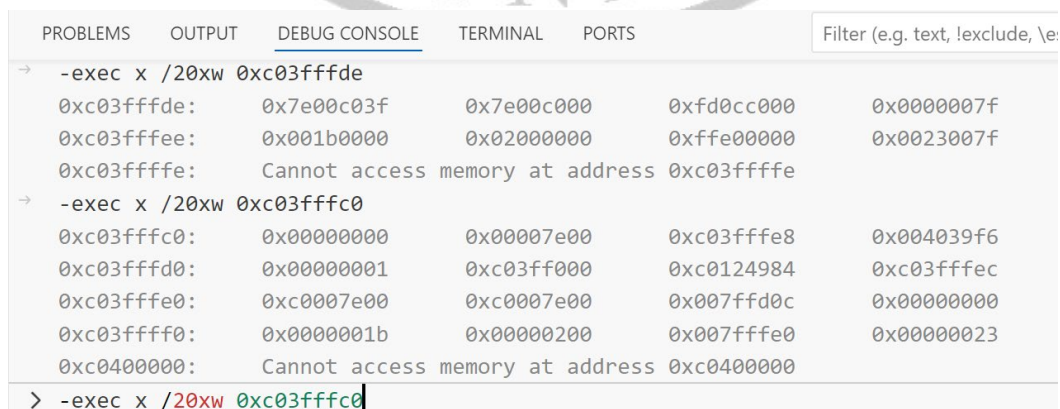


图 11：在 vscode 中查看内存

其中, x 表示要查看内存, nfu 表示含义为以 f 格式打印从 $addr$ 开始的 n 个长度单元为 u 的内存值。例如, 图 11 中, 可以查看从 $0xc03fffc0$ 开始的 20 个内存单元中每 4 个字节的值, 并以 16 进制显示。

查看内存单元在后续多个实验中都非常重要, 请读者务必掌握, 并能够灵活运用。

4.3. 观察 MAIN1 堆栈变化

这一节我们将通过分析汇编指令和查看内存单元, 来观察 C 语言执行过程中, 堆栈随着每一次函数调用的变化。

以主函数 `main1` 为例。如图 12 所示让你的程序停在 `main1` 函数中对 `sum` 函数的调用处, 可以看到此时寄存器 `esp` 和 `ebp` 的值显示了当前用户栈指针的位置。接下来, 我们将通过汇编指令的分析, 结合内存查看, 来确定 `main` 函数执行过程中, 用户栈的变化。

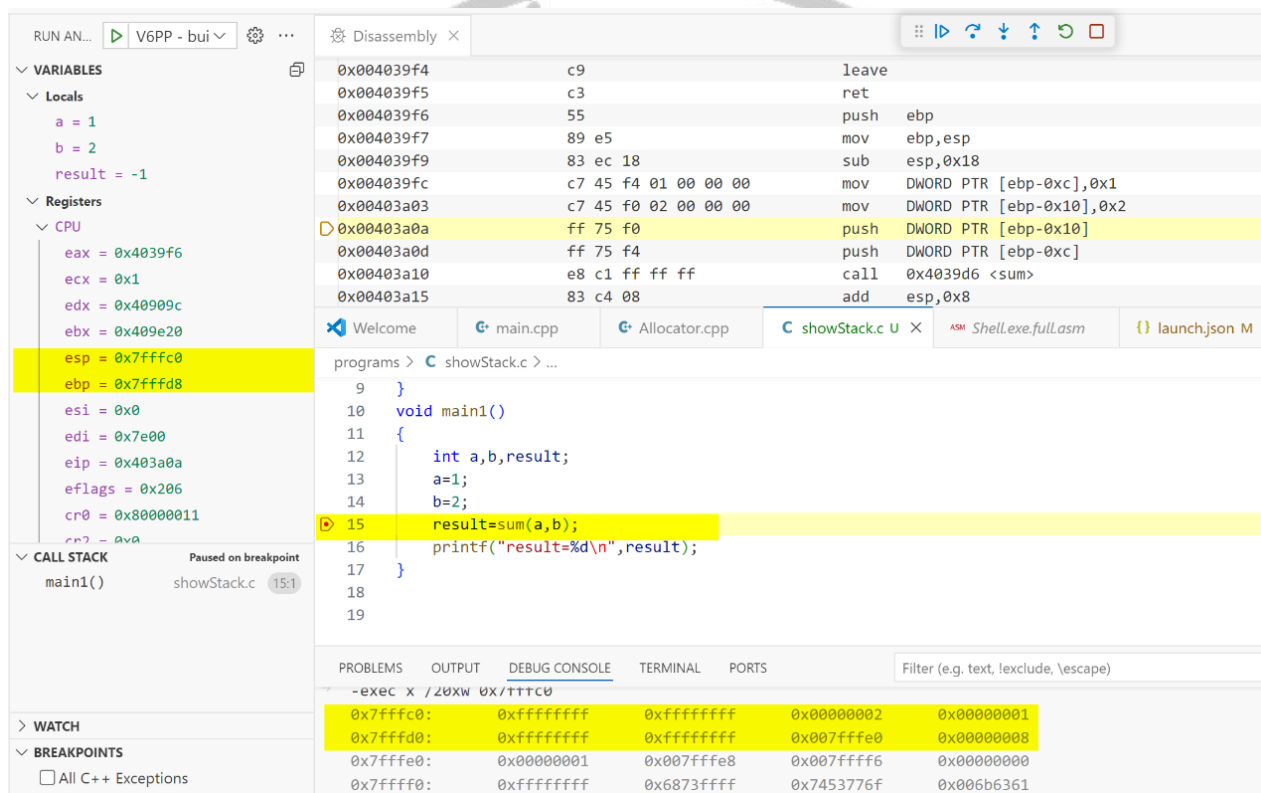


图 12: 观察程序执行过程中寄存器与内存单元值得变化

这里我们已经给 `main1` 函数的大部分指令添加了注释(需要注意的是, 这里形成的汇编指令可能会因编译器的版本不同而有差异)。

```
=====
push ebp      //前一栈帧的 ebp 存入当前栈
mov ebp, esp  //修改 ebp 指向当前栈帧, 此时 EBP=0x007fffd8
sub esp, 0x18 //esp 上移 6 个字, 空出 main 局部变量位置, 此时 ESP=0x007fffc0
```

//以下两句利用栈基址寄存器 EBP 向上访问 main 栈帧中的局部变量

```
mov DWORD PTR [ebp-0xc], 0x1 // main 的局部变量 a 赋值 1
mov DWORD PTR [ebp-0x10], 0x2 // main 的局部变量 b 赋值 2
```

//以下两句为 sum 栈帧准备参数

```

push DWORD PTR [ebp-0x10]      //把 sum 的参数 var2 的值 (2) 压栈
push DWORD PTR [ebp-0xc]      //把 sum 的参数 var1 的值 (1) 压栈

call 0x4039d6 <sum>           //调用 sum 函数，将返回地址压栈

add esp, 0x8
mov DWORD PTR [ebp-0x14], eax
sub esp, 0x8
push DWORD PTR [ebp-0x14]
push 0x406222
call 0x40257a <printf>
add esp, 0x10
nop
leave
ret

```

程序执行过程中，可随时停下来观察汇编指令的执行位置，和此时寄存器及内存单元的值。如图 12 中，当程序停在 `push DWORD PTR [ebp-0x10]` 处时，可以根据查看到的内存单元的值，绘制出如图 13 所示的用户栈状态。可见，此时，`main` 栈帧中局部变量部分已经准备好，但传递给 `sum` 的栈帧部分还没有完成。

地址			
main 栈帧	0x007fffc0	0xffffffff	sub esp, 0x18 后，将 ESP 指向此单元，空出 6 个字的局部变量区域
	0x007fffc4	0xffffffff	
	0x007fffc8	2	mov DWORD PTR [ebp-0x10], 0x2
	0x007ffcc	1	mov DWORD PTR [ebp-0xc], 0x1
	0x007fffd0	0xffffffff	push ebp 此后的 mov ebp, esp 将 EBP 指向此单元
	0x007fffd4	0xffffffff	
	0x007fffd8	007FFFE0	
	0x007fffdc	00000008	main 的返回地址

图 13: `main1` 函数执行过程中栈帧的变化 (1)

同样的方法，在 `call 0x4039d6 <sum>` 语句处添加断点，继续执行并查看内存，可以得到如图 14 的栈帧。

这里需要补充说明以下两点：

(1) 此处编译器具体实现的栈帧和课堂学习的实现方法略有不同，比如：采用 “`sub esp, 0x18`” 指令空出了 24 个字节（6 个字）的栈帧位置留给 `main1` 函数的局部变量。

(2) 图 13-14 中并没有完全画出核心栈的栈底，因为涉及过多 main1 函数被调用和调用 printf 的细节，这里不再展开。

地址			
	0x007fffb8	1	push DWORD PTR [ebp-0xc]
	0x007ffbbc	2	push DWORD PTR [ebp-0x10]
main 栈帧	0x007fffc0	0xffffffff	sub esp, 0x18 后, 将 ESP 指向此单元, 空出 6 个字的局部变量区域
	0x007fffc4	0xffffffff	
	0x007fffc8	2	mov DWORD PTR [ebp-0x10], 0x2
	0x007ffccc	1	mov DWORD PTR [ebp-0xc], 0x1
	0x007ffcd0	0xffffffff	push ebp 此后的 mov ebp, esp 将 EBP 指向此单元
ebp:	0x007ffcd4	0xffffffff	
	0x007ffcd8	007FFFE0	
	0x007ffcdc	00000008	main 的返回地址

图 14: main1 函数执行过程中栈帧的变化 (2)

4.4. 观察 sum 函数变化

请读者参考 4.3 节，给 sum 函数的汇编指令添加详细的注释。通过观察内存单元的值，参考图 14-15 绘制图表，详细说明在 sum 执行的过程中，用户栈的变化。结合你对 sum 函数的分析，尝试回答：在 main1 的汇编代码中，从 sum 返回后执行的指令“add esp, 0x8”的目的是什么？

```
=====
push ebp
mov ebp, esp
sub esp, 0x10
mov DWORD PTR ds:0x4042f4, 0x2
mov edx, DWORD PTR [ebp+0x8]
mov eax, DWORD PTR [ebp+0xc]
add eax, edx
mov DWORD PTR [ebp-0x4], eax
mov eax, DWORD PTR [ebp-0x4]
leave
ret
=====
```


5. 实验报告要求

本次实验报告需完成以下内容：

(1) (1 分) 参考实验指导完成实验 4.1~4.2，掌握在 UNIX V6++中添加自定义程序及编译、链接与运行的全过程，掌握 UNIX V6++中调试运行与观察结果的常规操作，截图说明上述过程。

(2) (1 分) 复现实验 4.3 中 main1 函数核心栈的变化，通过观察内存单元的值验证核心栈的变化。

(3) (1 分) 完成实验 4.4，通过观察内存单元的值，参考图 14-15 绘制图表，详细说明在 sum 执行的过程中，用户栈的变化。结合你对 sum 函数的分析，回答：在 main1 的汇编代码中，从 sum 返回后执行的指令“add esp, 0x8”的目的是什么？

(4) (1 分) 在 sum 的汇编代码中，“mov DWORD PTR ds:0x4042f4, 0x2”的作用是什么？结合课堂学习的知识，尝试解释 ds:0x4042f4 这个地址对应的是什么？为什么？

