

同济大学计算机系

操作系统实验报告



实验内容 UNIX V6++进程的栈帧

学 号 2251745

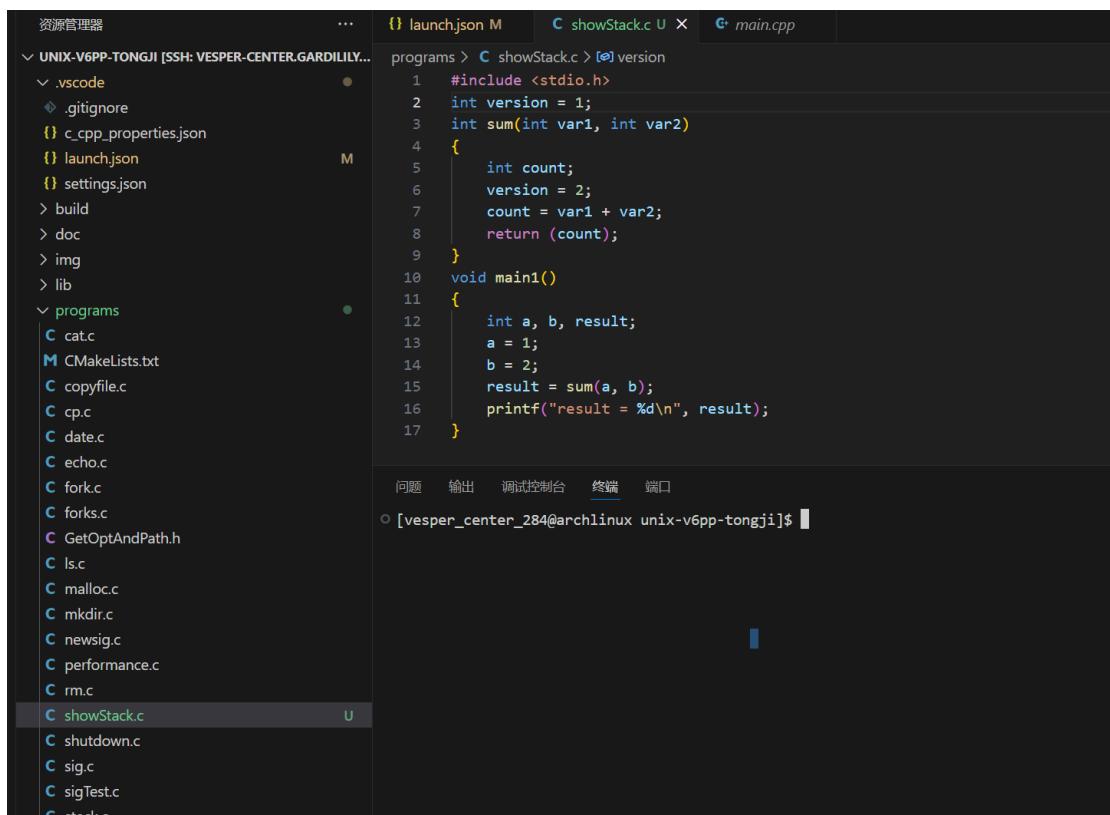
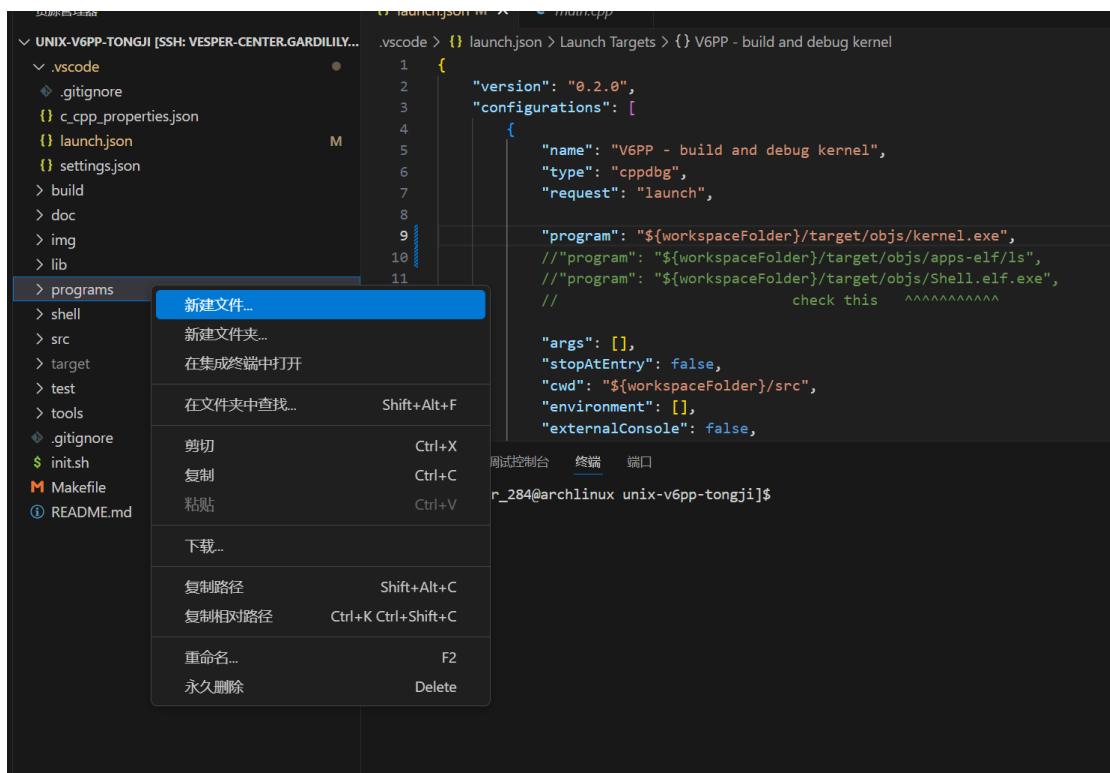
姓 名 张宇

专 业 计算机科学与技术

授课老师 方钰

# 一、UNIX V6++自定义程序的添加、编译、连接和运行

## 1.1 在 program 文件加入一个新的 c 语言文件



## 1.2 重新编译运行 UNIX V6++代码

问题    输出    调试控制台    终端    端口

```
[bin] > [info 5] 上传成功: performance
[bin] > [info 5] 上传成功: trace
[bin] > [info 5] 上传成功: cp
[bin] > [info 5] 上传成功: date
[bin] > [info 5] 上传成功: forks
[bin] > [info 5] 上传成功: malloc
[bin] > [info 5] 上传成功: cat
[bin] > [info 5] 上传成功: sig
[bin] > [info 5] 上传成功: shutdown
[bin] > [info 5] 上传成功: echo
[bin] > [info 5] 上传成功: testSTDOUT
[bin] > [info 5] 上传成功: copyfile
[bin] > [info 5] 上传成功: newsig
[bin] > [info 5] 上传成功: ls
[bin] > [info] 切换路径。
[bin/..] > [info 9] 创建文件夹: etc
[bin/..] > [info] 切换路径。
[bin/../../etc] > [info 5] 上传成功: v6pp_splash.bmp
[bin/../../etc] > [info] 切换路径。
[bin/../../etc/..] > [info 5] 上传成功: Shell.exe
[bin/../../etc/..] > bye!
cp target/img-workspace/c.img target/
build success (unix-v6pp-tongji).
○ [vesper_center_284@archlinux unix-v6pp-tongji]$ █
(w) 0
```

### 1.3 关于 UNIX V6++的调试目标

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
      forks malloc cat   sig   shutdown      echo   testSTDOUT      copyfile news
ig     ls
[/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
```

```

{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "V6PP - build and debug kernel",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/target/objs/kernel.exe",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}/src",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerServerAddress": "${workspaceFolder}/target/qemu-gdb.sock",
            "setupCommands": [
                {"text": "source ./target/elf/debug/Shell", "ignoreFailures": true}
            ]
        }
    ]
}

```

## 1.4 开始程序的调试运行

```

#include <stdio.h>
int version = 1;
int sum(int var1, int var2)
{
    int count;
    version = 2;
    count = var1 + var2;
    return (count);
}
void main()
{
    int a, b, result;
    a = 1;
    b = 2;
    result = sum(a, b);
    printf("result = %d\n", result);
}

```

Registers CPU

eax = 0x4039f6	c3	ret
ecx = 0x1	55	push ebp
edx = 0x40909c	89 e5	mov ebp,esp
ebx = 0x409e20	83 ec 18	sub esp,0x18
esp = 0x7fffcc	c7 45 f4 01 00 00 00	mov DWORD PTR [ebp-0xc],0x1
ebp = 0x7ffd8	c7 45 f0 02 00 00 00	mov DWORD PTR [ebp-0x10],0x2
esi = 0x0	ff 75 f0	push DWORD PTR [ebp-0x10]
edi = 0x7e00	ff 75 f4	push DWORD PTR [ebp-0xc]
eip = 0x403a0a	e8 c1 ff ff ff	call 0x4039d6 <sum>
eflags = 0x206	83 c4 08	add esp,0x8
cr0 = 0x80000011	89 45 ec	mov DWORD PTR [ebp-0x14],eax
cr2 = 0x0	83 ec 08	sub esp,0x8
cr3 = 0x200000	ff 75 ec	push DWORD PTR [ebp-0x14]
cr4 = 0x10	68 22 62 40 00	push 0x406222
cr8 = 0x0	e8 4f eb ff ff	call 0x40257a <printf>

Stack

0x00403a03	83 c4 10	add esp,0x10
0x00403a04	90	nop
0x00403a05	c3	ret
0x00403a06	ff	(bad)
0x00403a07	ff	(bad)
0x00403a08	ff	(bad)
0x00403a09	ff 00	inc DWORD PTR [eax]

The screenshot shows a debugger interface with two main panes. The left pane displays the 'Locals' and 'Registers' sections of the stack dump. The right pane shows the source code of `showStack.c` with line 15 highlighted.

**Locals:**

- a = 1
- b = 2
- result = -1

**Registers (CPU):**

- eax = 0x4039f6
- ecx = 0x1
- edx = 0x40909c
- ebx = 0x409e20
- esp = 0x7ffffc0
- ebp = 0x7ffffd8
- esi = 0x0
- edi = 0x7e00
- eip = 0x403a0a
- eflags = 0x206
- cr0 = 0x80000011
- cr2 = 0x0
- cr3 = 0x200000
- cr4 = 0x10
- cr8 = 0x0

**Source Code (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 }
```

The screenshot shows a terminal window titled 'QEMU [Paused]'. The window displays the following text:

```
welcome to Unix V6++ Tongji's Edition!
[/]#cd bin
[bin]#showStack
```

At the bottom of the terminal window, there is additional text from another process:

```
Process 1 execing
Process 1 finding dead son. They are Process 2 (Status:3) wait until child process Exit! Process 2
execing
```

```

0x004039f5      c3          ret
0x004039f6      55          push  ebp
0x004039f7      89 e5       mov   ebp,esp
0x004039f9      83 ec 18    sub   esp,0x18
0x004039fc      c7 45 f4 01 00 00 00  mov   DWORD PTR [ebp-0xc],0x1
0x00403a03      c7 45 f0 02 00 00 00  mov   DWORD PTR [ebp-0x10],0x2
D 0x00403a0a     ff 75 f0       push  DWORD PTR [ebp-0x10]
0x00403a0d      ff 75 f4       push  DWORD PTR [ebp-0xc]
0x00403a10      e8 c1 ff ff ff  call  0x4039d6 <sum>
0x00403a15      83 c4 08       add   esp,0x8
0x00403a18      89 45 ec       mov   DWORD PTR [ebp-0x14],eax
0x00403a1b      83 ec 08       sub   esp,0x8
0x00403a1e      ff 75 ec       push  DWORD PTR [ebp-0x14]
0x00403a21      68 22 62 40 00  push  0x406222
0x00403a26      e8 4f eb ff ff  call  0x40257a <printf>
0x00403a2b      83 c4 10       add   esp,0x10
0x00403a2e      90          nop
0x00403a2f      c9          leave
0x00403a30      c3          ret
0x00403a31      ff          (bad)
0x00403a32      ff          (bad)
0x00403a33      ff          (bad)
0x00403a34      ff 00       inc   DWORD PTR [eax]

```

问题 调试 控制台 终端 端口

0xc03ffffc0:	0x00000000	0x00007e00	0xc03ffffe8	0x004039f6
0xc03ffffd0:	0x00000001	0xc03ff000	0xc0124984	0xc03ffffec
0xc03ffffe0:	0xc0007e00	0xc0007e00	0x007ffd0c	0x00000000
0xc03fffff0:	0x0000001b	0x00000200	0x007ffffe0	0x00000023
0xc0400000:	Cannot access memory at address 0xc0400000			
→ -exec x /20xw 0xc03ffffc0	0x00000000	0x00007e00	0xc03ffffe8	0x004039f6
0xc03ffffc0:	0x00000000	0x00007e00	0xc03ffffe8	0x004039f6
0xc03ffffd0:	0x00000001	0xc03ff000	0xc0124984	0xc03ffffec
0xc03ffffe0:	0xc0007e00	0xc0007e00	0x007ffd0c	0x00000000
0xc03fffff0:	0x0000001b	0x00000200	0x007ffffe0	0x00000023
0xc0400000:	Cannot access memory at address 0xc0400000			
→ -exec x /20xw 0xc03ffffc0	0x00000000	0x00007e00	0xc03ffffe8	0x004039f6
0xc03ffffc0:	0x00000000	0x00007e00	0xc03ffffe8	0x004039f6
0xc03ffffd0:	0x00000001	0xc03ff000	0xc0124984	0xc03ffffec
0xc03ffffe0:	0xc0007e00	0xc0007e00	0x007ffd0c	0x00000000
0xc03fffff0:	0x0000001b	0x00000200	0x007ffffe0	0x00000023
0xc0400000:	Cannot access memory at address 0xc0400000			

运行规则 D V6PP - build and de ...

变量 Locals Registers CPU

```

program > C showStack.c > main1()
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 }

```

问题 调试 控制台 终端 端口

0x004039e9:	8b 45 0c	mov eax,DWORD PTR [ebp+0xc]
0x004039ec:	01 d0	add eax,edx
0x004039e9:	89 45 fc	mov DWORD PTR [ebp-0x4],eax
0x004039ec:	00	mov eax,DWORD PTR [ebp-f]
0x004039f5:	c9	leave
0x004039f6:	55	push ebp
0x004039f7:	89 e5	mov ebp,esp
0x004039f9:	83 ec 18	sub esp,0x18
0x004039fc:	c7 45 f4 01 00 00 00	mov DWORD PTR [ebp-0xc],0x1
0x00403a03:	c7 45 f0 02 00 00 00	mov DWORD PTR [ebp-0x10],0x2

Breakpoint 3, main1 () at /home/vesper\_center\_284/unix-v6pp-tongji/progs/showStack.c:15
15 result = sum(a, b);

Execute debugger commands using "-exec <command>", for example "-exec info registers" will list registers in use (when GDB is the debugger)

-exec 20xw 0xc03ffffc0

0x7ffffc0: 0xffffffff 0xffffffff 0x00000002 0x00000001

0x7ffffd0: 0xffffffff 0xffffffff 0x00000000 0x00000000

0x7ffffe0: 0x00000001 0x00000001 0x00000000 0x00000000

0x7fffff0: 0xffffffff 0xffffffff 0x7453776f 0x0000b6361

0x8000000: Cannot access memory at address 0x8000000

## 二、复现实验 4.3 中 main1 函数核心栈的变化

### 2.1 存前一栈帧的 ebp，修改 ebp 指向当前栈帧，esp 上移

main1:	
00401000:	push %ebp
00401001:	mov %esp,%ebp
00401003:	sub \$0x18,%esp
eax	4198400
ecx	1
edx	4223024
ebx	4226468
esp	0x007fffc0
ebp	0x007ffffd8
esi	917504
edi	65452
eip	0x00401006
eflags	[ PF IF ]
◆ 0x007fffc0	Address 0 - 3 4 - 7 8 - B C - F
◆ 0x007ffffdc	007FFFC0 00000000 00000000 00000000 00000000 007FFFFD0 00000000 00000000 ▲ E0FF7F00 08000000

内存单元的观察：观察可知为小端存储，在 ebp 指向的 0x007ffffd8 单元存储着上一栈帧基址 007FFE0，0x007ffffdc 存储着 main 的返回地址 00000008，并空出了局部变量和参数的值。

## 2.2 将 main 的局部变量送入栈中

8	a=1;
00401006:	mov 0x8(%ebp),%eax
00401009:	mov %eax,(%esp)
0040100c:	call 0x402129 <ftoa+115>
9	b=2;
0040100d:	sbb %dl,(%ecx)
0040100f:	add %al,(%eax)
00401011:	mov %eax,-0x4(%ebp)
eax	4198400
ecx	1
edx	4223024
ebx	4226468
esp	0x007fffc0
ebp	0x007ffffd8
esi	917504
edi	65452
eip	0x00401014
eflags	[ PF IF ]
cs	27
ss	35
ds	35
es	35
fs	0
◆ 0x007fffc0	Address 0 - 3 4 - 7 8 - B C - F
◆ 0x007ffffdc	007FFFC0 00000000 00000000 00000000 00000000 007FFFFD0 ▲ 02000000 01000000 E0FF7F00 08000000 007FFE0 01000000 E8FF7F00 F2FF7F00 00000000 007FFF0 00007368 6F775374 61636B2E 65786500

内存单元的观察：可以看到值 2 和 1 已经被放入栈中

## 2.3 将参数放入栈中

```

00401014: mov -0x4(%ebp), %eax
00401017: add 0x8(%ebp), %eax
0040101a: dec %eax
0040101b: mov %eax, -0x8(%ebp)
0040101e: mov -0x8(%ebp), %eax

```

Name	Value
>Main	
eax	1
ecx	1
edx	4223024
ebx	4226468
esp	0x007ffffb4
ebp	0x007ffffb8
esi	917504
edi	65452
eip	0x00401044
eflags	[ PF IF ]
cs	27
ss	25

	Address	0 - 3	4 - 7	8 - B	C - F
0x007ffffc0	007FFFC0	01000000	02000000	00000000	00000000
0x007ffffdc	007FFFD0	02000000	01000000	E0FF7F00	08000000
	007FFFE0	01000000	E8FF7F00	F2FF7F00	00000000
	007FFFF0	00007368	6F775374	61636B2E	65786500
	00800000	01000000	02000000	00000000	00000000
	00800010	02000000	01000000	E0FF7F00	08000000

内存单元的观察：参数已经被放入栈中

## 2.4 调用 sum 函数并返回

```

sum:
0040103e:    ret
0040103f:    push %ebp
00401040:    mov %esp,%ebp
00401042:    sub $0x4,%esp
17           version=2;
00401044:    add $0xc7,%al
00401046:    inc %ebp
00401047:    cld
00401048:    add %al,(%eax)
0040104a:    add %al,(%eax)
0040104c:    movl $0x0,0x407004
18           count=var1+var2;
0040104e:    add $0x70,%al
00401050:    inc %eax
00401051:    add %al,(%eax)
00401053:    add %al,(%eax)
00401055:    add %al,%bh
19           return(count);
00401057:    inc %ebp
00401058:    cld
00401059:    cld
20           }
0040105a:    leave
0040105b:    ret

```

寄存器观察窗口 (Registers) 显示了以下内容：

	Main
eax	3
ecx	1
edx	4223024
ebx	4226468
esp	0x007fffc0
ebp	0x007ffd8
esi	917504
edi	65452
eip	0x00401029
eFLAGS	PF IF 1

内存单元观察窗口 (Memory) 显示了以下内容：

	Address	0 - 3	4 - 7	8 - B	C - F
0x007fffc0	007FFFC0	01000000	02000000	00000000	03000000
0x007ffd8	007FFF80	02000000	01000000	E0FF7F00	08000000
	007FFFE0	01000000	E8FF7F00	F2FF7F00	00000000

内存单元的观察：可以看到为 result 预留出来的值变为了 3

2.5 打印，结果如下

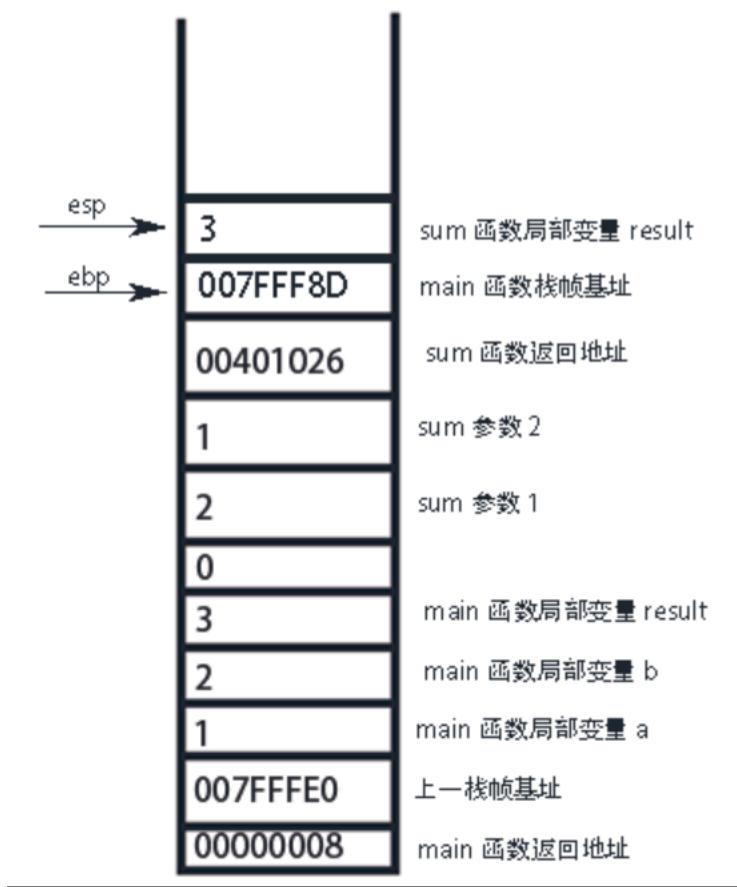
```
welcome to Unix V6++ Tongji's Edition!
[/]#cd bin
[/bin]#showStack
result = 3
[/bin]#
```

三、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？

3.1 给 sum 函数的汇编指令添加详细的注释

```
push ebp          // 1. 保存当前的栈帧基址（旧的 EBP），保护调用者的栈帧
mov ebp, esp      // 2. 将当前栈顶指针 ESP 赋值给 EBP，创建新的栈帧
sub esp, 0x10      // 3. 向栈中分配 16 字节 (0x10) 的局部变量空间，ESP 向下移动
mov DWORD PTR ds:0x4042f4, 0x2 // 4. 将数值 2 存入全局变量或静态数据段的地址 0x4042f4
mov edx, DWORD PTR [ebp+0x8] // 5. 从栈帧中读取第一个参数 (arg1)，存入寄存器 EDX (EBP+0x8 指向第一个参数)
mov eax, DWORD PTR [ebp+0xc] // 6. 从栈帧中读取第二个参数 (arg2)，存入寄存器 EAX (EBP+0xc 指向第二个参数)
add eax, edx       // 7. 将 EAX 中的第二个参数值与 EDX 中的第一个参数相加，结果保存在 EAX 中
mov DWORD PTR [ebp-0x4], eax // 8. 将相加后的结果存入局部变量区域 (EBP-0x4 表示 sum 函数的局部变量)
mov eax, DWORD PTR [ebp-0x4] // 9. 将局部变量中的计算结果加载回寄存器 EAX，准备返回
leave              // 10. 恢复调用者的栈帧 (等价于 mov esp, ebp; pop ebp)
ret                // 11. 从栈中弹出返回地址并跳转到返回地址处继续执行
```

3.2 完整的栈帧绘制



### 3.3 问题回答

在 `main1` 的汇编代码中，调用 `sum` 函数后，栈中还留有两个参数 `arg1` 和 `arg2`。由于在调用 `sum` 函数时，调用者会将参数压入栈中，这两个参数在 `sum` 函数返回后仍然存在于栈中。

执行 `add esp, 0x8` 的作用是调整栈顶指针 `ESP`，从而将 `sum` 函数使用的两个参数从栈中弹出，恢复调用 `sum` 之前的栈状态。这样可以避免栈顶指针错位，确保后续的栈操作不会受到这些参数的影响。

**四、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？**

作用是将常数 `0x2`（十进制值为 `2`）存储到内存地址 `0x4042f4` 所指向的内存单元中。

`ds:0x4042f4` 这个地址是全局变量所在的地址。它位于数据段（`data segment`，

通常通过 `ds` 寄存器访问) 中

原因:

**全局变量存储在 `.data` 段:** 根据编译器的工作原理, 全局变量通常分配在程序的数据段 (`.data`) 中, 它们的地址在编译时已经确定, 并在整个程序生命周期中保持不变。程序可以通过直接访问内存地址 (例如 `0x4042f4`) 来读取或修改全局变量的值。

**全局变量的生命周期:** 全局变量的生命周期是整个程序的执行周期, 这意味着在程序执行期间, 它们的值会一直存在。这个特性使得全局变量可以在不同的函数间共享, 并且可以在任何地方通过其固定的内存地址来访问。

**编译时分配的地址:** 全局变量的地址通常由编译器分配。当程序编译时, 编译器会将所有全局变量分配到数据段的特定地址中。`0x4042f4` 很可能是程序的全局变量存储区的一部分。