中国科学技术大学软件学院
SCHOOL OF SOFTWARE ENGINEERING OF USTC

操作系统分析所需的相关基础知识
Foundations For Hacking Linux
based on X86/Linux

孟宁
电话：0512-68839303
E-mail：mengning@ustc.edu.cn
主页：http://staff.ustc.edu.cn/~mengning
地址：苏州工业园区独墅湖高等教育区仁爱路188号思贤楼507室

Agenda

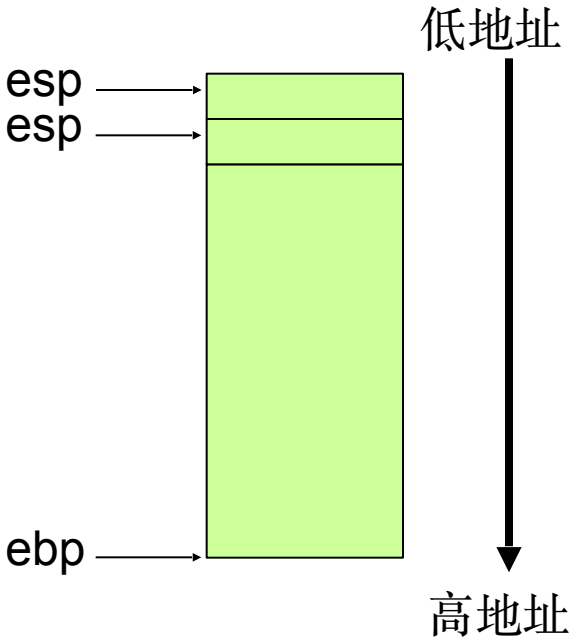
- C语言函数调用堆栈框架
- C代码中嵌入汇编代码
- 函数指针与回调函数
- 用户态和内核态
- 保护现场和恢复现场
- 虚拟内存
- 进程的地址空间

堆栈

- 堆栈是C语言程序运行时必须的一个记录调用路径和参数的空间
  - 函数调用框架
  - 传递参数
  - 保存返回地址
  - 提供局部变量空间
  - 等等
- C语言编译器对堆栈的使用有一套的规则
- 了解堆栈存在的目的和编译器对堆栈使用的规则是理解操作系统一些关键性代码的基础

堆栈寄存器和堆栈操作

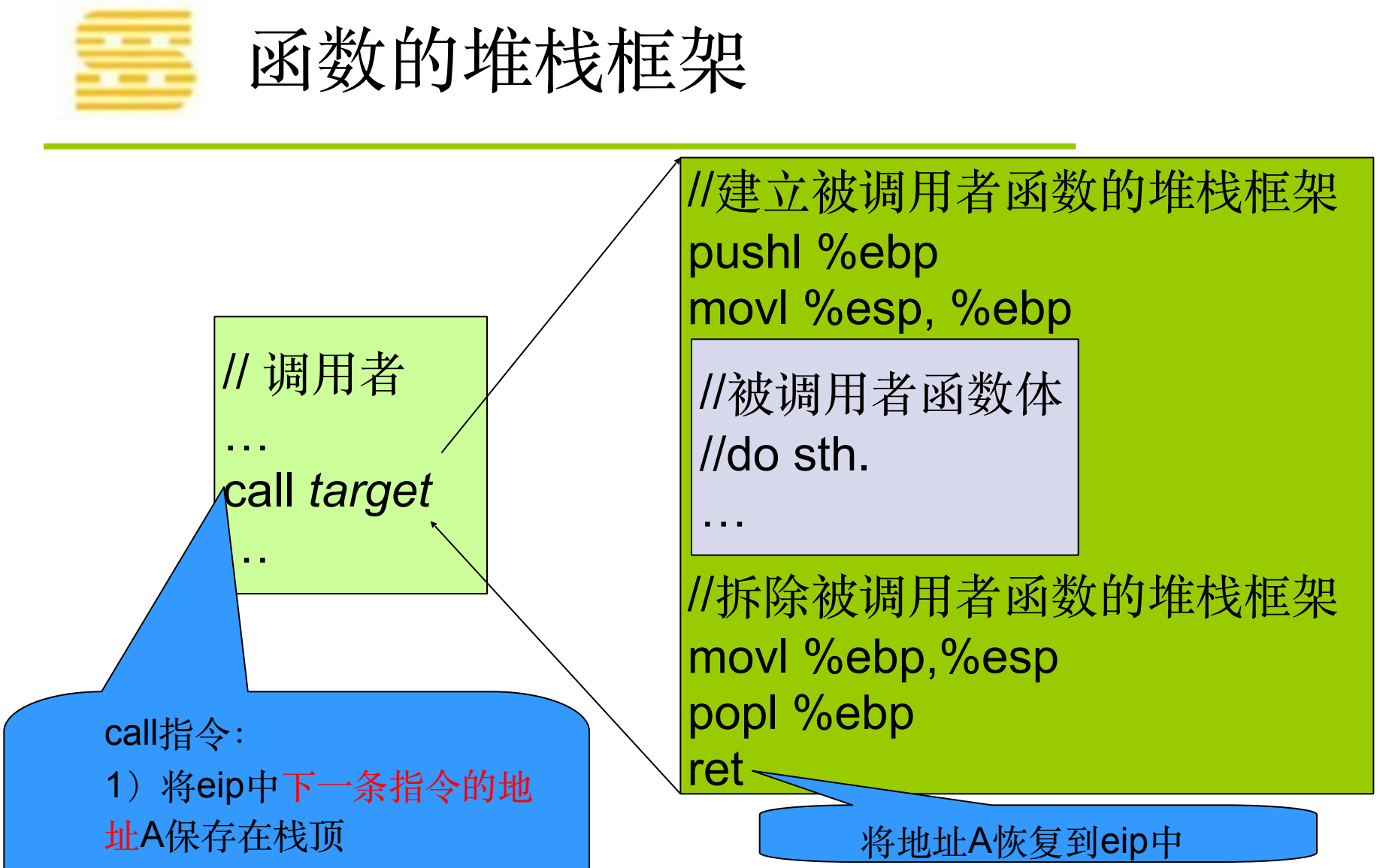
- 堆栈相关的寄存器
  - esp，堆栈指针（stack pointer）
  - ebp，基址指针（base pointer）
- 堆栈操作
  - push
    栈顶地址减少4个字节（32位）
  - pop
    栈顶地址增加4个字节
- ebp在C语言中用作记录当前函数调用基址



利用堆栈实现函数调用和返回

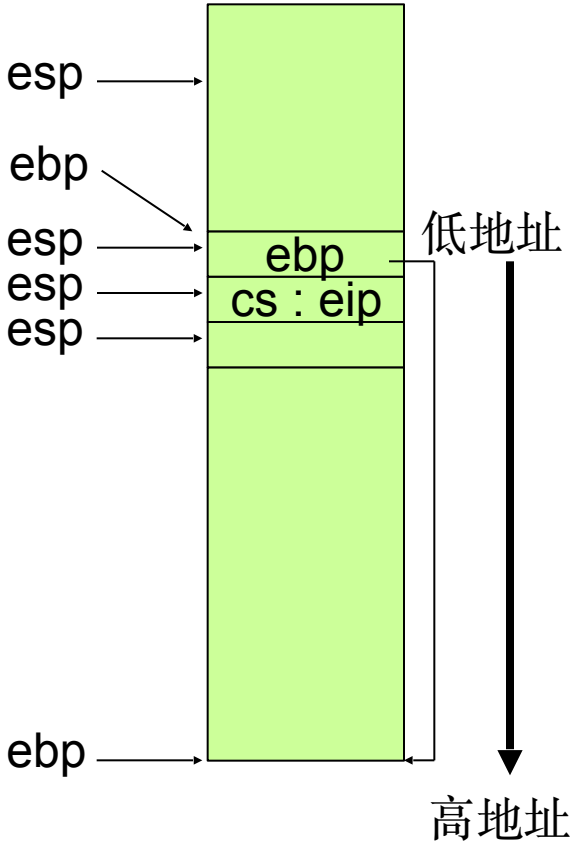
- 其他关键寄存器
  - cs：eip：总是指向下一条的指令地址
    - 顺序执行：总是指向地址连续的下一条指令
    - 跳转/分支：执行这样的指令的时候，cs：eip的值会根据程序需要被修改
    - call：将当前cs：eip的值压入栈顶，cs：eip指向被调用函数的入口地址
    - ret：从栈顶弹出原来保存在这里的cs：eip的值，放入cs：eip中
    - 发生中断时？

函数的堆栈框架


// 调用者
…
call *target*
..
//建立被调用者函数的堆栈框架
pushl %ebp
movl %esp, %ebp
//被调用者函数体
//do sth.
…
//拆除被调用者函数的堆栈框架
movl %ebp,%esp
popl %ebp
ret
call指令：
1）将eip中下一条指令的地址A保存在栈顶
2）设置eip指向被调用程序代码开始处
将地址A恢复到eip中

函数堆栈框架的形成

- call xxx
  - 执行call之前
  - 执行call时，cs : eip原来的值
    指向call下一条指令，该值被
    保存到栈顶，然后cs : eip的值
    指向xxx的入口地址
- 进入xxx
  - 第一条指令： pushl %ebp
  - 第二条指令： movl %esp, %ebp
  - 函数体中的常规操作，可能会压栈、出栈
- 退出xxx
  - movl %ebp,%esp
  - popl %ebp
  - ret



一段小程序

源文件：test.c

这是一个很简单的C程序

main函数中调用了函数p1和p2

首先使用gcc –g 生成test.c的可执行文件test

然后使用objdump –S获得test的反汇编文件

```
#include  <stdio.h>

void p1(char c)
{
        printf("%c\n",c);
}

int p2(int x,int y)
{
        return x+y;
}

int main(void)
{
        char c='a';
        int x,y,z;
        x=1;
        y=2;
        p1(c);
        z=p2(x,y);
        printf("%d=%d+%d\n",z,x,y);
}
```

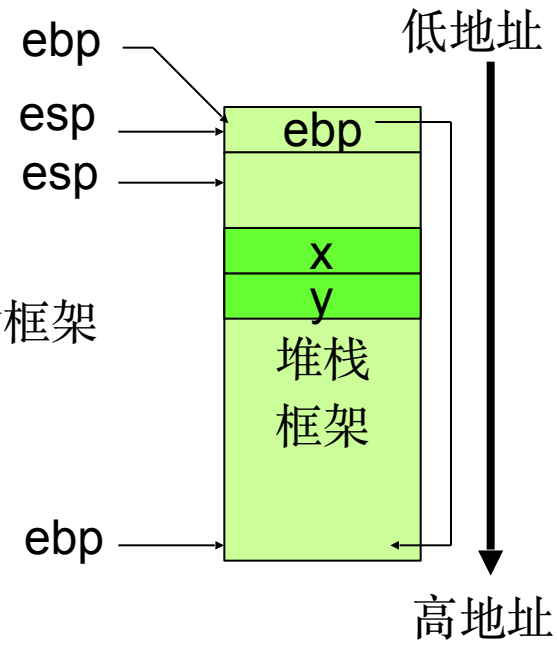
观察p2的堆栈框架

- 从test的反汇编文件中找到p2的反汇编代码

```
int p2(int x,int y)
{
        push   %ebp
        mov    %esp,%ebp
    return x+y;
        mov    0xc(%ebp),%eax
        add    0x8(%ebp),%eax
}
        movl %ebp,%esp
        pop    %ebp
        ret
```

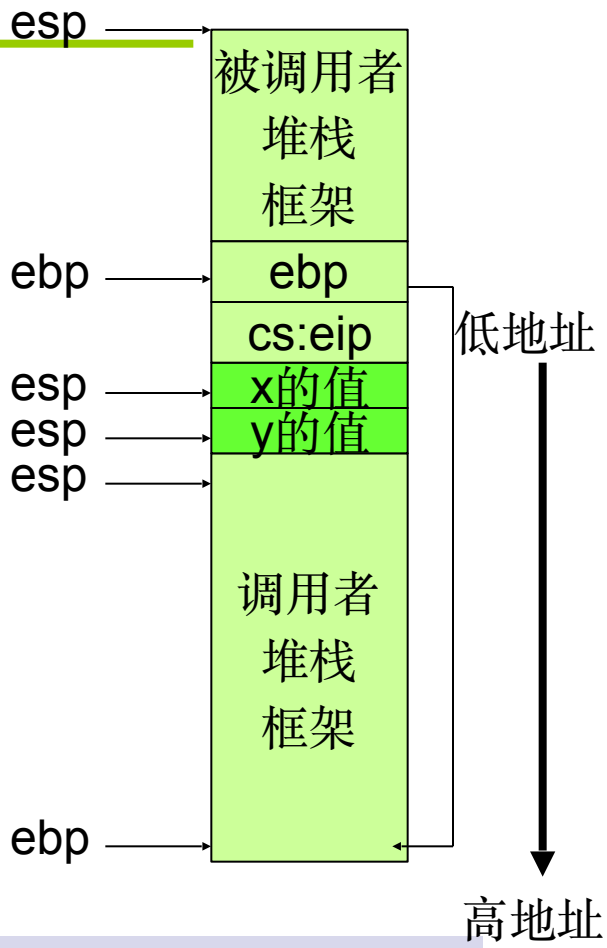
建立框架

拆除框架



如何传递参数给p2的

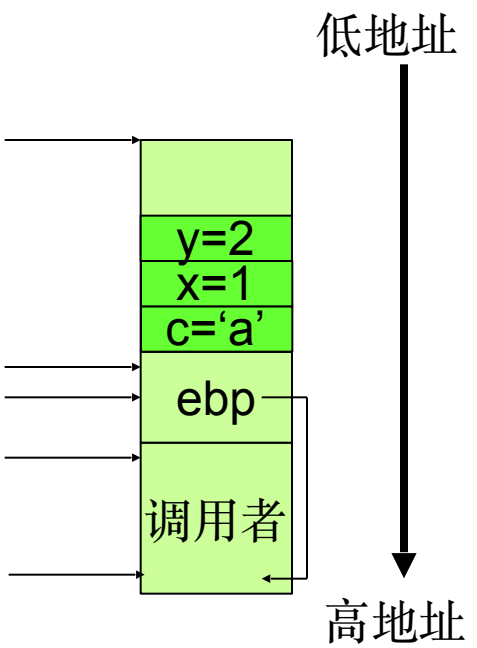
```
                …
    z=p2(x,y);
                pushl  0xffffffff8(%ebp)
                pushl  0xffffffff4(%ebp)
                call   804839b <p2>
                add    $0x8,%esp
                mov    %eax,0xfffffffc(%ebp)
printf("%d=%d+%d\n",z,x,y);
                pushl  0xffffffff8(%ebp)
                pushl  0xffffffff4(%ebp)
                pushl  0xfffffffc(%ebp)
                push   $0x8048510
                call   80482b0 <printf@plt>
                …
```



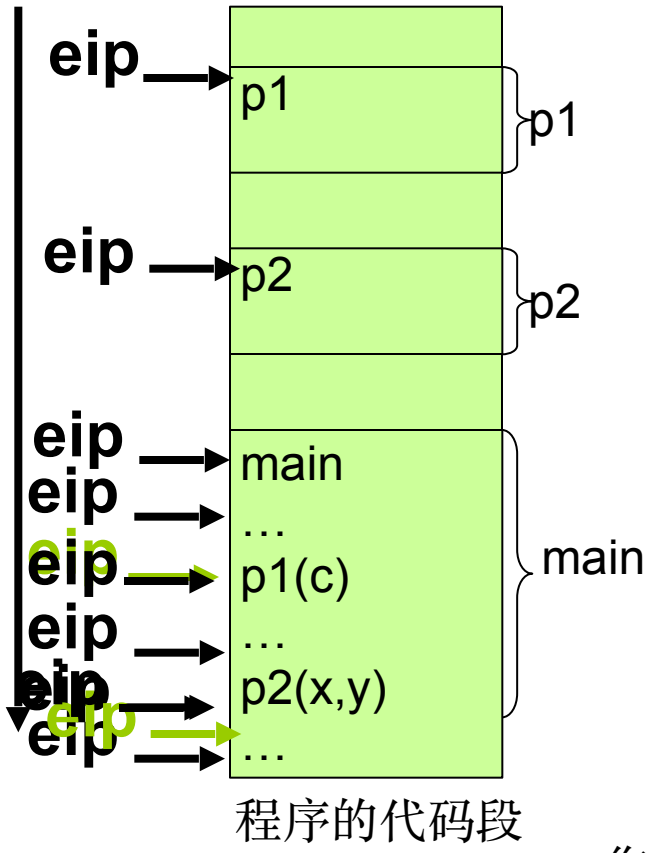
p2的返回值是如何返回给main的?

观察main中的局部变量

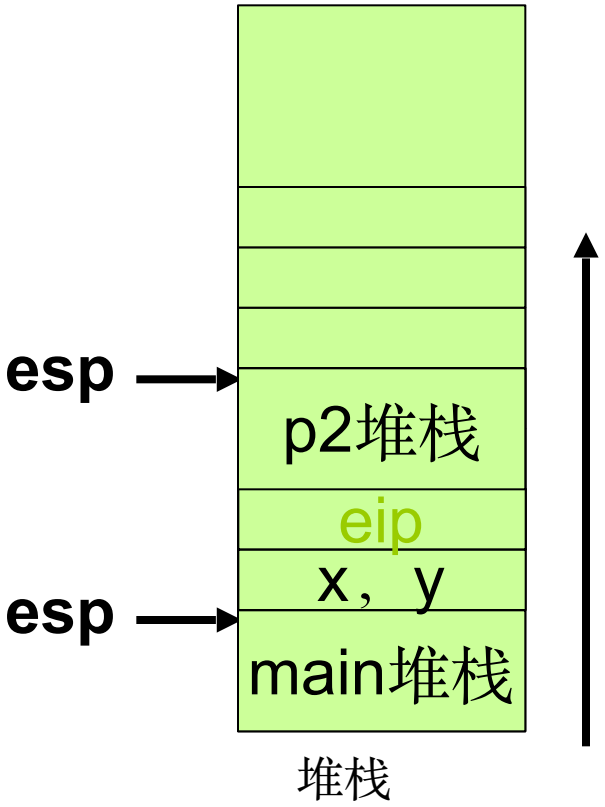
```
int main(void)
{
                push   %ebp
                mov    %esp,%ebp
                sub    $0x18,%esp
                …
char c='a';                                   esp
                movb   $0x61,0xfffffff3(%ebp)
int x,y,z;
x=1;
                movl   $0x1,0xfffffff4(%ebp)
y=2;                                          ebp
                movl   $0x2,0xfffffff8(%ebp)  esp
…                                             esp

                                              ebp
```



观察程序运行时堆栈的变化



程序的代码段



堆栈

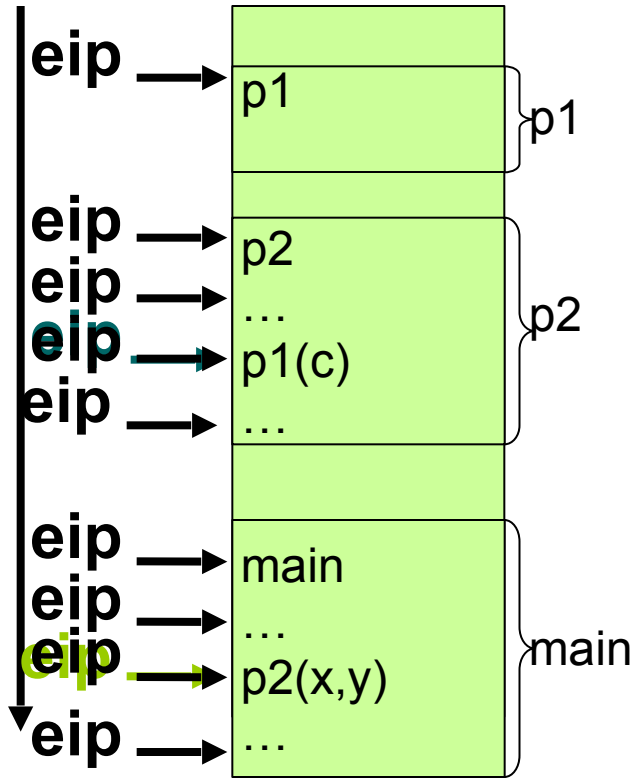
您能把main、p1和p2堆栈的内容补充完整吗?

三级函数调用程序

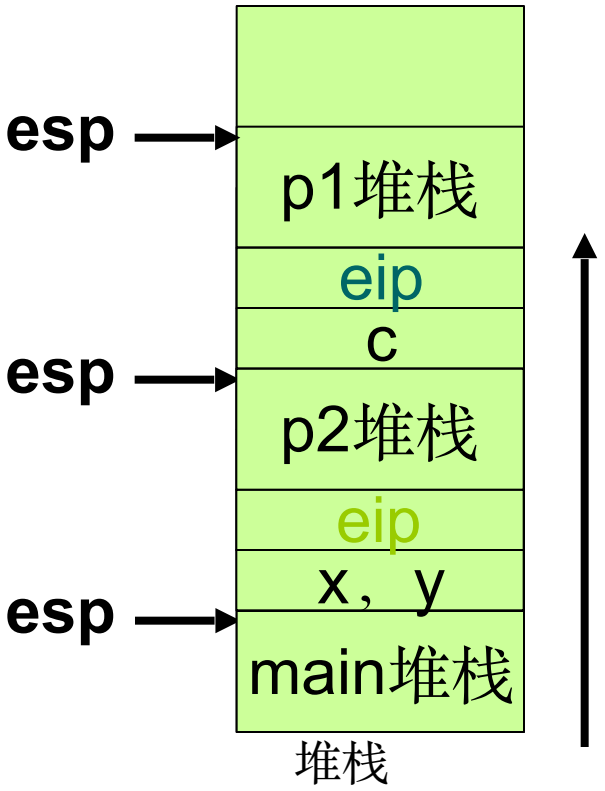
```
#include <stdio.h>
void p1(char c)
{
        printf("%c\n",c);
}
int p2(int x, int y)
{
        char c;
        c='a';
        p1(c);
        return x+y;
}
int main(void)
{
        int x,y,z;
        x=1;
        y=2;
        z=p2(x,y);
        printf("%d+%d=%d\n",x,y,z);
        return 0;
}
```

在这个小程序中，**main**函数中调用了函数**p2**，而在**p2**的执行过程中又调用了函数**p1**

观察程序运行时堆栈的变化



程序的代码段



堆栈

您能把main、p1和p2堆栈的内容补充完整吗?

C代码中嵌入汇编代码

- 内嵌汇编语法

__asm__(
    汇编语句模板:
    输出部分:
    输入部分:
    破坏描述部分);
    即格式为 asm ("statements" : output_regs : input_regs : clobbered_regs);

同时“asm”也可以由“__asm__”来代替，“asm”是“__asm__”的别名。在“asm”后面有时也会加上“__volatile__”表示编译器不要优化代码，后面的指令保留原样，“volatile”是它的别名，在这里值得注意的是无论“__asm__”还是“__volatile__”中的每个下划线都不是一个单独的下划线，而是两个短的下划线拼成的。再后面括号里面的便是汇编指令。

例子

```
#include <stdio.h>

int  main()
{
        /* val1+val2=val3 */
        unsigned int val1 = 1;
        unsigned int val2 = 2;
        unsigned int val3 = 0;
        printf("val1:%d,val2:%d,val3:%d\n",val1,val2,val3);
        asm volatile(
        "movl $0,%%eax\n\t"     /* clear %eax to 0*/
        "addl %1,%%eax\n\t"     /* %eax += val1 */
        "addl %2,%%eax\n\t"     /* %eax += val2 */
        "movl %%eax,%0\n\t"     /* val2 = %eax*/
        : "=m" (val3)           /* output =m mean only write output memory variable*/
        : "c" (val1),"d" (val2)     /* input c or d mean %ecx/%edx*/
        );
        printf("val1:%d+val2:%d=val3:%d\n",val1,val2,val3);

        return 0;
}
```

表 2-7 内嵌汇编常用限定符

	限定符	描述
具体的一个寄存器	“a”	将输入变量放入 eax
	“b”	将输入变量放入 ebx
	“c”	将输入变量放入 ecx
	“d”	将输入变量放入 edx
	“s”	将输入变量放入 esi
	“D”	将输入变量放入 edi
	“q”	将输入变量放入 eax、ebx、ecx、edx 中的一个
	“r”	将输入变量放入通用寄存器，也就是 eax, ebx, ecx, edx, esi, edi 中的一个
	“A”	放入 eax 和 edx，把 eax 和 edx，合成一个 64 位的寄存器 (useIong longs)
内存	“m”	内存变量
	“o”	操作数为内存变量，但是其寻址方式是偏移量类型
	“V”	操作数为内存变量，但寻址方式不是偏移量类型
	“<”	操作数为内存变量，但寻址方式为自动增量
	“p”	操作数是一个合法的内存地址（指针）
寄存器或内存	“g”	将输入变量放入 eax，ebx，ecx ，edx 中的一个或者作为内存变量
	“X”	操作数可以是任何类型
立即数	“I”	0-31 之间的立即数（用于 32 位移位指令）
	“J”	0-63 之间的立即数（用于 64 位移位指令）
	“N”	0-255 之间的立即数（用于 out 指令）
	“i”	立即数
	“n”	立即数，有些系统不支持除字以外的立即数，这些系统应该使用 “n”
操作数类型	“=”	操作数在指令中是只写的（输出操作数）
	“+”	操作数在指令中是读写类型的（输入输出操作数）
浮点数	“f”	浮点数
	“t”	第一个浮点寄存器
	“u”	第二个浮点寄存器
	“G”	标准的 80387
	%	该操作数可以和下一个操作数交换位置
	#	部分注释，从该字符到其后的逗号之间所有字母被忽略
	*	表示如果选用寄存器，则其后的字母被忽略

练习

```
int main(void)
{
    int input, output, temp;

    input = 1;
    __asm__ __volatile__    ("movl $0, %%eax;\n\t"
                "movl %%eax, %1;\n\t"
                "movl %2, %%eax;\n\t"
                "movl %%eax, %0;\n\t"
                : "=m" (output), "=m"(temp)
                : "r" (input)
                :"eax");
    printf("%d %d\n", temp,output);
    return 0;
}
```

这段代码的输出结果？

函数指针与回调函数

- 函数指针是指向函数的指针变量，即本质是一个指针变量。
  - int (*f) (int x); /* 声明一个函数指针 */
  - f=func; /* 将func函数的首地址赋给指针f */
- 指向函数的指针包含了函数的地址，可以通过它来调用函数。
- 回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。

函数指针的例子

```
void (*funcp)();
void FileFunc();
void EditFunc();

 main()
 {
    funcp=FileFunc;
    (*funcp)();
    funcp=&EditFunc;
    funcp();
 }
```

```
void FileFunc()
{
    printf(FileFunc\n);
}

void EditFunc()
{
    printf(EditFunc\n);
}
```

什么是用户态和内核态?

- 一般现代CPU都有几种不同的指令执行级别
- 在高执行级别下，代码可以执行特权指令，访问任意的物理地址，这种CPU执行级别就对应着内核态
- 而在相应的低级别执行状态下，代码的掌控范围会受到限制。只能在对应级别允许的范围内活动
- 举例:
  intel x86 CPU有四种不同的执行级别0-3，Linux只使用了其中的0级和3级分别来表示内核态和用户态

如何区分用户态和内核态?

- cs寄存器的最低两位表明了当前代码的特权级
- CPU每条指令的读取都是通过cs:eip这两个寄存器:其中cs是代码段选择寄存器，eip是偏移量寄存器。
- 上述判断由硬件完成
- 一般来说在Linux中，地址空间是一个显著的标志:0xc0000000以上的地址空间只能在内核态下访问，0x00000000- 0xbfffffff的地址空间在两种状态下都可以访问

注意:这里所说的地址空间是逻辑地址而不是物理地址

问题

- 我们编译好的普通应用程序（可执行文件）有无用户态和内核态之分?
- 我们将它运行起来，这时有没有用户态和内核态之分?
- 如果有的话，如何区分用户态和内核态?

用户态 vs 内核态

- 寄存器上下文
  - 从用户态切换到内核态时
    - 必须保存用户态的寄存器上下文
    - 要保存哪些?
    - 保存在哪里?
- 中断/int指令会在堆栈上保存一些寄存器的值
  - 如：用户态栈顶地址、当时的状态字、当时的cs:eip的值

保护现场和恢复现场

保护现场 就是 进入中断程序 保存 需要用到的 寄存器 的 数据，
恢复现场 就是 退出中断程序 恢复 保存 寄存器 的 数据，

```
#define SAVE_ALL \
        "cld\n\t" \
        "pushl %es\n\t" \
        "pushl %ds\n\t" \
        "pushl %eax\n\t" \
        "pushl %ebp\n\t" \
        "pushl %edi\n\t" \
        "pushl %esi\n\t" \
        "pushl %edx\n\t" \
        "pushl %ecx\n\t" \
        "pushl %ebx\n\t" \
        "movl $" STR(__KERNEL_DS) ",%edx\n\t" \
        "movl %edx,%ds\n\t" \
        "movl %edx,%es\n\t"
```

```
RESTORE_ALL     \
        popl %ebx;      \
        popl %ecx;      \
        popl %edx;      \
        popl %esi;      \
        popl %edi;      \
        popl %ebp;      \
        popl %eax;      \
        popl %ds;       \
        popl %es;       \
        addl $4,%esp;   \
        iret;           \
```

虚拟内存

- 物理内存有限，是一种稀缺资源
- 局部性原理
  - 空间局部性
  - 时间局部性
- 按需调页
  - 页框
- 利用磁盘上的交换空间

进程的地址空间

- 独立的地址空间（32位，4GB），每个进程一个
- 在Linux中，3G以上是内核空间，3G以下是用户空间
- 4G的进程地址空间使用进程私有的二级页表进行地址转换（虚拟地址→物理地址）
  - 页面大小：4KB
  - 页目录、页表
  - 若对应的内容在内存中，则对应的二级页表项记录相应的物理页框信息
  - 否则根据需要进行装载或者出错处理
- 进程调度后，执行一个新的被调度的进程之前，要先进行页表切换

Linux中的内核空间

- 每个进程3G以上的空间用作内核空间
- 从用户地址空间进入内核地址空间不经过页表切换
- 而是通过中断/异常/系统调用入口（也只能如此）

Homework

- 请实际编译运行课件中嵌入式汇编和函数指针的例子，体会它们的用法。
- 堆栈的作用是什么？请使用一个实例说明堆栈的作用。
- 为什么要有内核态与用户态的区别？请结合32位x86说明在Linux中，用户态与内核态有哪些区别？在什么情况下，系统会进入内核态执行？

“有两种生成一个软件设计方案的途径：一个是把它做得如此简单，以致于明显不会有漏洞存在。另一个是把它做的如此复杂，以致于不会有明显的漏洞存在。”——Tony Hoare

谢谢大家！

参考资料：

《深入理解Linux内核》第三版