目录

实验六实验报告 2

一、 【个人信息】 2

二、 【实验题目】 2

三、 【实验目的】 2

四、 【实验方案】 2

(一) 实验原理 2

1. 二状态的进程模型 2
2. 进程控制块（PCB） 3
3. 进程交替执行原理 4

五、 【实验过程】 5

(一) g++与汇编接口的研究 5

1. 命名空间解析规则探索 5
2. 类解析规则探索 6
3. 引用、指针传递修饰规则探索 9

(二) 多进程交替执行的研究 9

1. 保存当前进程的寄存器到 PCB 9
2. 切换进程 11
3. 回到操作系统 13
4. 测试 14

六、 【技术点与创新点】 15

(一) 工具创新：G++与 NASM 15

1. Name Mangling 概述 15
2. g++函数名修饰规则 16
3. g++变量修饰规则 17

(二) C++库的建立 17

1. malloc 和 free 17
2. new 和 delete 18
3. String 18
4. Vector 19

七、 【实验总结】 20

八、 【参考文献】 20

实验六实验报告

一、 【个人信息】

院系：数据科学与计算机学院
专业：计算机科学与技术（超算方向）
年级：2016
班级：教务 2 班
姓名：劳马东
学号：16337113
邮箱：laomd@mail2.sysu.edu.cn

二、 【实验题目】

保留原型原有特征的基础上，设计满足下列要求的新原型操作系统：

1. 在 c 程序中定义进程表，进程数量至少 4 个。
2. 内核一次性加载多个用户程序运行时，采用时间片轮转调度进程运行，用户程序的输出各占 1/4 屏幕区域，信息输出有动感，以便观察程序是否在执行。
3. 在原型中保证原有的系统调用服务可用。再编写 1 个用户程序，展示系统调用服务还能工作。

三、 【实验目的】

1. 在内核实现多进程的二状态模，理解简单进程的构造方法和时间片轮转调度过程。
2. 实现解释多进程的控制台命令，建立相应进程并能启动执行。
3. 至少一个进程可用于测试前一版本的系统调用，搭建完整的操作系统框架，为后续实验项目打下扎实基础。

四、 【实验方案】

(一) 实验原理

1. 二状态的进程模型

- 进程模型就是实现多道程序和分时系统的一个理想的方案
  - ✓ 多个用户程序并发执行

- ✓ 进程模型中，操作系统可以知道有几个用户程序在内存运行，每个用户程序执行的代码和数据放在什么位置，入口位置和当前执行的指令位置，哪个用户程序可执行或不可执行，各个程序运行期间使用的计算机资源情况等等

- 二状态进程模型
  - ✓ 执行和等待
  - ✓ 目前进程的用户程序都是 COM 格式的，是最简单的可执行程序
  - ✓ 进程仅涉及一个内存区、CPU、显示屏这几种资源，所以进程模型很简单，只要描述这几个资源
    - 现在的用户程序都很小，只要简单地将内存划分为多个小区，每个用户程序占用其中一个区，就相当于每个用户拥有独立的内存。根据我们的硬件环境，CPU 可访问 1M 内存，我们规定 MYOS 加载在第一个 64K 中，用户程序从第二个 64K 内存开始分配，每个进程 64K
    - 用内存单元保存 8086CPU 的所有寄存器，将一个物理 CPU 模拟为多个独立的逻辑 CPU
    - 参考内存划分的方法，将 25 行 80 列的显示区划分为多个区域，在进程运行后，操作系统的显示信息是很少的我们就将显示区分为 4 个区域，用户程序如果要显示信息，规定在其中一个区域显示。
  - ✓ 以后扩展进程模型解决键盘输入、进程通信、多进程、文件操作

2. 进程控制块（PCB）

汇编语言中描述PCB

```
PCB:
        dw 0B800h; GS \
        dw 6000h  ; FS   | 部分段寄存器，用PUSH指令一个个压入栈
        dw 6000h  ; ES   |
        dw 6000h  ; DS /
        dw 0      ; DI   \
        dw 0      ; SI    | 指针寄存器 \
        dw 0      ; BP    |              |
        dw 100h-4; SP /                  | 通用寄存器，用PUSHA指令一起压入栈
        dw 0      ; BX \                 |
        dw 0      ; DX  | 主寄存器     /
        dw 0      ; CX  |
        dw 0      ; AX  /
        dw 6000h ; SS 堆栈段寄存器，手工赋值
        dw 100h  ; IP 指令指针寄存器                    \
        dw 6000h ; CS 代码段寄存器                       | 中断时由CPU压入栈
        dw 512   ; Flags 标志寄存器(中断允许IF=1)  /
        dw 1     ; ID 进程ID
        db 'ProcessA' ; Name 进程名（8个字符）
```

图 1 PCB-汇编

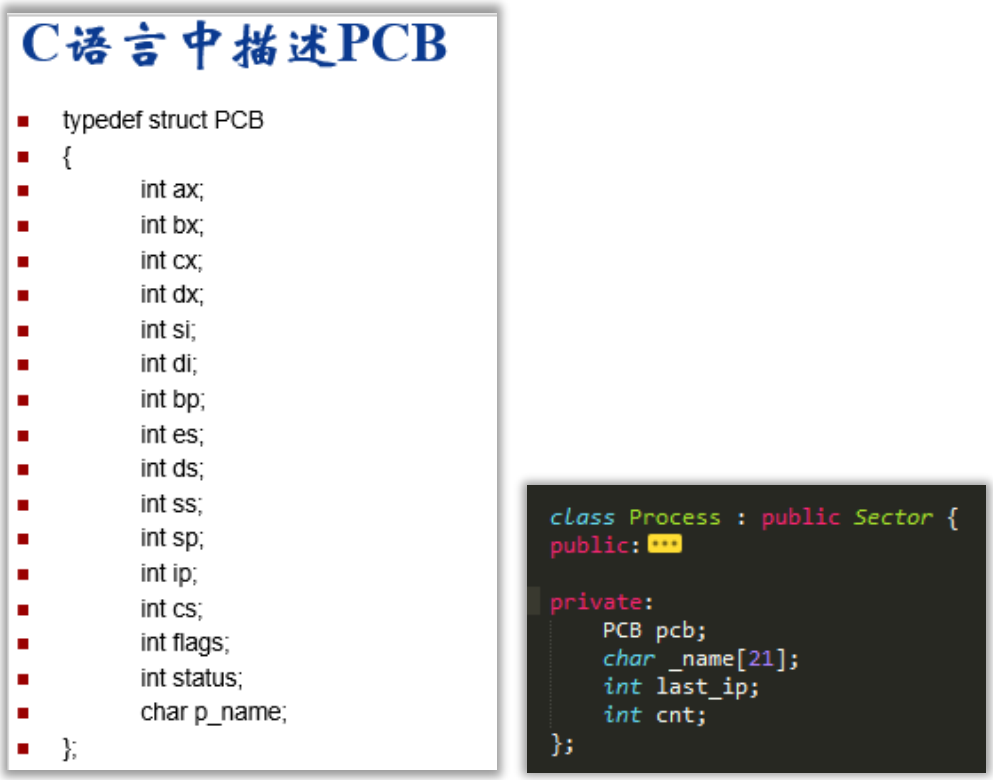

图 2 Process 中的 PCB 变量

3. 进程交替执行原理

- 采用时钟中断打断执行中的用户程序实现 CPU 在进程之间交替
  - ✓ 每次发生时钟中断，中断服务程序就让 A 换 B 或 B 换 A。
  - ✓ 要知道中断发生时谁在执行，还要把被中断的用户程序的 CPU 寄存器信息保存到对应的 PCB 中，以后才能恢复到 CPU 中保证程序继续正确执行。中断返回时，CPU 控制权交给另一个用户程序。
- 现场保护：save 过程
  - ✓ save 是一个非常关键的过程，保护现场不能有丝毫差错，否则再次运行被中断的进程可能出错。
  - ✓ 涉及到二种不同的栈：应用程序栈、进程表栈、内核栈。其中的进程表栈，只是我们为了保存和恢复进程的上下文寄存器值，而临时设置的一个伪局部栈，不是正常的程序栈
  - ✓ 在时钟中断发生时，实模式下的 CPU 会将 FLAGS、CS、IP 先后压入当前被中断程序（进程）的堆栈中，接着跳转到（位于 kernel 内）时钟中断处理程序（Timer 函数）执行。注意，此时并没有改变堆栈（的 SS 和 SP），换句话说，我们内核里的中断处理函数，在刚开始时，使用的是被中断进程的堆栈
  - ✓ 为了及时保护中断现场，必须在中断处理函数的最开始处，立即保存被中断程序的所有上下文寄存器中的当前值。不能先进行栈切换，再来保存寄存器。因为切换栈所需的若干指令，会破坏寄存器的当前值。

这正是我们在中断处理函数的开始处，安排代码保存寄存器的内容

- ✓ 我们 PCB 中的 16 个寄存器值，内核一个专门的程序 save，负责保护被中断的进程的现场，将这些寄存器的值转移至当前进程的 PCB 中。

- 进程切换：restart 过程
  - ✓ 用内核函数 restart 来恢复下一进程原来被中断时的上下文，并切换到下一进程运行。这里面最棘手的问题是 SS 的切换。
  - ✓ 使用标准的中断返回指令 IRET 和原进程的栈，可以恢复（出栈）IP、CS 和 FLAGS，并返回到被中断的原进程执行，不需要进行栈切换。
  - ✓ 如果使用我们的临时(对应于下一进程的)PCB 栈，也可以用指令 IRET 完成进程切换，但是却无法进行栈切换。因为在执行 IRET 指令之后，执行权已经转到新进程，无法执行栈切换的内核代码；而如果在执行 IRET 指令之前执行栈切换（设置新进程的 SS 和 SP 的值），则 IRET 指令就无法正确执行，因为 IRET 必须使用 PCB 栈才能完成自己的任务。
  - ✓ 解决办法有三个，一个是所有程序，包括内核和各个应用程序进程，都使用共同的栈。即它们共享一个（大栈段）SS，但是可以有各自不同区段的 SP，可以做到互不干扰，也能够用 IRET 进行进程切换。第二种方法，是不使用 IRET 指令，而是改用 RETF 指令，但必须自己恢复 FLAGS 和 SS。第三种方法，使用 IRET 指令，在用户进程的栈中保存 IP、CS 和 FLAGS，但必须将 IP、CS 和 FLAGS 放回用户进程栈中，这也是我们程序所采用的方案。
- 进程调度： scheduler 过程
  - ✓ 计算下一个 PCB 地址
  - ✓ 设置当前进程变量 curPro
  - ✓ 控制权交给 curPro 进程

五、 【实验过程】

（一） g++与汇编接口的研究

编译命令：g++ -masm=intel -ml6 -S <if>.cpp -o <of>.asm

1. 命名空间解析规则探索

命名空间是一种描述逻辑分组的机制，可以将按某些标准在逻辑上属于同一个集团的声明放在同一个命名空间中。在 C++中，名称可以是符号常量、变量、宏、函数、结构体、枚举、类和对象等等。为了避免在大规模程序的设计中，以及在程序员使用各种各样的 C++库时，这些标识符的命名发生冲突，标准 C++引入了命名空间，可以更好地控制标识符的作用域。那么，C++的命名空间是如何解析的呢？

1） 全局命名空间

```
int var = 10;

void foo(int a, int b)
{

}
```

图 3 全局命名空间（::）中的 foo 函数和 var 变量

gcc 编译后对应的符号表中，几乎没有对标识符做任何修饰，g++对全局变量的处理同样不加修饰，但函数使用_Z 开头（C99 标准）。函数名之后的一连串字母是形参列表，如 i 代表 int，b 代表 bool。通过不同的形参列表后缀，C++实现了重载机制，但是函数的返回值并没有作为标签名的一部分出现，这也就解释了为什么返回值不能作为重载的标志。

另外，在下文会看到，引用传递的形参被加上 R 前缀，指针传递的形参加上 P 前缀（后面可以看到），因此 C++中三种不同的传递方式也成为区分不同重载函数的依据。

图 4 foo 和 var 解析后的标签

2） 局部命名空间

```
namespace laomd {
    int var = 10;

    void foo(int a, int b)
    {

    }
} // laomd
```

命名空间信息

图 5 命名空间 laomd 中 foo 和 var

局部命名空间中的标签解析规则和全局命名空间最大的不同是加入命名空间相关信息的前缀——命名空间长度和命名空间名字，并且分别用 N 和 E 作为开头和结尾，E 之后才是形参列表后缀。

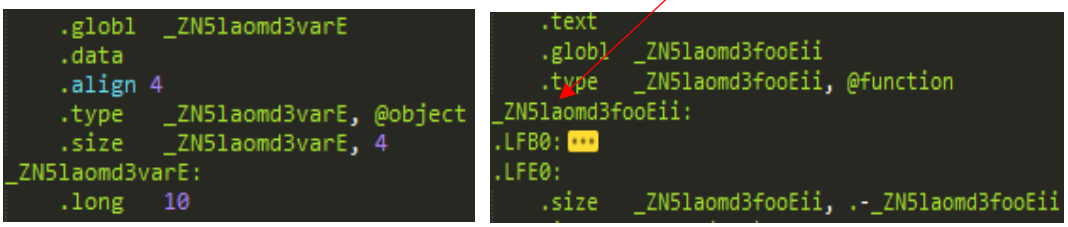

图 6 foo 和 var 解析后的标签

2. 类解析规则探索

C++类主要包括一下元素：构造和析构函数、类静态变量、类静态函数、成员函数（包括普通成员函数和运算符函数）、成员变量。在图 3 的测试代码中，

MyClass 类依次含有普通构造函数、拷贝构造函数、析构函数、foo 普通成员函数、小于运算符函数、类静态函数 toInt、类静态变量 cnt 和类成员变量 x。

```
class MyClass
{
public:

    MyClass(int i) : x(i) {
        cnt++;
    }
    MyClass(const MyClass& other) {
        x = other.x;
        cnt++;
    }
    ~MyClass() {
        cnt--;
    }

    int foo(int u, int v)
    {
        return u + v + x;
    }

    bool operator<(const MyClass& b)
    {
        return x < b.x;
    }

    static int toInt(bool b)
    {
        return b;
    }

private:
    int x;
    static int cnt;
};

int MyClass::cnt = 0;
```

图 7 测试代码

对比局部命名空间解析规则，很容易发现，g++对于局部命名空间和类的解析规则完全一样！类的函数和静态变量相当于局部命名空间的全局函数和全局变量。当然，类中也有一些函数名比较特殊的函数——构造函数、析构函数和运算符函数。

g++对成员函数和类静态函数的解析规则是一样的，这有点出乎意料。类静态函数和成员函数的区别在于传参，成员函数需要 this 指针而类静态函数不需要。这样的话，按照上面的分析，成员函数解析出来本应带有 S_后缀。但是 g++并没有这样做，而是直接在传参的时候控制是否传递 this。

```
    .section    .text._ZN7MyClass5toIntEb,"axG",@progbits,_ZN7MyClass5toIntEb,comdat
    .weak   _ZN7MyClass5toIntEb
    .type   _ZN7MyClass5toIntEb, @function
_ZN7MyClass5toIntEb:
.LFB10: ...
.LFE10:
    .size   _ZN7MyClass5toIntEb, .-_ZN7MyClass5toIntEb
```

类名信息

图 8 类静态函数 toInt

```
    .section    .text._ZN7MyClass3fooEii,"axG",@progbits,_ZN7MyClass3fooEii,comdat
    .align 2
    .weak   _ZN7MyClass3fooEii
    .type   _ZN7MyClass3fooEii, @function
_ZN7MyClass3fooEii:
.LFB9:...
.LFE9:
    .size   _ZN7MyClass3fooEii, .-_ZN7MyClass3fooEii
```

图 9 普通成员函数 foo

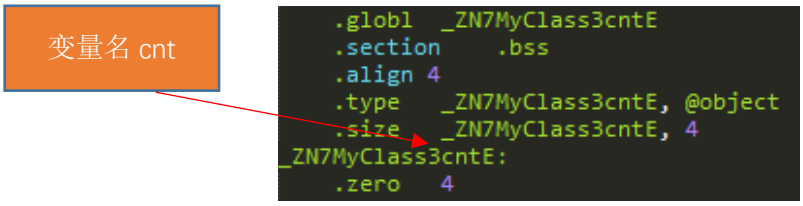

图 10 类静态变量 cnt

g++编译器把构造函数的函数名解析为 C（即 constructor），析构函数解析为 D(destructor),运算符函数解析为相应运算符英文缩写(如小于为 lt,大于为 gt)，并且前面都不带函数名长度。

```
    .section    .text._ZN7MyClassC2Ei,"axG",@progbits,_ZN7MyClassC5Ei,comdat
    .align 2
    .weak   _ZN7MyClassC2Ei
    .type   _ZN7MyClassC2Ei, @function
_ZN7MyClassC2Ei:
.LFB1:...
.LFE1:
    .size   _ZN7MyClassC2Ei, .-_ZN7MyClassC2Ei
    .weak   _ZN7MyClassC1Ei
    .set    _ZN7MyClassC1Ei,_ZN7MyClassC2Ei
```

图 11 第一个构造函数

```
    .section    .text._ZN7MyClassC2ERKS_,"axG",@progbits,_ZN7MyClassC5ERKS_,comdat
    .align 2
    .weak   _ZN7MyClassC2ERKS_
    .type   _ZN7MyClassC2ERKS_, @function
_ZN7MyClassC2ERKS_:
.LFB4:...
.LFE4:
    .size   _ZN7MyClassC2ERKS_, .-_ZN7MyClassC2ERKS_
    .weak   _ZN7MyClassC1ERKS_
    .set    _ZN7MyClassC1ERKS_,_ZN7MyClassC2ERKS_
```

图 12 拷贝构造函数

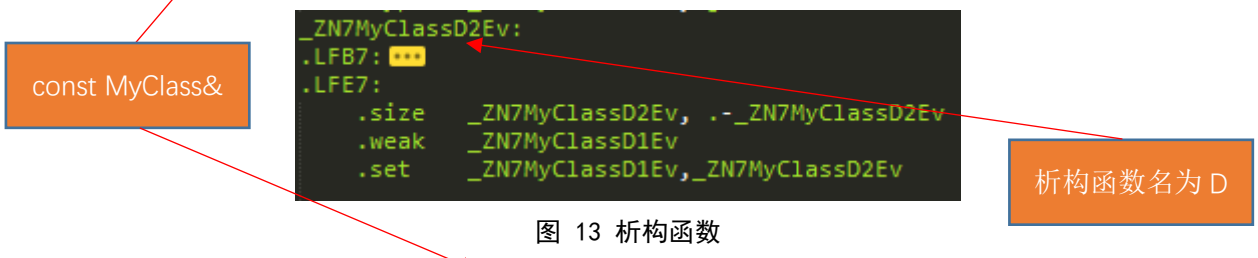

图 13 析构函数

```
_ZN7MyClassltERKS_:
.LFB10:...
.LFE10:
    .size   _ZN7MyClassltERKS_, .-_ZN7MyClassltERKS_
```

图 14 小于运算符函数

3. 引用、指针传递修饰规则探索

```
void inc(int& a, int b)
{
    a += b;
}
```

```
void inc(int* a, int b)
{
    *a += b;
}
```

图 15 测试代码

和按值传递唯一的不同是在类型前加上了修饰，引用位 R（reference），指针为 P（pointer）。另外，const 为前缀 K，加在 R 或 P 与类型之间。

图 16 按引用传递

图 17 按指针传递

(二) 多进程交替执行的研究

1. 保存当前进程的寄存器到 PCB

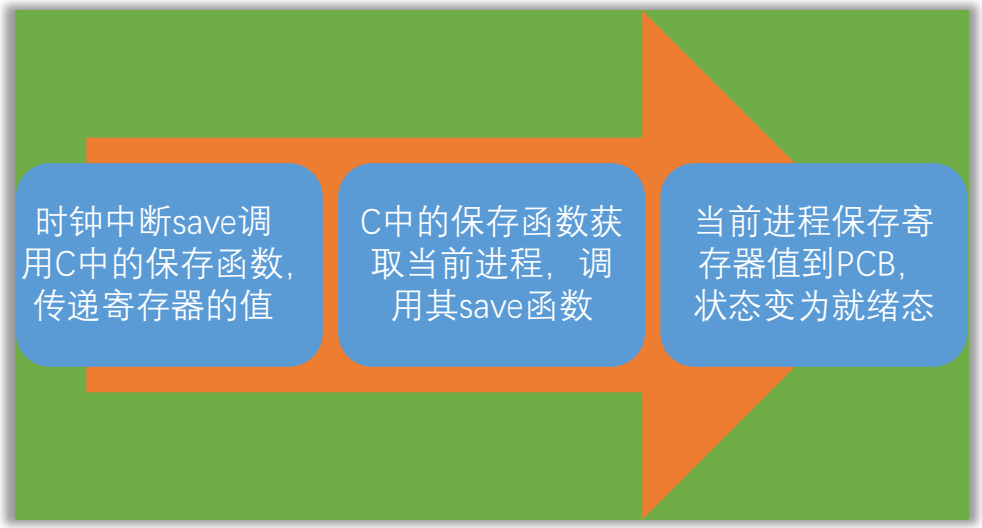

图 18 保存过程基本思路

保存的过程相对简单，只需要将所有寄存器的值 push 到栈，一路传递给当前进程就行。其中 ip、cs、flags 三个寄存器值的获取需要一点技巧。当时钟中断发生时，栈顶

的三个寄存器就是这三个，因此只要 pop 出来保存，然后以 4 个字节为单位 push 即可。代码如下：

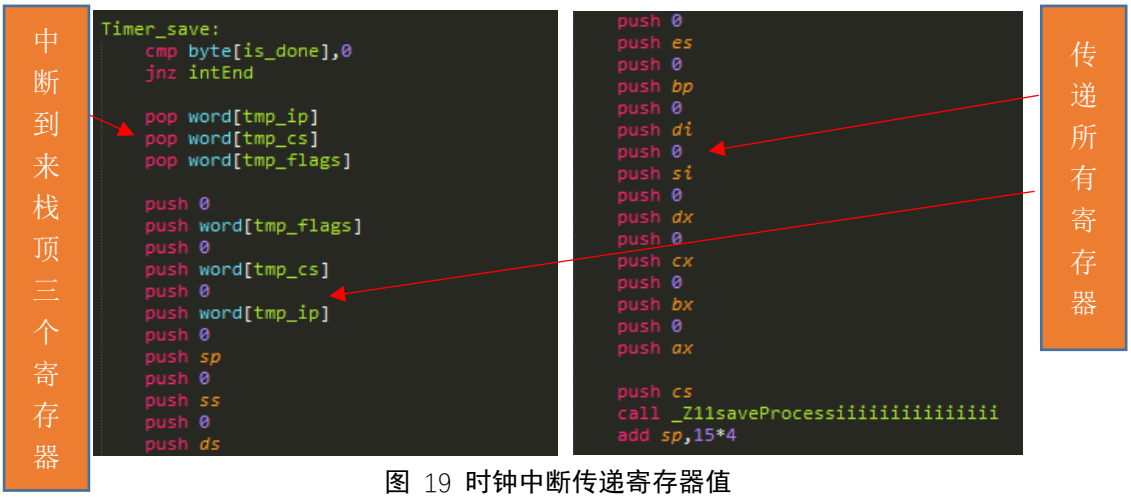

图 19 时钟中断传递寄存器值

```
void saveProcess(
    int ax, int bx, int cx, int dx,
    int si, int di, int bp,
    int es, int gs, int ds, int ss, int sp,
    int ip, int cs, int flags)
{
    ProcessTable& table = ProcessTable::instance();
    Process* cur = table.curPro();
    if(cur) {
        cur->save(ax,bx,cx,dx,si,di,bp,es,gs,ds,ss,sp,ip,cs,flags);
    }
}
```

图 20 获得当前进程并调用其 save

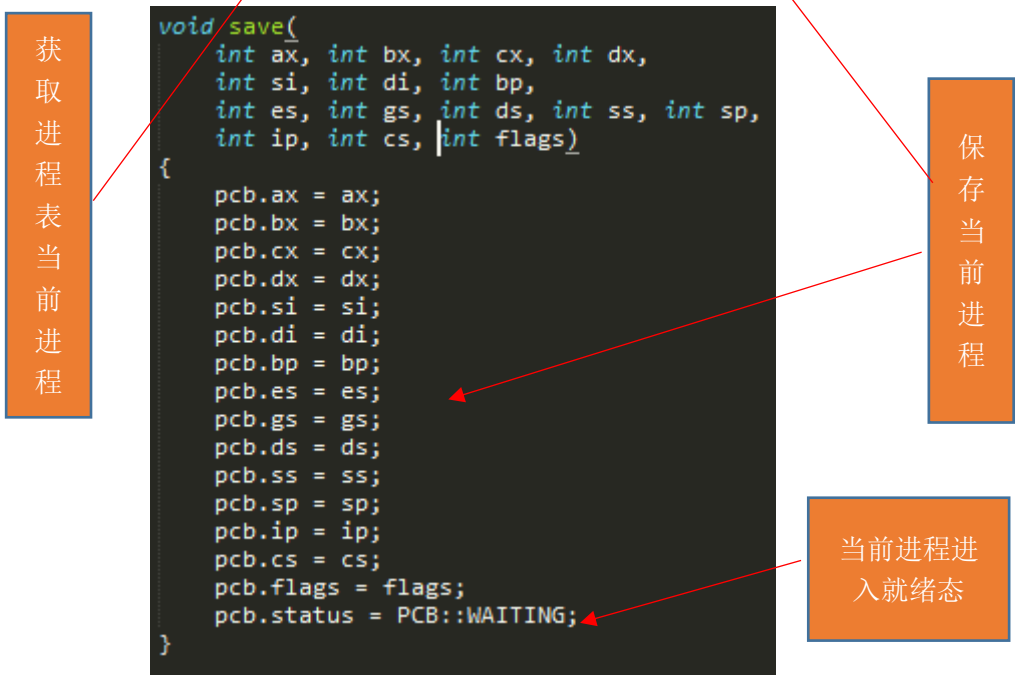

图 21 保存 PCB 并进入就绪态

2. 切换进程

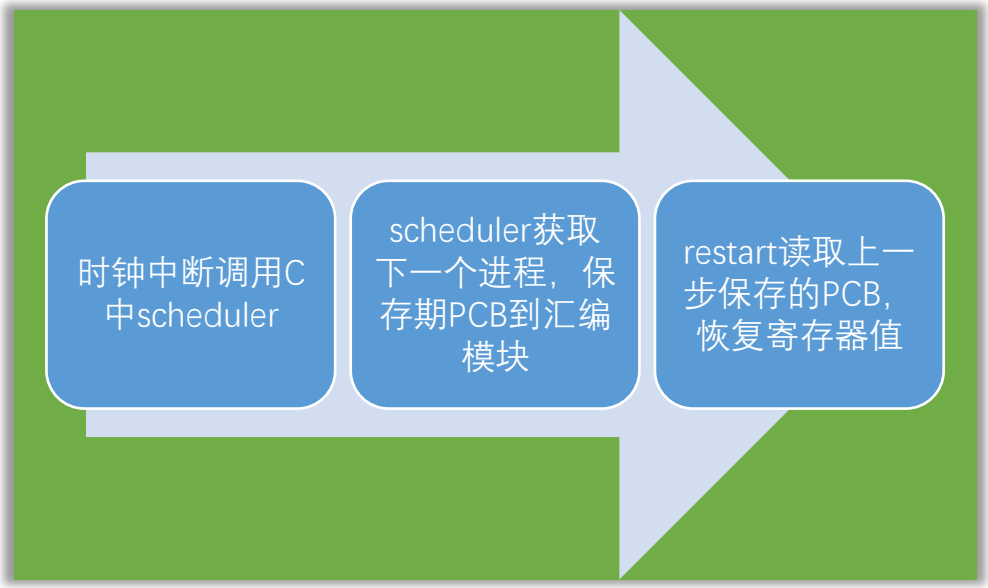

图 22 切换过程基本思路

我所做的进程切换逻辑与参考原型 2 有所不同。参考原型 2 在 restart 过程中直接获取到当前进程的 PCB 的指针放到 bp 中，然后用[ds：bp+结构体偏移量]的方式来访问当前 PCB。这在 PCB 数组是全局变量的时候可以通过 ds 存取，因为全局变量被 gcc 汇编后实际上是一个全局标签，存在于数据段。但是，一切都要从实际出发，原型只能给个思路，做法还是要适应自己的代码。由于我把进程的 PCB 作为 Process 类的成员变量来表示，而 C++中成员变量是在栈上分配的，不在全局数据区，也就不能通过 ds 来存取。当然，既然 PCB 在栈上，通过 ss 来访问也可以。但是如此一来，汇编是直接去访问 C 模块中的内存，这样的代码未免耦合度过高，也容易出错。

于是，我打算换一种 restart 的方法。restart 不直接访问当前进程，而是调用 C 模块的 scheduler，让它把当前进程的 PCB 信息保存到汇编模块中，然后 restart 去读取保存的临时 PCB，恢复一系列寄存器，然后切换到下一个进程。这样，与参考原型 2 相比，scheduler 就多了一项传递 PCB 的功能，也就是下面代码中的 saveProcess。

简单起见，我们非官方地把所有寄存器归为 3 类，非关键寄存器（包括 ax、bx、cx、dx、si、di、bp、es、gs）、段切换寄存器（ds、ss 和 sp）、控制切换寄存器（ip、cs、flags）。需要注意是，restart 时应该先恢复非关键寄存器，这是仍然在上一个进程的用户栈或操作系统的内核栈中，然后恢复 ss 和 sp，切换到下一个进程的用户栈，这时把 PCB 的 ip、cs、flags 入栈，最后 iret，就切换到了下一个进程，包括 CPU 控制权和用户栈的切换。

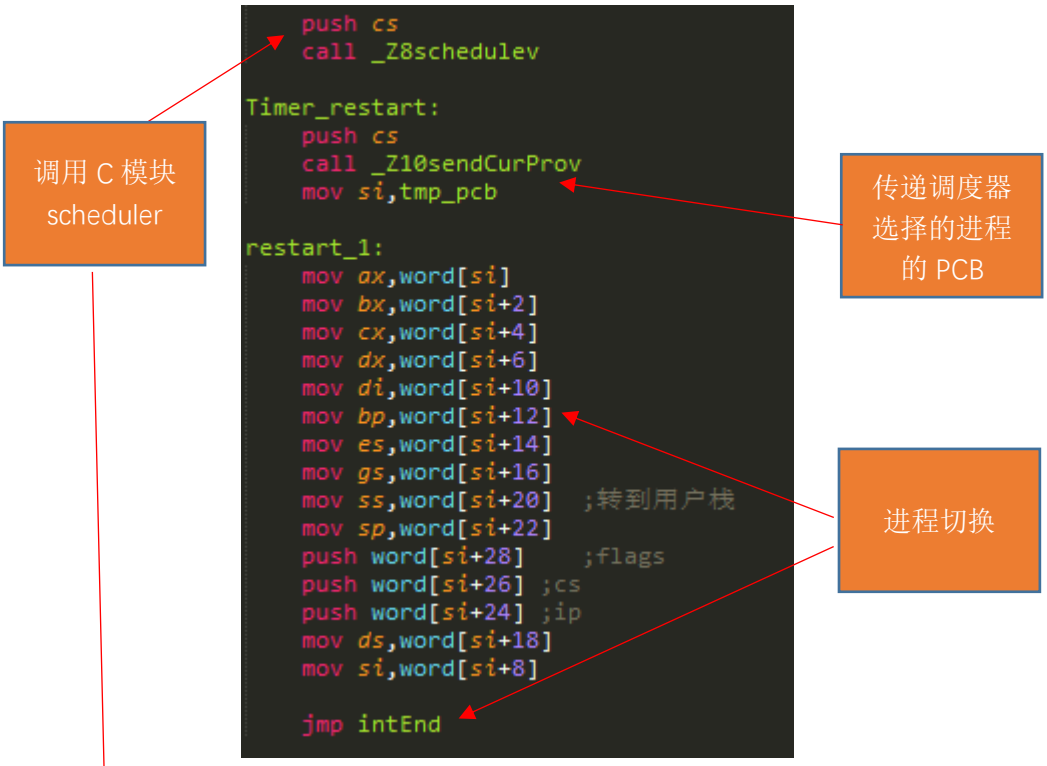

图 23 汇编进程切换 restart

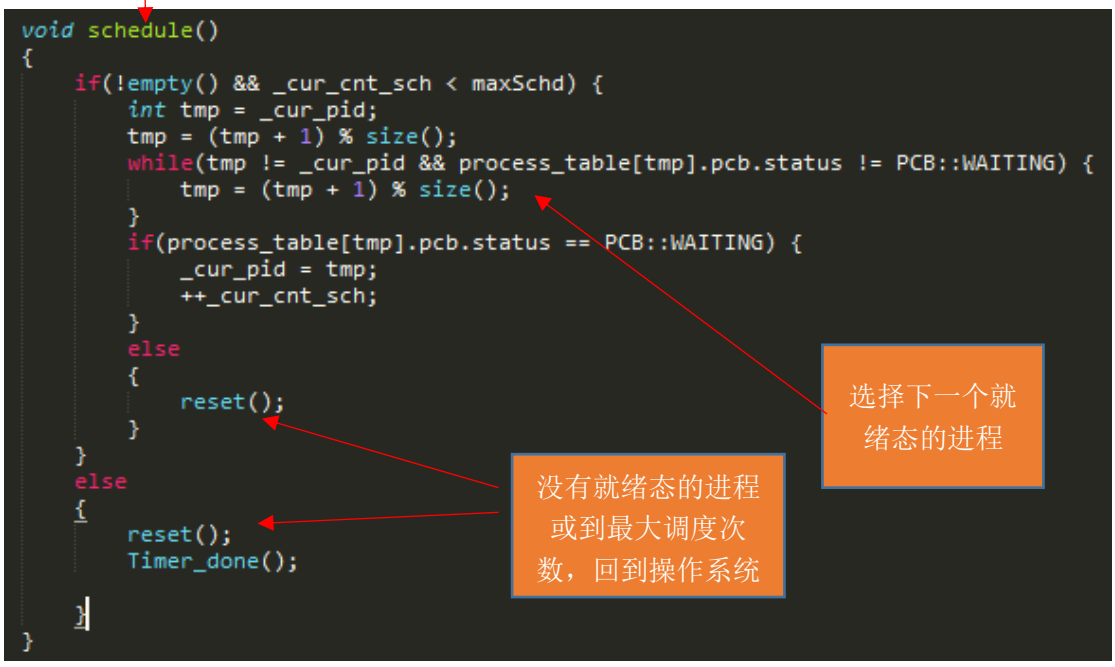

图 24 scheduler 选择下一个进程

```
void sendCurPro()
{
    Process* curPro = ProcessTable::instance().curPro();
    sendPCB(
        curPro->pcb.ax,
        curPro->pcb.bx,
        curPro->pcb.cx,
        curPro->pcb.dx,
        curPro->pcb.si,
        curPro->pcb.di,
        curPro->pcb.bp,
        curPro->pcb.es,
        curPro->pcb.gs,
        curPro->pcb.ds,
        curPro->pcb.ss,
        curPro->pcb.sp,
        curPro->pcb.ip,
        curPro->pcb.cs,
        curPro->pcb.flags);
}
```

传递进程的 PCB 到汇编

图 25 传递 scheduler 所选进程的 PCB

```
tmp_pcb: times 15 dw 0
GLOBAL _Z7sendPCBiiiiiiiiiiiiiiii
_Z7sendPCBiiiiiiiiiiiiiiii:
    push bx
    push di
    push si
    push bp
        mov bp,sp
        add bp,2*4+4
        xor di,di
        mov si,tmp_pcb
        loopPop:
            cmp di,15
            jae sendEnd
            mov bx,word[bp]
            mov word[si],bx
            inc di
            add bp,4
            add si,2
            jmp loopPop
sendEnd:
    pop bp
    pop si
    pop di
    pop bx
    retf
```

restart 要用来切换进程

图 26 汇编 sendPCB-保存到 tmp_pcb

3. 回到操作系统

当所有用户进程都执行完毕后，CPU 的控制权需要交回给操作系统。这里有两个难题：何时保存操作系统的寄存器到内核 PCB？怎么判断所有进程都结束了？

对于第一个问题，由于时钟中断是在操作系统内核安装的，因此第一次时

钟中断到来时，所有寄存器的值必定是内核的，这个时候保存内核寄存器。

对于第二个问题，这个实验还未做进程退出的系统调用，因此用户程序在执行完成后不返回（jmp $），因此最简单的做法是设置一个时间，超过这个时间后，scheduler 直接罢工，认为所有进程都执行完了，于是调用多进程结束的函数，restart 回操作系统。在实验中，我采取的做法是给 scheduler 一个最大的切换次数，超过了就罢工。

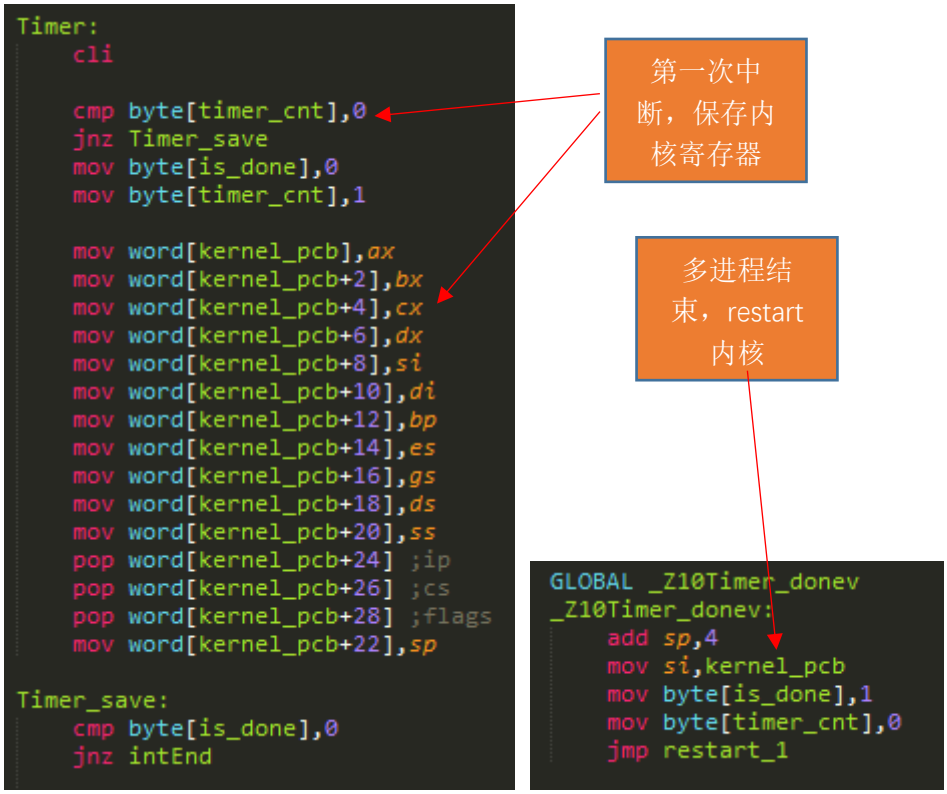

图 27 保存操作系统寄存器与回到操作系统

4. 测试

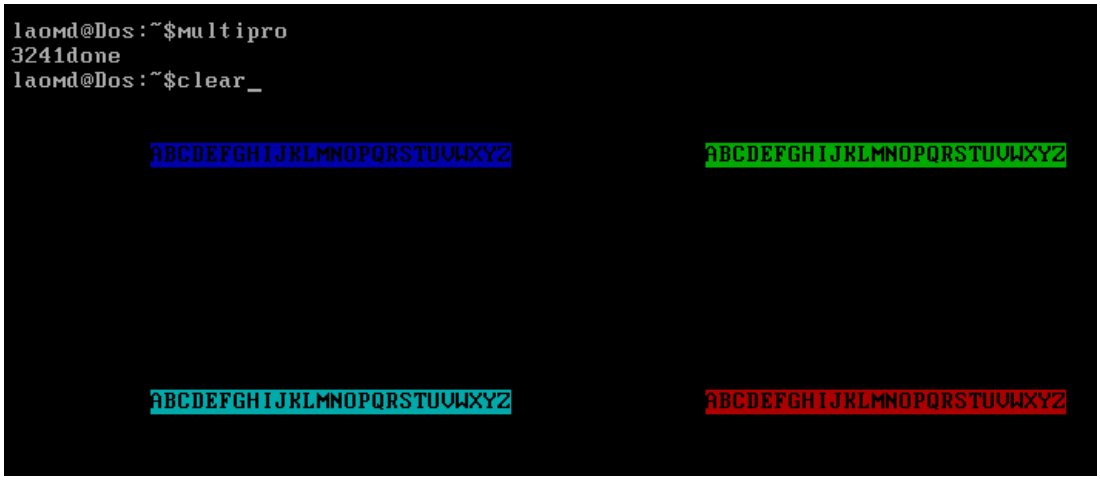

图 28 测试-多进程执行与回到操作系统

在该图中，终端输入 multipro，开始执行多进程，四个简单的用户程序分别在屏幕的 1/4 区域输出 A 到 Z，完成后分别打印 1、2、3、4，当所有用户程序都执行完成后返回操作系统，可以在终端输入命令 clear，说明正确回到了操作系统。

由于这四个用户程序调用了 21h 中断输出字符 1234，因此没必要再另外设计用户程序验证系统调用。

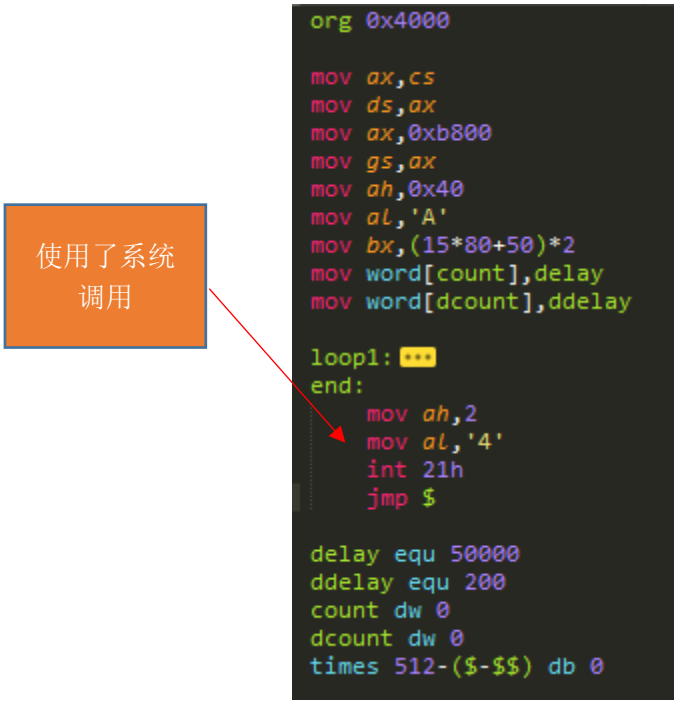

图 29 第四个用户程序

六、 【技术点与创新点】

（一） 工具创新：G++与 NASM

1. Name Mangling 概述

大型程序是通过多个模块构建而成，模块之间的关系由 makefile 来描述。对于由 C++语言编制的大型程序而言，也是符合这个规则。

程序的构建过程一般为：各个源文件分别编译，形成目标文件。多个目标文件通过链接器形成最终的可执行程序。显然，从某种程度上说，编译器的输出是链接器的输入，链接器要对编译器的输出做二次加工。从通信的角度看，这两个程序需要一定的协议来规范符号的组织格式。这就是 Name Mangling 产生的根本原因。

C++的语言特性比 C 丰富的多，C++支持的函数重载功能是需要 Name Mangling 技术的最直接的例子。对于重载的函数，不能仅依靠

函数名称来区分不同的函数，因为C++中重载函数的区分是建立在以下规则上的：函数名字不同|| 参数数量不同||某个参数的类型不同。那么区分函数的时候，应该充分考虑参数数量和参数类型这两种语义信息，这样才能为却分不同的函数保证充分性。

当然，C++还有很多其他的地方需要 Name Mangling，如namespace, class, template 等等。

总的来说，Name Mangling 就是一种规范编译器和链接器之间用于通信的符号表表示方法的协议，其目的在于按照程序的语言规范，使符号具备足够多的语义信息以保证链接过程准确无误的进行。

不同编译器使用不同的方式进行 Name Mangling, 所以一种编译器生成的目标文件并不能被另外一种编译器生成的目标文件使用。你可能会问为什么不将 C++的 name mangling 标准化，这样就能实现各个编译器之间的互操作了。事实上，编译器由于内部实现的不同而不同，内部实现包括对象在内存中的布局，继承的实现，虚函数调用处理等等。所以如果将 Name Mangling 标准化，不错，你的程序确实能够链接成功，但是运行肯定要崩的。因此，ARM 鼓励不同编译器使用不同的 Name Mangling 方式。这样在编译的时候，不兼容的库就会被检测到，而不至于发生“链接时通过，但是运行时崩溃了”的现象。显然，这是基于“运行时崩溃比链接时失败的代价更大”这个原则而考虑的。

GNU C++采用 IA 64 的 Name Mangling 方案，此方案定义于 Intel IA64 standard ABI。

2. g++函数名修饰规则

A. 全局函数：_Z+函数名长度+函数名+函数参数表

例如 C++模块定义了 foo(int, char)函数，修饰后的标签为_Z3fooic。以下为一些常用数据类型的修饰规则：

数据类型	修饰
void	v
bool	b
char	c
short	s
int	i
unsigned int	j
long	l
float	f
double	d
类	类名长度+类名
引用	前缀 R
指针	前缀 P
常亮	前缀 K

表格 1 常用数据类型与修饰符在函数中的简写

B. 命名空间/类函数

函数种类	格式
构造函数	_Z+N+类名长度+类名+C2+E+参数表（C-constructor、NE-namspace）
拷贝构造函数	_Z+N+类名长度+类名+C2+E+R[可选 K]S(RKS_是对该类对象的常引用，R-reference，K-const）
析构函数	_Z+N+类名长度+类名+D2+E+v（D-destructor）
运算符函数	_Z+N+类名长度+类名+运算符英文简写+E+参数表
普通成员函数、类静态函数、命名空间中的函数	_Z+N+命名空间/类名长度+命名空间/类名+函数名长度+函数名+E+参数表

表格 2 命名空间/类函数修饰规则

3. g++变量修饰规则

变量种类	格式
类静态变量、命名空间全局变量	_Z+N+命名空间/类名长度+命名空间/类名+变量名长度+变量名+E
类成员变量	成员变量不是以符号的形式出现，而是通过类对象的符号和偏移量的方法存取

表格 3 变量修饰规则

注意：对于单个字母的类名、函数名和变量名，修饰时名字长度是 1 而不会省略。

（二） C++库的建立

1. malloc 和 free

```
void* malloc(size_t nbytes)
{
    Header *p,*newp;
    size_t nunits;
    nunits=(nbytes+sizeof(Header)-1)/sizeof(Header)+1;
    if(memptr==NULL)
    {
       memptr->s.next=memptr=mem;
       memptr->s.usedsize=1;
       memptr->s.freesize=MEMSIZE-1;
    }
    for(p=memptr;(p->s.next!=memptr) && (p->s.freesize<nunits);p=p->s.next);
    if(p->s.freesize<nunits) return NULL;
    newp=p+p->s.usedsize;
    newp->s.usedsize=nunits;
    newp->s.freesize=p->s.freesize-nunits;
    newp->s.next=p->s.next;
    p->s.freesize=0;
    p->s.next=newp;
    memptr=newp;
    return (void*)(newp+1);
}
```

```
void free(void* ap)
{
    Header *bp,*p,*prev;
    bp=(Header*)ap-1;
    for(prev=memptr,p=memptr->s.next;
    (p!=bp) && (p!=memptr);prev=p,p=p->s.next);
    if(p!=bp) return;
    prev->s.freesize+=p->s.usedsize+p->s.freesize;
    prev->s.next=p->s.next;
    memptr=prev;
}
```

2. new 和 delete

```
void* operator new(size_t size)
{       // try to allocate size bytes
    void *p;
    while ((p = malloc(size)) == 0);
    return p;
}

void* operator new[](size_t size)
{
    return ::operator new(size);
}
```

```
void operator delete( void * p )
{
    free( p );
}

void operator delete[](void * p)
{
    ::operator delete(p);
}

void operator delete[](void * p, size_t)
{
    ::operator delete[](p);
}
```

图 30 new 和 delete 运算符

3. String

```
#include <string.h>

class String
{
public:
    typedef int size_type;
    typedef char value_type;
    typedef value_type* pointer;
    typedef value_type* iterator;
    typedef const value_type* const_iterator;

    String() : _data(nullptr), _capacity(0) { }

    String(const char* s) : String()
    {
        *this = s;
    }

    String(const String& other) : String()
    {
        *this = other;
    }

    ~String()
    {...
    }

    size_type size() const
    {...
    }

    bool empty() const
    {
        return size() == 0;
    }

    void push_back(char c)
    {...
    }
```

```
    void pop_back()
    {...

    }

    void reverse()
    {...
    }

    iterator begin()
    {
        return _data;
    }

    iterator end()
    {...
    }

    const_iterator begin() const
    {
        return _data;
    }

    const_iterator end() const
    {...
    }

    bool starts_with(const String& sub)
    {...
    }

    value_type* data()
    {
        return _data;
    }
```

```cpp
String& operator+=(const String& other)
{ ...
}

String& operator=(const char* s)
{ ...
}

String& operator=(const String& other)
{ ...
}

char& operator[](int index)
{
    return _data[index];
}

char operator[](int index) const
{
    return _data[index];
}
```

```cpp
bool operator==(const String& other) const
{
    return !(*this < other || other < *this);
}

bool operator!=(const String& other) const
{
    return !(*this == other);
}

bool operator<(const String& other) const
{
    return strcmp(_data, other._data) < 0;
}

bool operator>(const String& other) const
{
    return other < *this;
}
```

4. Vector

```cpp
#include <new>

template <typename T>
class Vector {
public:
    using size_type = size_t;
    using value_type = T;
    using pointer = value_type*;
    using iterator = value_type*;
    using const_iterator = const value_type*;
    using reference = value_type&;
    using const_reference = const value_type&;

//  constructors
    Vector() {
        init(0);
    }
    explicit Vector(size_type n) {
        init(n);
    }

//  destructor
    ~Vector() {
        clear();
    }

    size_type size() const { return _size; }
    bool empty() const { return size() == 0; }

//  iterators
    iterator begin() {
        return _data;
    }
    const_iterator begin() const {
        return _data;
    }
    iterator end() {
        return _data + _size;
    }
    const_iterator end() const {
        return _data + _size;
    }
```

```cpp
//  element access
    reference at(size_type n) {
        return _data[n];
    }
    const_reference at(size_type n) const {
        return _data[n];
    }
    reference operator[](size_type n) {
        return at(n);
    }
    const_reference operator[](size_type n) const {
        return at(n);
    }
    reference front() {
        return at(0);
    }
    const_reference front() const {
        return at(0);
    }
    reference back() {
        return at(_size - 1);
    }
    const_reference back() const {
        return at(_size - 1);
    }

    void push_back(const value_type& val) { ...
    }

    void pop_back() { ...
    }

    void clear() { ...
    }
```

七、 【实验总结】

这次实验的收获是这么多次实验以来最多的。其一，由 C 语言转为 C++，对 C++底层实现原理有了更深的理解；其二，解决了许多尘封已久的 bug；其三，对多进程交替执行原理有了新认识。

为了能够顺利从 gcc 转为 g++，我按照当初研究 gcc+nasm 的方法，对 g++编译生成的汇编代码做了许多分析，分别探索了命名空间、类中的函数和变量在汇编中的表示方法，也研究了类成员变量的存取方法。上个学期学 C++，感觉 C++中的一系列特性（函数重载、模板、namespace 等）十分神奇，也看过很多书去研究它们的实现原理。但问题是很多书对于这些原理的解释只是停留于理论层面，不涉及具体的底层汇编代码，因此也是看得将信将疑。这一次，我亲手把这一系列特性编译成真真实实的汇编代码来分析，研究之后发现，原来一切都那么简单——重载，只需要把函数汇编后对应的标签加上参数表的后缀；类和命名空间，就是加类或命名空间的长度和名字；模板，就是把模板信息和模板参数的信息拼接成标签……

对于多进程，我一开始的理解是多个进程同时并发地执行。要实现这样的效果，需要多核的 CPU，但是我们的实验好像没有提及多核？仔细再分析一遍发现，这个实验中的多进程实际上是多道程序交替执行。的确是多个进程，但不是同时执行，而是轮流执行。采用时间片轮转，每个进程都被分配一个时间片，当时间片用完时切换到下一个程序。这样，由于时间片极小，进程之间切换地很快，就造成了多个进程“同时”执行的假象。在进程切换中，保存和恢复现场是极其关键的过程。进程的时间片用完时，需要准确无误地保存它当前的状态，包括所有寄存器和它的用户栈；而恢复时，寄存器的恢复顺序也是很讲究的，从一个栈切换到另一个栈成功与否常常取决于此。

八、 【参考文献】

1. 初探 c/c++ 与 汇编 之间的交叉编译 命令行实现. CSDN G_Spider
2. 实验参考文献/蔡日俊 GCC+NASM 环境
3. 参考原型 2