In your main board, you have your BIOS. Attached to MB, you have harddrive, CPU, and RAM.

When you start your computer:

* MB takes the data from the BIOS (sth like assembly code) and copy content of BIOS to RAM. Now you have firmware at your RAM.
* CPU puts its instruction pointer into position of firmware in RAM.
* Firmware tells to CPU: “HELLO CPU! Please talk to the harddrive and load boot loader to RAM.“
* Then CPU IP points to boot loader.
* Boot loader knows about partition tables, file systems… So it will be able to go to second partition at disk and look at the /boot/grub/grub.cfg then load this file.
* Boot loader prints list of OSs on your screen. With arrow keys, you can select one of them.
* Grub will say this OS is, lets say, on the third partition of the disk.
* Now I need to load /boot/kernel.bin file to RAM, for example.
* Then CPU IP points to kernel bin.
* This is the point our (C++) code will start to execute.
* Boot loader doesn’t set stack pointer register and this is a problem because C++ programs expect stack pointer to be set before it is running.
* We have to write 2 different files:
  + loader.s ---> sets the stack pointer
    - compiled using GNU assembler then we get file loader.o
  + kernel.cpp
    - compiled using g++ then we get file kernel.o
* Now we have 2 different object files from 2 different programming languages they need to be combined to 1 common file
* To do this we use program called ld which is a linker. We get combined file kernel.bin
* This is the file we put into “boot” directory and we will have to write an entry into grub.cfg so that boot loader can find it.
* When you start your computer, CPU will start out in 32-bit compatibility mode. So in start of the kernel, we are in 32-bit mode. We don’t know how to switch to 64-bit mode. We make all loader.s and kernel.cpp 32 bits to be compatible.

We make 4 files:

* loader.s
* kernel.cpp
* linker.ld
* makefile

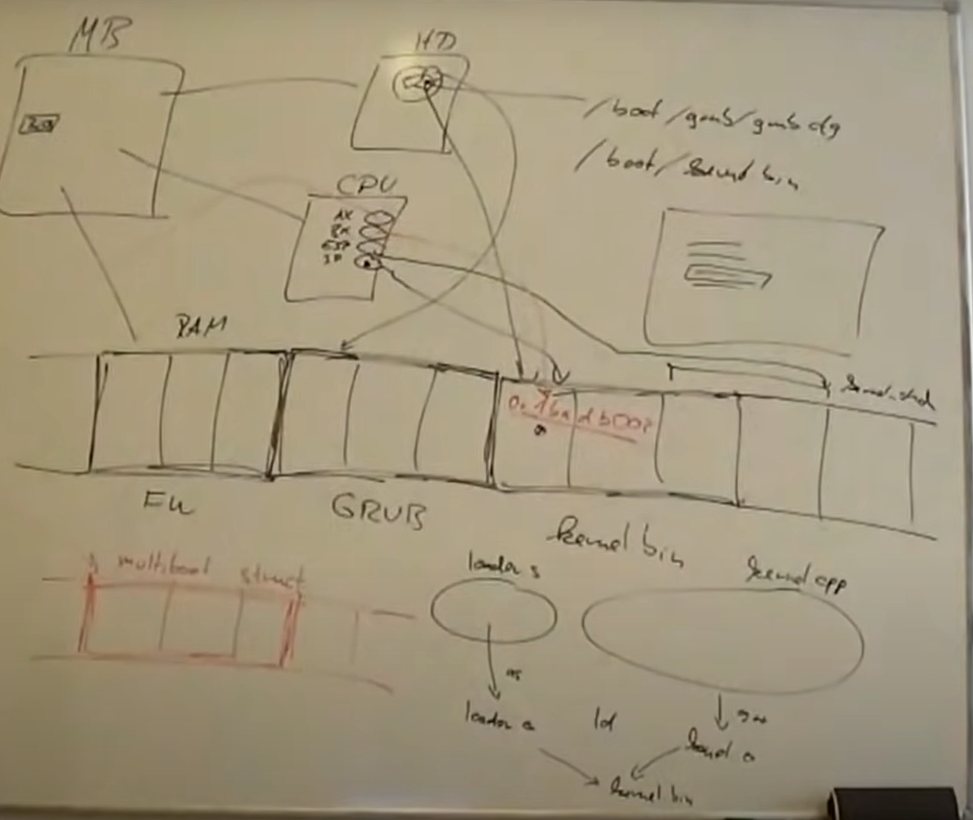
Bootloader will not recognize kernel bin as a kernel bc when boot loader looks at kernel.bin it will look for metric number and if it is not there, boot loader won’t believe this is the kernel.We need to put this metric number in there.

Metric number: 0x1BADB002

When bootloader decides “OK this is a kernel and I will load it and jumps into it”, before it does that, it stores some informations on somewhere in the RAM.

At some point, there is multiboot structure containing some information such as **size of your RAM**.

Bootloader creates multiboot structure and puts pointer to that in AX register. Also it copies metric number into BX register.



BIOS

BOOT LOADER – knows about partition tables etc., – look GRUB.CFG and loads it – creates multiboot structure

KERNEL.BIN (kernel.cpp+loader.s)

printf

Regular printf that you can include with stdio at kernel.cpp is outside of the OS. It is inside dynamic library.

Linux will see OK, your program is using stuff from a library.

Linux will use dynamic linking to load glibc and connect your program to glibc.

Problems:

* We don’t have glibc for our OS yet.
* We also don’t have OS around us, like Linux, which connects library for us.

So we have no dynamic linking and we have no libraries.

We don’t have dynamic memory management so we cannot use malloc, free, etc.

We have nothing to write sth on screen.

We have nothing.

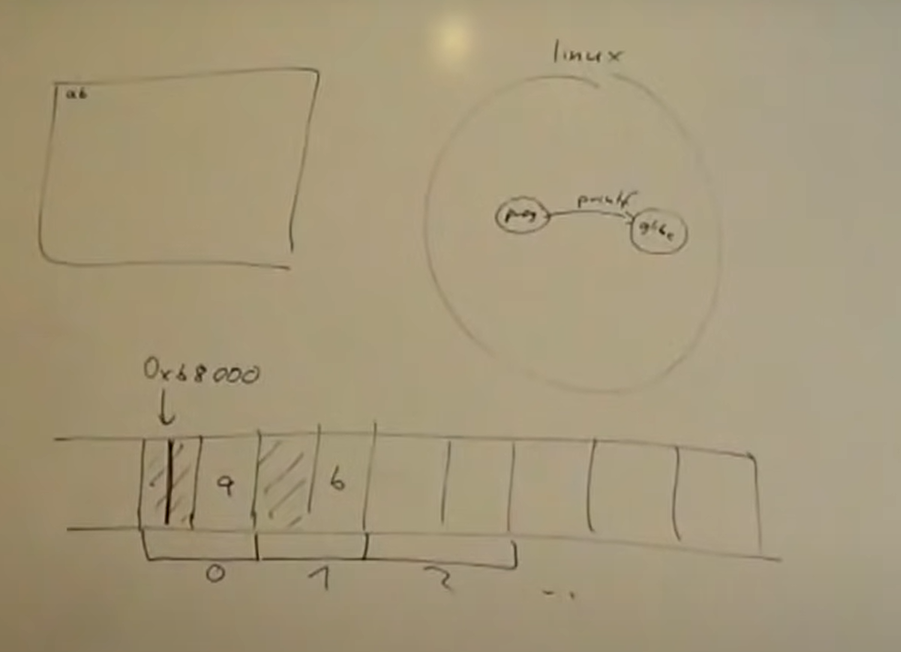
Everything we want to do, we have to write ourselves.

We write our simple printf code.

There is a specific memory location: 0xb8000

Whatever you write to that location, will be put on the screen by graphics card.

What are these striped bytes?

* They are for color information.
* We will not use them.
* When your program starts, these bytes already have same values in them so you don’t have to set this colors explicitly for white text on black background. We just have to make sure that we don’t overwrite that.

screen

dynamic memory

We are not inside Linux environment where there is a glibc library.

We also don’t have other things like exception handling, dyamic memory that Linux deals for us.

Compiler assumes there is an OS, I can do them.

We need to tell to compiler, don’t assume that there is a glibc, OS that deals with exception and gives you answer to interrupt for dynamic memory management.

We do this by GPPPARAMS fno at makefile.

* -fno-use-cxa-atexit ---> deal with memory management
* -nostdlib ---> we don’t have glibc
* -fno-builtin
* -fno-rtti ---> run time type identification
* -fno-exceptions
* -fno-leading-underscore ---> naming of the sections in the resulting .o file. If we don’t have that, loader will not be able to call kernelMain, it would have to be \_kernelMain. It is a minor thing.

linker.ld

If we had different files from same programming languages we could tell to compiler do linking for us.

When we have files from different programming languages, we have to take care of it ourselves.

* make loader.o
* make kernel.o
* make mykernel.bin

Now you have your mykernel.bin

* make install
* ls /boot/
  + You can see mykernel.bin
  + Now you have to enter it into grub.cfg
* sudo vim /boot/grub/grub.cfg

Text

Description automatically generated

At very end of the grub.cfg write this into white part above:

* ### BEGIN MYKERNEL ###
* menuentry ‘My Operating System’ {
* multiboot /boot/mykernel.bin
* boot
* }
* ### END MYKERNEL ###