Operating systems. Lecture 1

Michał Goliński

2018-10-02

Practical information

Contact

- B3-10
- phone: +48 61 829 5362
- office hours: Tuesday 12:00–13:00, Friday: 13:30–14:30
- golinski@amu.edu.pl
- homepage: https://golinski.faculty.wmi.amu.edu.pl/

Content of the course

- Current operating systems – theory and some implementation details
    - scheduling
    - virtual memory
    - security
    - filesystems

Content of the course cont.

- `bash` as a shell
    - files
    - processes
    - pipes
    - text manipulation
- Programming in `bash` – scripts
- Linux administration basics

Content of the course cont.

- POSIX programming in C (system functions)
- Concurrency – theory and practice with BACI

Literature

- Book for *LPI Linux Essentials*
- The Linux Command Line
- Avi Silberschatz, *Operating System Concepts*
- Andrew Tannenbaum, *Modern Operating Systems*
- Michael Kerrisk, *The Linux Programming Interface*

Presence

- Presence is obligatory (lectures and classes)
- Up to 2 unjustified absences

Grades

- Classes
  - script (or scripts) in `bash`
  - project in C (system programming)
  - maybe something else
- Lecture
  - exam (both theory and practice)

Final remarks

- Ask questions anytime
- I am no native speaker and will make mistakes
- Assume that I do not understand Polish

Questions?

Introduction

Operating system

Operating system is the main software running on a computer, managing hardware:

- **processor** (CPU)
- **memory** (RAM)
- storage
- graphics adapters and screens
- network adapters and network

Other tasks

Besides these tasks a full-fledged operating system

- manages running programs (*user processes*)
- enforces security of running processes/services

- gives a consistent interface, regardless of specific hardware
- manages users, credentials and their capabiliities

Process

Process is the name used for a running program together with all its associated resources (mapped memory pages, opened files, network connections).

Main tasks – CPU

Virtual processor – every process (unless uses some special functionality) can think that it runs on the processor on its own. If the OS decides that the current process used too much time it is *paused* and later *resumed* as if nothing happened.

Main tasks – RAM

Virtual memory – every process starts with memory looking exactly the same. OS cares that while the process is running *logical addresses* (used by the process) are translated into *physical addresses* (used by the hardware). Besides the obvious simplifications for the programmer, this adds security (its impossible/difficult to access memory of another process) and allows the OS to pretendit has more memory than physically available (through *paging* and *swap*).

Kernel

The main part of the OS is called the **kernel** – it's a privileged process running as long as the computer is working, usually in the lowest security mode – kernel has unrestricted access to memory, hardware etc.

Monolithic kernel

If most device drivers, services implementing filesystems etc. run in common memory, then we call such a kernel **monolithic** (e.g., Linux, Windows 9x). This is conceptually simple and gives more performance, yet one bug in a not so important service may bring the whole system down.

Microkernel

If the kernel contains only the core functionality (pausing and resuming processes, virtual memory) and the rest of system services runs as processes with their own memory, we call it a **microkernel**. (e.g., GNU Mach, GNU Hurd). This adds security but lessens performance.

Hybrid kernel

Some operating systems try to mix both approaches – these are called **hybrid kernels** (e.g. Windows NT, macOS, iOS).

. . .

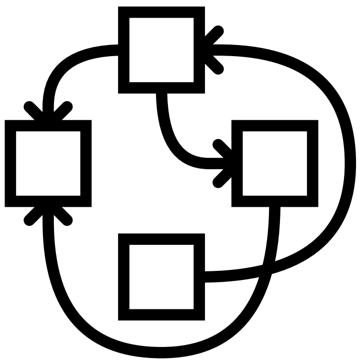

Figure 1: GNU Hurd logo

Some people believe these are really just monolithic kernels with good PR.

Booting the OS

Overview

- OS is us usually stored on disk, it needs to be loaded to memory in order to run
- when computer is turned on a series of more and more complex programs is run that achieve the task
  - firmware tests and initializes hardware
  - FW finds a suitable bootloader in connected storage
  - bootloader runs, often in stages
  - in the end OS is up and running

Historical perspective

- First computers were pre-wired for specific operations (no true booting)
- First commercial computers used some loading procedures, in particular I/O was often already implemented
- Afterwords a myriad of different mechanisms were implemented
- 1981 – first IBM PC, its clones not much later (BIOS)
- 2005 – first computers with UEFI, BIOS successor

BIOS

What is BIOS?

BIOS is the old standard by which the system may communicate with firmware. Communication was really used with MS-DOS, BIOS is not used during normal computer operation today, just at the very start.

Booting the OS with BIOS

- POST (power on self test)
- disks are probed in prescribed order if they are bootable (by checking the MBR)
- if a bootable disk is found, its MBR is loaded into memory and run as a program
- with a modern OS the BIOS is not used afterwords

Master Boot Record

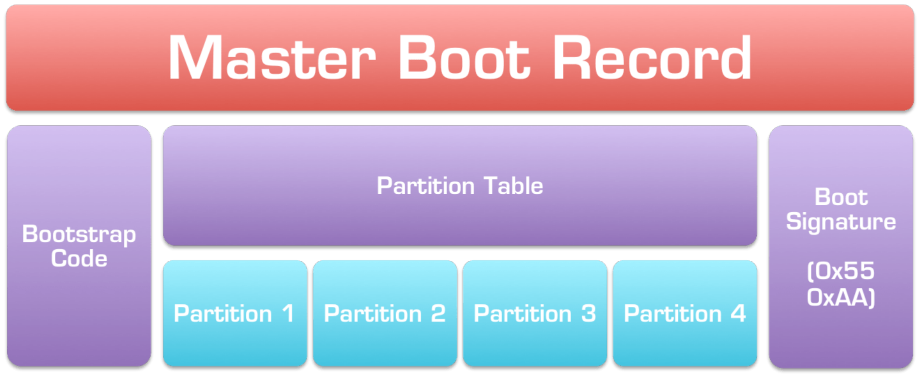

Figure 2: Master Boot Record

Master Boot Record cont.

- Code (bootloader stage 1) – 446 bytes
- Partition table – 4×16 bytes
- Boot signature – 2 bytes (`0xAA55` means this drive is bootable)

Bootloader stage 1

Stage 1 runs under a heavily constrained environment (very little memory), so it needs to be simple.

Standard bootloader (used under Windows) looks for a partition (one of the four) that is flagged as active and loads its first sector as the next stage. It does not try to understand the filesystem structure on that partition.

. . .

GRUB behaves similarly, but the next stage location is hardcoded in the bootloader at installation (usually it is just the next sector). The partition table is not consulted at this point, that is the task of the second stage.

Bootloader Stage 2

This stage is usually the last before loading an OS. The environment it runs in allows for more complex program, so it is usually configurable and it understands the structure of filesystems, i.e., you can use usual filenames with paths, not only sector numbers.

Chainloading

This mechanism is complicated but flexible. To run a bootloader you only load it into memory and perform a JMP instruction. It allows for one bootloader (say GRUB) to easily load another bootloader (say Windows Boot Manager from Win 10). This is called *chainloading*.

Limitations and problems of MBR

- Installation of an OS may overwrite the MBR of an already installed OS, leading to problems if one does not know how the system works.
- Partitions are described by starting sector and length in sectors. 32-bit numbers are used. This gives a limit on partition size: $2^{32} \times 512B = 2$TiB. This was the main motivation to create and adopt GPT (and UEFI with it).

UEFI

What is UEFI?

UEFI is a new interface by which computer firmware and operating system communicate. It was created around 2005 by Intel first for Itanium, later ported to x86. This allowed to cut most of the historical baggage that was necessary to be carried in BIOS. In particular the new partition table – GPT – lifts the 2 TiB limit.

Booting the OS with UEFI

- POST (power on self test)
- a program from a prescribed list is loaded
- if that is not possible disks are probed in order if they are bootable (by checking whether that have a special EFI System Partition)
- if a bootable disk is found, a program from the EFI partition is loaded (by default `\EFI\boot\bootx64.efi`)
- with a modern OS the UEFI is not used afterwords

GUID Partition Table

- Although MBR partition table could be used it is better to use GPT
- 128 partitions
- Limit on partition size: $2^{64} \times 512B = 2^{33}$TiB $= 8$ZiB.
- Comparison: in 2013 about 4 ZiB of data was created worldwide.

EFI System Partition

- FAT12, FAT26 or FAT32, has to be marked as ESP
- can have multiple EFI programs (bootloaders etc.)
- if configured, UEFI can boot Linux kernel directly
- in principle UEFI can show a menu with many option to choose at boot

Secure Boot

UEFI allows to have Secure Boot, i.e., to force a cryptographic signature on loaded EFI programs. This makes some attacks impossible (i.e., run a kernel modified with a rootkit), but may make it harder to run a self-compiled kernel.

Final remarks

- In principle it is possible to have BIOS+GPT (because of a special feature of GPT that allows for an MBR). This is not allowed under Windows. Windows allows only:
  - BIOS + MBR
  - UEFI + GPT