

## 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，观察程序运行时核心栈的变化。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 UNIX V6++运行和调试环境的 PC 机一台。

### 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~图 2 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 3 所示的代码。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) 修改编译需要使用的 Makefile 文件

为了完成上述测试程序源文件的编译，需要修改 `program` 文件夹下的 `Makefile` 文件。需要修改的两个地方分别如图 4 和图 5 所示。

##### (3) 重新编译运行 UNIX V6++代码

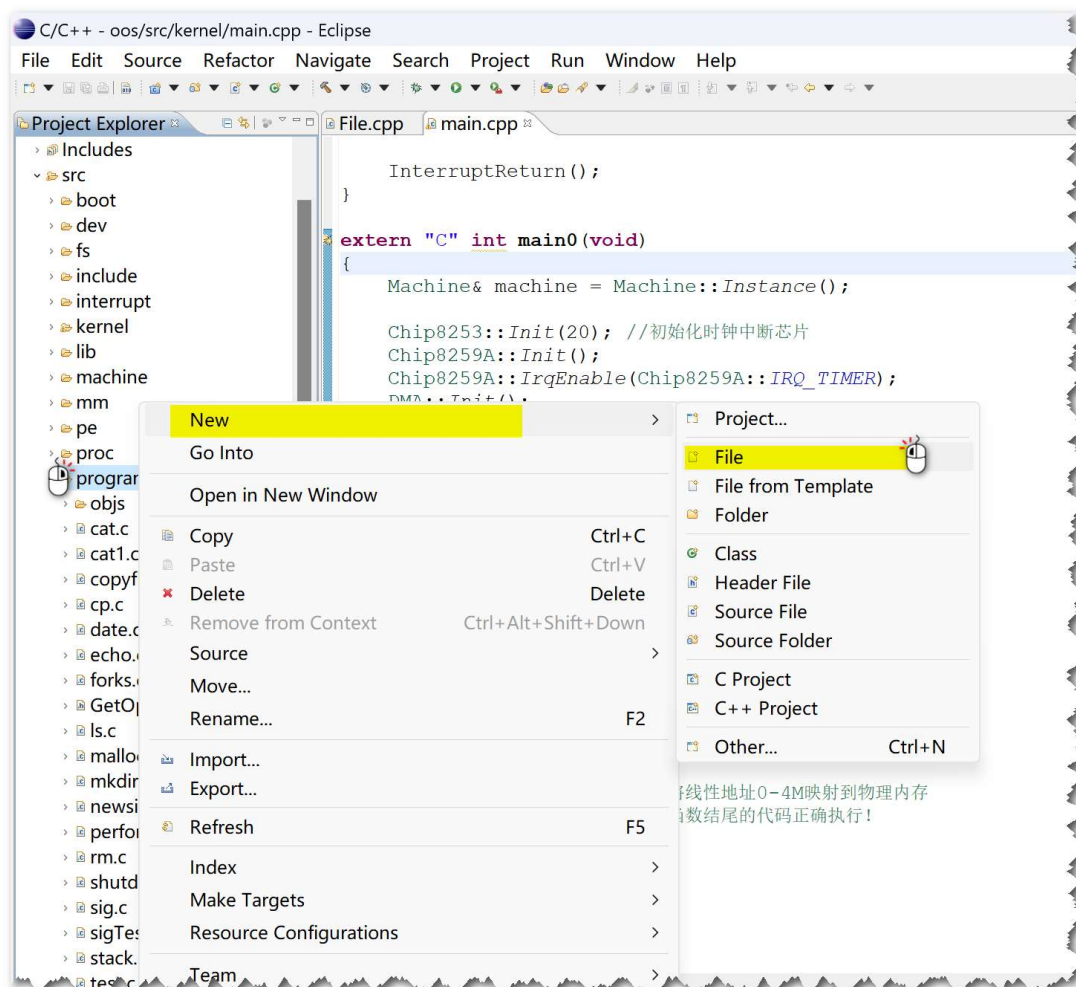


图 1：在 program 下新建一个文件

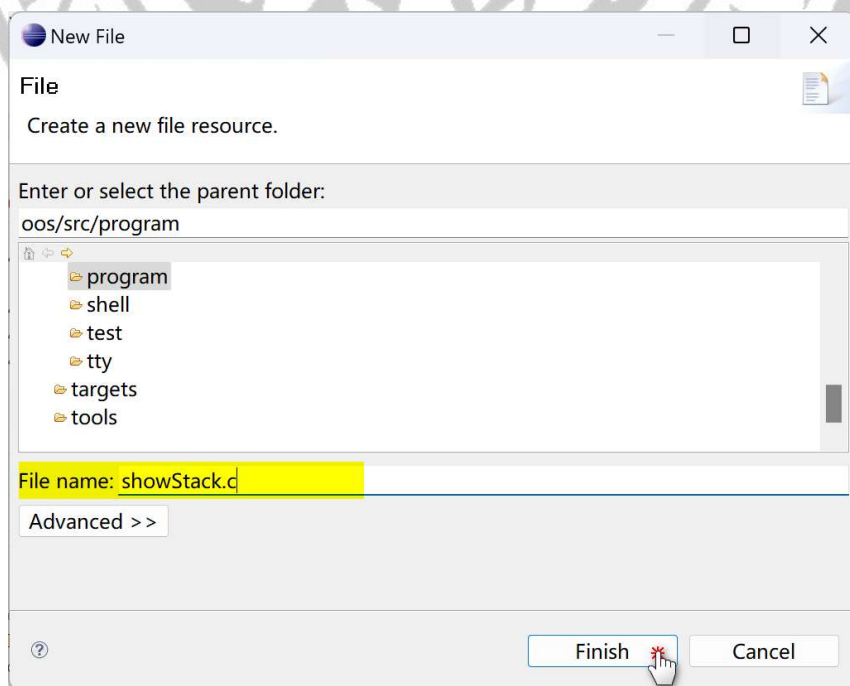


图 2：为新建的文件命名

```

showStack.c
#include <stdio.h>

int version = 1;

main1()
{
    int a,b,result;
    a=1;
    b=2;
    result=sum(a,b);
    printf("result=%d\n",result);
}

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

```

图 3: 示例代码

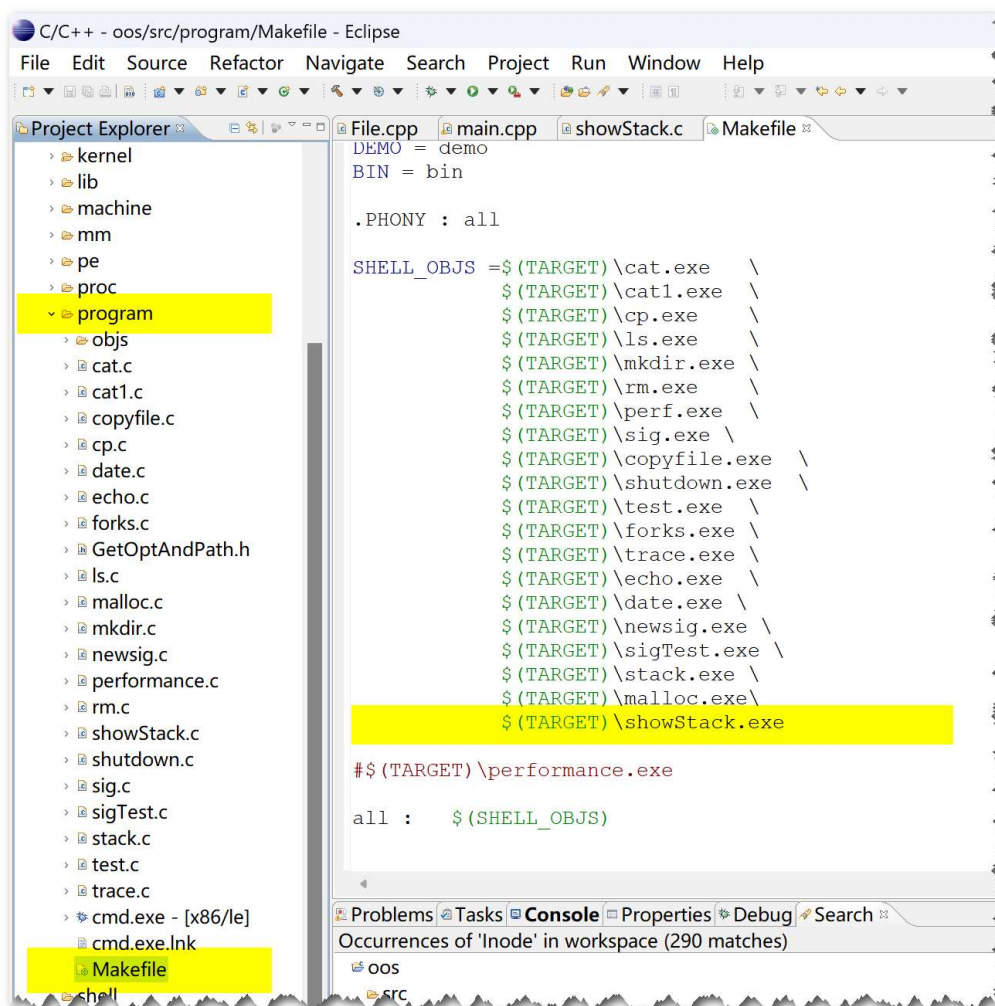


图 4: program 文件夹下的 Makefile 需要修改的地方

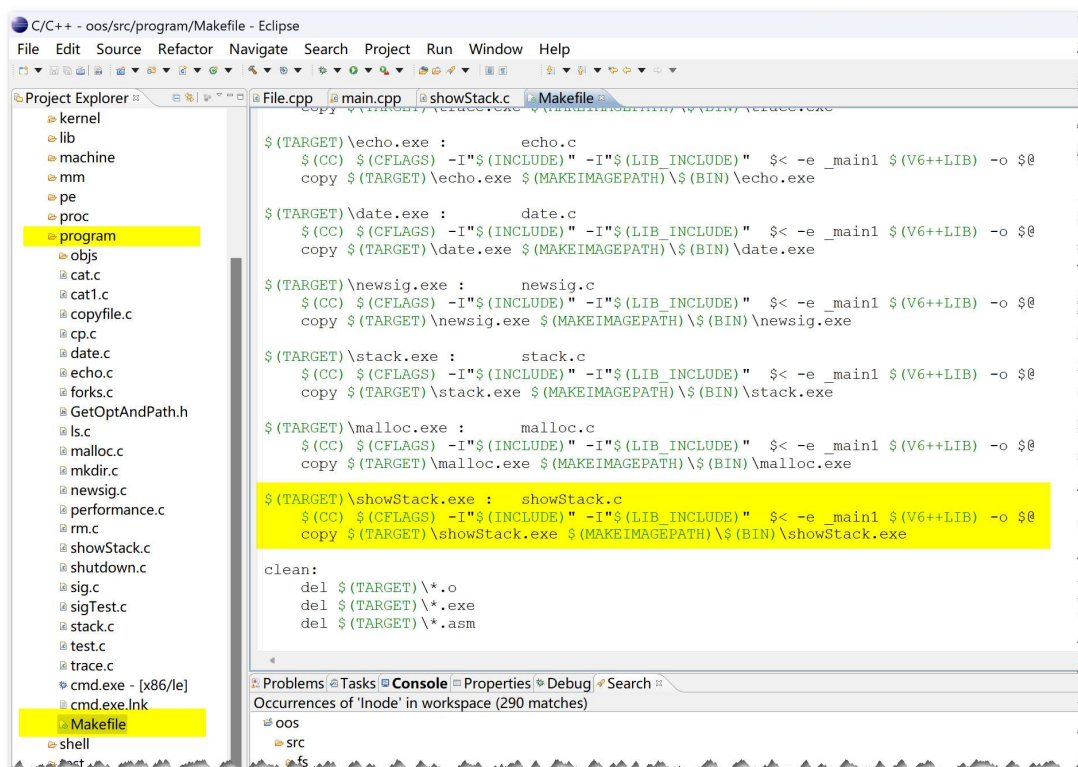


图 5: program 文件夹下的 Makefile 需要修改的地方

在 eclipse 中选择“project” - “Build All”，完成 UNIX V6++代码的重新编译，如图 6 所示。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.exe（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.exe，可以看到程序的运行结果（如图 7 所示）。

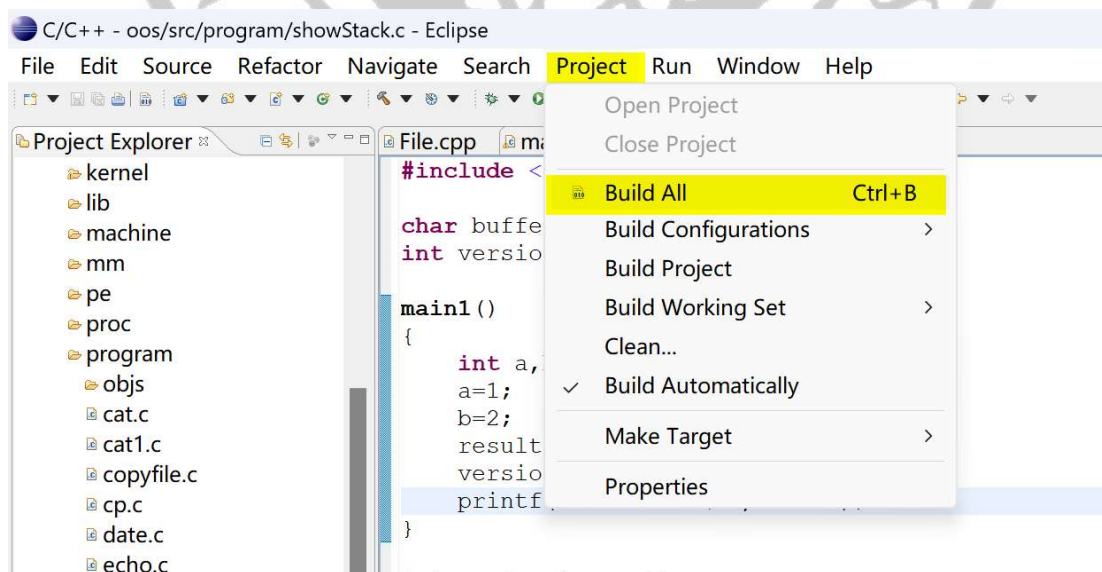
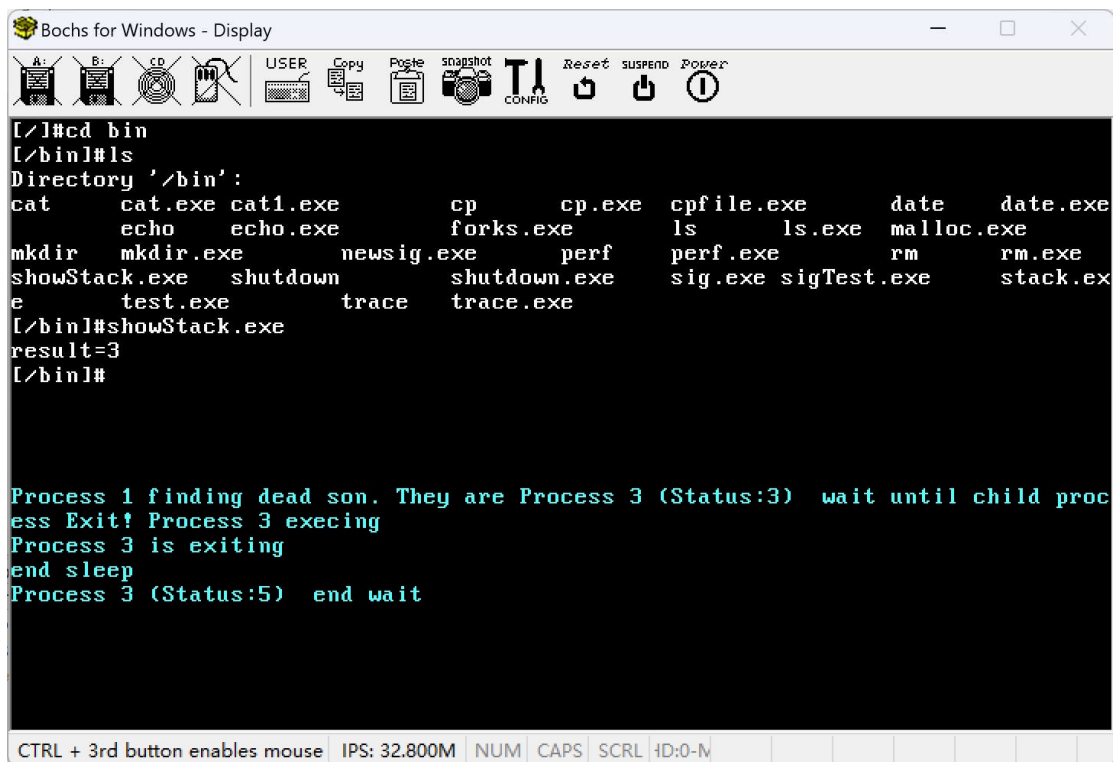


图 6: 重新编译 UNIX V6++的源代码



```

Bochs for Windows - Display
[~/1#cd bin
[/bin1#ls
Directory '/bin':
cat      cat.exe cat1.exe      cp      cp.exe  cpfile.exe  date      date.exe
          echo      echo.exe  forks.exe  ls      ls.exe  malloc.exe
mkdir    mkdir.exe shutdown  newsig.exe perf     perf.exe   rm        rm.exe
showStack.exe shutdown  shutdown.exe sig.exe sigTest.exe stack.exe
e        test.exe  trace      trace.exe
[/bin1#showStack.exe
result=3
[/bin1#

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

```

CTRL + 3rd button enables mouse | IPS: 32.800M | NUM | CAPS | SCRL | HD:0-N

图 7: showStack.exe 程序运行结果




## 4.2. 完成程序的调试运行

### (1) 关于 UNIX V6++ 的调试对象和调试起点

在实验一中配置 UNIX V6++ 调试环境的时候，我们曾经完成了对调试对象和调试起点的设置，当时我们将调试对象设置为 Kernel.exe，起点设置为 main0，是因为我们希望从 main0 开始调试内核。而本实验中，我们希望调试的是自己编写的程序而非内核，所以需要将该设置修改。如图 8~图 10 所示，将调试对象设置为 showStack.exe 程序，调试起点设置为该程序的入口 main1（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）。

### (2) 开始调试

再次以调试模式启动 UNIX V6++。当 UNIX V6++ 启动成功，等待调试指令时，点击工具条上的 ，开始调试。

在 UNIX V6++ 环境下完成“cd bin ”和“showStack.exe ”的执行（如图 11），在此过程中，UNIX V6++ 可能会多次暂停执行，可直接点击工具条上的  按钮，直到程序停在 main1 处，如图 12 所示。



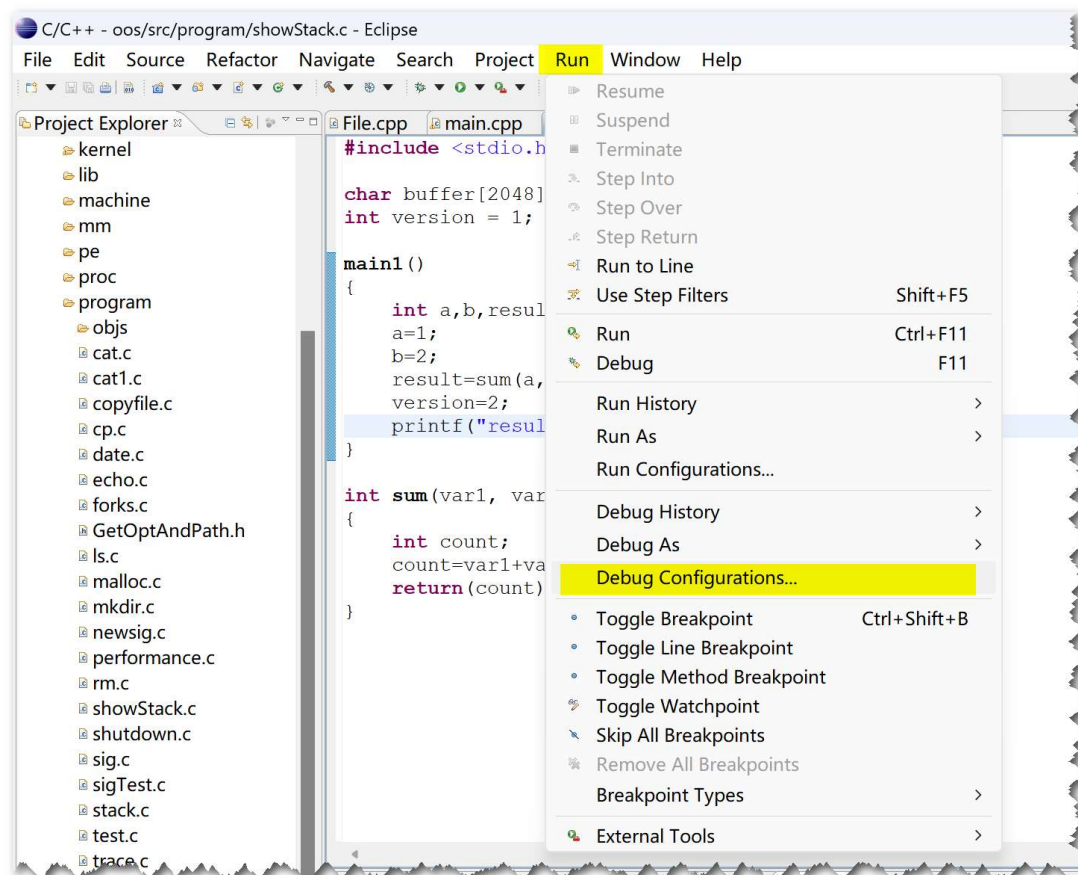


图 8：进入调试环境设置

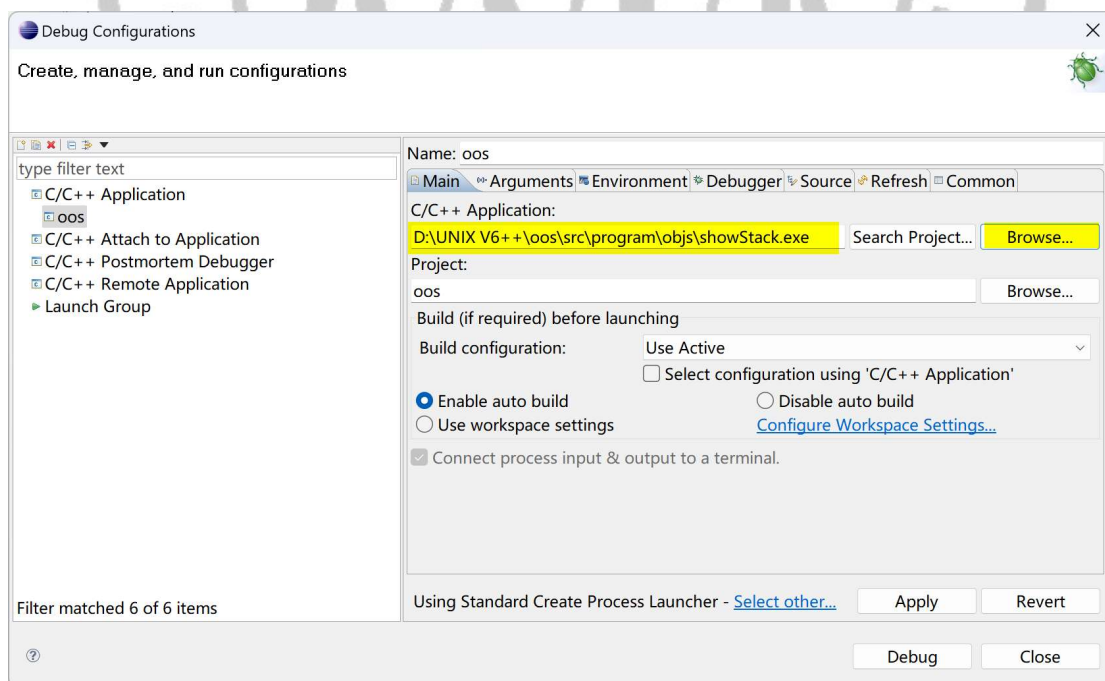


图 9：修改调试对象

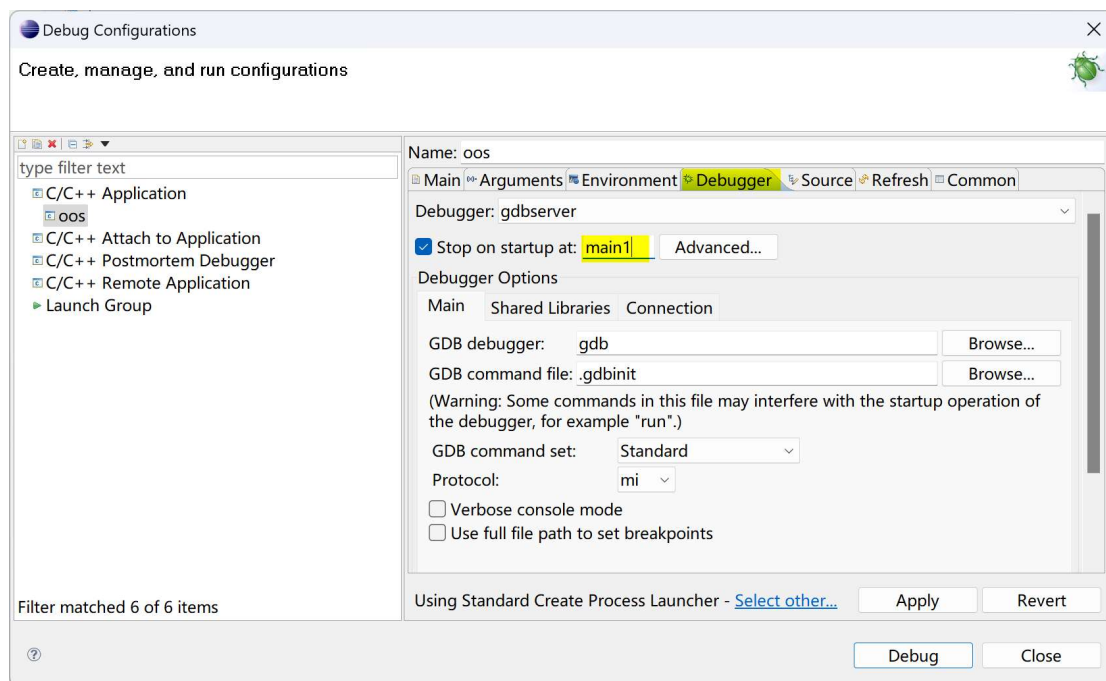


图 10: 修改调试起点

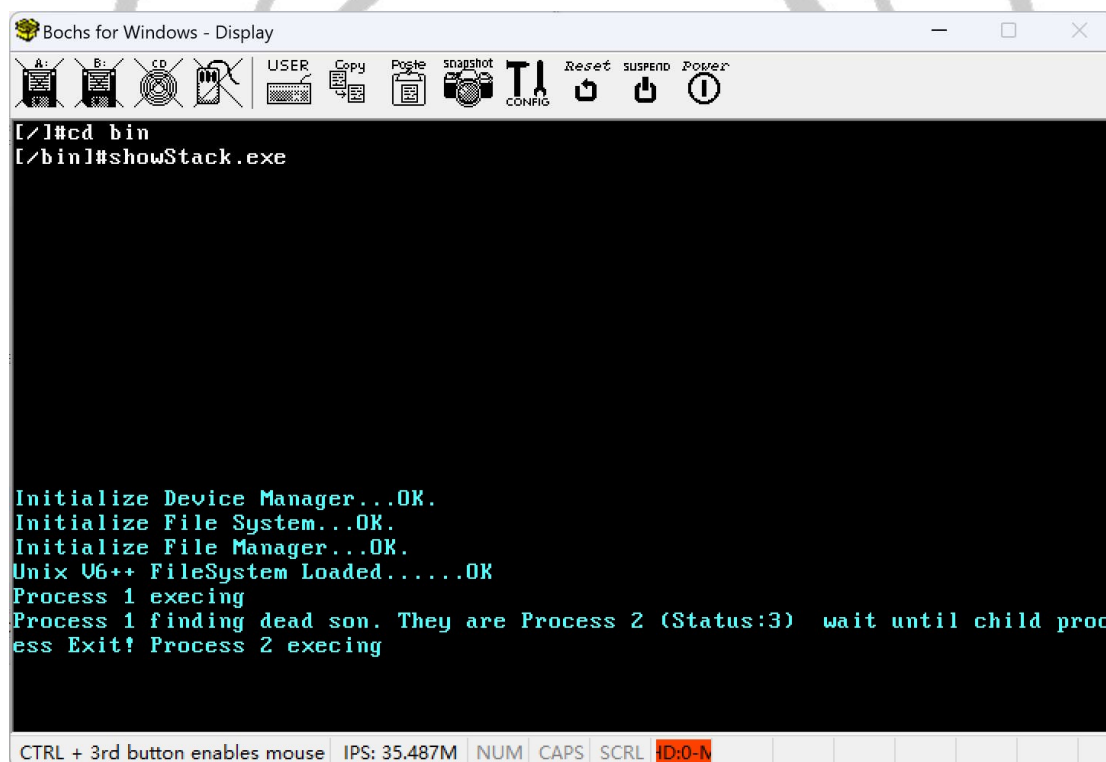


图 11: UNIX V6++调试界面

图 12 中显示了 Eclipse 调试过程中的几个重要窗口，建议全部打开，以便随时程序执行过程中各方面的变化情况。窗口的格局可按个人喜好摆放。找不到的，可以从菜单栏寻找添加，如图 13 所示。

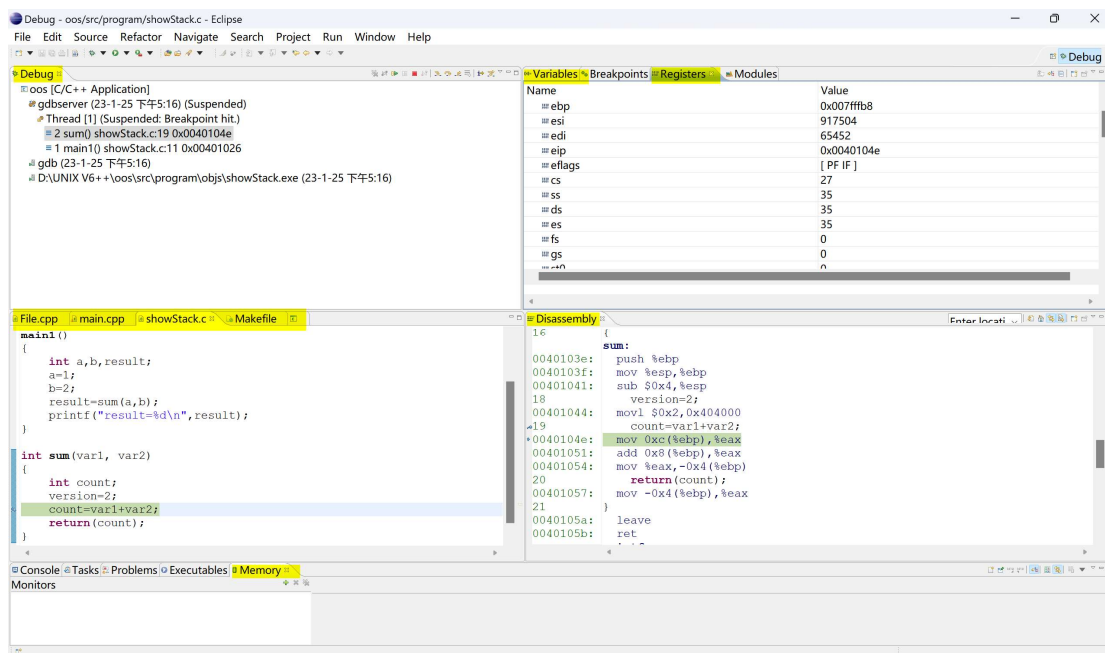


图 12: Eclipse 调试界面

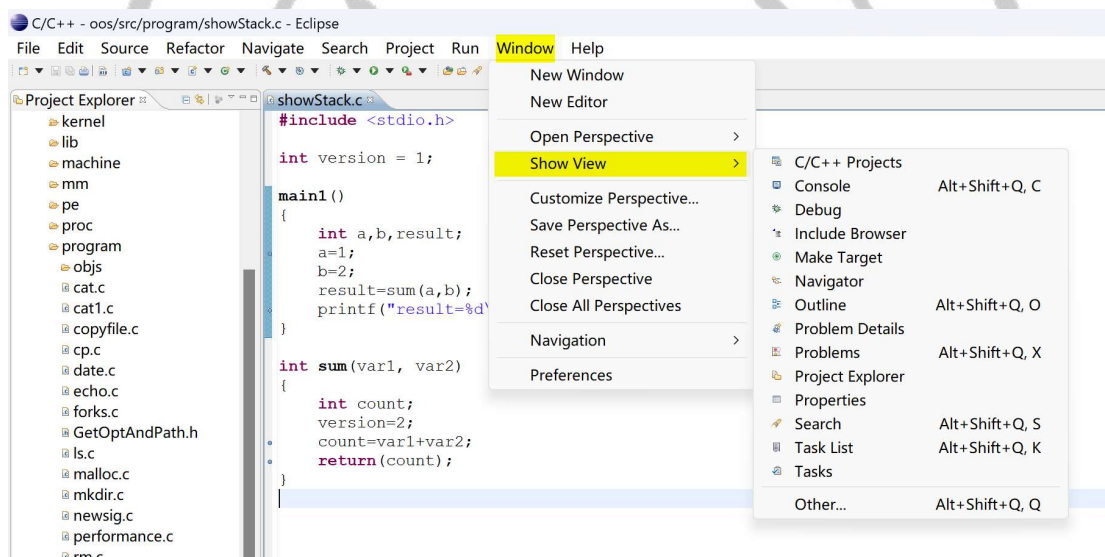


图 13: 定制 Eclipse 调试界面

### (3) 调试运行

此后,可在你希望程序停下来的地方设置断点,观察程序的运行。以图 14 为例。在 Debug 窗口中,可以看到程序当前停下的位置①,进而在 Register 窗口中看到当前 eip 寄存器的值也指向该位置③,Disassembly 窗口中可以查到该位置的汇编指令④。

另外需要说明的是,在 Memory 窗口中添加需要查看的地址时,如果是 16 进制值,前面必须加上 0x。Eclipse 提供了 6 种查看的模式,读者可根据需要灵活选择。



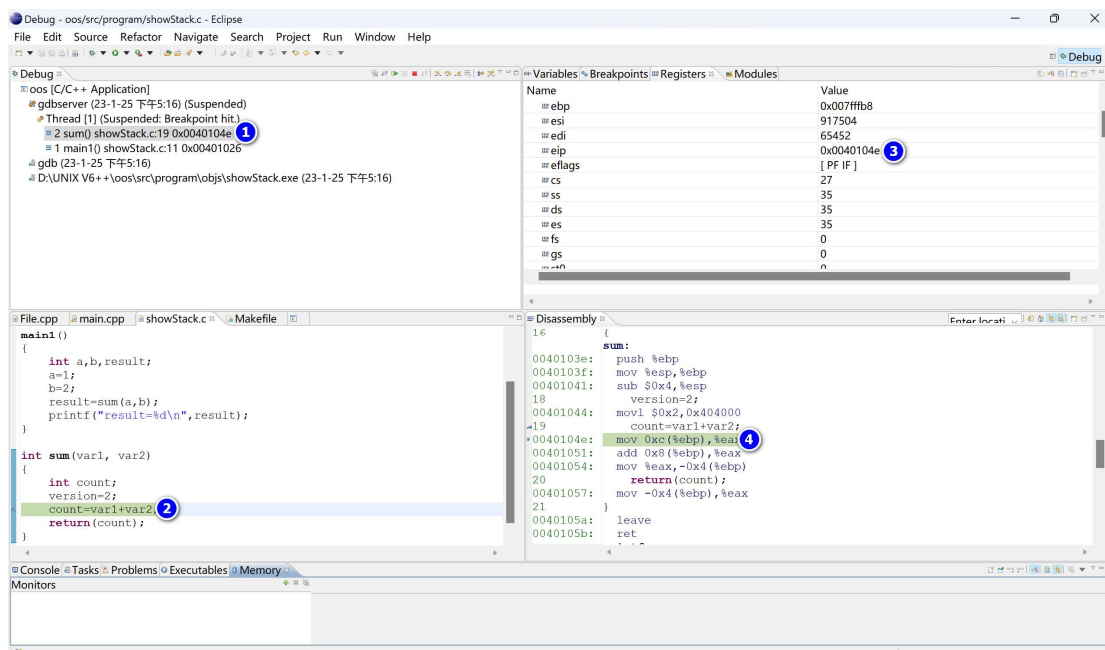


图 14：调试过程中的结果查看

### 4.3. 观察堆栈变化

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。在其中设置断点让程序停下来后，即可以在 Disassembly 窗口中获取到全部的汇编代码如下所示。这里我们已经给其中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和代码停止的位置不同而有差异，所以请确保使用的是正确的编译器版本，且程序一定停止在 `main1` 内的某一条语句，建议可多尝试几次）

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一般实现方法略有不同，比如：采用“`sub $0x18,%esp`”指令空出了 24 个字节（6 个字）的栈帧位置留给调用函数的局部变量和被调用函数的参数。

（2）图 17 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。

#### main1:

```
00401000:  push %ebp           //前一栈帧的 ebp 存入当前栈
00401001:  mov %esp, %ebp      //修改 ebp 指向当前栈帧
00401003:  sub $0x18, %esp      //esp 上移 6 个字，空出 main 局部变量和 sum 参数的位置
8      a=1;
```

```

00401006:  movl $0x1, -0x4(%ebp) //把 1 送入%ebp-4, 即 main 的局部变量 a
9      b=2;
0040100d:  movl $0x2, -0x8(%ebp) //把 2 送入%ebp-8, 即 main 的局部变量 b
10     result=sum(a,b);
00401014:  mov -0x8(%ebp), %eax //把 main 的局部变量 b 的值送入 eax
00401017:  mov %eax,0x4(%esp) //把 eax 的值送入%esp+4, 即 sum 的参数 var2
0040101b:  mov -0x4(%ebp),%eax //把 main 的局部变量 a 的值送入 eax
0040101e:  mov %eax,%esp //把 eax 的值送入%esp, 即 sum 的参数 var1
00401021:  call 0x40103e <sum> //调用 sum 函数
00401026:  mov %eax,-0xc(%ebp) //将 eax 带回的 sum 返回值赋值给 result
11     printf("result=%d\n",result);
00401029:  mov -0xc(%ebp),%eax
0040102c:  mov %eax,0x4(%esp)
00401030:  movl $0x405000,(%esp)
00401037:  call 0x40105c <printf>
12     }
0040103c:  leave
0040103d:  ret

```

在图 15 中可以看到,当程序暂停在 sum 中某一条时,利用栈顶指针 esp 的值 0x007fffc0,可以在 Memory 窗口中利<Hex Integer>模式获取到此时堆栈内所有单元的值。

利用这个方法,我们将 main 的代码在执行完 00401026 语句,即: sum 函数调用结束为止,用户栈的变化整理到图 16 中,并加以说明。

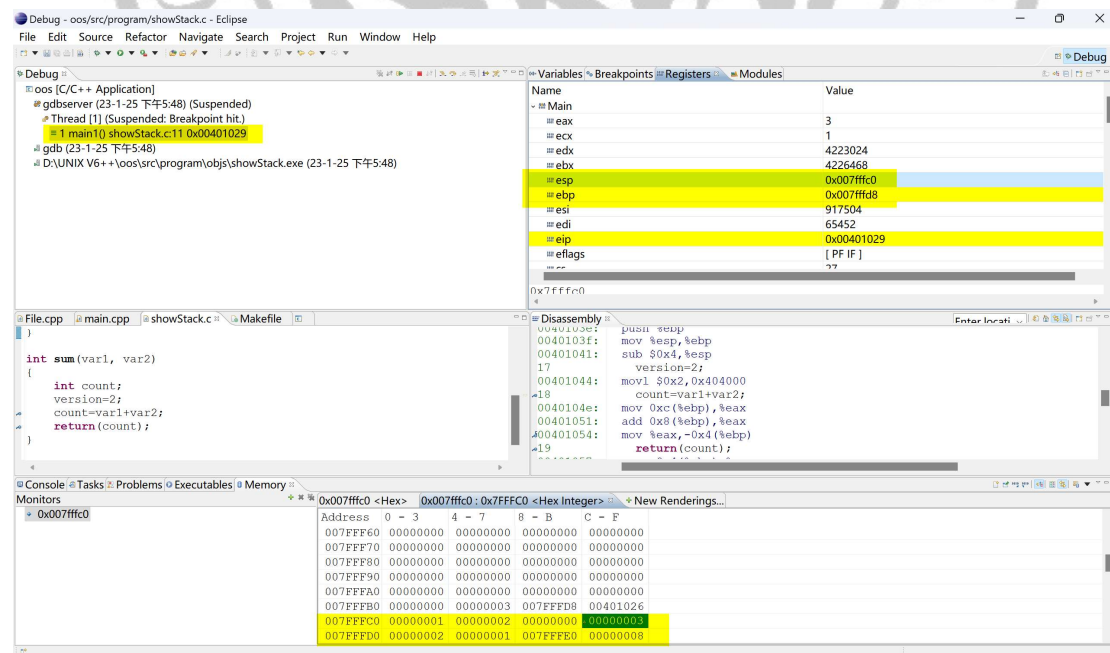


图 15: 查看 main1 运行过程中的堆栈

esp: (call 之后)	0x007fffb0	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
	0x007fffc8	0	main 的局部变量 result, 00401026 语句后变为 3
	0x007ffcc	3	
	0x007fffd0	2	
	0x007fffd4	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

图 16: main1 函数的栈帧

读者请按照上述方法自行分析如下 sum 函数的汇编代码, 添加注释, 并绘制核心栈栈帧。

```

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
0040105a: leave
0040105b: ret

```

## 5. 实验报告要求

实验报告需包含以下内容:

(1) (1 分) 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, 通过在 Memory 窗口中观察地址单元的值验证核心栈的变化。

(3) (1 分) 完整分析 sum 的汇编代码, 添加注释, 并仿照图 16, 绘制完整的堆栈。

(4) (1 分) 选择以下两个问题其一回答:

问题一: 在 main1 的汇编代码中都出现了 “sub \$0x18,%esp” 语句来预留出 6 个字, 请查阅资料, 解释这么做的原因是什么。

问题二: 在 sum 的汇编代码中都出现了 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你的答案。

