kmain() is the main routine of the kernel

ram detection - hardware/software – boot sequence

boot sequence

* Identify bootable medium
* Master boot record
  + Magic bytes
    - 511 and 512, 0x55 and 0xAA respectively
    - These bytes make it bootable
    - This signature is represented (in binary) as 0b1010101001010101. The alternating bit pattern was thought to be a protection against certain failures (drive or controller). If this pattern is garbled or 0x00, the device is not considered bootable

BIOS physically searches for a boot device by loading the first 512 bytes from the bootsector of each device into physical memory, starting at the address 0x7C00 (1 KiB below the 32 KiB mark). When the valid signature bytes are detected, BIOS transfers control to the 0x7C00 memory address (via a jump instruction) in order to execute the bootsector code.

CPU is in 16bit mode, needs to switch to protected for 32bit

Gnu grub first thing booted

Simple load, multiboot, external memory modules

Grub uses multiboot, executable binary is 32 bits and has the multiboot header in first 8192 bytes

Kernel is usually ELF, executable and linkable format, used in most UNIX

First boot sequence is in assembly (start.asm) and we use linker to define what is executable (linker.ld)

Also defines c++ runtime

Multiboot Header:

struct multiboot\_info {

u32 flags;

u32 low\_mem;

u32 high\_mem;

u32 boot\_device;

u32 cmdline;

u32 mods\_count;

u32 mods\_addr;

struct {

u32 num;

u32 size;

u32 addr;

u32 shndx;

} elf\_sec;

unsigned long mmap\_length;

unsigned long mmap\_addr;

unsigned long drives\_length;

unsigned long drives\_addr;

unsigned long config\_table;

unsigned long boot\_loader\_name;

unsigned long apm\_table;

unsigned long vbe\_control\_info;

unsigned long vbe\_mode\_info;

unsigned long vbe\_mode;

unsigned long vbe\_interface\_seg;

unsigned long vbe\_interface\_off;

unsigned long vbe\_interface\_len;

};

**To validate header, mbchk kernel.elf**

Disk image for kernel grub

Create hard disk image (c.img) using qemu-img

qemu-img create c.img 2M

Then partition using fdisk (format disk)

fdisk ./c.img

# Switch to Expert commands

> x

# Change number of cylinders (1-1048576)

> c

> 4

# Change number of heads (1-256, default 16):

> h

> 16

# Change number of sectors/track (1-63, default 63)

> s

> 63

# Return to main menu

> r

# Add a new partition

> n

# Choose primary partition

> p

# Choose partition number

> 1

# Choose first cylinder (1-4, default 1)

> 1

# Choose last cylinder, +cylinders or +size{K,M,G} (1-4, default 4)

> 4

# Toggle bootable flag

> a

# Choose first partition for bootable flag

> 1

# Write table to disk and exit

> w

Now we have partition – attach to the loop device to allow it to be accessed using losetup, then the offset is passed and calced using offset=startsector\*bytes\_by\_sector

Using fdisk -l -u c.img, you get: 63 \* 512 = 32256.

losetup -o 32256 /dev/loop1 ./c.img

We create a EXT2 filesystem on this new device using:

mke2fs /dev/loop1

We copy our files on a mounted disk:

mount /dev/loop1 /mnt/

cp -R bootdisk/\* /mnt/

umount /mnt/

Install GRUB on the disk:

grub --device-map=/dev/null << EOF

device (hd0) ./c.img

geometry (hd0) 4 16 63

root (hd0,0)

setup (hd0)

quit

EOF

Now we’re done using loop device so lets detach it

losetup -d /dev/loop1

The compiler will assume that all the necessary C++ runtime support is available by default, but as we are not linking in libsupc++ into your C++ kernel, we need to add some basic functions that can be found in the cxx.cc file.

Caution: The operators new and delete cannot be used before virtual memory and pagination have been initialized.

**Basic C/C++ functions**

The kernel code can't use functions from the standard libraries so we need to add some basic functions for managing memory and strings:

void itoa(char \*buf, unsigned long int n, int base);

void \* memset(char \*dst,char src, int n);

void \* memcpy(char \*dst, char \*src, int n);

int strlen(char \*s);

int strcmp(const char \*dst, char \*src);

int strcpy(char \*dst,const char \*src);

void strcat(void \*dest,const void \*src);

char \* strncpy(char \*destString, const char \*sourceString,int maxLength);

int strncmp( const char\* s1, const char\* s2, int c );

These functions are defined in [string.cc](https://github.com/SamyPesse/How-to-Make-a-Computer-Operating-System/blob/master/src/kernel/runtime/string.cc), [memory.cc](https://github.com/SamyPesse/How-to-Make-a-Computer-Operating-System/blob/master/src/kernel/runtime/memory.cc), [itoa.cc](https://github.com/SamyPesse/How-to-Make-a-Computer-Operating-System/blob/master/src/kernel/runtime/itoa.cc)

**C types**

During the next step, we are going to use different types in our code, most of the types we are going to use unsigned types (all the bits are used to stored the integer, in signed types one bit is used to signal the sign):

typedef unsigned char u8;

typedef unsigned short u16;

typedef unsigned int u32;

typedef unsigned long long u64;

typedef signed char s8;

typedef signed short s16;

typedef signed int s32;

typedef signed long long s64;

**Compile our kernel**

Compiling a kernel is not the same thing as compiling a linux executable, we can't use a standard library and should have no dependencies to the system.

Our [Makefile](https://github.com/SamyPesse/How-to-Make-a-Computer-Operating-System/blob/master/src/kernel/Makefile) will define the process to compile and link our kernel.

For x86 architecture, the followings arguments will be used for gcc/g++/ld:

# Linker

LD=ld

LDFLAG= -melf\_i386 -static -L ./ -T ./arch/$(ARCH)/linker.ld

# C++ compiler

SC=g++

FLAG= $(INCDIR) -g -O2 -w -trigraphs -fno-builtin -fno-exceptions -fno-stack-protector -O0 -m32 -fno-rtti -nostdlib -nodefaultlibs

# Assembly compiler

ASM=nasm

ASMFLAG=-f elf -o