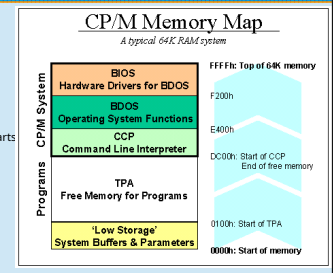
INF-2203
02- Starting OS and basic I/O

John Markus Bjørndalen
2026

Historical note: CP/M and BIOS

- Basic Input/Output System (BIOS)
  - The portability layer of CP/M
  - Porting to new computers: in principle, only the BIOS would need to be specific for the computer.
- Complexity
  - The BIOS must be mapped into addr 0 from the start (CPU starts executing there)
  - The operating system stores information from addr 0
  - => need to move the BIOS before executing the operating system
    - One option: let the BIOS
      - boot computer
      - load CP/M.
      - Then the relevant part of the BIOS can be copied to higher addresses before switching out the BIOS ROM.



https://obsolescence.wixsite.com/obsolescence/cpm-internals

Booting on an old PC

- Computer turns on in **16-bit mode** and starts running BIOS
- BIOS loads a "Boot Loader" from a designated storage device (typically your first hard disk)
  - The boot loader is a tiny piece of code that has one task: find and load the operating system kernel
  - After the kernel is loaded, the boot loader jumps to a predefined start location in the kernel and the operating system is running
  - The operating system can choose to use the BIOS to handle I/O, but can also choose to interface directly with hardware
- This is similar to CP/M, but adds the boot loader
- To get to 64-bit mode, the operating system has to (see next foil)

Booting a newer PC through legacy mode

Here be dragons:

- Load and start 16-bit operating system using BIOS
  - Real mode
  - Limited address space
    - Segmented memory: 16 bit addr => 64KB segments. Combine with segment register to create 20-bit addrs (1MB).
- Kernel switches to 32-bit mode. Rough description:
  - Protected mode (introduced with 80286)
  - Historical baggage: need to fiddle with keyboard driver to enable pin 20 on the address bus
  - Set up memory mapping (more about this later in the course): global descriptor table (GDT).
  - Enable 32-bit by setting Protection Enable bit in control register 0 (CR0).
  - Execute "long jump" (ljmp)
- Kernel can now switch to 64-bit mode:
  - Long mode
  - https://wiki.osdev.org/Setting_Up_Long_Mode
- Problems:
  - Complicated due to old baggage (backwards compatibility) – it's actually much more convoluted than the description above
  - Modern computers are dropping support for this (can no longer boot like this on many computers)
    - Booting directly using UEFI instead of BIOS
    - Future: dropping support for 16 and 32-bit mode:
      - https://www.intel.com/content/www/us/en/developer/articles/technical/envisioning-future-simplified-architecture.html
    - Or maybe not.. see updated note above on X86S

Booting a modern PC - implications

Implications for our OS kernel

- The old inf-2201 OS no longer boots on modern hardware
- UEFI takes care of initialisation: our kernel can start directly from 64-bit mode
  - Security mechanisms may cause issues
- Newer hardware is more complicated and may require more drivers (ACPI, ...)
- May need to consider switch to simpler architectures

Booting a modern PC – the modern way

Slightly simplified

- UEFI Firmware (ROM/something) initially starts booting a computer
- The firmware looks for an EFI boot/system partition and loads a boot loader
  - The bootloader initializes computer and prepares to load an operating system
  - Bootloaders can provide menus or escape into modes for configuring the computer
- In this course, we'll use the GRUB bootloader
  - Going into details will take too much time of the course
  - Provides a boot menu
  - Locates and loads the operating system kernel and
  - See multiboot protocol for how Grub
    - Inspects the kernel binary for requests
    - Prepares the computer to correspond to requests (ex: select graphics mode)
    - Provides information to the kernel before handing over to the kernel
- If this gets complicated, don't worry. We will ingore most of it when developing the OS!

Kernel as a "program launcher"

Topics

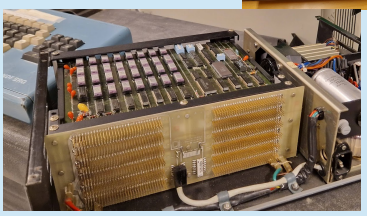
- Basic I/O
  - Memory mapped – start with how memory reads/writes are I/O and how it's done
  - Separate I/O instructions (similar, but tagged I/O and separate addr space effectively)
- Multiboot protocol
  - Why do we need something like this?
  - Specifying what the kernel wants
  - Scanning the info the kernel gets
- Starting and runnning a program from the kernel
  - Need to find the files
    - how to read cpio "fs"
    - VFS

I/O – let's start with old computers

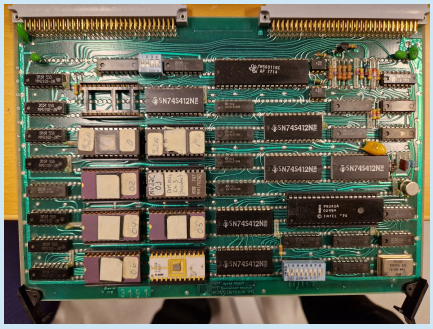
Example computer: Mycro-1

- Norwegian computer from 1974
  - Similar age to the Altair
- The first were Intel 8080 based (DIM-1001)
  - Said to be the first single-board Microcomputer (it had cpu, prom, memory and some I/O on a single card)
- Later an upgraded Z80 CPU card came (DIM-1003) that we will look at
-
- Some info: https://en.wikipedia.org/wiki/MYCRO-1

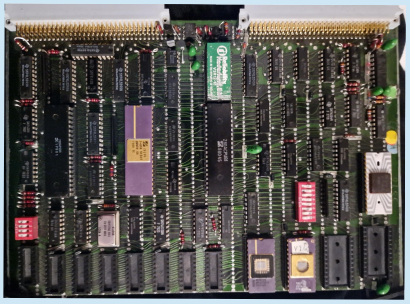
Opening the computer



DIM-1001 – CPU board (Intel 8080)

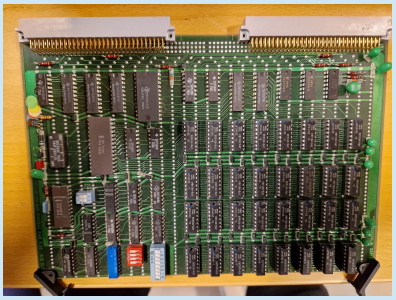


DIM-1003 CPU Card (Z80)



DIM-1016 64 KB of dynamic RAM

- 16-bit addressing
  - => 64KB max



Accessing memory

- Simplified "Z80-inspired" memory read (real version is slightly more complicated)
  - CPU sets address pins on CPU and asserts /MREQ (memory request pin set low) and /RD (read pin)
  - Any memory device decodes address pins to determine if it is responsible
  - Responsible chip(s) looks up memory cells an sets content on shared data pins
  - Responsible chip(s) asserts "data ready" and CPU can copy in content from data pins
- Writes similar, just flip the /RD bit and let the CPU set the data pins while respinsible chips reads the data pins

I/O - v1 – memory mapped I/O

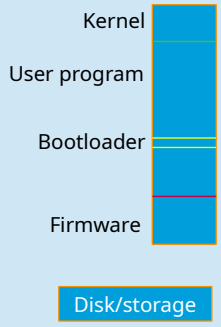
- Observation: RAM or ROM are just chips that talk to CPU over a bus using signaling.
- Same type of reads/writes can be used by other devices, but interpreted in a different way
  - Serial port device could interpret writes to addr X and Y as commands to the serial port device
    - Writes to addr X could be writes to control register
    - Writes to addr Y could be data bytes you want to send over the serial port
  - The CPU doesn't need to know what the I/O device is, only that it can read and write bytes to and from the given addresses
- The 6502 processor use memory mapped I/O
  - Simple programming, sacrifices a little bit of memory space

I/O - v2 – I/O ports

- Intel 8080 and Z80 used the same address + data pins for I/O but added another mode of communication: I/O ports.
  - Set /MREQ high in a request, singalling that the CPU is not doing a memory request
  - Memory chips ignore the operation
  - I/O port devices (like the Z80 serial port) then responds to the read/write operations
  - You could almost think of it like an extra address pin adding an I/O address space
- Modern Intel processors (x86/x64) still have the I/O ports. We will use them in the course

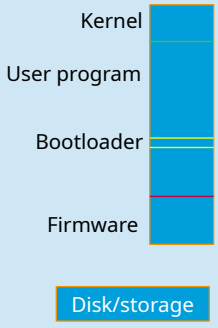
Multiboot protocol

- Firmware finds a bootloader on some storage device
  - Loads it into memory and starts it
- Bootloader then starts looking for an operating system to load
  - Loads operating system into memory and starts it
- What if:
  - The kernel needs a few more files (device drivers etc) before it can fully use the computer?
  - The kernel wants the computer set up in a way it understands? (known minimal state with graphics mode, text mode, ...)



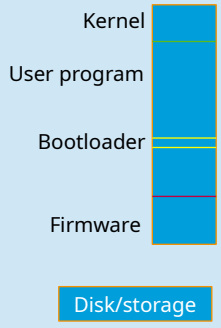
Multiboot protocol

- Methods for kernel to request things from bootloader
  - Setting up options in the bootloader config (like grub config), but that's done by sysadmins and not kernel
  - Specify options in the kernel binary that the bootloader can find
- The second option requires a method/protocol so the kernel can put information
  - A place the bootloader can find it
  - In a format the bootloader can understand
- This is part of the multiboot protocol



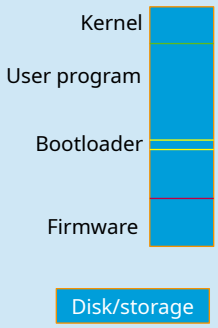
Multiboot protocol

- Time for some code
  - grub-multiboot2/test-kern
  -



Multiboot protocol

- How does the kernel know what the bootloader gave it?
- Bootloader fills in a data structure in memory that describes things like
  - State of computer (memory maps...)
  - Where loaded files are located
  - ...
- Before starting the kernel, the address of that data structure is written to a register
- The first thing a kernel does is to store that address somewhere and then start parsing the data structure
- Back to the example code



Multiboot protocol, resources

For more information

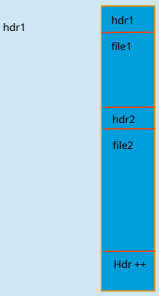
- We use the multiboot protocol for our kernel, so you can have a look there
- https://www.gnu.org/software/grub/manual/multiboot/multiboot.html
- https://wiki.osdev.org/Multiboot

Bundling files for the kernel – initrd

- Initrd = Initial RAM disk
- A binary file that contains a simple file system
  - One of the simplest in common use is "cpio" (we will get back to this)
- Bootloader loads it into memory when the kernel is loaded, then the bootloader tells the kernel where the initrd is (using multiboot)
- Typically contains support files that the kernel needs to boot up
  - Drivers
  - Scripts for initializing and setting up a system
  - Initial binaries and programs
- In this course, we will use a cpio based initrd to contain things like programs we want to start running

The CPIO file format (simplified)

- There are multiple versions and options for headers, but we will look at a simple version
- The basic format is as follows:
  - Files are appended to the binary file with the following:
    - A header that describes one file, with information such as file name, size, modes, user ids, file types etc.
    - The binary file added just after the header
  - Similar in idea to a tar or zip file (except zip files have indexes to where files are)
- To find a file, you scan through the file until you find a maching file name.
- Code example: ramfstst
  - You will get a cleaner example in the P1 source code



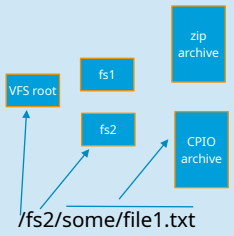
Using a CPIO / initrd file as initial file system

- You can view any archive file like cpio as a file system
- To use it from programs, you need functions for
  - Locating files
  - Opening files
  - Reading files
  - We're ignoring file writes for now
- We could provide "cpio filesystem" functions to programs
  - Would reuire all progams to know about all file systems we want to use, and figure out which to use

Using a CPIO / initrd file as initial file system - VFS

- Alternative approach: provide an abstraction to any filesystem
  - Think OOP or abstract data types
  - Provide a file system data structure that describes the file system and operations on it (opening files, finding files etc)
  - First part of a file path could determine filesystem, then look inside fs
  - In our operating system: VFS
    - lib/drivers/vfs_fs.h and vfs_file.h
    - Mike?

(over)simplified example of VFS idea



TODO

- ELF format
- Loading and starting programs (jump or function call)