# 引言

一个记录了Fulton于2016~2017年期间开发的一个操作系统的过程

制定计划，然后开始行动。

源代码见 ~/Desktop/bochs/devel/x2 system

Sometimes it’s difficult to see things to be desired.But it’s worth for trying and indeed we will finally succeed.

## 计划

撰写基本启动过程，重新理解.code16 .code16gcc .code32的区别(done，2017年2月10日23:52:08）

c+重构启动代码和进入保护模式代码（done，2017年2月11日23:44:05）

学习as高级汇编特性和ld脚本编写（done，2017年2月10日23:52:20）

**（不要熬夜，恢复基本的生活习惯，现在时间：2017年2月9日03:14:08）**

启动第一个任务，需要建立C++对于TSS的描述 （done，2017年2月12日19:20:05）

进入第3特权级并调用中断 (done，2017年2月13日01:40:22。兼重构Make以及宏)

启动定时中断 (done,2017年2月14日4:27:02)

编写键盘中断处理程序开始接受输入

分页策略和内存映射

基于类的全局资源管理器

进程管理

一个内存式的简单shell

文件系统

## 环境

windows10 bochs binutils

## 基础知识

编写引导器使用16位模式，架构选择8086

as宏汇编：

使用c++来编写原始汇编代码

访问限制是在编译器做出的

注意事项：

class定义要加分号

内联汇编：

\_\_asm\_\_() 不需要输入和输出，没有优化。用于函数体内外都可以

\_\_asm\_\_ \_\_volatile\_\_ (

code

:output

:input

:condition #可取”memory”,”cc”,”ecx”

) 输入的形式：”a”(

输出的形式：”=rm”(flags) r/寄存器/a=eax b=ebx c=ecx d=edx S=esi D=edi r=任意一个 g=寄存器、内存、立即数(由gcc确定) m/存储器 flags/变量名 i/立即数

m：当不希望通过寄存器而是直接内存操作时，可以通过声明”m”(var)，然后使用%n的形式来引用这个内存。譬如

\_\_asm\_\_ (“sidt %0 \n\t”: : “m”(loc));

q: a、b、c或者d

.codeXX系列指令在于指定为带有和不带尺寸指示的汇编语句生成何种机器码。.codeXX为带有XX位尺寸和不带XX尺寸的语句生成标准机器码，XX可取16，32。为XX之外的(16或32)位代码生成前缀。

.code16gcc不同于.code16的地方在于处理与堆栈相关的操作pop,push,ret,iret,int,call时生成的是32位代码，即带有前缀66。这样做的原因是为了和gcc调用c语言函数一致。因为c语言的函数框架默认是32位的。

声明静态常量

在类中以const/static声明，在类外初始化，一般在.cpp文件中，因为通常这些常量会被其他文件用到，但是初始化只进行一次。访问使用“类::域”访问。

默认参数

gcc编译器相关

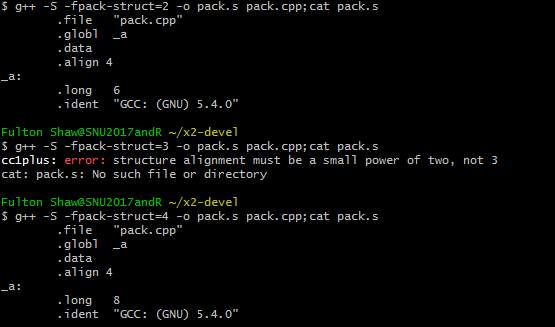
生成汇编文件： g++ -S main.cpp -o main.s

生成对象文件: as main.s -o main.o

链接（用于生成正确的地址）：ld main.o ...

\_\_attribute\_\_ ((section(“sec”)))：用于对一个函数重新定位

\_\_attribute\_\_((packed)):对一个结构体或者类不使用优化对其。但是这个用gcc和g++测试都没有作用。但是选项-fpack-struct=n(n是2的幂), n=1的时候，就是没有优化的。这已经测试：下图显示的2 ， 4的情况，a就是sizeof(struct)的大小。



\_\_attribute\_\_((interrupt)) 生成中断处理程序，这个特性可以用与ARM（如果用于ARM，还可以指定中断子类型）、m68k、H8/300、H8/300H、H8S、SH等架构上。如果用于intel的处理器上就会被忽略。

\_\_gxx\_personality\_sj0 相关错误：

main.o:main.cpp:(.test\_section+0x14)：对‘\_\_gxx\_personality\_sj0’未定义的引用

main.o:main.cpp:(.test\_section+0x4b)：对‘\_Unwind\_SjLj\_Register’未定义的引用

main.o:main.cpp:(.test\_section+0xd1)：对‘\_Unwind\_SjLj\_Resume’未定义的引用

main.o:main.cpp:(.test\_section+0xe1)：对‘\_Unwind\_SjLj\_Unregister’未定义的引用

这些符号的引入是为了异常处理，当使用g++ -fno-exceptions,就会去掉这些异常预定义符号。

gcc：编写内联函数替代宏

~~需要开启-O 优化输出文件才能对inline函数进行作用。但是-O选项的开启对于文件的生成不利，因为总是不能预测优化的程度。如果程序出错了，先考虑与-O是否有关。~~

与inline直接相关的不是-O，而是\_\_attribute\_\_((always\_inline))。

-Winline 对不能替换的inline函数进行警告

声明：

inline 将所有的inline声明函数替换成本地调用，如果能不引用函数地址，就不会保留函数定义。即 “一定替换（不管有多长），尽量不保留”,递归会破坏inline规则。当然，-O对于递归函数总是会预先生成多个递归前置。

static inline 同inline

extern inline 就是一个宏定义，所有的调用都会被替换，也尽量不保留定义。

注：这些声明可以和定义分开。

选项：

-finline-functions 将一些函数，如果能inline(如果代码很长就不inline，甚至1行代码也不会inline，适用情况几乎除了空函数没有），就使用其inline代码，但是会保持其原来代码。“尽量不inline，一定会保留”

-fkeep-inline-functions 为了防止定义被抹除而在其他文件中引用不到的情况，必须加上这个选项。

实际操作体验：跨文件的情况下必须将inline函数放在.h文件中使用，并且为了防止生成定义，必须使用extern inline。~~而由于-O的不确定性，最好还是放弃这种想法，尽管这是带有命名空间的宏。采用def.h的形式就可以了。~~现在知道了attribute always inline,就可以自由控制了。注意，在编写类的静态函数作为宏时，必须使用inline声明，尽管类的函数默认是inline的。通常的做法是在.h文件中声明和定义要使用的宏。标准的准则有三条，满足其中一条就可以使用：

1. 函数很少被调用 （有限次调用）
2. 函数体很短，主体只有1~3条语句
3. 一个函数调用必须以本地的方式实现，不能涉及堆栈操作。（就像曾经的ENTER\_DS)

还有一条但不是必须的：为宏预留命名空间。

宏替换的一个关键就是必须在编译阶段知道函数定义。

看gcc帮助：

-finline Enable inlining of function declared "inline",

disabling disables all inlining

-finline-atomics Inline \_\_atomic operations when a lock free

instruction sequence is available.

-finline-functions-called-once Integrate functions only required by their

single caller

-finline-small-functions Integrate functions into their callers when code size is known not to grow

-finline-limit=<number> Limit the size of inlined functions to <number>

-fkeep-inline-functions Generate code for functions even if they are

fully inlined

gcc：分区

函数： 在.text

静态量： 初始值为0 --> .bss

初始值不为0 --> .data

字符串常量： 在.rdata

静态常量： 在.rdata

gcc版本信息： 在.rdata$zzz

gcc：编写16位和32位通用代码

使用 #ifdef CODE16

\_\_asm\_\_(“.code16gcc \n\t”)

#elseif defined CODE32

\_\_asm\_\_(“.code32 \n\t”)

#

以及其他将函数区域包围的形式来确定代码的编译。但是另一种虽然不太美观但是有效的方式是将所有的通用函数复制两份，一份放在 #ifdef CODE16 ... #endif ,一份放在 #ifdef CODE32 ... #endif 下。不过我也发现如果真这样做的话，变量名、函数名都是冲突的。因此还是采用第一种方式，如果需要编译32位代码，就定义CODE32单独编译。

gcc：命令行定义宏

-D<macro>[=<val>] Define a <macro> with <val> as its value. If

just <macro> is given, <val> is taken to be 1

比如：g++ -E -D ‘CODE32(a,b,c)=a##b##c’ x.cpp

gcc:多行宏定义

#define MULTI\_LINE(a,b) ({\

...\

})

gcc:编写中断

相关属性：\_\_attribute\_\_((noreturn))

gcc：类函数的具体汇编内容

1. 构造函数是ecx作为this指针，参数个数正常
2. 析构函数ecx作为this指针，没有参数
3. 成员函数返回：void、普通类型、引用、非类，则ecx作为this指针，其他正常

如果返回一个本地引用：

T& T::returnRef()

{

T local;

return local;//返回的是 &local

}

void test()

{

T t1;

T t2=t1.returnRef();

T& t3=t1.returnRef();

T\* t4=&(t1.returnRef());

}

则实际上分为两个过程：t1返回地址，t2分配空间，然后调用t2的operator=函数。

t3,t4具有相同的作用，那就是取得返回值中eax的值，也就是local的地址。

不要与情况4搞混。然而返回一个栈引用之所以不正确可以通过汇编来看，因为函数返回之前的一件事情就是调用析构函数，析构函数的调用意味着对象不可用（因为析构函数可能改变了类实例的数据），但是如果析构函数没有影响，则正确。

1. 成员函数返回：类， 如果类的尺寸小于等于8字节，则同3，且eax返回第一个字节，edx返回第二个字节；否则，ecx是返回参数的地址，隐含第一个参数是this指针，所以实际参数个数多于声明个数一个，如果想要在内联汇编正确引用某个参数，必须是考虑到this指针的存在。在函数返回类的情况下，即使不使用 “=”来复制（=在c++中具有完全复制的作用，可以通过operator=重载，但是对返回class的函数不适用，因为那在返回函数的内部就已经做了） ，**函数的返回类型其实对于调用者来说也是相当于在调用者的空间申请的空间**）

这条规则的意思是，如果有下列语句：

class T{....};

T T::createT()

{

T temp;

....

return temp;

}

int main()

{

T t1;

T t2=t1.createT();

}

则方式是现在本地栈创建t2的空间，然后传入ecx，调用createT，在createT中使用ecx操作返回值。如果return语句中有return t;之类的语句，就会通过逐位复制来返回参数，通常这种效率很低下，直接操纵返回参数会更好。（通常所说的 allocate local, copy at return)

可以编写宏 classType\* target;\_\_asm\_\_ \_\_volatile\_\_(“”:”=c”(target)::);来获取返回参数的地址，target定义为classType \*.，这个宏有效性只能在函数的第一句保证。 规定用target来表示返回参数的地址。（在区分基本类型和引用类型方面，c++真是编程语言特别是脚本语言的鼻祖）。

如果仅仅是t1.createT();则由于没有变量接受，返回值被认为是废弃的对象，所以下一个隐含的调用就是调用这个废弃对象的析构函数。

**效率低下是必须强调的**

如果实际情况与上述不符，那是g++做的优化导致的。

还有一点：如果一个函数具有返回值但是调用者没有使用返回值，则返回值立刻被丢弃，普通类型丢弃就是不做任何事，如果是类，则类由于已经预先分配了空间，所以立刻调用析构函数。

g++调用返回类的的函数特点：先分配空间，再以ecx作为地址参数传递。

g++返回类的函数特点： copy on return。发生条件就是return temp;在汇编层面上与 (\*target)=temp；作用是相同的。所以copy on return指的就是return temp这句话。

g++返回类的函数与operator=有关系吗？ 有，在copy on return 阶段。

gcc:初始化列表

用于加速对象初始化，初始化列表的外侧变量，因为默认就是this中不允许使用this指针，而括号中的表达式则默认使用局部变量，可以使用this.

init(int x,int y):x(x),y(x+y)

则 x+y默认来自参数而非成员变量，可以使用y(this->x+y)特指。注意，初始化顺序和与定义顺序相同，而和列表定义顺序不同，因此最好不要使用变量之间的相互引用，存在这种引用的放到函数体里面执行。尽管这种引用不会报错，但是如果提前引用了后面定义的变量，结果就会未知。

gcc：生成代码

仅当文件中涉及对某个函数的调用，这个函数的定义才会被保留。

gcc：const static 的变量引用会涉及内存吗？

gcc：const属性的成员？

这种如果对这种成员进行写操作就会报错。但是这种变量不会被放到只读区域，因为它本身是属于类的。

gcc:内联函数宏的返回值

不要依赖eax，必须指明返回的类型，否则极有可能在eax有效值到达变量之前eax被修改，就像下面这条：

c语言 汇编

char code=k.readScanCode()； in al, dx

movzx eax, byte ptr ss:[ebp-92]

mov byte ptr ss:[ebp-9], al

int readScanCode()

{

\_\_asm\_\_(...);//默认对eax改变，但是没有明确返回值。

}

可以看到在赋值的中间eax被改变。

正确写法：

int readScanCode()

{

int temp;

\_\_asm\_\_(...);

return temp;

}

gcc：模板编程

模板语法：template<typename T>

class C{

};

如果想要特化某个方法，也必须指明是模板函数，即使没有模板参数，如下：

template <>

void V<int>::get()

{

}

模板为空的函数定义会被保留。

模板不为空的函数必须放在.h文件中，因为只有实例化模板的定义才能被保留。

g++:new

符号：\_\_Znwj

参数：unsigned int ... / sizeof(something)

返回值：void\*指针，（但是类型检查不受限）

cpp:

class T{

public:

int a;

};

void test()

{

T\* t=new T;

int c=t->a;

}

汇编代码：

\_\_Z4testv:

pushl %ebp

movl %esp, %ebp

subl $40, %esp

movl $4, (%esp) //接受一个参数

call \_\_Znwj

movl %eax, -12(%ebp) //返回指针类型

movl -12(%ebp), %eax

movl (%eax), %eax

movl %eax, -16(%ebp)

nop

leave

ret

g++:delete

符号：\_\_ZdlPv

参数：void\*指针

g++:异常

-fno-exceptions 禁止异常

-fexceptions 允许异常

-funwind-tables Just generate unwind tables for exception handling

-fasynchronous-unwind-tables Generate unwind tables that are exact at each

instruction boundary

-fenforce-eh-specs Generate code to check exception specifications

-fhandle-exceptions

-fnothrow-opt Treat a throw() exception specification as

noexcept to improve code size

-fuse-cxa-get-exception-ptr Use \_\_cxa\_get\_exception\_ptr in exception handling

-fnon-call-exceptions Support synchronous non-call exceptions

-fdelete-dead-exceptions Delete dead instructions that may throw exceptions

-fobjc-sjlj-exceptions Enable Objective-C setjmp exception handling

runtime

-fobjc-exceptions Enable Objective-C exception and synchronization

syntax

g++：类继承

构造函数：不继承构造函数，但是1.若父类没有默认构造函数，则子类的构造函数必须明确调用一个父类构造函数 2.若父类有默认构造函数，则子类不必显式调用，但会生成调用父类构造函数的代码

而且必须注意：所有的调用都必须在初始化列表中完成。

同名成员变量：不会被覆盖，可以通过 Son.Father::a来引用属于Father的变量a。

同名函数：不会被覆盖，正常的调用。

不存在的函数：继承。

virtual关键字：仅仅在多态中用得着。即仅当一个类指针指向自己的子类时，如果调用的函数时virtual函数，则会调用子类的该函数，否则就调用父类函数。其他时候没有作用。在生成代码时，会依据当前指针取寻找该函数的偏移。

as：.linkonce discard|one\_only|same\_size|same\_contents

这条指令用于.section指令下，对分区进行说明。所有的同名分区都会标记为一个分区。然后根据后面的指令来发出警告或者丢弃分区。

ld：选项

-T 使用脚本文件

-nostdlib 不使用标准库

-l 标准库，如-lc代指libc.a，以.a结尾

-Ttext/-Tbss/-Tdata hexaddr 指定相应标准段的起始地址,可以省略0x前缀

ld：错误

main.o:main.cpp:(.text+0x1): relocation truncated to fit: 16 against `.text'

在.text中使用了16位的地址。

原因在于，ld假定.text段的地址范围是32位，在重定位的时候如果遇到一个16位的偏移地址，则ld认为如果前面的.text基址超过16位，这里的16位就不足以容纳正确地址，所以拒绝链接。

解决方法：加上 -Ttext 0指定.text偏移地址，ld经过计算会发现其可容纳。

ld：分区

as允许的符号是由 字母，数字和\_.$组成的，不允许数字开头

标准分区: .text .data .bss 按顺序存放，如果一个区域没有指定分区，默认放入.text分区

absolute区：地址始终保持不变，但是不存在.absolute分区

undefined区：对于一个未知section的地址的引用就处于undefined分区

.data [SECNAME=0]：声明一个.data子分区

.section A

.subsection 1 声明一个任意子分区

.struct EXPRESSION 跳转到absolute分区的EXPRESSION偏移处，之后将停留在absolute分区

.offset LOC 将location counter设置到absolute区绝对地址，不要与org搞混。此命令用于定义与绝对分区相关的符号等。struct和offset指令都只能用于定义符号

.rdata$zzz 存放'GCC: (GNU) 5.4.0'之类的版本信息

符号.: .指代当前地址，对.的赋值就是org指令

ld：如何在一个绝对位置定义内存？

通过将待指定的变量定义到一个自定义的分区，然后在脚本中指定起始位置。如果存在对其问题，就再外加.org来指定位置

ld：脚本教程

SECTIONS命令是必须的，它规定了输出文件的布局，如位置和顺序等。SECTIONS内部的语句可以完成三件事情：

定义起始地址

给符号赋值

定义由多个区组成的单个区的位置

MEMEORY则用于补充SECTIONS的描述，它描述了目标架构上的可用内存。全局至多有一个MEMORY描述

注释使用/\*\*/

数学表达式：所有的表达式计算结果都是long或者unsigned long，所有的C操作符都是可用的，可以使用变量，可以调用内置函数

后缀：K,M 4K=4096,即K=1024

symbol约定：使用:””双引号可以包括任意符号，如 ”This is a variable” = b+1,不使用双引号则必须符合C语言规则，不能与关键字冲突（双引号可以做任何事）。不使用双引号的符号必须以字母、下划线、点号开始，后接字母，下划线，数字，点，连字符。 A-B是一个符号，A - B是一个减式。必须在赋值表达式之后使用分号；，PHDRS后也必须，其他情况可选。

. : 总是指向当前位置，必须出现在SECTIONS指令内，是一个上下文变量

函数：

ABSOLUTE() //即使在一个section内部，该符号也被赋予绝对值

如 \_edata = ABSOLUTE(.);

ADDR(section) //返回section的绝对地址

ALIGN(exp) //和NEXT相似

DEFINED(sym) //1 0,用法 DEFINED(begin)?1:0

NEXT(exp) //返回最接近于exp倍数的未分配空间地址

SIZEOF(section) //

SIZEOF\_HEADERS

MAX(exp1,exp2)

MIN(exp1,exp2)

MEMORY{

name (attr) : ORIGN/org/o = origin, LENGTH/len/l = len(如4M)

...

} name适用于linker内部，可以与symbol冲突。

attr可用 : R W X A(已经分配) I(已经初始化) L(同I) !(not操作符) 表示这个区间可以容纳的section类型

在SECTIONS中可以使用 > mem 来指定放置的位置。如果区间不足以容纳就会报错。

SECTIONS{

secname : { /\*定义输出section，其内容，内存位置。secname可以使用””\*/

contents

}

} /DISCARD/区将会被丢弃

ld会丢弃没有任何内容的区。

contents包括

filename 指定哪些文件的区会被包含到此区,

.data : { afile.o bfile.o cfile.o }

filename( section , section, ... ) 同上，“，”可以换成空格。\*意味着所有未提及过的文件。section=COMMON时表示放置未初始化的数据。filename支持通配符。

典型脚本

SECTIONS {

.text : { \*(.text) }

.data : { \*(.data) }

.bss : { \*(.bss) \*(COMMON) }

}

例子:

SECTIONS{

.sec1 0:{

main.o(.text) main.o(.rdata)

}

}

和

SECTIONS{

.sec1 0:{

main.o(.text)

}

.sec2 :{

main.o(.rdata)

}

}

有很重要的区别，前者中如果rdata有符号，则符号的值还是相对于0的；后者中的则相对于.（当前值）。采用后一种脚本才能得到正确的符号引用。

objdump:

使用-m i8086查看16位汇编代码

## 引导器和启动过程

跳转到0x7c0:起始处

重新设置ds,es,ss,sp

读取剩余的扇区然后移动数据，进入第一个内核程序

内核程序1：

启用A20，设置idtr,gdtr，进入保护模式

再重新设置cr3，gdtr中的表，设置分页

初始化各个资源管理器

进入第一个shell

如何进入保护模式：

mov %cr0,%eax

or $0x1,%eax

mov %eax,$cr0

ljmp $SEG\_SEL,$OFF #注意，这里的SEG，OFF只能使用立即数而不能使用。当然，如果ljmp也能使用寄存器就好。

问题1：如何让代码接着执行？

在OFF:处，使用.code32使之可以继续执行。但是最好不要将16位代码和32位代码掺和在一起。

或者在进入保护模式前，读取32位代码到指定位置。

总之，在进入32位代码之后必须和所有的16位代码说再见，因此中断也不能再用了。

问题2：ljmp这条语句算32位还是16位代码？

因为ljmp在内容上还属于16位代码文件，理应和16位代码放在一起，但是此时代码已经进入了32位模式，应当按照32位代码编译。

也就是说，位置上和16位代码一起，编译上属于32位代码

## 代码布局和内存分配

0~15 扇区（按照LBA编址）属于实模式，其中前两个分区用于栈

0x7c00~0x9c00

0x9c0:0是第一个可用的空闲地址

16~49 扇区属于保护模式核心代码

将会被加载到 0x0~0x4400，前4个扇区作为栈

因此0x440:0又是第一个可用的地方

留10个TSS结构

总大小：50\*512 = 25600

## 全局常量

由gcc处理，在Makefile中 DRIVER SECSTART SECNUM CODESEG CODEOFF,规定了保护模式代码所在的驱动器，逻辑扇区（从0开始），占有的扇区数目，被读入的段和偏移

由as处理，在Makefile中JMPSEG JMPOFF，规定跳转的保护模式代码的段和偏移。

gcc: SEG\_CURRENT = 0x10000; 表示取当前值

TSS\_MIN\_SIZE = 104; TSS最小结构

TSS\_AREA\_START = SEC\_NUM\*512,TSS\_AREA\_SIZE //为TSS\_AREA\_START分配的空间，相对于0的偏移

IDT\_START,IDT\_SIZE

GDT\_START,GDT\_SIZE

这些参数的写入方式见Makefile中对于数组imageValues和args参数的处理

## 关系类图

pLDT pTSS只能存在于GDT表中

pDescriptor也叫门

所有GDT管理的项都叫Descriptor

## 分页策略

## BIOS中断参考

int 0x10

int 0x13

dh 磁头号

dl 驱动器号（0号驱动器，0磁头）

cx 高10位磁道号，低6位扇区，从1开始

ah 功能号 1

2 读取软盘

al 读取的扇区数目

es:bx 读取的目的地

返回值，如果进位被设置(jc)，就是出现错误

## 汇编语言参考

串指令(操作数都是隐含的)

lods ds:si --> al

stos al --> es:di

cld 清除方向位

movs 串复制

cmps

scas

rep cx!=0

repe cx!=0 and zf==0

ins/outs

ljmp指令：

ljmp $,$ -- 即两个操作数都是立即数

ljmp

lxx/sxx指令：

ltr/str

lidt/sidt

lldt/sldt

lgdt/sgdt

iret指令如果涉及特权级切换，堆栈顶部如下：

00 SS

ESP

EFLAGS

00 CS

EIP

enter/leave:

enter $0,$0

...

leave

IDT的结构是另一种，它里面的描述符的基址是指向LDT或者GDT的

jmp用法：

jmp .

jmp 0

jmp HERE

$ cat jmp.s;objdump -D jmp.o

.code32

jmp .

jmp 0

jmp 5

jmp \*0

jmp \*5

jmp.o： 文件格式 pe-i386

Disassembly of section .text:

00000000 <.text>:

0: eb fe jmp 0 <.text>

2: e9 03 00 00 00 jmp a <.text+0xa>

7: e9 0d 00 00 00 jmp 19 <.text+0x19>

c: ff 25 00 00 00 00 jmp \*0x0

12: ff 25 05 00 00 00 jmp \*0x5

pusha(16位） pushad（32位）：依次压入如下寄存器：

eax ecx edx ebx esp ebp esi edi

as中没有pushad指令，只有pusha指令

EFLAGS各个位：

0 1 2 3 4 5 6 7 8 9 10 11,12 13 14 15 16

CF 1 PF 0 AF ZF SF TF IF DF OF IOPL NT 0 RF V

push 段寄存器：

push %es -- 32位代码下会压入4字节

pop %es -- 弹出4字节，其中低2字节给es

0: 06 push %es

1: 66 06 pushw %es

3: 6a 00 push $0x0

5: 66 07 popw %es

7: 66 07 popw %es

9: 07 pop %es

上述代码表明只有后缀w能够强制开启16位模式。

lss和pop/mov ss：

由于中断可能在任何时候发生，设想下面的代码

1：mov %eax,%ss //ss选择子

2：mov $512\*4,%eax

3：mov %eax,%esp

4：mov $512\*4,%esp

设想1之后异常发生，那么，栈就处于不一致的状态。所以intel强制任何对ss的pop/mov操作，直到下一条指令完成之前，都不会发生中断。所以可以肯定的是，1、2之间不会有任何中断，但是4之前都有可能。

所以正确做法是在pop/mov ss之后立即对esp进行改变。或者转而使用lss。

中断分类：

interrupts：

hardware interrupts -- 硬件产生

software interrupts -- int n指令产生

exceptions

program error

software generated -- intO ,int 3,BOUND指令 以及其他由int n指向的内定异常，但是这样引发方式不会push出错码，也就意味着可能发生未知的错误。int n如果指向trap则是引起了trap。

machine-check

异常分类：

Faults -- 返回指令之前

Trap -- 返回指令之后

Aborts -- 不应当返回

## 硬件参考

1.44 M软盘：2（磁头）\*80（磁道/柱面）\*18（扇区）\*512 = 1.44M

LBA = (S-1) + H\*18+C\*18\*2

\* C = LBA/18/2

\* H = (LBA-18\*2\*C)/18

\* S = LBA%18 + 1

视频缓冲区：0xb800:0~0xb800:0x0f9f

字符属性：Flash(7) BG(6~4) FG(3~0)

R G B = 1 0 0 : 红色

8259A（主片和从片编程）：

主从芯片的地址是 0x20,0xA0,0x21,0xA1.

发送到A0=0，且D4=1,表示设置ICW1

如果在ICW1之后发送且A0=1，则是ICW2

如果是在ICW2之后发送且A0=1，且方式为嵌套方式，则是ICW3

如果在ICW2或者ICW3之后发送且ICW1明确需要ICW4且A0=1，则是ICW4。

发送到A0=1则为OCW1

发送到A0=0,D4D3=00,则为OCW2

ICW1用于设置中断触发方式，是否多片8259A级联，是否需要ICW4

ICW2用于设置中断的起始地址高5位。

ICW3主片表示端口的级联方式，从片低3位表示连接到主片的位置，高5位为0

ICW4 设置嵌套方式、缓冲、自动结束、兼容系统

OCW1 用于设置要屏蔽的中断

OCW2用于发送EOI（非自动结束时发送）

linux所采用的命令字

主 从

ICW1 0x11 0x11

ICW2 0x20 0x28

ICW3 0x4 0x2

ICW4 0x1 0x1

OCW1 无 无,

OCW2 0x20 两次0x20 0x20

8253/8254编程

该芯片拥有3个独立的计时通道，每个通道依次用于中断、动态DRAM刷新、主机扬声器合成，3个计数器的输入时钟频率都为1.193180MHz.

其IO端口地址为0x40,0x41,0x42,0x43,前三个对应各个计数器通道，0x43对应控制器写端口。计数通道0被连接到8259A的主片0级中断请求。

初始时（系统上电时）BIOS将通道0设为方式3（方波发生器），初始计数器位0（被当作65535），因此频率为1.193180MHz/65535=18.2Hz,这是最小频率，也就是说最大的时间间隔。Linux系统设置的计数器为1193180/100，间隔10ms，方式3（方波）。向8254写入的控制字为0x36,随后写入计数器的低字节和高字节。

控制字格式：

7~6 选择计数器通道 0x3回读命令

5~4 选择读写 00：锁存寄存器 01:低字节 10：高字节 11：先低字节，再高字节

3~1 工作方式 000 011：方式3，方波发生器

0 格式选择 0：16进制 1：BCD

键盘：

现在使用的标准键盘是PC/XT，这种键盘中，多数普通的键具有一个固定的扫描码，当按下一个键和松开一个键时会产生依据最高位（位7）区分的接通码和断开码。而其他一些扩展键，比如右边的ctrl键，则会先产生一个0xe0前缀，然后才是常规的接通和断开码。

0xe0实际上一个转义字符，如同串行网络协议中的转义字符一样。

另外PrtScn产生的序列是：

0xe0 0x2a 0xe0 0x37

如果一直按住，则附加一个后缀0xaa。松开则产生对应的0xe0加上断开码。

如果按住shift或者ctrl，则仅发送0xe0 0x37，松开时对应。如果按住alt，具有普通扫描码（接通和断开）0x54。

而Pause产生的序列是：

0xe1 0x1d 0x45 0x9d 0xc5

如果按下了任意ctrl键，将产生 0x46， 而其他任何情况都是上面的序列（包括按住、外加其他键的情况），如果松开不会产生断开码。

可以这样认为，0xe0意味着后面还有1个字符，0xe1则意味着有两个。

上面的规则被称为扫描集，目前有三种扫描集，键盘默认采用第二套扫描集，但是在内部会将所有的扫描集转换为第一套。

检测键盘布局的”算法”：（本地算法）

写一个简单的中断，每次都读取扫描码，当扫描码是接通码时，打印预定义的字符，将其和本地字符对比，然后逐步修正，直到所有的键修正完毕

## Make参考

**规则**

target:

(tab)cmd

**指令**

.phony 不检查文件

.oneshell

If '.ONESHELL' is provided, then only the first line of the recipe

will be checked for the special prefix characters ('@', '-', and '+').

注意，该命令之后的所有代码都被影响

@ 关闭指令回显

- 忽略错误继续

# 注释

.secondexpansion

多数情况下make变量在第一次扩展时已经展开，但是如果$被$所转义，则会被第二次展开。

.SECONDEXPANSION:

AVAR = top

onefile: $(AVAR)

twofile: $$(AVAR)

AVAR = bottom

onefile被立即展开，而twofile则是AVAR的最后值。

但是有意义的用处是可以使用特殊变量 $@ $\*等

$@ 引用目标的名称

$< 第一个前提

$^ 所有的前提

$+ 所有的前提（有重复）

$? $\*无效

注意，一个目标多次出现，最终值为最后一个。这个指令只影响它后面出现的所有目标，并且不影响命令，只影响前提集合。

.SECONDEXPANSION:

main\_OBJS := main.o try.o test.o

lib\_OBJS := lib.o api.o

main lib: $$($$@\_OBJS)

.SECONDEXPANSION:

foo: foo.1 bar.1 $$< $$^ $$+ # line #1

foo: foo.2 bar.2 $$< $$^ $$+ # line #2

foo: foo.3 bar.3 $$< $$^ $$+ # line #3

**变量引用**

$(foo) ${foo} 都是等价的

$foo 相当于 $(f)oo

$($(foo)) 相当于2级指针，也叫Computed Names引用

**赋值**

= 递归赋值。使用原则：不要使用 a = $(a) -o , 因为这是递归进行的。当变量扩展中有后出现的变量时，使用=。多数情况

:= 或者 ::=(由POSIX描述) 直接赋值

objects := program.o foo.o utils.o

注：值从=后的第一个非空白符算起，直到换行符。make中只有一种类型：字符串。

+=

!= 操作符 $(shell ...)

1 x1 = 1 2 3

2 x2 = 3 4 5

3 x3 :=$(shell a=($(x1));b=($(x2));for i in $$(seq 0 $$(( $(words $(x1)) - 1 )));do echo $${a[$$i]}.$${b[$$i]};done)

数组操作的标准模板

**函数（使用${} $() 来call）**

shell

subst FROM,TO,string -- string中的FROM被换成TO

patsubst PATTERN,REPLACE,TEXT -- TEXT中的PATTERN被换成REPLACE。

PATTERN %匹配任意个字符,并且是贪婪式的，也就是c.c.c 将会匹配最后一个.c

\ 转义特殊字符

如 $(patsubs %.c,%.o,a.c b.c c.c)生成 a.o b.o c.o

filter PATTERN...,TEXT -- PATTERN可以是数组，返回过滤

word N,var -- 将var视为以空白字符分割的数组，返回自下标1起的字符

words var -- 相当于数组长度

wordlist nstart,nend,var -- 返回数组切片

firstword/lastword var -- 返回第一个/最后一个

（文件类函数）

dir NAMES... -- 返回数组中每一个文件名的目录 src/x hack 返回 src/ ./

**条件**

ifeq (a,b) //全部按照字符串条件

....

endif

## 编程错误和启示汇总

c语言中任何时候，使用 &操作符必须确保正确，所以加括号是必须的

实模式下，不能将数据直接读到0x50:0以下的地方，因为那是中断处理程序和数据所在的地方。必须先将其读到其他地方，再进行移动。并且在覆盖之后就不能再使用那个区域了

ld脚本中定义的空区(如.stack，预留的栈区)可能不会被objcopy保留，即使加上-j .stack，而在原本文件中就存在的区一定能被保留。

/\*

.stack 0 : {

. = 512\*4;\*/ /\*栈占有4个扇区\*/ /\*

}

\*/ /\*This is discarded by objcopy even said -j .stack\*/

.stack 0 : {

protected\_main\_32.o(.stack)

}

注释中的stack和没有注释的唯一区别就是产生方式，然后objcopy 对前者进行了舍弃，而后者得以保留。

实模式下在调用了lidt/lgdt之后再调用int 0x13出错。原因未知，最后在调用lidt之后就不再使用中断。

段描述符中不应当将向下扩展的数据段看成栈，而应把ss指向的正常数据段看作段。这是linux 0.00 所采用，也是intel指南中说明的。

在进入保护模式应当刷新所有的段寄存器而非仅仅刷新ds,ss。因为如es等也经常被用到，而且一般其使用方法都是

push %es

...

pop %es

如果没有刷新es的值，如其仍旧等于0x7c0，则pop的时候有可能出错。这已经被证实。

as的-(-)defsym存在bug，如果-defsym A=0b110,-defsym B=4\*512,那么最后的值实际上是A=0,B=4会将后面所有的值舍弃。如果定义的值是作为立即数（在ljmp中使用，则不能将--defsym放到ld），应当将定义的符号写成0x,整数的形式。

ltr指令异常情况包括：加载busy的tss结构。不可能加载一个busy的任务。

gcc中可以在class中声明一个函数，但是不定义它。只要不引用这个函数就不会存在错误。同时，声明中的static在定义中可以省略，但是const不能。

gcc的static数据在编译器是常量，可以用前面定义的数据来定义当前数据。譬如

static int a,b,c;

int a=0,b=1;

int c=a+b; //这里c可以在编译器算出，尽管a,b都是变量。

这说明g++还是有很大的智能的。

如果在\_\_asm\_\_中定义了一个符号，想在c++中使用怎么办？使用extern声明即可。

相关选项 -fleading-underscore Give external symbols a leading underscore

extern int extern\_sym;

\_\_asm\_\_(

"\_extern\_sym: \n\t"

);

int test()

{

int a=extern\_sym;

}

但是必须注意一点，a=extern\_sym编译出来的语句是

mov \_extern\_sym,%eax

mov %eax,a

也就是说，这个extern\_sym被视为整数时，其值实际上就是该处的代码值，而不是地址值。为了获得标号的值，可以使用int \*a=&extern\_sym.这也提醒我们，可以通过这种方式获得代码。

**其他用法：参见下面的代码**

extern int a;

extern "C" void x();

extern int &e;

extern int\* &o;

extern int z[];

extern int \*y;

void test()

{

int b,c,d;

//b=&(int)a; //impossible

c=x;

c=&x;

c=e;

c=&e;

c=o;

c=&o;

c=a;

c=&a;

c=z;

c=&z;

c=y;

c=&y;

**}**

汇编代码：

movl $\_x, -4(%ebp)

movl $\_x, -4(%ebp) -- 函数本身直接使用就是标号值

movl \_e, %eax

movl (%eax), %eax

movl %eax, -4(%ebp) -- 引用被当作二级指针（如果标号本身算是立即值）

movl \_e, %eax

movl %eax, -4(%ebp) -- 引用的地址是一级指针

movl \_o, %eax

movl (%eax), %eax

movl %eax, -4(%ebp)

movl \_o, %eax

movl %eax, -4(%ebp) -- 指针的引用还是引用，被当作二级指针。

movl \_a, %eax

movl %eax, -4(%ebp)

movl $\_a, -4(%ebp) -- 基本类型被当作一级指针

movl $\_z, -4(%ebp)

movl $\_z, -4(%ebp) -- 数组类型被当作标号

movl \_y, %eax

movl %eax, -4(%ebp)

movl $\_y, -4(%ebp) -- 指针是基本类型

**如果内联汇编有标号，则将其要么申明为数组类型，要么声明为函数类型。**

不要妄图改变标号本身的值。因为c++中函数名本身是一个标号，而标号是不可以被改变的，即使使用.set也不可以。（下面的做法试图将一个函数名称修改为真正的中断起点）

extern "C" void int0x24()

{

\_\_asm\_\_(

"INT\_24: \n\t"

“enter $4,$0 \n\t” //esp指向当前值，所以为了防止被修改，将其下移4个byte

);

Util::printStr();

\_\_asm\_\_(

“leave \n\t”

"iret \n\t"

);

}

\_\_asm\_\_(

".set \_int0x24,INT\_24 \n\t" //将int0x24的值设为中断起点

);

编写中断的注意事项：现在g++的函数框架较以前的有所不同。比如

以前：

call\_1(1)

编译成 push $1

call call\_1

add $4,esp

包括堆栈的操作

现在：

编译成

mov $1,(%esp)

call call\_1

不包括堆栈的操作，但是破坏了之前堆栈的内容。这对于优化显然是好的，但是对iret等严重依赖堆栈的操作，这就很致命。或许有某个gcc选项可以关掉这种优化，但是相对于全局来说这样是不必要的。

将中断写成函数的形式，用enter,leave包围起来也不行，因为你不知道函数预留了多少空间，没法正确引用局部变量。

正确方法是：

//============中断处理程序:0x24============

/\*\*

\*打印字符串

\* dl 指定属性

\* ecx:ebx 指定要打印字符串地址，遇到0结束

\*

\*/

extern "C" void int0x24()

{

int dsSel,off,attr;

\_\_asm\_\_ \_\_volatile\_\_(

""

:"=c"(dsSel),"=b"(off),"=d"(attr) //引用局部变量

:

:

);

Util::enter\_ds(dsSel);

Util::printStr(off,attr);

Util::leave\_ds(dsSel);

\_\_asm\_\_(

"leave \n\t" //中断返回

"iret \n\t"

);

}

//===================================================

使用extern “C” 编译的原因是使符号已知。leave/iret正常放置。

c语言(c++)属于编译期语言，一旦编译成代码，无需系统支持就能运行。

中断相关：理解门描述符中的DPL和其指向的代码段DPL关系，假设分别称为DPL和CDPL，则DPL确定一个来自CPL的请求能不能通过，而CDPL确定是否发生特权级转换。

而CDPL的Code段是否一致也有要求，如果是一致的，则特权级限制发生。否则不发生。Intel手册上说，只要handler放在0级非一致代码段，CPL将被忽略。现在测试发现确实如此。

为这，我探索了很久。（下面将RPL看成与CPL等价）

Intel手册上说，只要handler放在0级非一致代码段，CPL将被忽略。现在测试发现确实如此。下面测试的是一致代码段。

RPL=0 DPL=0 CDPL=0 一致 -- 能

RPL=0 DPL=3 CDPL=0 一致 -- 能，不切换

RPL=3 DPL=0 CDPL=0 一致 -- 不能

RPL=3 DPL=3 CDPL=0 一致 -- 能，不切换（intel手册上讲明：这种模式下只保证访问ss上的数据不发生错误。经测试正确，一旦发现mov ...,%es/%ds/%..的CPL和RPL不兼容，都会立即导致错误）

extern “C” 对于变量属于定义不属于声明。如果你使用了 extern “C” int a;实际上汇编为a保留了空间。extern才属于声明。

extern “C”对函数而言可以可以是定义，也可以是声明。 extern “C” void f();会生成符号\_f。如果一个函数声明为extern “C”，则即使定义中不加extern “C”,也会生成c符号。

对了，extern “C” { }可不必加分号。

STRING宏： #define \_\_STRING(x) #x

#define STRING(x) \_\_STRING(x)

.h文件中不要引入其他文件。关闭优化： -Wdisabled-optimization

push/pop不工作：如下定义4个宏：

#define ENTER\_DS(seg) ({\

if(seg!=SEG\_CURRENT){\

\_\_asm\_\_ \_\_volatile\_\_(\

"push %%ds \n\t"\

"movw %%ax,%%ds \n\t"\

:\

:"a"(seg)\

:\

);\

}})

#define LEAVE\_DS(seg) ({if(seg!=SEG\_CURRENT){\_\_asm\_\_("pop %ds \n\t");}})

#define ENTER\_ES(seg) ({\

\_\_asm\_\_("push %es \n\t");\

if(seg==SEG\_CURRENT){\

\_\_asm\_\_(\

"push %ds \n\t"\

"pop %es \n\t"\

);\

}else{\

\_\_asm\_\_ \_\_volatile\_\_(\

"movw %%ax,%%es \n\t"::"a"(seg):\

);\

}\

})

#define LEAVE\_ES(seg) ({\_\_asm\_\_("pop %es \n\t");})

每一个都构成一对push / pop的关系，但是已经说过g++对现在函数参数的设置方式有所改变，如果在中间代码中涉及函数调用，就可能改变当前的堆栈值。理想的方法是明确设置一个变量来接受ds/es的值，或者在clobber list中告知“memory”。前者是正确的，后者可能错误。下面是正确方式：

//===================设置ds,es的上下文环境====================

/\*\*

\*涉及堆栈平衡，不可能由非内联函数函数来实现。

\*/

#ifndef SEG\_CURRENT

#error("SEG\_CURRENT Not Defined")

#endif

#define ENTER\_DS(seg,saver) \

int saver;\

if(seg!=SEG\_CURRENT){\

\_\_asm\_\_ \_\_volatile\_\_(\

"mov %%ds,%%ebx\n\t"\

"movw %%ax,%%ds \n\t"\

:"=b"(saver)\

:"a"(seg)\

:\

);\

}

#define LEAVE\_DS(seg,saver) ({\

if(seg!=SEG\_CURRENT){\

\_\_asm\_\_ \_\_volatile\_\_(\

"mov %%ax,%%ds \n\t"\

:\

:"a"(saver)\

:);\

}})

#define ENTER\_ES(seg,saver) \

int saver;\

({\

\_\_asm\_\_ \_\_volatile\_\_(\

"mov %%es,%%ax \n\t"\

:"=a"(saver)\

:\

:);\

if(seg==SEG\_CURRENT){\

\_\_asm\_\_(\

"push %ds \n\t"\

"pop %es \n\t"\

);\

}else{\

\_\_asm\_\_ \_\_volatile\_\_(\

"movw %%ax,%%es \n\t"::"a"(seg):\

);\

}\

})

#define LEAVE\_ES(seg,saver) ({\_\_asm\_\_ \_\_volatile\_\_("mov %%ax,%%es \n\t"::"a"(saver):);})

使用ljmp切换任务时必须把偏移加上，虽然偏移没有实际用处。

中断的高阶正确写法，必须考虑以下事实：

1. 中断过程中可能发生别的中断
2. 中断使用寄存器传递参数

最近我遇到的一个错误十分巧合：在处理0x24号中断（字符串打印中断）没有关中断，因为中断的开关对于中断本身来说可能并不重要。我使用ecx作为ds的参数寄存器。而我设置的0x20号（定时中断）处理过程中需要发送0x20（EOI）。

在一定的时间之后，总是会发生#GP错误，调试了很长时间，最后发现（就是现在发现，2017年2月14日01:50:31），在原本ecx应当为常数0x10000的一过程中居然变成了0x20，于是0x20被作为段寄存器寻址，但是0x20是堆栈段，其限长只有2048 byte，不可能寻址高地址部分的数据，所以产生了GP错误。

我想原因应该是在int 0x24过程中发生了int 0x20中断，将ecx改变，返回的时候就变成了这个样子。但为什么是一段时间之后？因为int 0x24过程分为使用ecx之前和之后两个过程，只有发生在之前的改变才有效。

正确处理中断的方式：

1. 如果此中断不是硬件中断或者不由硬件中断调用（即纯软中断），则如果无论屏蔽中断与否，都可以不保存寄存器。这样做的原因是因为这种中断永远都不会影响任何执行上下文。
2. 属于硬件中断，则其自身必须先保存所有的通用寄存器。因为即使其他中断屏蔽了中断，即使其第一句就是cli，但是在cli的过程中还有可能发生中断。所以这种中断必须保存上下文。

（错误找到了，我他娘的很高兴啊）。

（2017年2月15日18:50:50）到目前为止，我总结的关于中断的编写黄金准则就是：不要在函数框架之内有任何内联栈操作。即pusha/popa等只能放在函数框架之外。看看下面的例子你就能明白。以后，**凡是在函数内部内联汇编中push/pop堆栈的都是错误！！！除了那种立即平衡了堆栈的pop/push对能够正确。**

\_\_asm\_\_(

".text \n\t" //如果是第一个函数，就要将其汇编进入.text中

".global \_int0x20 \n\t"

"\_int0x20:\n\t "

"pusha \n\t"

);

void \_int0x20() //保护现场只能发生在堆栈框架之前，所以编写中断有所不同

{

//CALL\_INT\_3(0x24,c,SEG\_CURRENT,b,"int 0x20.\n",d,Util::MODE\_COMMON);

Util::insertMark(0x55520);

IO\_8259A p1;

p1.sendOCW2(0,0x20);

if(Int\_0x20::current==0)

{

Int\_0x20::current=1;

\_\_asm\_\_("ljmp $0b1001000,$0 \n\t");

}else{

Int\_0x20::current=0;

\_\_asm\_\_("ljmp $0b101000,$0 \n\t");

}

\_\_asm\_\_(

"leave \n\t"

"popa \n\t"

"iret \n\t"

);

}

调试中断：断点放在起始位置有很大可能miss，放在第二个好一些。

新建任务tss结构的时候必须指定一下几个值：

SS,ESP -- 堆栈

SS0,ESP0 -- 调用系统中断

CS,EIP -- 代码段

EFLAGS -- 其中IF标志位很重要

同时，为了正确引用数据，还应设置ds。

测试如果内联函数调用其他的内联函数会如何。

int mark(int a)

{

}

\_\_attribute\_\_((always\_inline)) inline void L1()

{

mark(0x1);

}

\_\_attribute\_\_((always\_inline)) inline void L2()

{

L1();

mark(0x2);

}

void test()

{

int a=10;

printf("10 \n");

L2();

}

最后L1 L2都被内联，其中L1先被内联，L2再被内联，然后定义消失。

如果L1,L2相互调用，即假定不满足内联要求，则编译报错。

inline宏替换的关键之处：

如果一个局部变量多次用到，那么传给inline宏的将是它的值，一定不是它的地址。

如下：

int a;

Util::enterEs(0x10000,a);

Util::leaveEs(a);

Util::leaveEs(a);

Util::enterEs(0x10000,a);

每一次展开都会预留一个int空间，然后将a的值复制到此处，然后所有操作与此处相关。应当注意，如果函数宏需要使用局部变量，则会专门为其预留一个空间，而所有的操作都是基于此空间的。因此，函数宏无法改变局部变量的值。

最后不要妄图在函数宏中改变局部变量的值，我已经为此付出代价，时间的代价。

如果希望改变，就传地址吧。宏函数的本体函数函数。

当 leaveEs(a)变成了leaveEs(&a),就没有人认为你在改变局部变量了。记住，你每一次调用一个带参数的宏都是对栈的一种伤害，因为那些复制的空间用了就丢。

但是如果是引用，就会完全正确，它会引用原来的地址。

所以，如果以前说引用和指针本质相同，只是使用方式不同，那么那条规律在此不适用。这里，引用就是引用，其他类型就是其他类型。

ljmp必须使用已知的立即数而不能使用符号

为了使常数在编译期确定而不是在运行期寻址，但是如果将const static 常数放在.h文件中，就会产生定义，多份文件就会产生多份定义。 所以常规做法是吧定义放在.cpp文件中，但为了享受编译期常数带来的快捷，~~一般也在.h的class体内放相同定义。因为在class体内的常数只会被看成常数而不产生定义~~（错误，虽然不引用就不产生定义，但是再定义就有错误，编译期常数的情况除非在同一个.cpp文件中）。但如果放到.h文件中class的体外，则产生定义。

class声明体内绝对不要放任何定义。

另外ljmp指令必须使用已知的常数而不是未知符号。

类符号生成规则 ： class K{const static int A}-->\_\_ZN1$K1$AE

即

static常量： \_\_ZN{类名长度}{类名}{名称长度}{名称}E

析构函数： \_\_ZN{类名长度}{类名}D1Ev

构造函数： \_\_ZN{类名长度}{类名}C1Ev

如果有模板，就是类名就等于={原类名}I{类型}E

使用template的codegen与否：

template是用于语法检查的，并且用于在编译期生成代码。但是，如果在编译期对模板没有实现的话，就不会为其生成代码。

一种方法是将要用到的所有方法放在一个Capella类中，然后引用之。

但是第二种方法（是第一种想法的完全升级，但是语法不同）。模板语法实际上有两个：1.模板声明 2.模板实例化

模板实例化就是将类型具体化。如果template class 类名<实例模板>。注意，template是必不可少的，template表示需要到template定义的空间查找。

则解决方法就是：将涉及使用的类实例 实例化在.cpp的首部。

如果使用extern template ... ，则不尝试为其生成代码，而是看成外部实例。

环形依赖：

如果A.h需要引用B.h , 而B.h同时也引用A.h，则产生环形依赖问题。对于继承特别严重。因为继承必须实现知道被继承类的具体结构。解决方法：将它们放到一个文件中并合理安排位置。

使构造函数可以多次调用：

非模板化的构造函数名称

\_\_ZN{类名长度}{类名}C1E{参数}

（构造函数可能会生成C2的定义，但是所有的调用都指向C1）

i : int

j: unsigned int

h:unsigned char

v : void

Px: x\* x的指针

Rx:x& x的引用

S\_: 当前类（虽然你可以定义以\_结尾的类，但是通常不能使用）

{类名长度}{类名}：类

当然，为了可以使用类的构造函数（你当然只会在有限的地方这样做），可以在函数的上方定义标号来引用这个函数。使用call %esp之类的函数。

传参方式：ecx为this值

析构函数将C1换成D1，参数永远是v

replacement new

构造函数的地址

要想调用一个类的构造函数，可以从下面这个方面考虑：传递ecx给构造函数，然后让其在内部修改。一般情况下并不修改，只有当某个参数表明需要修改ecx参数时，才修改。

或者更加直接点：传递的第一个参数就当作this参数。

到头来，还是最初的解决方案最简单。

类的成员如果是引用类型，则只能在构造函数中初始化。