```
movl    $0x2a, %ebx
xorl    %eax, %eax
incb    %al
int     $0x80
```

DC0D32

```
movl    $STDOUT,%ebx
movl    %eax,%ecx
movl    fdlen,%edx
movl    $SYS_write,%eax
int     $0x80
```

Real mode in C with gcc : writing a bootloader

Usually the x86 boot loader is written in assembler. We will be exploring the possibility of writing one in C language (as much as possible) compiled with gcc, and runs in real mode. Note that you can also use the 16 bit bcc or TurboC compiler, but we will be focusing on gcc in this post. Most open source kernels are compiled with gcc, and it makes sense to write C bootloader with gcc instead of bcc as you get a much cleaner toolchain :)

As of today (20100614), gcc 4.4.4 officially only emits code for protected/long mode and does not support the real mode natively (this may change in future).

Also note that we will not discuss the very fundamentals of booting. This article is fairly advanced and assumes that you know what it takes to write a simple boot-loader in assembler. It is also expected that you know how to write gcc inline assembly. Not everything can be done in C!

getting the tool-chain working

.code16gcc

As we will be running in 16 bit real mode, this tells gas that the assembler was generated by gcc and is intended to be run in real mode. With this directive, gas automatically adds *addr32* prefix wherever required. For each C file which contains code to be run in real mode, this directive should be present at the top of effectively generated assembler code. This can be ensured by defining in a header and including it before any other.

```
1 #ifndef _CODE16GCC_H_
2 #define _CODE16GCC_H_
3 __asm__(".code16gcc\n");
4 #endif
```

This is great for bootloaders as well as parts of kernel that must run in real mode but are desired written in C instead of asm. In my opinion C code is a lot easier to debug and maintain than asm code, at expense of code size and performance at times.

Special linking

As bootloader is supposed to run at physical 0x7C00, we need to tell that to linker. The mbr/vbr should end with the proper boot signature 0xaa55.
All this can be taken care of by a simple linker script.

```
ENTRY(main);
SECTIONS
{
    . = 0x7C00;
    .text : AT(0x7C00)
    {
        _text = .;
        *(.text);
        _text_end = .;
    }
    .data :
    {
        _data = .;
        *(.bss);
        *(.bss*);
        *(.data);
        *(.rodata*);
        *(COMMON)
        _data_end = .;
    }
    .sig : AT(0x7DFE)
    {
        SHORT(0xaa55);
    }
    /DISCARD/ :
    {
```

```
        *(.note*);
        *(.iplt*);
        *(.igot*);
        *(.rel*);
        *(.comment);
/* add any unwanted sections spewed out by your version of g
    }
}
```

gcc emits elf binaries with sections, whereas a bootloader is a monolithic plain binary with no sections. Conversion from elf to binary can be done as follows:

```
$ objcopy -O binary vbr.elf vbr.bin
```

The code

With the toolchain set up, we can start writing our hello world bootloader!
vbr.c (the only source file) looks something like this:

```
/*
 * A simple bootloader skeleton for x86, using gcc.
 *
 * Prashant Borole (boroleprashant at Google mail)
 * */

/* XXX these must be at top */
#include "code16gcc.h"
__asm__ ("jmpl  $0, $main\n");


#define __NOINLINE  __attribute__((noinline))
#define __REGPARM   __attribute__ ((regparm(3)))
#define __NORETURN  __attribute__((noreturn))

/* BIOS interrupts must be done with inline assembly */
void    __NOINLINE __REGPARM print(const char   *s){
        while(*s){
                __asm__ __volatile__ ("int  $0x10" : : "a"(0
                s++;
        }
}
/* and for everything else you can use C! Be it traversing tl

void __NORETURN main(){
    print("woo hoo!\r\n:)");
    while(1);
}
```

compile it as

```
$ gcc -c -g -Os -march=i686 -ffreestanding -Wall -Werror -I.
$ ld -static -Tlinker.ld -nostdlib --nmagic -o vbr.elf vbr.o
$ objcopy -O binary vbr.elf vbr.bin
```

and that should have created vbr.elf file (which you can use as a symbols file with gdb for source level debugging the vbr with gdbstub and qemu/bochs) as well as 512 byte vbr.bin. To test it, first create a dummy 1.44M floppy image, and overwrite it's mbr by vbr.bin with dd.

```
$ dd if=/dev/zero of=floppy.img bs=1024 count=1440
$ dd if=vbr.bin of=floppy.img bs=1 count=512 conv=notrunc
```

and now we are ready to test it out :D

```
$ qemu -fda floppy.img -boot a
```

and you should see the message!

Once you get to this stage, you are pretty much set with respect to the tooling itself. Now you can go ahead and write code to read the filesystem, search for next stage or kernel and pass control to it.

Here is a simple example of a floppy boot record with no filesystem, and the next stage or kernel written to the floppy immediately after the boot record. The next image LMA and entry are fixed in a bunch of macros. It simply reads the image starting one sector after

boot record and passes control to it. There are many obvious holes, which I left open for sake of brevity.

```
/*                                                                  ?
 * A simple bootloader skeleton for x86, using gcc.
 *
 * Prashant Borole (boroleprashant at Google mail)
 * */

/* XXX these must be at top */
#include "code16gcc.h"
__asm__ ("jmpl  $0, $main\n");


#define __NOINLINE  __attribute__((noinline))
#define __REGPARM   __attribute__ ((regparm(3)))
#define __PACKED    __attribute__((packed))
#define __NORETURN  __attribute__((noreturn))

#define IMAGE_SIZE  8192
#define BLOCK_SIZE  512
#define IMAGE_LMA   0x8000
#define IMAGE_ENTRY 0x800c

/* BIOS interrupts must be done with inline assembly */
void    __NOINLINE __REGPARM print(const char   *s){
        while(*s){
                __asm__ __volatile__ ("int  $0x10" : : "a"((
                s++;
        }
}

#if 0
/* use this for the HD/USB/Optical boot sector */
typedef struct __PACKED TAGaddress_packet_t{
    char                size;
    char                :8;
    unsigned short      blocks;
    unsigned short      buffer_offset;
    unsigned short      buffer_segment;
    unsigned long long  lba;
    unsigned long long  flat_buffer;
}address_packet_t ;

int __REGPARM lba_read(const void   *buffer, unsigned int
        int i;
        unsigned short  failed = 0;
        address_packet_t    packet = {.size = sizeof(addres
        for(i = 0; i < 3; i++){
                packet.blocks = blocks;
                __asm__ __volatile__ (
                                "movw   $0, %0\n"
                                "int    $0x13\n"
                                "setcb  %0\n"
                                :"=m"(failed) : "a"(0x4200)
                /* do something with the error_code */
                if(!failed)
                        break;
        }
        return failed;
}
#else
/* use for floppy, or as a fallback */
typedef struct {
        unsigned char   spt;
        unsigned char   numh;
}drive_params_t;

int __REGPARM __NOINLINE get_drive_params(drive_params_t
        unsigned short  failed = 0;
        unsigned short  tmp1, tmp2;
        __asm__ __volatile__
            (
             "movw  $0, %0\n"
             "int   $0x13\n"
             "setcb %0\n"
             : "=m"(failed), "=c"(tmp1), "=d"(tmp2)
             : "a"(0x0800), "d"(bios_drive), "D"(0)
             : "cc", "bx"
            );
        if(failed)
                return failed;
        p->spt = tmp1 & 0x3F;
        p->numh = tmp2 >> 8;
        return failed;
}

```

```
 85  int __REGPARM __NOINLINE lba_read(const void    *buffer, uns
 86          unsigned char   c, h, s;
 87          c = lba / (p->numh * p->spt);
 88          unsigned short  t = lba % (p->numh * p->spt);
 89          h = t / p->spt;
 90          s = (t % p->spt) + 1;
 91          unsigned char   failed = 0;
 92          unsigned char   num_blocks_transferred = 0;
 93          __asm__ __volatile__
 94              (
 95               "movw  $0, %0\n"
 96               "int   $0x13\n"
 97               "setcb %0"
 98               : "=m"(failed), "=a"(num_blocks_transferred)
 99               : "a"(0x0200 | blocks), "c"((s << 8) | s), "d"
100              );
101          return failed || (num_blocks_transferred != blocks)
102  }
103  #endif
104
105  /* and for everything else you can use C! Be it traversing
106
107  void __NORETURN main(){
108          unsigned char   bios_drive = 0;
109          __asm__ __volatile__("movb  %%dl, %0" : "=r"(bios_d
110
111          drive_params_t  p = {};
112          get_drive_params(&p, bios_drive);
113
114          void    *buff = (void*)IMAGE_LMA;
115          unsigned short  num_blocks = ((IMAGE_SIZE / BLOCK_S
116          if(lba_read(buff, 1, num_blocks, bios_drive, &p) !=
117              print("read error :(\r\n");
118              while(1);
119          }
120          print("Running next image...\r\n");
121          void*   e = (void*)IMAGE_ENTRY;
122          __asm__ __volatile__("" : : "d"(bios_drive));
123          goto    *e;
124  }
```

removing __NOINLINE may result in even smaller code in this case. I had it in place so that I could figure out what was happening.

Concluding remarks

C in no way matches the code size and performance of hand tuned size/speed optimized assembler. Also, because of an extra byte (0x66, 0x67) wasted (in addr32) with almost every instruction, it is highly unlikely that you can cram up the same amount of functionality as assembler.

Global and static variables, initialized as well as uninitialized, can quickly fill those precious 446 bytes. Changing them to local and passing around instead may increase or decrease size; there is no thumb rule and it has to be worked out on per case basis. Same goes for function in-lining.

You also need to be extremely careful with various gcc optimization flags. For example, if you have a loop in your code whose number of iterations are small and deducible at compile time, and the loop body is relatively small (even 20 bytes), with default -Os, gcc will unroll that loop. If the loop is not unrolled (-fno-tree-loop-optimize), you might be able to shave off big chunk of bytes there. Same holds true for frame setups on i386 - you may want to get rid of them whenever not required using -fomit-frame-pointer. Moral of the story : you need to be extra careful with gcc flags as well as version update. This is not much of an issue for other real mode modules of the kernel where size is not of this prime importance.

Also, you must be very cautious with assembler warnings when compiling with .code16gcc. Truncation is common. It is a very good idea to use --save-temp and analyze the assembler code generated from your C and inline assembly. Always take care not to mess with the C calling convention in inline assembly and meticulously check and update the clobber list for inline assembly doing BIOS or APM calls (but you already knew it, right?).

It is likely that you want to switch to protected/long mode as early as possible, though. Even then, I still think that maintainability wins over asm's size/speed in case of a bootloader as well as the real mode portions of the kernel.

It would be interesting if someone could try this with c++/java/fortran. Please let me know if

you do!

at 6/15/2010 02:05:00 AM

20 comments:

Megha Tuesday, June 15, 2010 at 6:12:00 PM GMT+5:30

Dokyaawarun 10 foot.. kiwwa jaastach.
:-|

Reply

descent Tuesday, December 21, 2010 at 1:10:00 PM GMT+5:30

Hi,
Thank you for your sharing.

in void __NOINLINE __REGPARM print(const char *s)
I change the print function to access pointer,
like this:
videoram[0]='H';

but I got the warning message:
/tmp/cc5qsy9l.s:33: Warning: 00000000000b8000 shortened to 0000000000008000

Do I miss something?

Reply

descent Tuesday, December 21, 2010 at 2:05:00 PM GMT+5:30

Hi,

I use gcc-3.4 to compile again.
I see no warning message, but in qemu,
I still cannot see char H.
videoram is static variable.

static unsigned char *videoram = (unsigned char *) 0xb8000;

Reply

descent Tuesday, December 21, 2010 at 3:16:00 PM GMT+5:30

Hi,
I got something. In 16bit mode, the pointer is 16bit length. So 0xb8000 shortened to 0x8000.
I write a c file and a function,

```
void put_char()
{
unsigned char *videoram = (unsigned char *) 0xb8000;
videoram[0]='H';
videoram[2]='H';
videoram[40]='H';

}
```

no include code16gcc.h, I think the pointer is 32bits length, but I still can not see the H character.

Reply

Prashant Tuesday, December 21, 2010 at 7:16:00 PM GMT+5:30

@descent: check the '--save-temps' preserved assembler version of the C function.

This article assumes that the reader has low level programming experience with x86.

To access the vidmem with b8000h, you have 2 options:
1. write inline assembly to set es to b800h, and di to the address in the real mode segment. Then write byte/word to es:di.
2. Enter unreal mode. Then you can use the full 4G memory, one-to-one mapped.

I personally would not recommend any of these methods for printing - BIOS int 10h is pretty good. Remember - do not try and do anything fancy in the (m/v)br; do it in the next stage instead as you have pretty much unconstrained image size in later stages.

Reply

descent Wednesday, December 22, 2010 at 9:41:00 AM GMT+5:30

Hi Prashant,

Thank you for your explanation.

Because in protected mode, I can use C,
and direct access 0xb8000, so I am confused.
real/protect mode, gcc/gas 16/32 bit also confuse me.
They are very complicate.

Reply

TheComputaNerd Saturday, March 12, 2011 at 6:26:00 PM GMT+5:30

you are a genius!

Reply

abraker95 Sunday, April 17, 2011 at 5:48:00 AM GMT+5:30

I've got that infamous runtime error...

bootloader.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

Reply

boskov1985 Saturday, May 21, 2011 at 2:39:00 AM GMT+5:30

Managed to do it in C++.
Code is the same.

Linker file needs to discard eh_frame.

When building on x86-64 add -m32 to g++ and -melf_i386 on ld command line.

Trying to rewrite it in a more c++-ish style.

My e-mail is boskovits@cogito-top.hu .

Reply

Prashant Saturday, May 21, 2011 at 3:02:00 AM GMT+5:30

@abraker95: are you trying to run the MZ/PE image in windows? that is like sinning and then spitting on the devil when in hell.

@boskov1985: cool man! let us know how it goes :D

Reply

rpfh Friday, November 25, 2011 at 2:50:00 AM GMT+5:30

It's easier to to this without objcopy. Modern ld versions support --oformat=binary , so just one line does the direct compilation job.

gcc -g -Os -march=i686 -ffreestanding -Wall -Werror -I. -static -nostdlib -Wl,-Tlinker.ld -Wl,--nmagic -Wl,--oformat=binary -o loader.bin loader.c

Reply

Prashant Friday, November 25, 2011 at 8:01:00 AM GMT+5:30

I can't verify right now whether it works, but thanks for letting us know, rpfh!

Reply

descent Sunday, December 4, 2011 at 9:42:00 PM GMT+5:30

Hi,

The c code uses function call, why need not set stack (ss:esp)?

Reply

Prashant Tuesday, December 6, 2011 at 10:18:00 AM GMT+5:30

good point @decent. I guess you will need to set up the stack first in main, probably in assembler.

Reply

descent Saturday, December 24, 2011 at 8:02:00 PM GMT+5:30

I change %ss:%esp to 0x07a0:0000,
Is any side effect?

```
void __NORETURN main(){
__asm__ ("mov %cs, %ax\n");
__asm__ ("mov %ax, %ds\n");
__asm__ ("mov $0x07a0, %ax\n");
__asm__ ("mov %ax, %ss\n");
__asm__ ("mov $0, %esp\n");
print("woo hoo!\r\n:)");
while(1);
}
```

Reply

descent Monday, July 30, 2012 at 8:16:00 AM GMT+5:30

Hi,
I test c bootloader in real machine, in my eeepc 904, need add some code to setup stack.

http://descent-incoming.blogspot.tw/2012/05/x86-bootloader-hello-world.html
The article is written by Chinese, but the code, picture can give some reference.

cppb.cpp is cpp version (compile by g++), it can work, I test it in real machine(eeepc 904).

Reply

Chris Evans Saturday, April 20, 2013 at 10:46:00 AM GMT+5:30

linker fails whats up with it..?

Reply

Prashant Sunday, April 21, 2013 at 9:34:00 AM GMT+5:30

Fails how? Can you please elaborate?

Reply

abhinandan s Wednesday, November 13, 2013 at 12:51:00 PM GMT+5:30

Thank you for detaile explanation
Linker failed nt sure why..ld: error: load segment overlap [0x7c00 -> 0x7e50] and [0x7dfe -> 0x7e00]

Reply

Jing Peng Thursday, February 5, 2015 at 8:39:00 PM GMT+5:30

Thank you for your nice post! I'm trying to run it on my x86-64 linux box, but gcc reports errors like "bad register name rax", I'm a little confused by the various compiler options here, could you please give me suggestions on how to compile the C source file on x86-64 machines? Thanks

Reply

Enter your comment...

Comment as: Unknown (Google **Sign out**

Publish **Preview** Notify me

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Awesome Inc. template. Powered by Blogger.