

Operating Systems

Basic architecture and operation

Tamás Mészáros

<http://www.mit.bme.hu/~meszaros/>

Hussein Al-Rikabi

rhussein@mit.bme.hu

Budapest University of Technology and Economics (BME)
Department of Measurement and Information Systems (MIT)

Let's design an operating system!

The Operating System

is a collection of software that
control the operation of the computer's hardware
in order to **support** the execution of user's tasks.

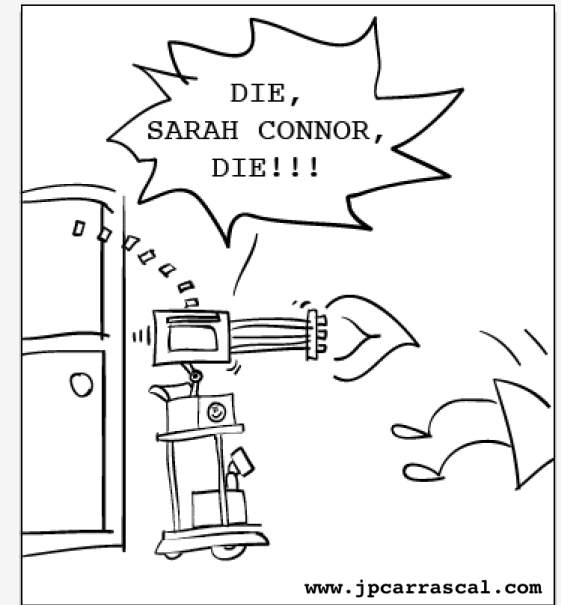
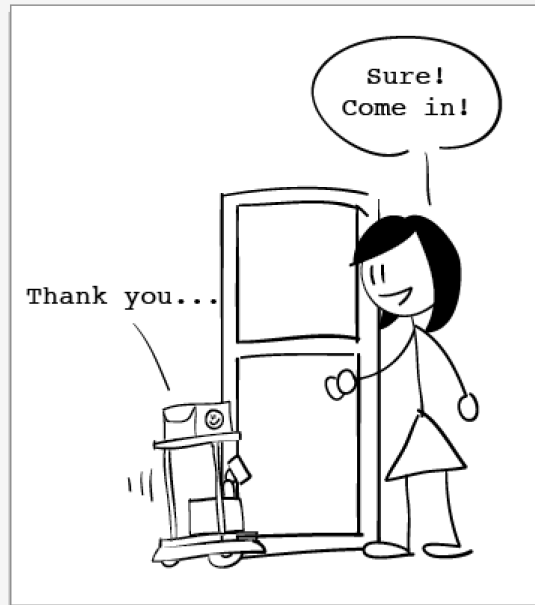
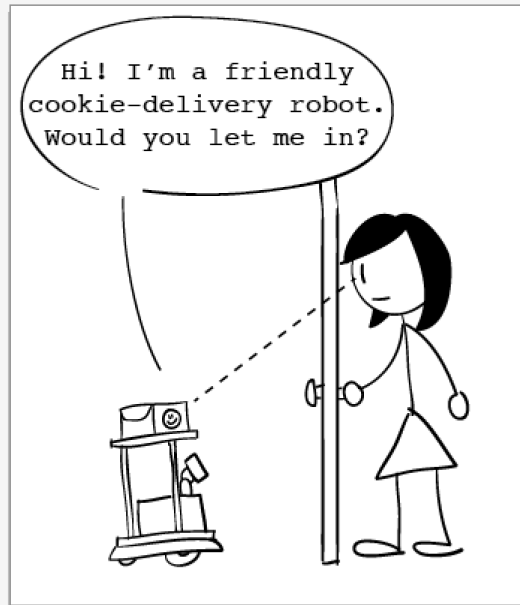
What we expect

- handles multiple tasks
- Reliable (If error happens, other tasks should not be affected)
- Safe

We run software that

- solve our tasks
- came from different sources ([OS](#), app store, Web etc.)

Can we trust the software?



OS architecture: multitasking



OS architecture: governance

AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICH RENDERS DOZENS OF VIDEO FRAMES EVERY SECOND

BECAUSE I WANTED TO SEE A CAT JUMP INTO A BOX AND FALL OVER.

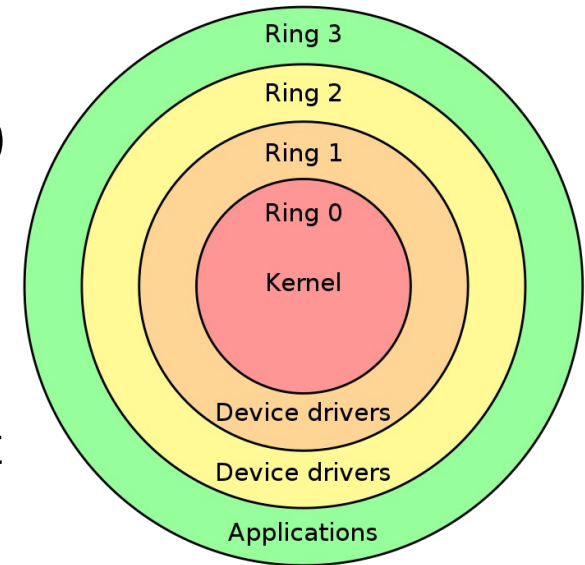


I AM A GOD.

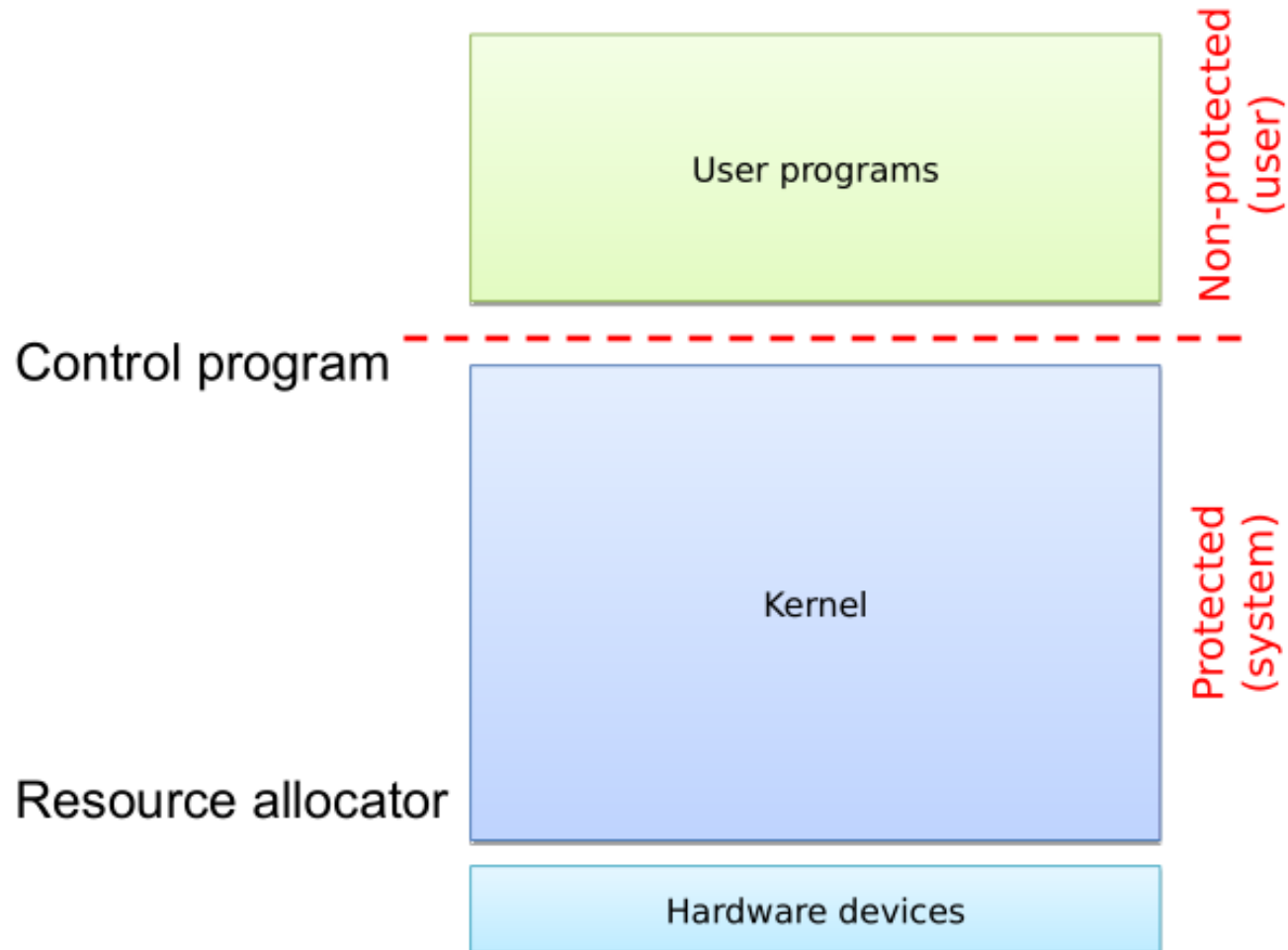
forrás: [xkcd](#)

How to implement governance?

- Can a software govern another one?
 - Recap: CPU protection modes ([HW Architecture](#))
 - at least two operational modes
 - Level 0. (protected)
 - Level 1- (user)
 - restricted access to HW, restricted instruction set
-
- Some part of the OS is running at Level 0 protected mode
 - this governs all other software
 - handles their life cycle (creation, operation, termination)
-
- *The **kernel** is a part of an OS software that is running in protected mode, has complete control over user level programs and grants resources for their operation.*
 - Everything else is running in User mode (enforced by hardware)



Kernel operating in protected mode



The kernel Jobs

- Controls user-mode processes
 - life-cycle management (creation, operation, [termination](#))
 - event management (passes hardware and software events to processes)
 - provides common services to simplify software development
- Manages resources
 - set ups hardware elements
 - provides functions to access them
 - handles their events (e.g. interrupts)
 - resolves [conflicts](#), allows simultaneous access when it is safe (Write , Read)
- Keeps the system reliable and secure
 - protects resources from programming errors and malicious requests
 - [separates](#) processes from each-other to protect them
 - provides security functions to user-level programs

Other OS parts

System Library is a part of the OS that provides common user-level functions to programs.

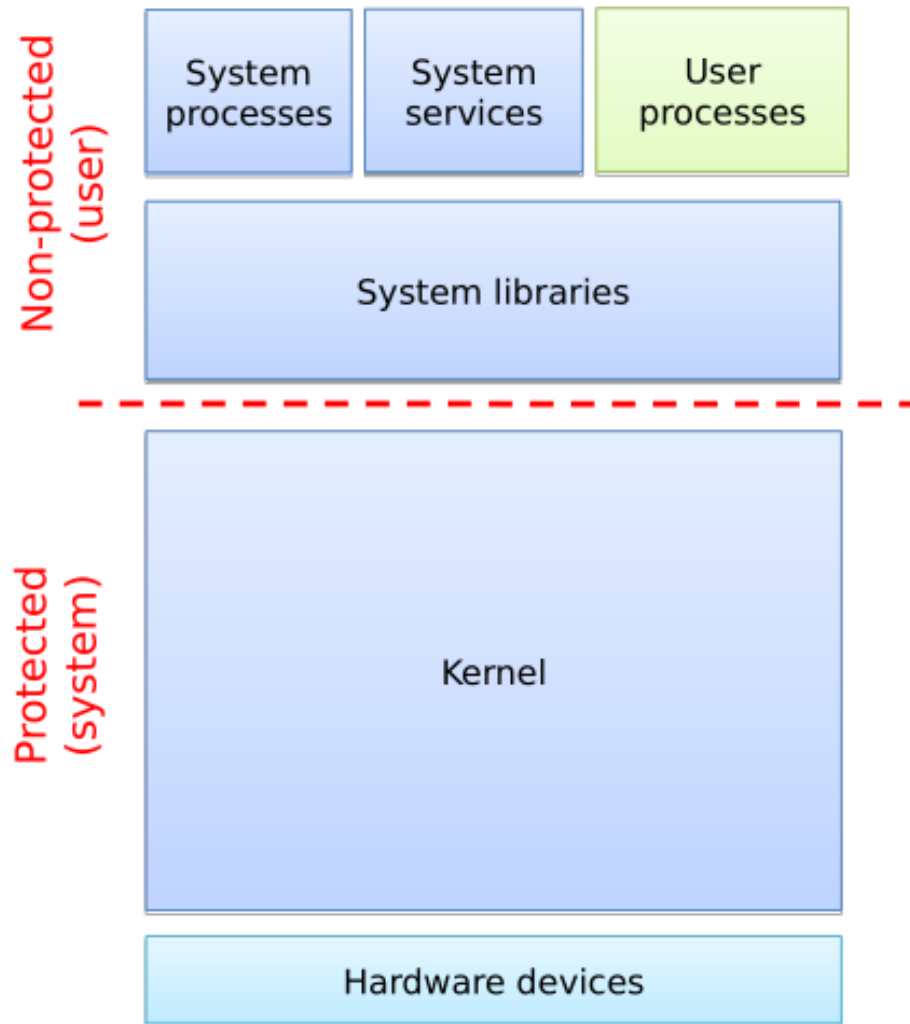
System Programs are software tools that solve tasks related to the operation of the OS.

(Ex. Task Manager , File Explorer)

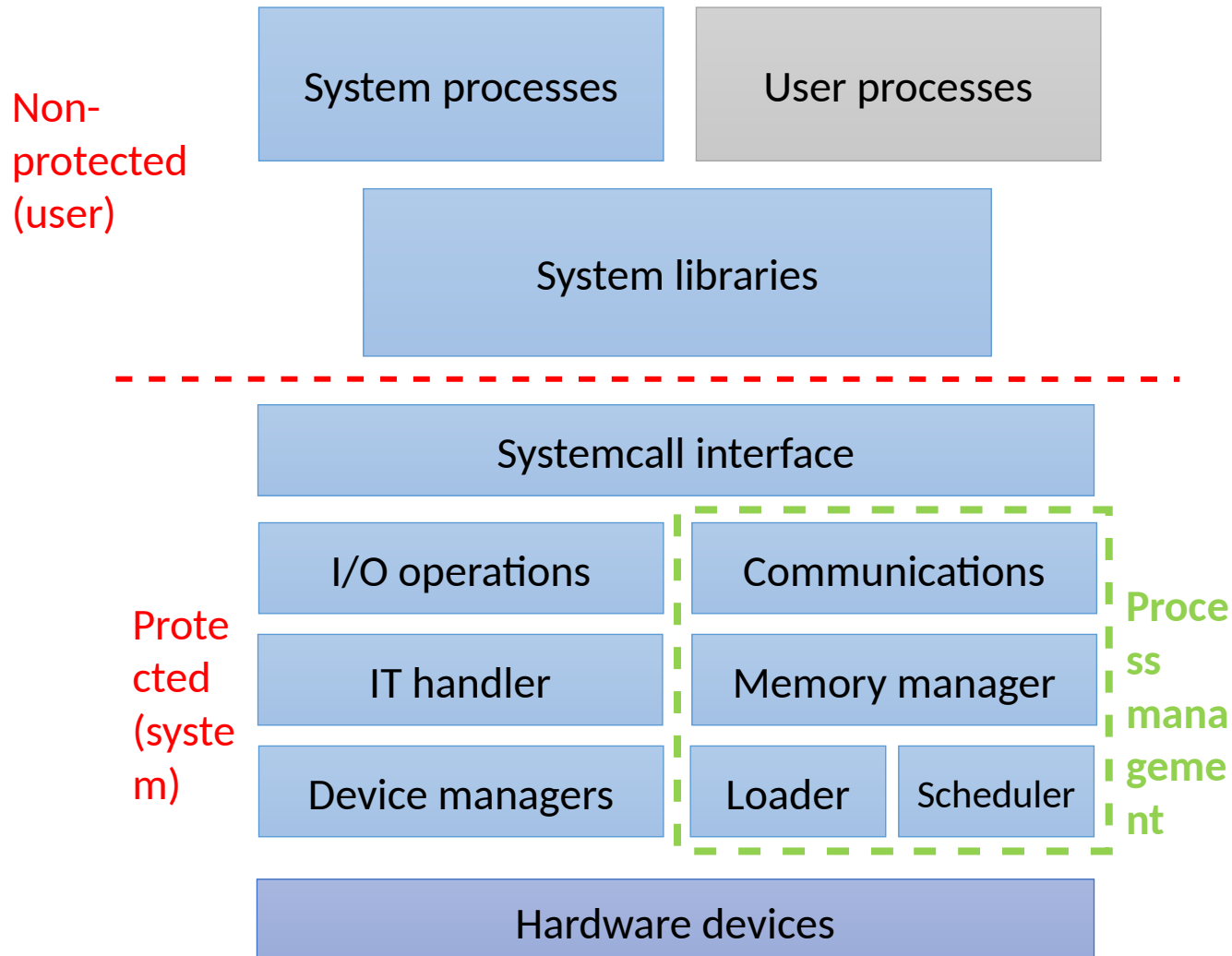
System Services are system programs that provide continuously available services.

(Ex. Print spooler)

Main parts of an Operating System



The main blocks of the OS and the kernel (recap)



The OS structures in detail: principles and models

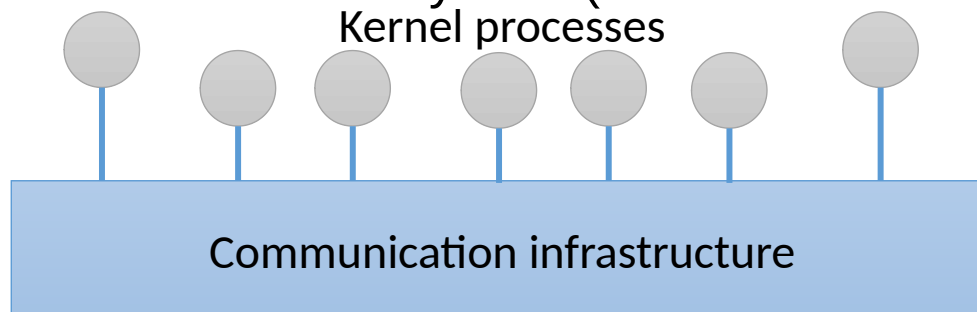
- The kernel of the OS is typically a complex system
- Monolithic** vs. **Microkernel** (see later also)
- The **Monolithic** kernel is ONE program
 - Pro: better performance, simple implementation
 - Con: error sensitive, more security risks
- The **Microkernel** is a distributed system
 - Pro: more reliable and secure (one part fails, other parts can still work)
 - Con: more complex structure, harder to implement, slower operation
- Layered structure** (from HW to user apps.)
 - »Comprehensible, flexible, extendable, less complex development
 - »The layers are separated by **well defined interfaces** (can be standardized)
 - »The more layers and interfaces, the more overhead -> slower operation
- Modular structure**
 - »Different HW architectures (x86, ARM, ...), different vendors -> huge code-base (million lines of code)
 - »It isn't necessary to support all devices at once
 - »» Decompose the kernel into modules and only load the necessary ones
 - Static (offline): at compile time, or during a system reboot
 - Dynamic (online): loading and unloading during runtime

What's the problem with kernel structures?

- When did the TV say?
 - Don't turn me off, 220 important updates are pending
 - Needs reboot, because updates were performed – while watching a movie
- When can a vehicle control system crash?
 - Because a dirty CD is inserted
 - One of the components are changed during a maintenance
- Why do such phenomena occur when using an OS?
 - Complex systems, numerous devices and functions
 - Monolithic kernels are typical
 - One mistake causes the whole system to struggle or crash
 - Hard to isolate and repair (debug) the problems
 - Hard to maintain the integrity of the system
 - A bug can cause **security weaknesses** (for the whole system)
 - Programmers are not super humans

What can be done to amend the situation?

- Isolate the sensitive parts, keeping the monolithic structure
 - Most of the problems are caused by **device drivers**
- **Isolate** them: sandboxing
- **Armored OS**: wrap the device driver functions in a protective function which is able to detect problems and, for example stop the driver function
- A **user-mode agent** managing the detected problems
 - Decompose the system using virtualization (see later in this semester)
- These techniques are often used in the current OS-s
- Throw away the **monolithic** structure
- Build the kernel as a distributed system (workers and communication)



The concept of microkernel

- Distributed system in general
 - Consists of independent units (computational and storage)
 - It is transparent to the user, the only differences are in the internal operation
 - Can be distributed physically
- The microkernel as a distributed system
 - A **kernel mode task manager** is necessary to perform the distributed tasks
- Tasks: memory management and scheduling
- Distributed: the workers are communicating and cooperating
- (optionally the most relevant device drivers)

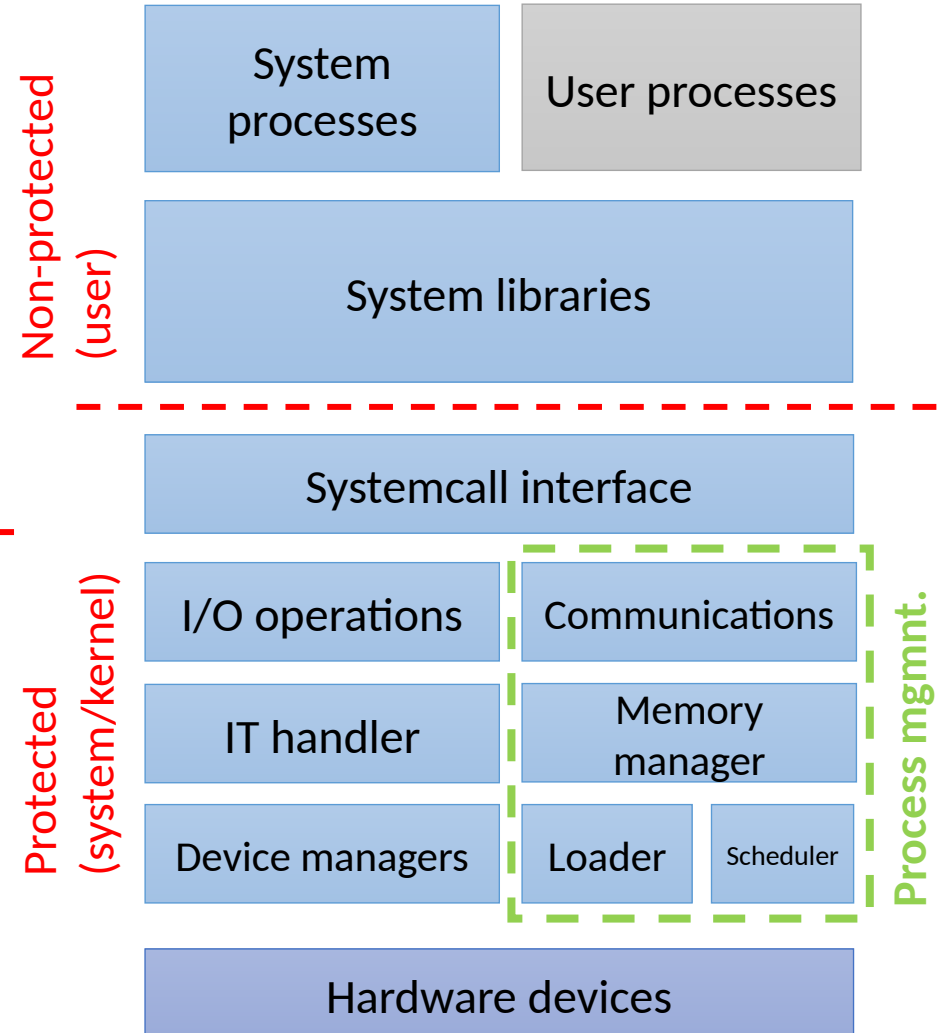
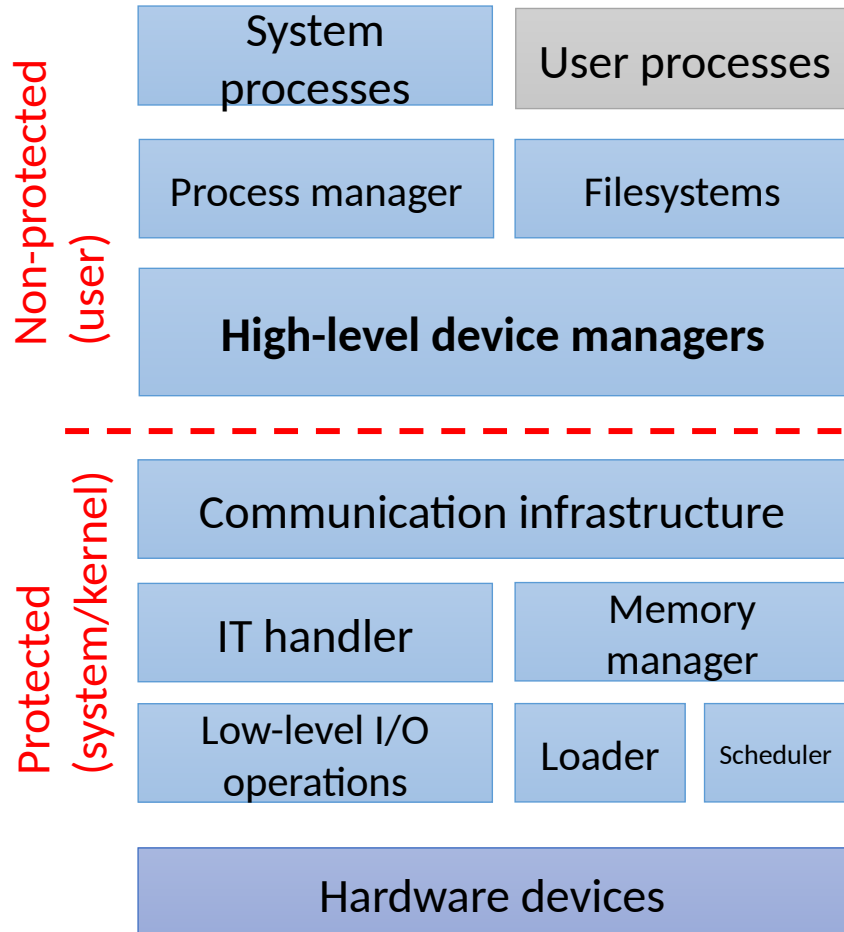
- Separated

- Pros and Cons (see [Tanenbaum-Torvalds debate](#))
 - Flexibility: multiple API-s together, dynamic expansion
 - Reliability: only a small section of the code has to be „good” (may be verified by formal methods)
 - Fault tolerant: errors in the user mode programs can be handled by kernel mode section
 - Slow

Microkernel

vs.

Monolithic kernel



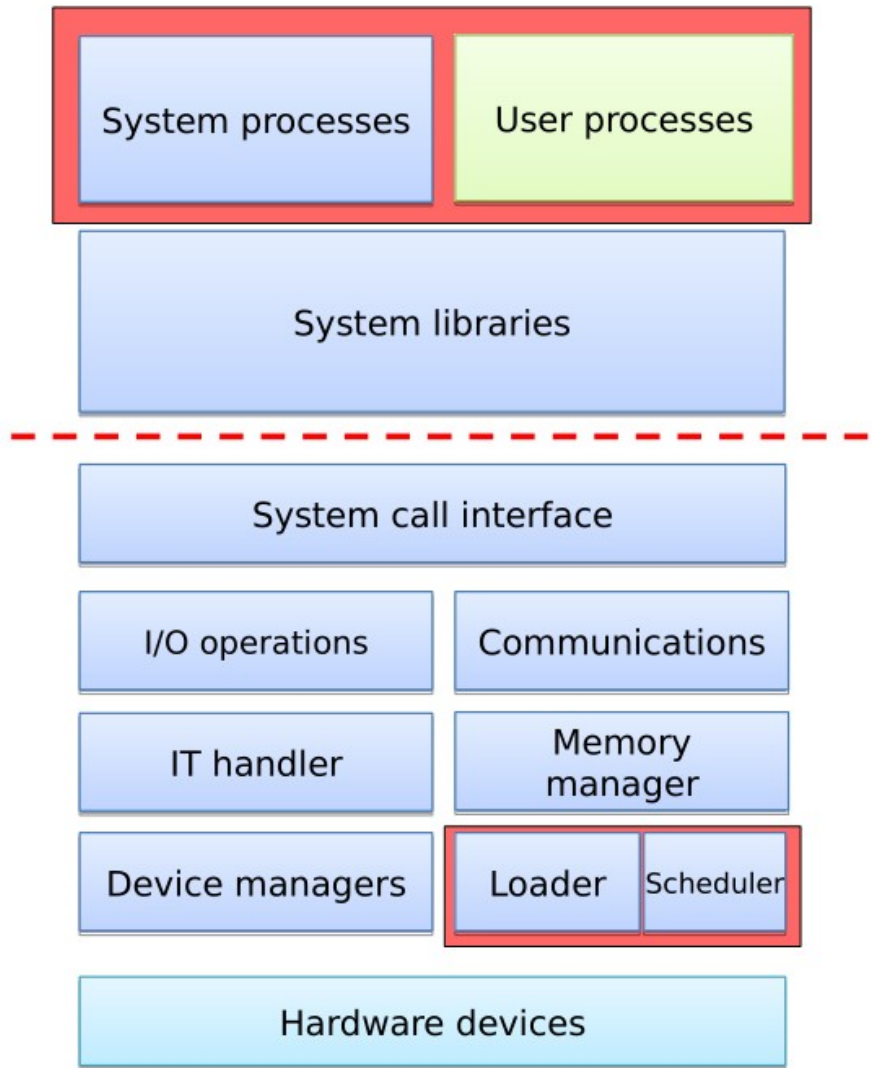
Second generation microkernels

- IPC instead of RPC
 - **RPC** (Remote Procedure Call) are forwarded as
 - **IPC** (Inter Process Communication) messages
 - This method is really slow, compared to system calls
- **The second generation** microkernel improves the IPC speed
 - **Exokernel**: minimized kernel, simple and fast system calls
 - **L4 microkernel**: very fast IPC (may be forwarded through CPU registers)
 - 10-20 times faster than classic microkernel
 - Very few kernel functions (7 functions)
 - The protected kernel section is small (5-15K LoC)
 - The small kernel makes possible the **formal modeling and [verification](#)**
- **Multiserver**: more than one server are running on the same microkernel
- **Hybrid kernel**: monolithic kernel over a microkernel
 - Windows is containing microkernel elements, but it isn't microkernel based

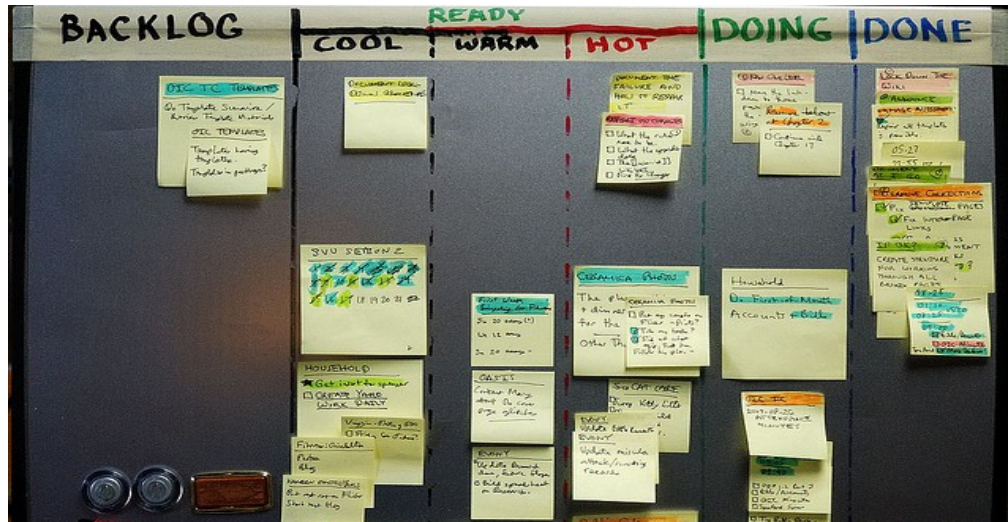
The OS as a control program

The kernel as a control program – overview

- The Operating System
 - helps solving user's tasks
 - **control program**
 - **resource allocation**
- Expectations
 - handles multiple tasks
 - reliable, secure
 - meets users' requirements



How to handle tasks?

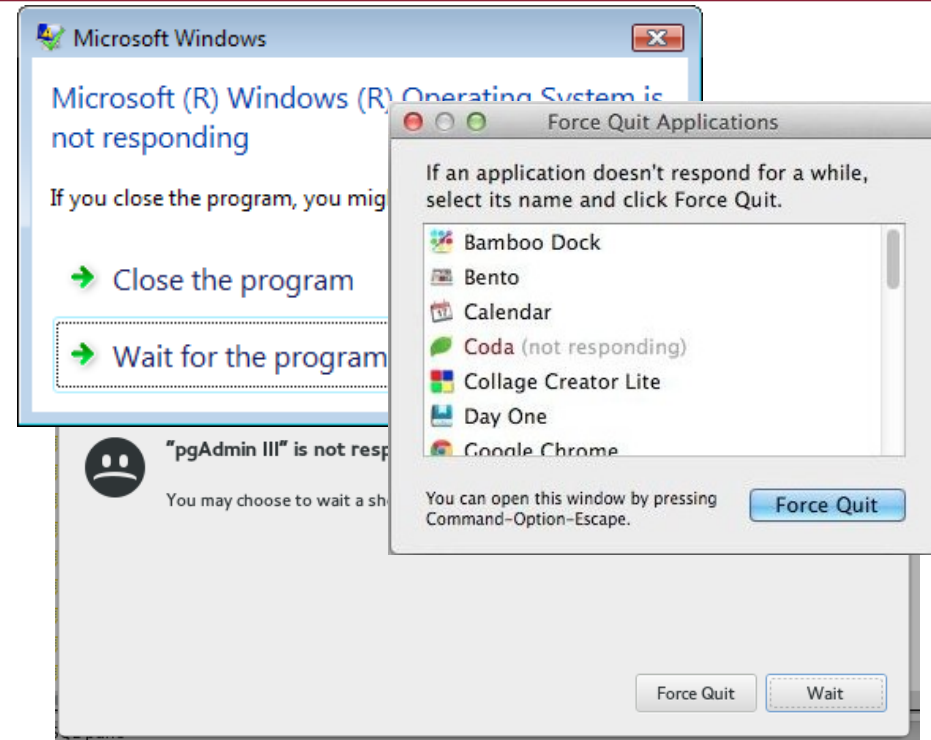


Let's try to characterize tasks (Kind of tasks)

- I/O-bound
 - mostly waiting for I/O (reading, writing, events)
 - need less CPU time
 - examples: Web server, File storage, Email etc.
- CPU-bound
 - the CPU is their most required resource
 - need less I/O
 - example: simulations, mathematical algorithms, machine learning etc.
- Memory-intensive
 - need large amount memory
 - enough memory → CPU-bound
 - not enough → I/O-bound (swapping)
 - e.g. large matrix operations, document indexing and search etc.
- There are many others...
 - Real-time

User's expectations

- Wait less
 - waiting time
 - turnaround time
 - response time
- Work efficiently
 - CPU utilization
 - throughput
 - overhead
- Be deterministic



The optimal task execution system

- Ideally...

- assures that all tasks are performed in time
- minimizes wait and response times
- maximizes the resource usage
- has no overhead



- In practice...

- some programs run slowly or even freeze
- the OS require lot of resources (Chrome)
- the battery depletes fast
- sometimes event the entire OS freezes for a time
- we can't answer calls on mobile
- ...



- Why?

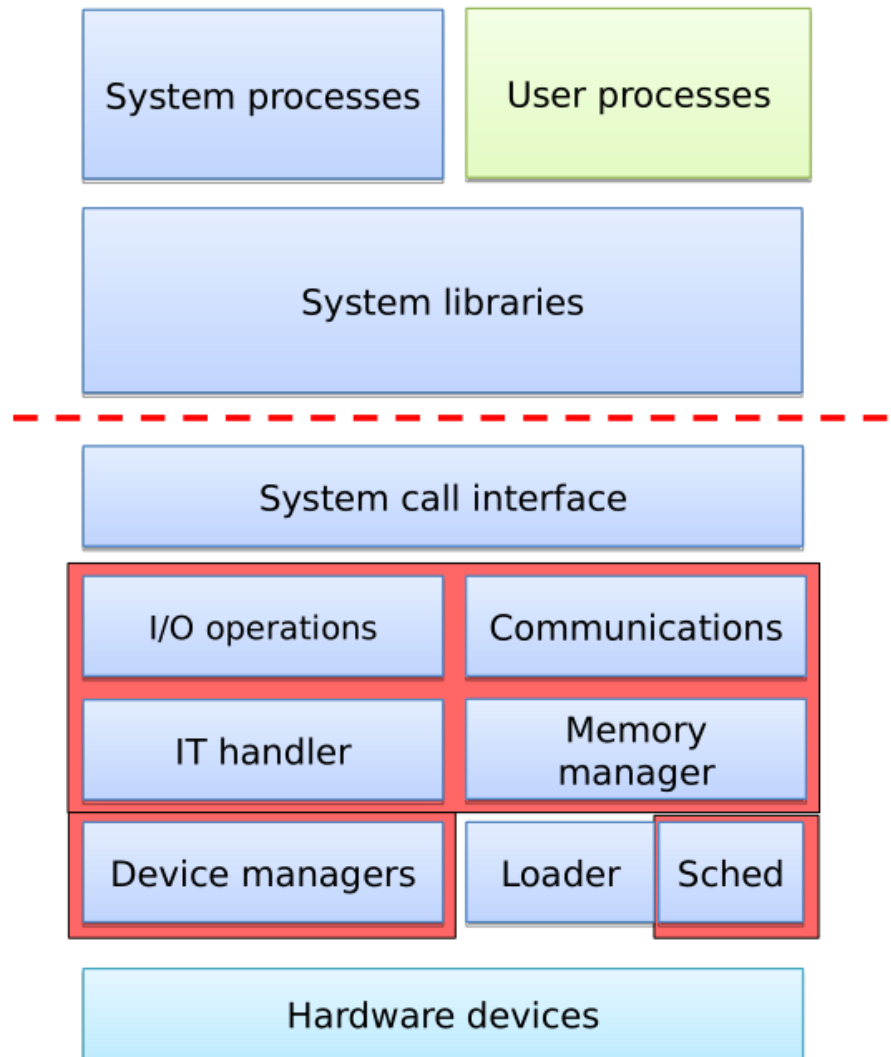
Why is it hard to design a good OS?

- We can't see into the future
 - what tasks are to come
 - what will be their characteristics
- There are many tasks running at the same time
 - they have different requirements
 - and different goals and optimums
 - sometimes the system collapses under the heavy load
- Tasks affect each-other
 - cooperation
 - competition
- There are errors
 - programming
 - hardware

The OS as a resource allocator

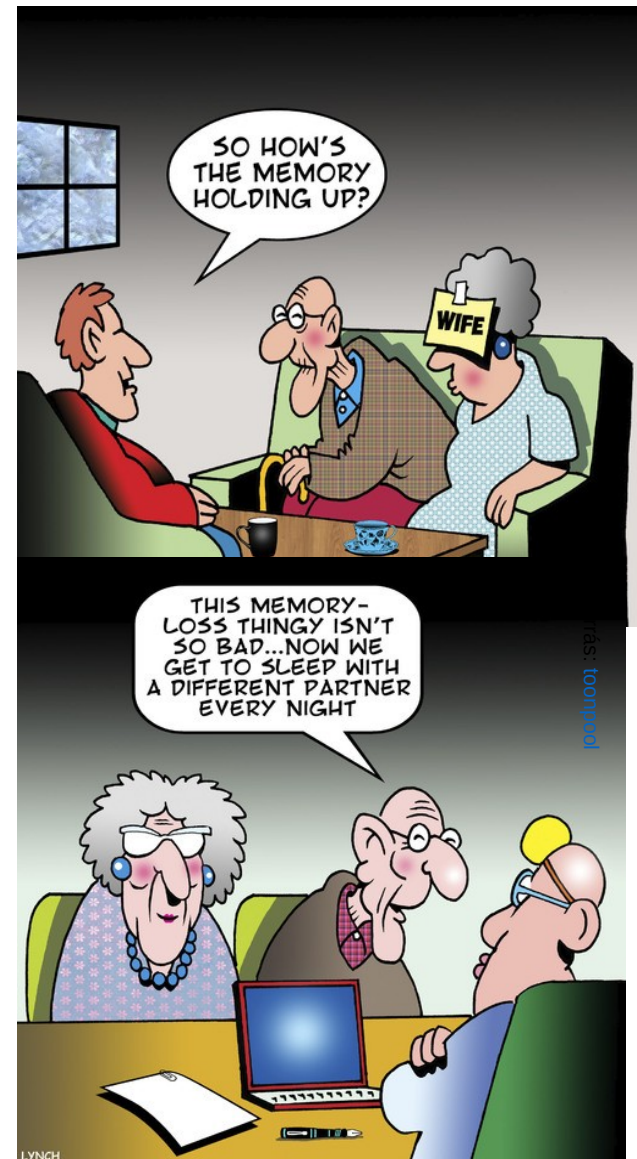
OS as a resource allocator – overview

- The Operating System
 - helps solving user's tasks
 - control program
 - **resource allocation**
- Expectations
 - handles multiple tasks
 - reliable, secure
 - **highly utilizes resources**
- Resources
 - processing units (CPU, VGA)
 - system memory
 - storage systems
 - computer peripherals
 - other hardware components
 - software resources



How to handle memory as a resource?

- Allocation
 - resource: physical memory
 - requested by: user tasks and kernel
- Store tasks
 - program
 - data
- Provide memory for the kernel
 - program
 - administrative data
- Security and reliability
 - separation of users' tasks
 - error detection and handling
- Support data sharing
 - communication between separated programs



Sharing File systems and storage

- User-level access
 - end user (store photos , videos) Privacy,
 - Administrator
 - Programmer
- Internal operation (OS)
 - various data organization (**NTFS**, FAT32
 - modular structure
 - internal interfaces
- Storage Level
 - Physical (HDD , SSD)

End of Today's Lecture

Operating Systems

Basic architecture and operation

Continued

Tamás Mészáros

<http://www.mit.bme.hu/~meszaros/>

Hussein Al-Rikabi

rhussein@mit.bme.hu

Budapest University of Technology and Economics (BME)
Department of Measurement and Information Systems (MIT)

The OS boot procedure

Unix – Windows

System Calls (Kernel operation)

Summary

Labs

The OS boot procedure

The boot procedure

- Booting is not a trivial process, from powering up the system to the user logging in.

- **The main sequence of things happening is as follows:**

1. The CPU starts its operation from a fixed ROM loader address
 - Level 0: The ROM loader, It is built-in the computer by the manufacture
 - In that ROM address there is a system initiator program
 - It is in a fixed memory range, e.g.: ROM, EEPROM, flash, etc.
 - So, The CPU knows that address and look for it, when booting up.
 - This program is called: BIOS (in Windows), bootROM (in Android)
2. It loads the bootloader program to the RAM, then starts it
3. Level 1: The RAM loader (small program loaded from the hard drive)
 - It's located in the MBR (Master Boot Record) of the hard drive
 - It checks the HDD structure
 - It starts to look for the OS in the Hard drive and start it.

----- All of the above steps are the same for different OSs (Windows, Linux) -----

The boot procedure

4. Level 2: The OS loader

- It's located in the PBR (Partition Boot Record) or VBR (Volume Boot Record)
 - This part is OS specific (the OS installed it at the OS installation)
 - It can optionally load further boot loaders (Windows: Bootmgr, Linux: GRUB2)
 - It can have a GUI also (e.g. to select the OS to load)
 - Initiates the OS, loading the kernel's code then starts it
- The kernel is started
-
- So the Boot Manager (Bootmgr) looks like this:

Windows Boot Manager

Choose an operating system to start, or press TAB to select a tool:
(Use the arrow keys to highlight your choice, then press ENTER.)

Windows 7

>

Windows 7 Safe Mode

To specify an advanced option for this choice, press F8.

Tools:

Windows Memory Diagnostic

ENTER=Choose

TAB=

Chainload into GRUB 2

When you have verified GRUB 2 works, you can use this command to complete the upgrade: `upgrade-from-grub-legacy`

Debian GNU/Linux, kernel 2.6.28-11-generic

Debian GNU/Linux, kernel 2.6.28-11-generic (recovery mode)

Debian GNU/Linux, kernel memtest86+

Other operating systems:

Windows Vista (loader)

Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, 'e' to edit the
commands before booting, or 'c' for a command-line.

Booting the OS (Unix kernel)

5. The kernel's self loading process

- Starts it's operation in user mode
- A small utility uncompresses the kernel's code
- First initializations: memory manager, stack, interrupts, other descriptors.
- Initializing basic devices (e.g. keyboard, video card, ...)

6.Changing into kernel mode (Protected)

- First steps in kernel mode
- Writing page descriptor tables ()

----- before this point the CPU can use a small partition of the memory,
now it can use the whole RAM-----

- Setup IT vectors and handlers (Interrupt vectors are addresses that inform the interrupt handler as to where to find the ISR)
- Starting protected memory management (It sets the kernel own pages in memory)
- Creating process description table and starting the first process
- Loading the driver necessary for the system start (initrd: initial ram disk) which are architecture dependent init. Functions (drives, DMA, etc.)
- Starting the scheduler (which is responsible for which tasks to run)
- Setting up the data structures for parallel computing (context changes)
- Loading and starting the code **init**, witch is the first user level program.

Booting the OS (Windows kernel)

- The VBR (Volume Boot Record) finds, loads and runs the Level 2. loader (Bootmgr)
 - It starts in 32-bit user mode (does not know yet if 64 OS is installed)
 - Showing the boot menu if it is necessary
 - Changes to 64-bit mode if it is possible and loads the next program: the OS loader
- Winload.exe is the kernel's loader
 - Runs in 32/64-bit kernel mode
 - Loads the Ntoskrnl.exe and its dependencies and the device drivers necessary to the system start
- Ntoskrnl.exe – the kernel
 - Runs in 32/64-bit kernel mode
 - Phase 0 – initialization with disabled interrupts
 - Initializing: boot processor, kernel data structures, lock tables, etc..
 - Setup IT vectors and handlers
 - Initializing: memory and process manager
 - Phase 1
 - Switches to a normal process with the highest priority
 - Binding the physical and logical processors, setup CPU cores (Before that, it was using only one core)
 - Initializing: video card, I/O, and many other subsystems
 - The kernel is up and running

Critical user processes in Windows (Booting the OS 2.)

- SMSS.exe – session manager (Manages individual used sessions)
 - Performs special user mode tasks
- **Then Using only low-level (one core executive) system calls**
 - Checking file systems integrity, attempts to repair (Autochk.exe)
 - Sets up the basic environmental variables
 - Initializing the memory management
 - Builds the whole registry database (that is necessary to configure the system for one or more users, applications, and hardware devices.)
 - Starts the Windows initialization program - Wininit.exe
 - Creates the default sessions (1 typically which is the administrator session)
 - Starts the Client Server Runtime csrss.exe
 - Starts the login manager (Winlogon.exe)
 - Wininit.exe – further user mode init. Steps (session 0)
 - Winlogon.exe – user login (session 1+)
- **The system is ready for user login**

Critical user processes in UNIX (Booting the OS 2.)

- The first user-mode program started: init
 - This is the parent of all the other processes (ex. when interrupt process ends, it has to clean up)
 - Running constantly
 - It's task to reach a given system state and maintain it
 - The configuration of init defines the runlevel of the system
- Runlevel
 - A complex state description which defines:
 - The operating mode of the system (maintenance, multiuser, graphical, etc.)
 - The tasks (services) of the OS to perform
 - It is marked with a number (e.g. 0-6), or sometimes with a single letter
 - The meaning is different in different Linux distributions, but typically:
 - Level 0: full shutdown
 - 1 or S: single-user administrator mode
 - ~2-5: multi-user mode, with or without GUI
 - 6: reboot
 - The system admin may change the mode

Critical user processes in UNIX (Booting the OS 3.)

- The configured system state is set up by init to the corresponding runlevel
- The init performs some steps that
 - are running in a **predefined order** (some devices or services depend on other to be initialized)
 - These orders set up the system including:
 - Defining the drives and file systems
 - Starting services (user login, GUI, webserver, etc...
 - **The system is ready after init commands**

Alternatives of sysinit

- Problems with init
 - Dependencies between scripts (order)
 - Cannot run in multiple threads, Hence become slower
 - Error handling is not sophisticated (If a previous service fails, it still might try to do perform it, and it will also fail)
- Systemd is an alternative, used now in (RedHat, Linux, Ubuntu 15.04+, Arch CentOS, Debian etc),
 - Declarative description of the services, precise dependency trees
 - Parallel and scheduled starting of the services (lowers booting time)
 - better in detecting and managing errors
 - command set is changed..
- But it is very **difficult to use**.
- Many unhappy Debian/Ubuntu user...

How the kernel works

The kernel is a very complex software

- Examples:

- Windows XP: [45 million LOC](#) (the entire OS)
- Linux kernel: 20 million LOC

- See

- <http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>
- <http://www.pabr.org/kernel3d/kernel3d.html>
- <http://www.jukie.net/bart/blog/linux-kernel-walkthroughs>
- http://en.wikiversity.org/wiki/Reading_the_Linux_Kernel_Sources
- Linux vs. Windows kernel (videó, Mark Russinovich)

How to design such a big software

- **Layered architecture**

- with (standardized) interfaces

- *The **system call interface** is a programming interface that separates the protected and user mode operation and provides common functions for user mode programs*

- **Monolithic design**

- the kernel has a single, large address space
 - eases the tasks of kernel programmers
 - But it is hard to make it resilient

*Today's OSeS: modular and monolithic
(not distributed not microkernel)*

Linux: vmlinux

Windows: ntoskrnl.exe

- **Modular design**

- avoids loading the entire kernel into the memory
 - can be loaded in (compile time / configurable / runtime)

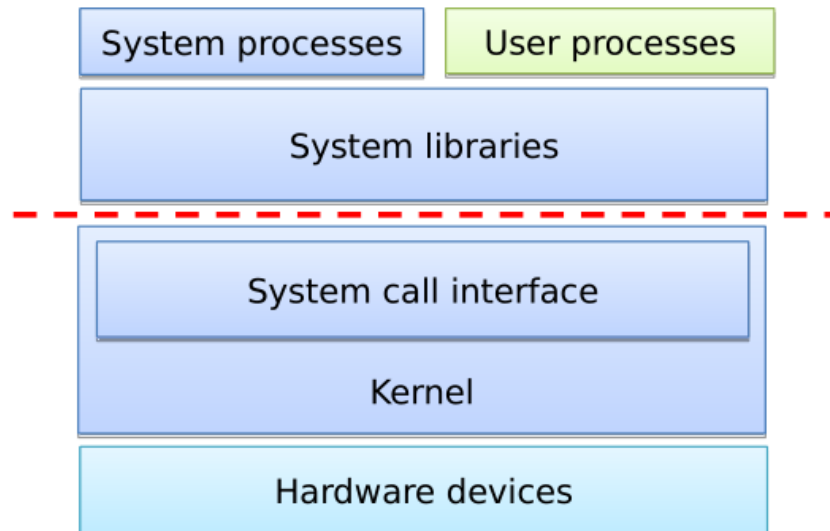
- **Distributed**

- the kernel is composed of multiple, separated address spaces
 - it also implements a communication system between them

The syscall interface

- The system call is NOT like a conventional function call
 - it changes the CPU mode (user → protected)
 - using a special CPU instruction


```
trap, syscall, sysenter
```
 - the kernel's interrupt handler
 - recognizes the interrupt (that it is a syscall)
 - performs the syscall
 - and returns from the interrupt (CPU: `iret`, `sysexit`)



System calls under UNIX

- Performing the system call (e.g. `read()`, `write()`,...)
 - it is similar to a standard function call
 - implemented by a system library (**libc**),
- The libc performs the SYSCALL instruction (generating an IT)
 - the interrupt changes the CPU-mode (to protected)
 - the kernel's SYSCALL interrupt handler function is invoked
- The SYSCALL
 - collects and checks the arguments (usually from CPU registers)
- The system call is performed
 - the return values are also stored in CPU registers
- The kernel returns from the interrupt (`sysexit`)
 - the CPU changes back to user mode
 - the CPU goes back to the last address from the stack, and continue its normal operation



Virtual system calls

- **Default** Syscalls are frequent and they have overhead
 - software IT
 - CPU mode change
 - handling the IT
 - passing arguments to and from kernel address space
- How to cut down the overhead?
 - idea: avoid mode change
 - only works for a few system calls but worth to try
- **Virtual system calls**
 - a part of the kernel address space is mapped into the user space
 - some safe system calls are implemented using this technique
 - they work as a simple function call without mode change and interrupt
 - no changes necessary to the user programs (the syscall is the same)
 - examples: `gettimeofday()`

What can we do to achieve better design?

- Kernel sandboxing [armored OS](#)
 - create a wrapper around susceptible calls (error detection)
 - provide a kernel component to detect and recover from such errors
 - OS/app sandboxing, [MirageOS](#), [Docker](#), [BlueStacks](#)
 - smaller attack surfaces, more control and governance
- Doing this virtualization adds one more level of control
- containers: completely separated subsystems on the same kernel
 - Change the kernel design from monolithic to distributed
 - distributed system
 - only necessary functions are implemented in protected mode
 - (due to huge over head it is rarely used for critical cases)

Summary: monolithic vs microkernel kernel

- The kernel is a complex piece of software
 - layered, modular, monolithic or microkernel design
- The boot process is also quite complex
 - ROM, RAM, OS and kernel loaders

Monolithic kernel

In a monolithic kernel, user services and kernel services are kept in the same address space.

The monolithic kernel is larger in size.

Execution speed is faster in the case of a monolithic kernel.

The monolithic kernel is hard to extend.

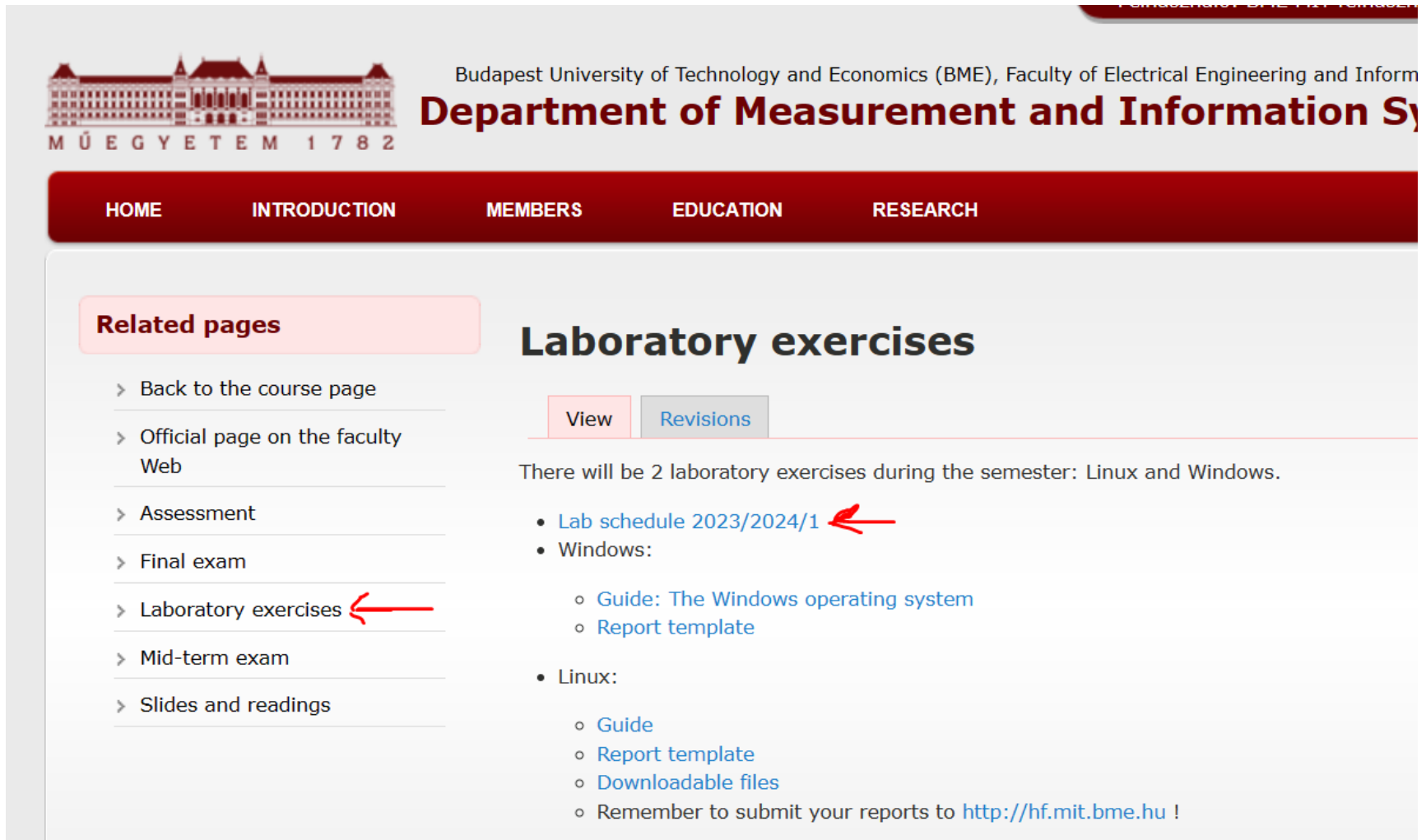
The whole system will crash if one component fails.

Fewer lines of code need to be written for a monolithic kernel.

Debugging and management are complex in the case of a monolithic kernel.

Monolithic OS is easier to design and implement.

Lab: Operating Systems BME MIT



The screenshot shows the website for the Budapest University of Technology and Economics (BME) Faculty of Electrical Engineering and Informatics, Department of Measurement and Information Systems. The page features a navigation bar with links to HOME, INTRODUCTION, MEMBERS, EDUCATION, and RESEARCH. On the left, a 'Related pages' sidebar lists links such as 'Back to the course page', 'Official page on the faculty Web', 'Assessment', 'Final exam', 'Laboratory exercises' (highlighted with a red arrow), 'Mid-term exam', and 'Slides and readings'. The main content area is titled 'Laboratory exercises' and includes tabs for 'View' and 'Revisions'. Below the tabs, a text block states: 'There will be 2 laboratory exercises during the semester: Linux and Windows.' This is followed by a bulleted list of links: 'Lab schedule 2023/2024/1' (highlighted with a red arrow), 'Windows:', 'Guide: The Windows operating system', 'Report template', 'Linux:', 'Guide', 'Report template', 'Downloadable files', and a reminder to submit reports to 'http://hf.mit.bme.hu !'.

Budapest University of Technology and Economics (BME), Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

HOME INTRODUCTION MEMBERS EDUCATION RESEARCH

Related pages

- › Back to the course page
- › Official page on the faculty Web
- › Assessment
- › Final exam
- › Laboratory exercises ←
- › Mid-term exam
- › Slides and readings

Laboratory exercises

View Revisions

There will be 2 laboratory exercises during the semester: Linux and Windows.

- [Lab schedule 2023/2024/1](#) ←
- Windows:
 - [Guide: The Windows operating system](#)
 - [Report template](#)
- Linux:
 - [Guide](#)
 - [Report template](#)
 - [Downloadable files](#)
 - Remember to submit your reports to <http://hf.mit.bme.hu> !

Operating Systems Laboratory - Tuesday 14:15 - 16:00 - Room IB141

Week	Date	Group A	Group B	Group C	Teacher 1	Teacher 2
1	2023-09-05	-	-	-		
2	2023-09-12	-	-	-		
3	2023-09-19	-	-	-		
4	2023-09-26	-	-	-		
5	2023-10-03	-	-	-		
6	2023-10-10	Windows	-	-	Vetró Mihály	Tagelsir Zahraa
7	2023-10-17	-	Windows	-	Vetró Mihály	Tagelsir Zahraa
8	2023-10-24	-	-	Windows	Vetró Mihály	Tagelsir Zahraa
9	2023-10-31	Linux	-	-	Alekszejenkó Levente	Tagelsir Zahraa
10	2023-11-07	-	Linux	-	Alekszejenkó Levente	Tagelsir Zahraa
11	2023-11-14	-	-	Linux	Alekszejenkó Levente	Tagelsir Zahraa
12	2023-11-21	-	-	-		
13	2023-11-28	-	-	-		
14	2023-12-05	-	-	-		

Group members:

Group A	Group B	Group C
A7L40E	HU3TIL	RQ74QO
ACM7BI	H7KYNF	RRB7VB