Operating Systems

Basic architecture and operation

Lecturer: András Millinghoffer (milli@mit.bme.hu)
Coordinator: Tamás Mészáros (http://www.mit.bme.hu/~meszaros/)

Budapest University of Technology and Economics (BME)
Department of Artificial Intelligence and Systems Engineering (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.

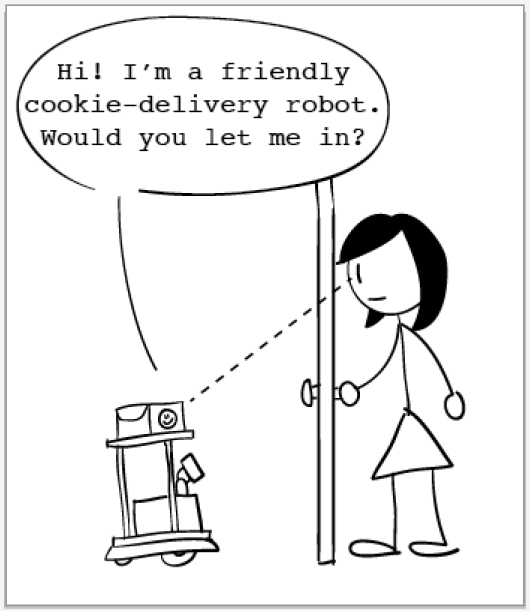
What we expect

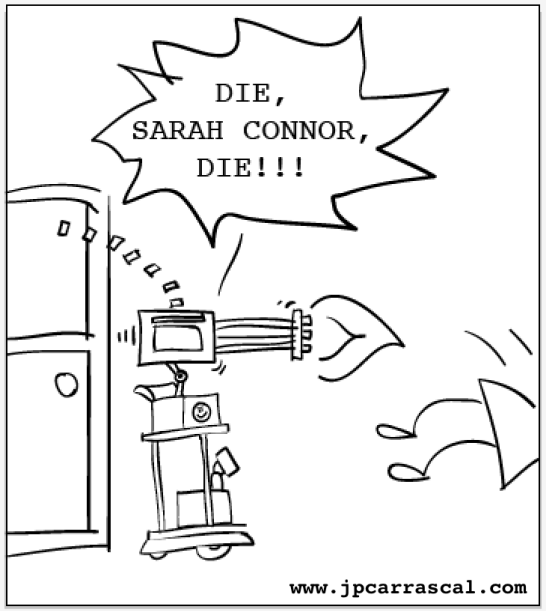
- handles multiple tasks
- reliable
- Safe

We run software that

- solve our tasks
- came from different sources (OS, app store, sw repo, Web erc.)

Can we trust the software?





OS architecture: multitasking

OS architecture: governance

forrás: xkcd

How to implement governance?

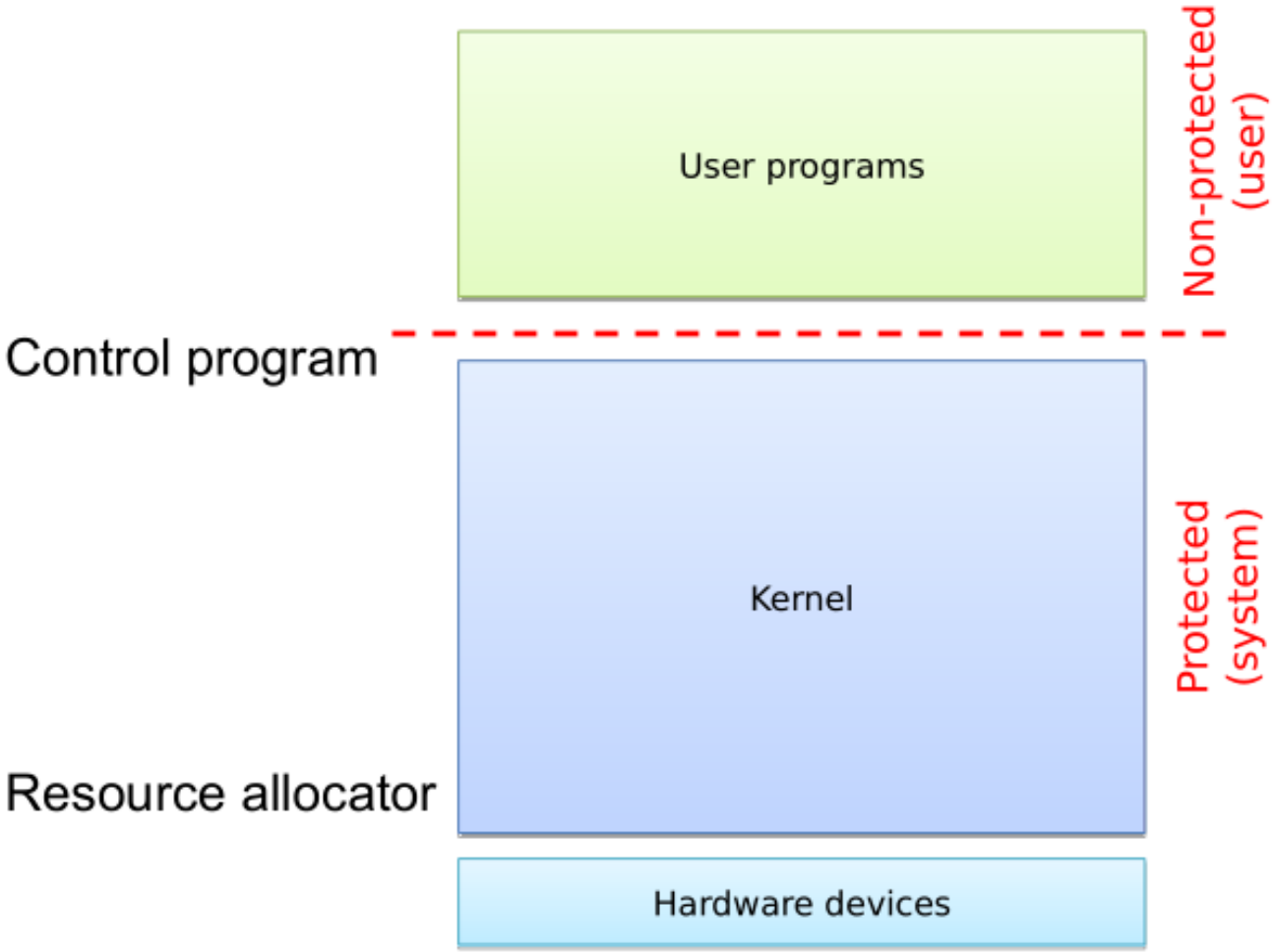
- Can a software govern another one?
- Recap: CPU protection modes (HW Architecture)
  - at least two operational modes
  - Level 0. (protected)
  - Level 1- (user)
    - restricted access to HW, restricted instruction set

- 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

- 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

- 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

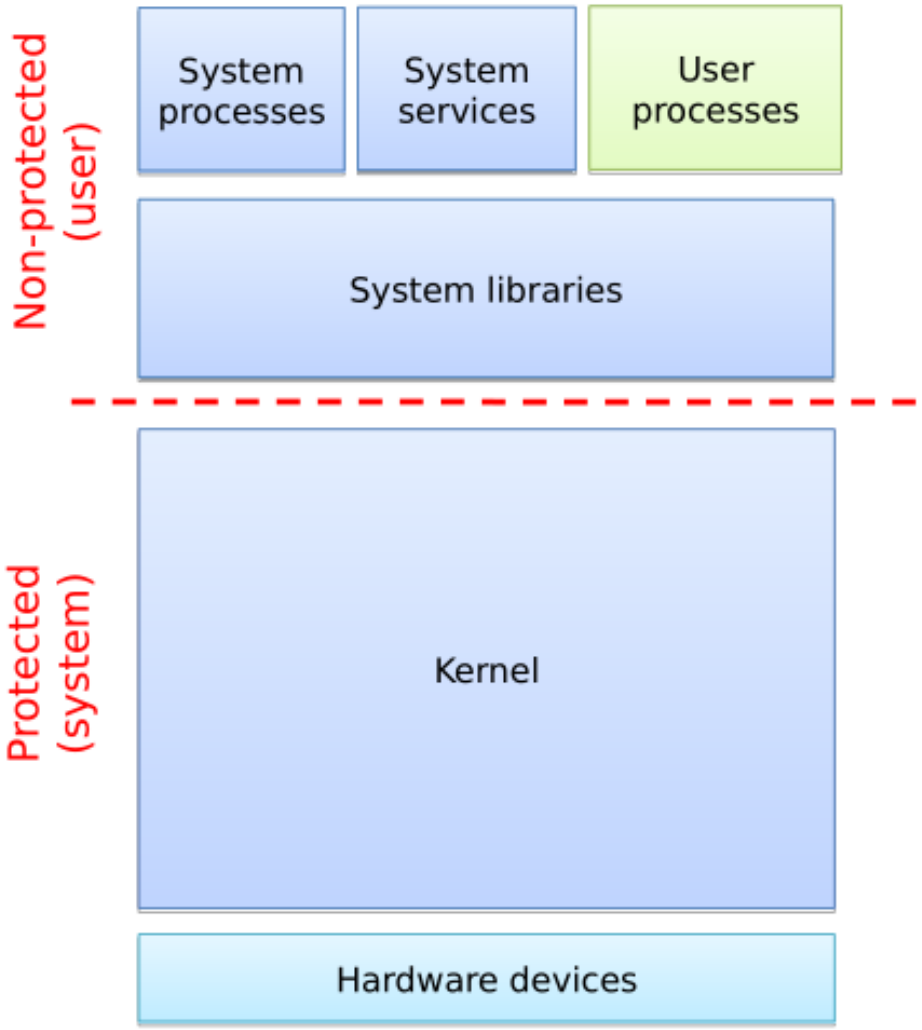
Other OS parts

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

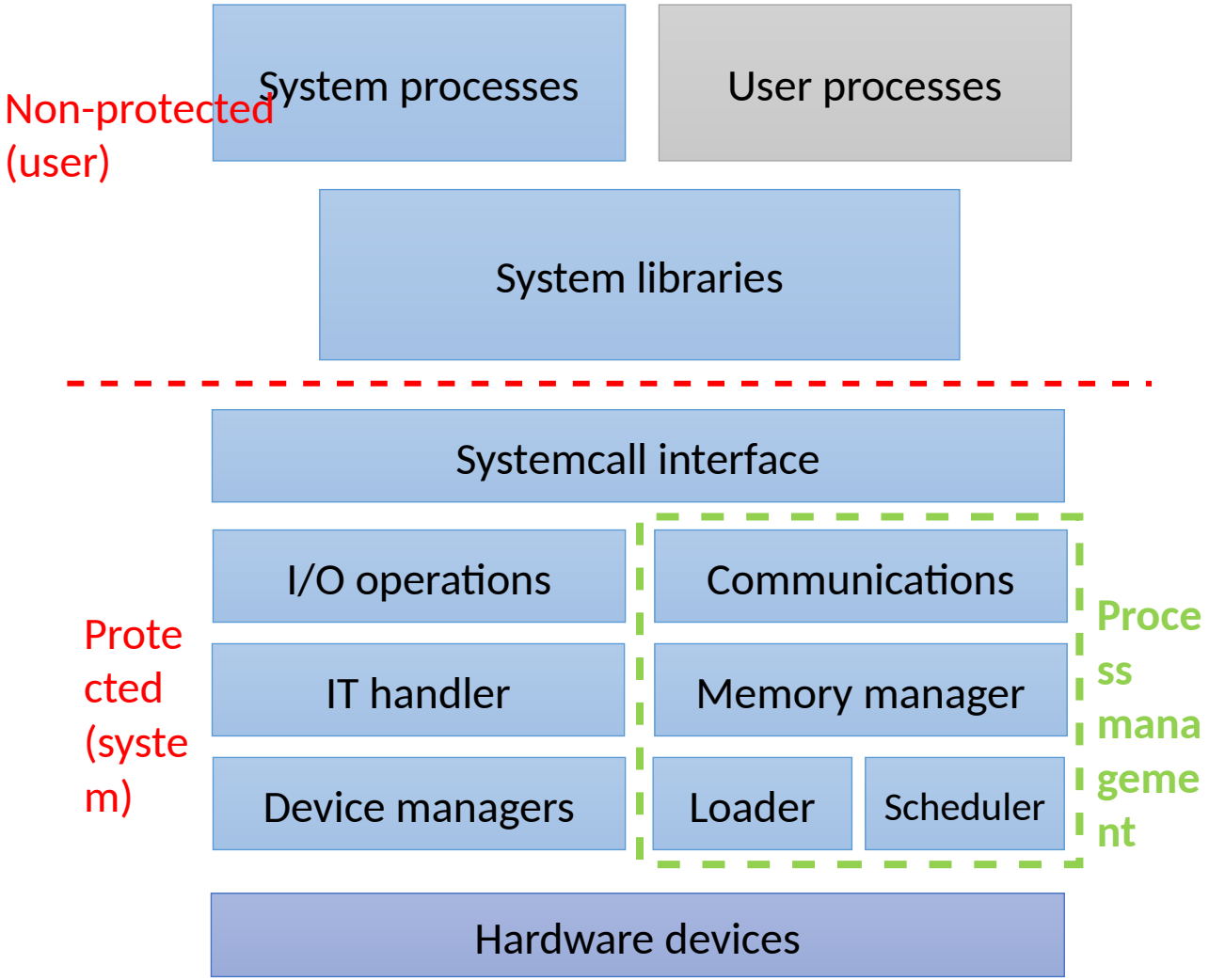
System Programs *are software tools that solve tasks related to the operation of the OS.*

System Services *are system programs that provide continuously available services.*

Main parts of an Operating System



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



The OS structures in detail: principles and models

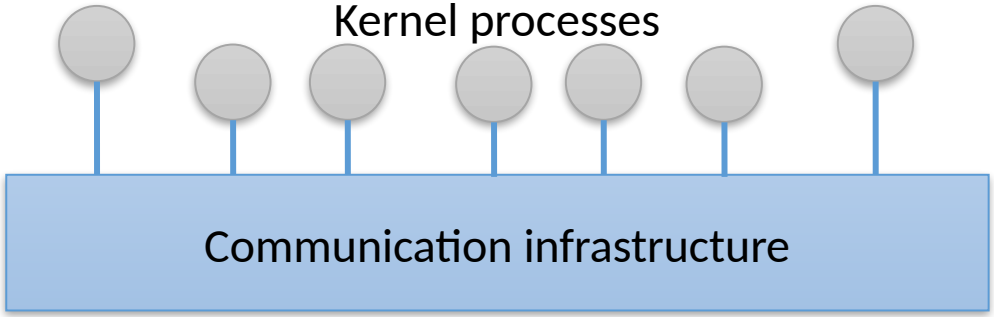
- The kernel of the OS is typically a complex system
- Monolithic vs. Microkernel (see later also)
  - The monolithic kernel is ONE program
    - Pro: great performance, simple implementation
    - Con: error sensitive, more security risks
  - The microkernel is a distributed system
    - Pro: more reliable and secure
    - 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
  - Pay €400 or all of your channels will be encoded
- When can a vehicle control system crash?
  - Because a dirty CD is inserted
  - One of the components are changed during a maintainance
- Why do such phenomena occur when using an OS?
  - Complex systems, numerous devices and functions
    - Real and imaginary urges to develop more 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
    - 1 small bug / 100 lines of code is typical
  - Not only the kernels are problematic, user programs as well…

What can be done to amend the situation?

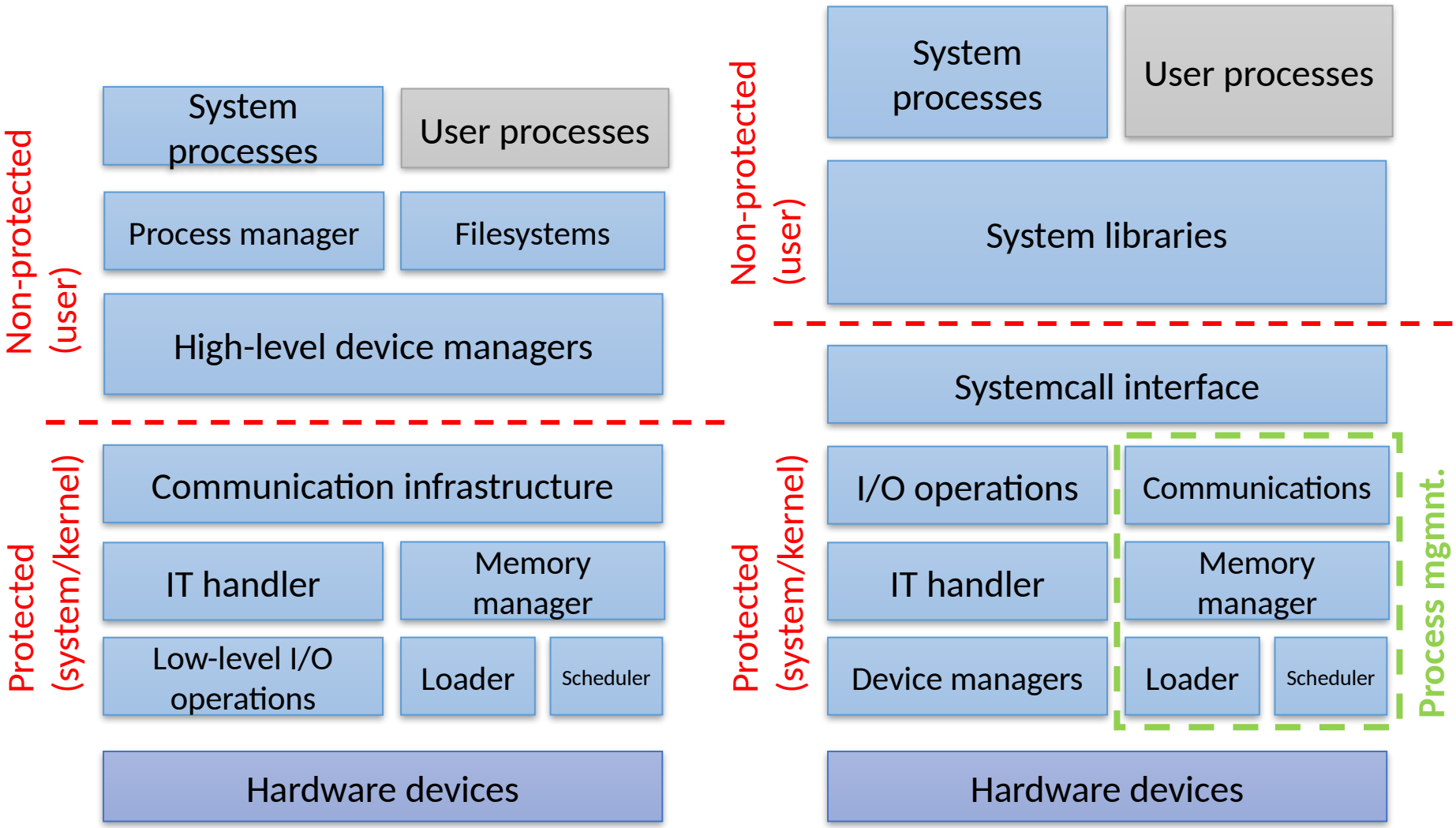
- Isolate the sensitive parts, keeping the monolithic structure
  - Most of the problems are caused by device drivers
    - Isolate them: sandboxing
    - Armored OS: wrap the device driver functions in a protective function which is able to detect problems and, for example stop the driver function
    - A user-mode agent managing the detected problems
  - Decompose the system using virtualization (see later in this semester)
    - Multiple virtualization methods
    - Causing performance drop
  - 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)
  - Only the most essential functions use kernel mode



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)
  - Individual programs implementing the kernel's other tasks
    - Running in user mode
    - Separated from each other, like every other user task
- 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
  - Using the right programming patterns are mandatory: modular coding, interfaces
  - SLOW: communication is multiple times slower than system calls

Microkernel vs. Monolithic kernel



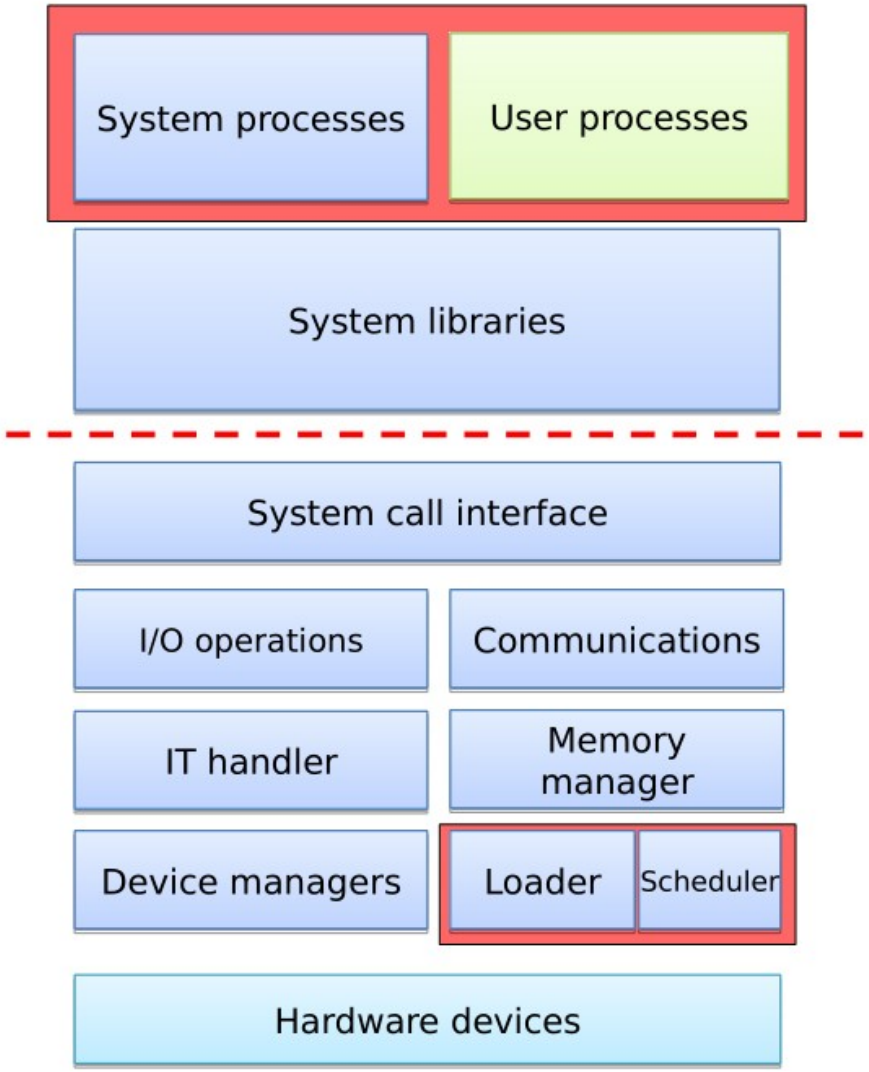
Second generation microkernels

- IPC instead of RPC
  - **RPC** (Remote Procedure Call) are forwarded as
  - **IPC** (Inter Process Communication) messages
  - Which are forwarded by the communication infrastructure/subsystem
  - 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 trhough CPU registers)
  - 10-20 times faster than classic microkernel
  - Very few kernel functions (e.g. L4 provides 7 functions)
  - The protected kernel section is small (5-15K LoC)
  - HW dependencies are more important
  - 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
    - OS X XNU: Mach microkernel + BSD UNIX hybrid kernel
    - Windows is containing microkernel elements, but it isn't microkernel based

The OS as a control program

The kernel as a control program – overview

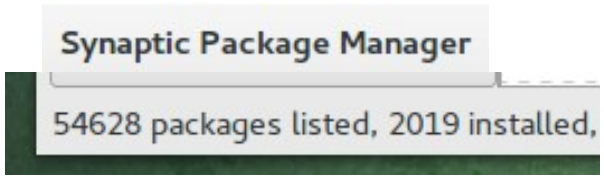
- The Operating System
  - helps solving user's tasks
  - **control program**
  - **resource allocation**
- Expectations
  - handles multiple tasks
  - reliable, secure
  - meets users' requirements



How to handle tasks?

What kind of tasks?

- All kinds of computers (from servers to small devices)
- → Tasks of all kinds → huge variety of software tools




```
> yum list all | wc -l
22747
```

source: statistica.com

Let's try to characterize tasks

- I/O-bound
  - mostly waiting for I/O (reading, writing, events)
  - need less CPU time
  - examples: Web server, File storage, Email etc.

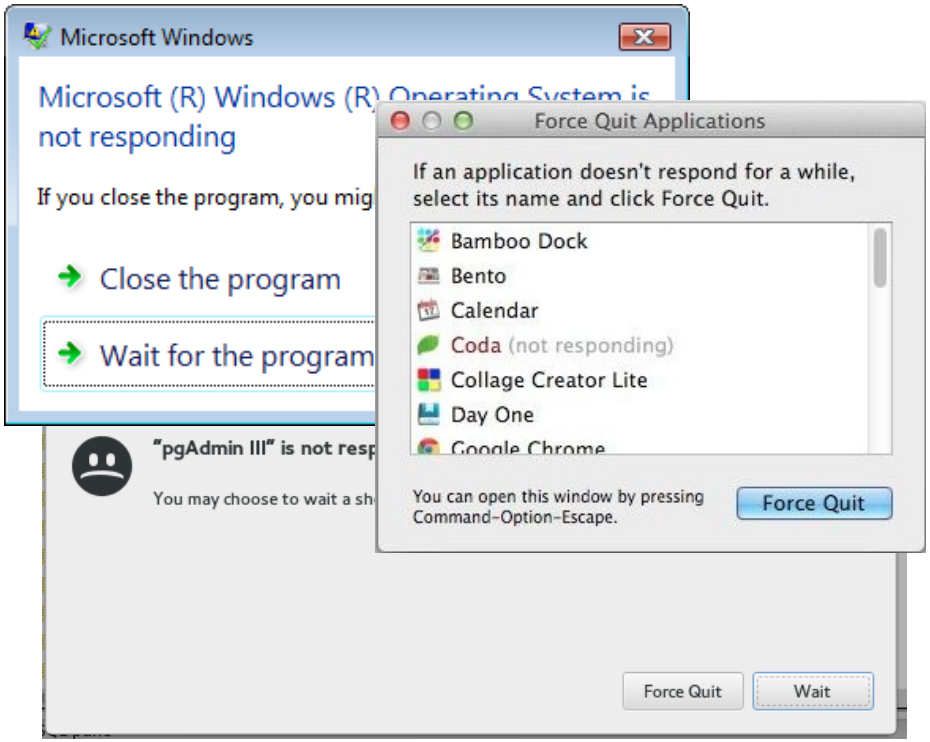
- CPU-bound
  - the CPU is their most required resource
  - need less I/O
  - example: simulations, mathematical algorithms, machine learning etc.

- 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, graph DB, etc.

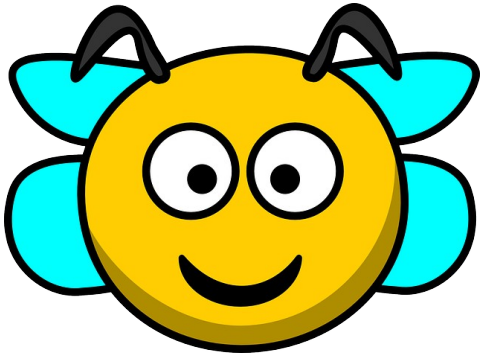
- The are many others...
  - Real-time
  - watching movies etc.

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
  - the battery depletes fast
  - sometimes even the entire OS freezes for a time
  - we can't answer calls on mobile
  - ...

- Why?

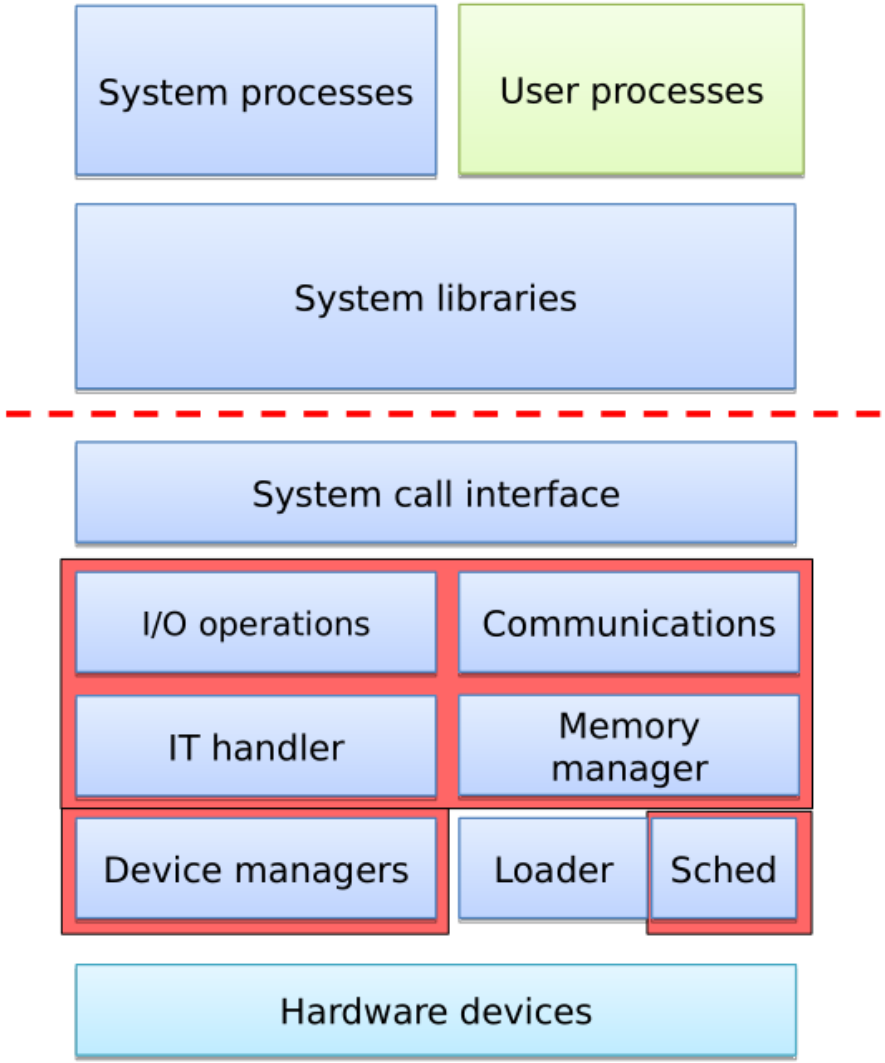
Why is it hard to design a good OS?

- 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

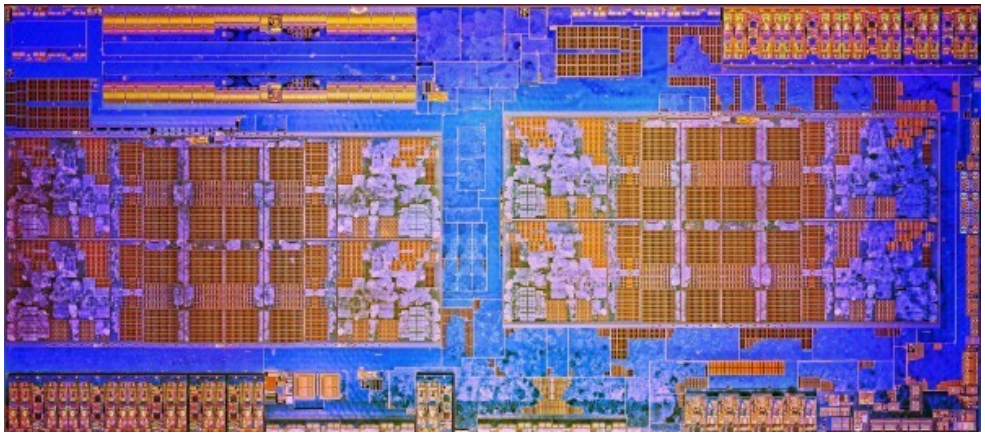
OS as a resource allocator – overview

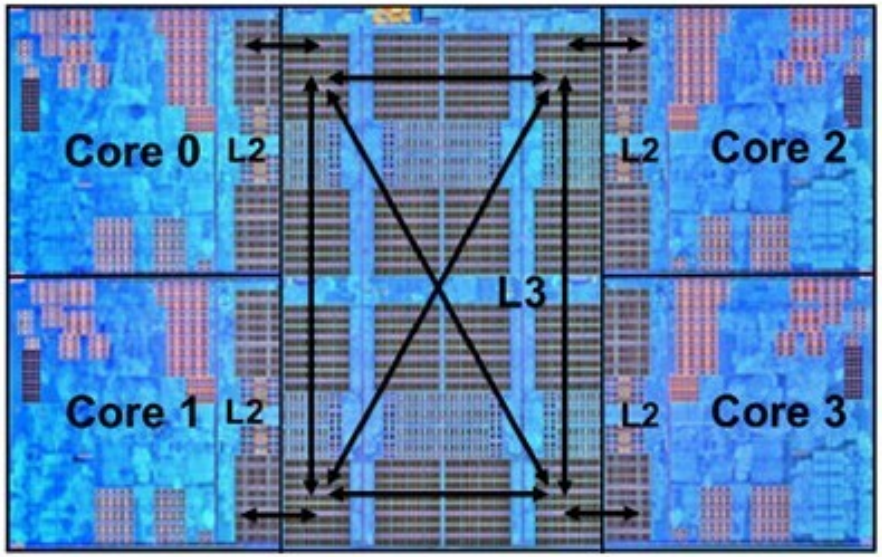
- The Operating System
  - helps solving user's tasks
  - control program
  - **resource allocation**
- Expectations
  - handles multiple tasks
  - reliable, secure
  - **highly utilizes resources**
- Resources
  - processing units (CPU, VGA)
  - system memory
  - storage systems
  - computer peripherals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



AMD Ryzen

- 8 CPU core
  - 2 „Core Complex” (CCX)
  - Infinity Fabric interconnect
- Core complex
  - 4 CPU core (SMP)
  - 8MB L3 cache
- CPU core
  - 2 thread (SMT)
  - 512K L2 cache, 64K+32K L1 cac
- Many aspects for kernel programmers
  - How to handle CCX
  - Multithreading

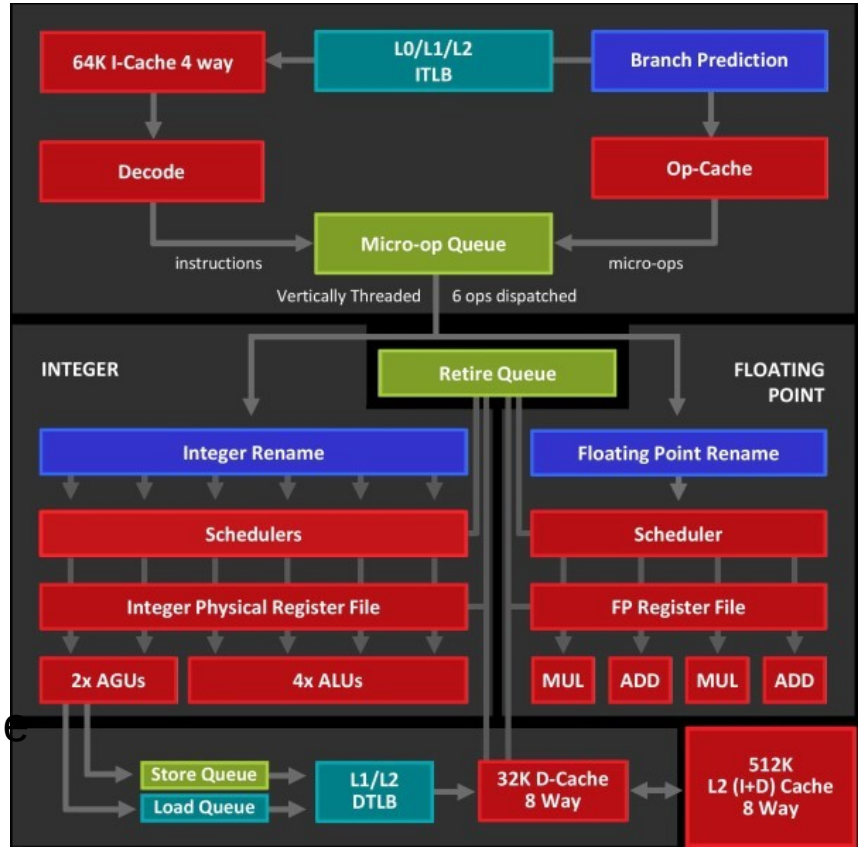




Source: AMD

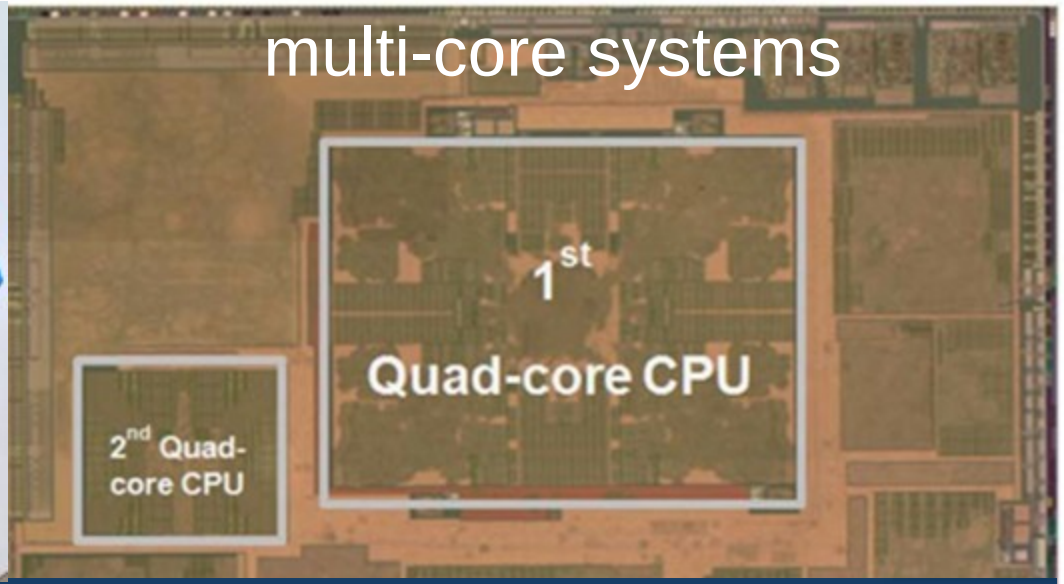
AMD Ryzen

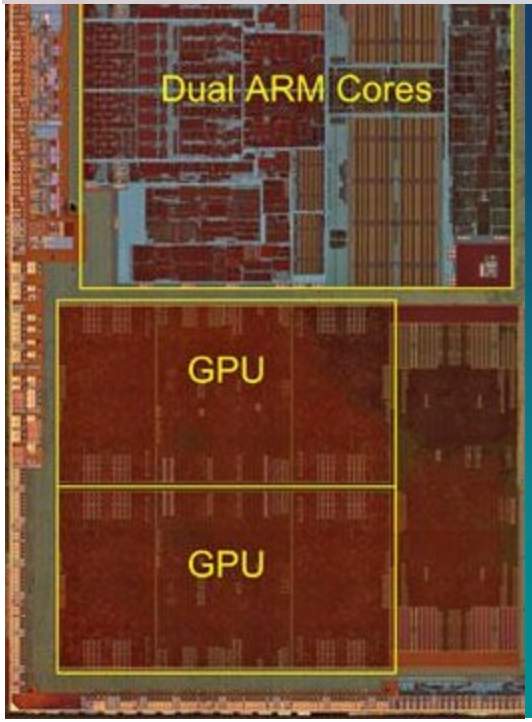
- 8 CPU core
  - 2 „Core Complex” (CCX)
  - Infinity Fabric interconnect
- Core complex
  - 4 CPU core (SMP)
  - 8MB L3 cache
- CPU core
  - 2 thread (SMT)
  - 512K L2 cache, 64K+32K L1 cache
- Many aspects for kernel programmers
  - How to handle CCX
  - Multithreading

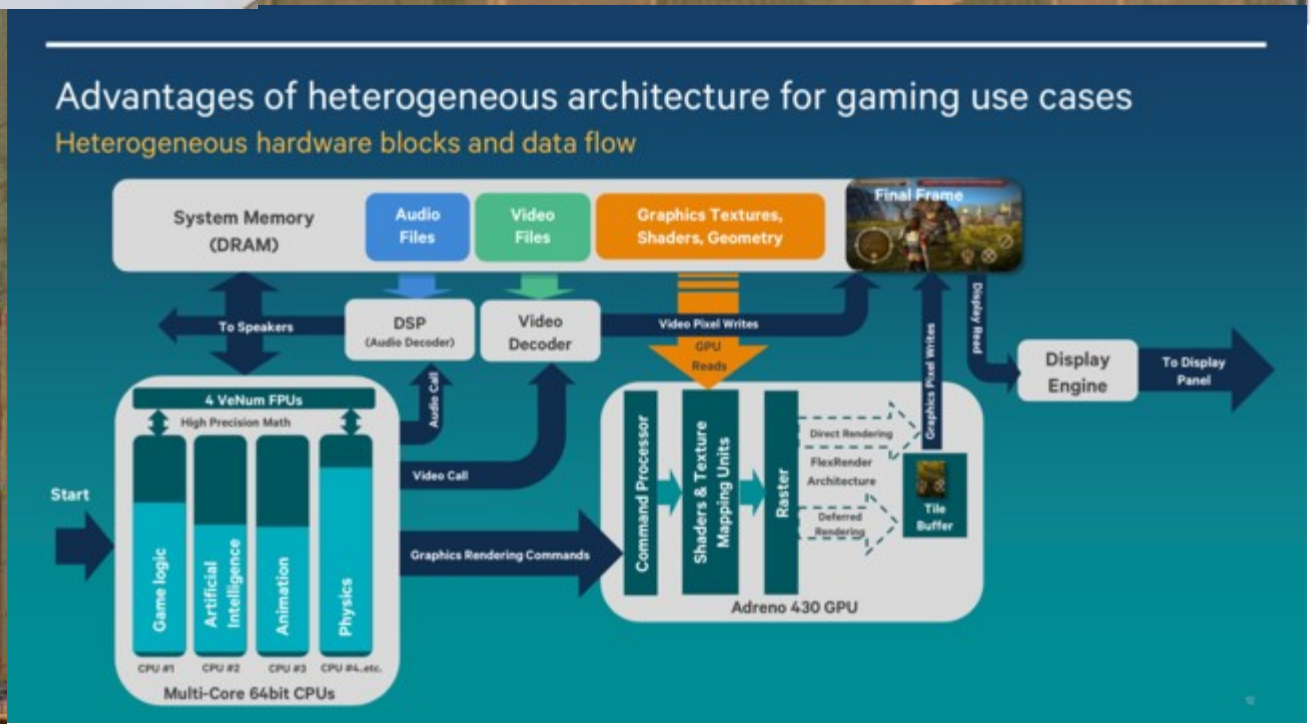


Source: AMD

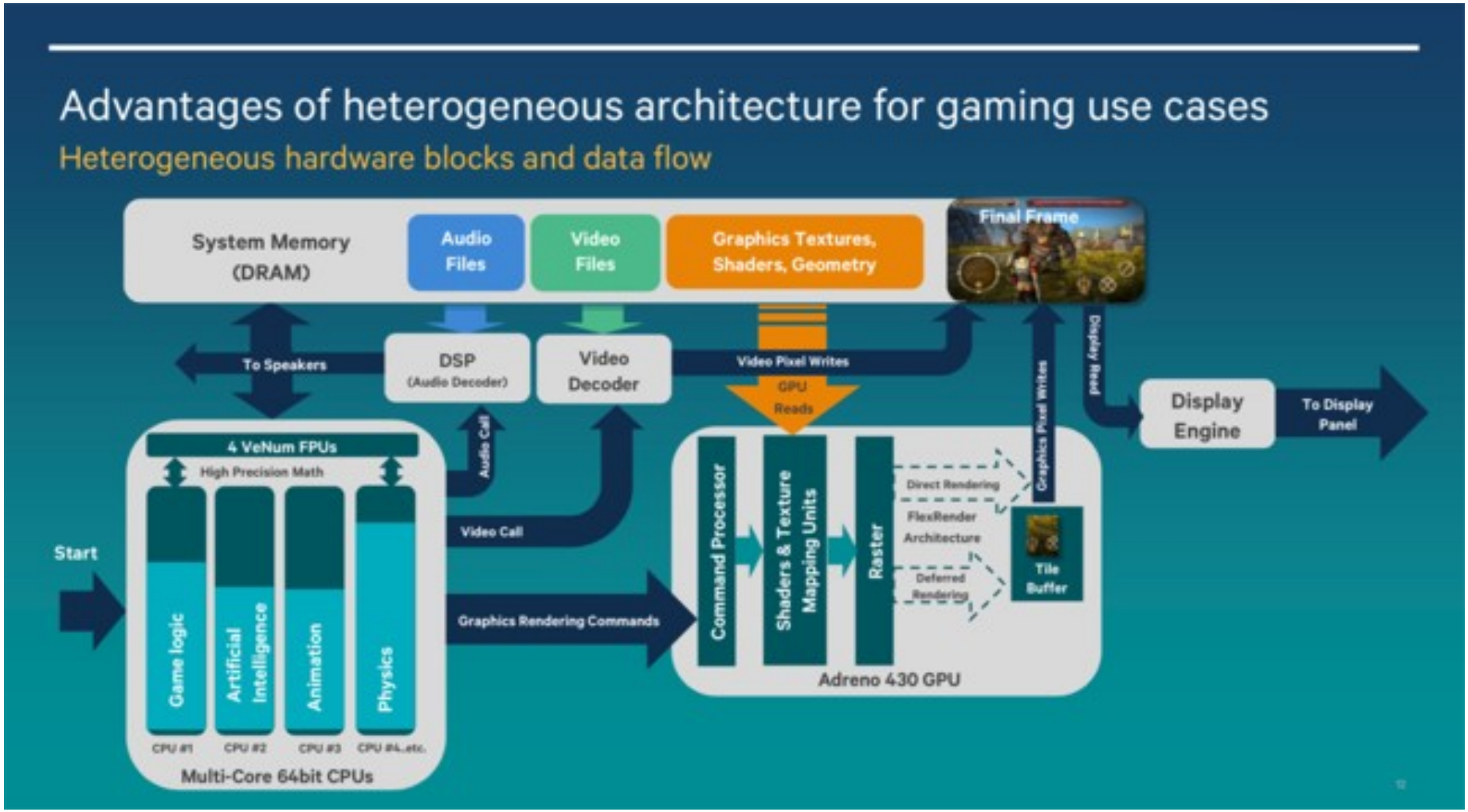
Heterogeneous multi-core systems





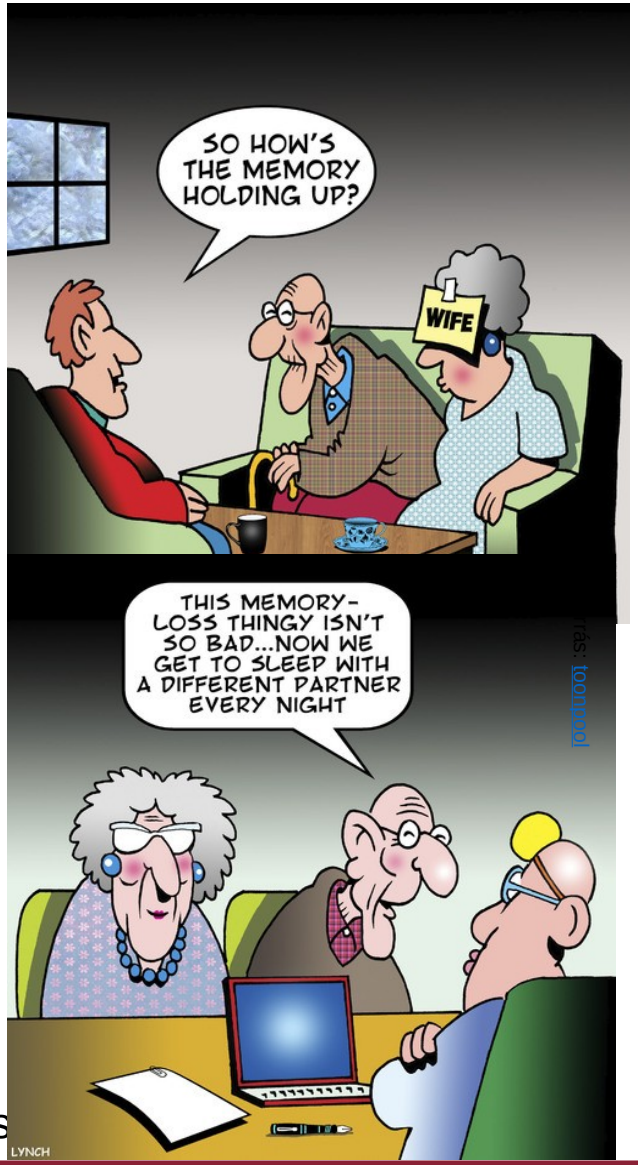


Assigning tasks to execution units



How to handle memory as a resource?

- Allocation
  - resource: physical memory and storage
  - requested by: user tasks and kernel
- Store tasks
  - program
  - data (dynamic and static)
- 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



File systems and storage

User-level access

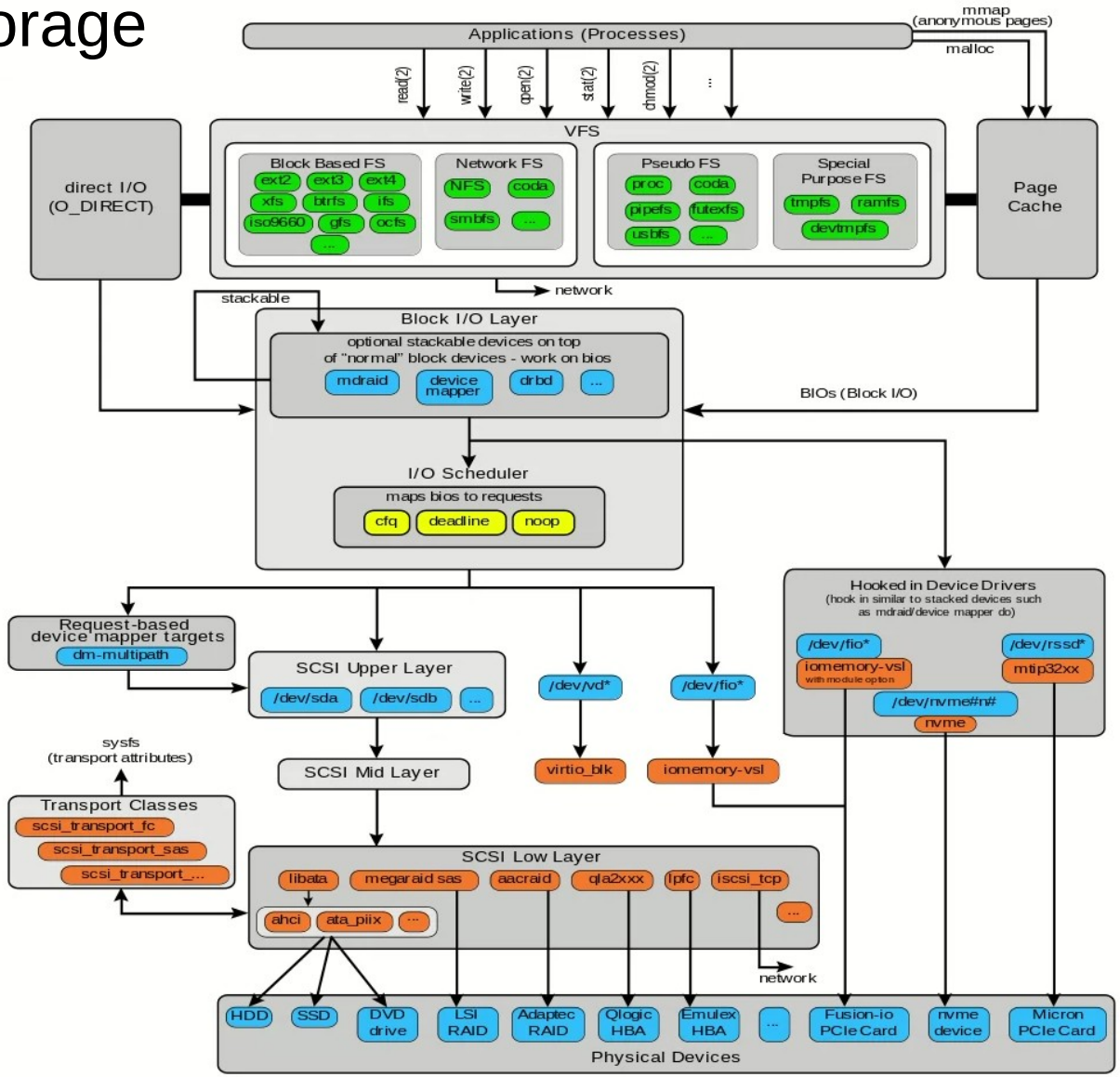
- end user
- administrator
- programmer

Internal operation

- various data organization
- modular structure
- internal interfaces

Storage level

- physical devices (HDD, SSD)
- I/O operation and scheduling
- virtualized storage systems:
  - local (RAID, LVM)
  - network (SAN, NAS)
- distributed storage solutions
  - e.g. RADOS / Ceph

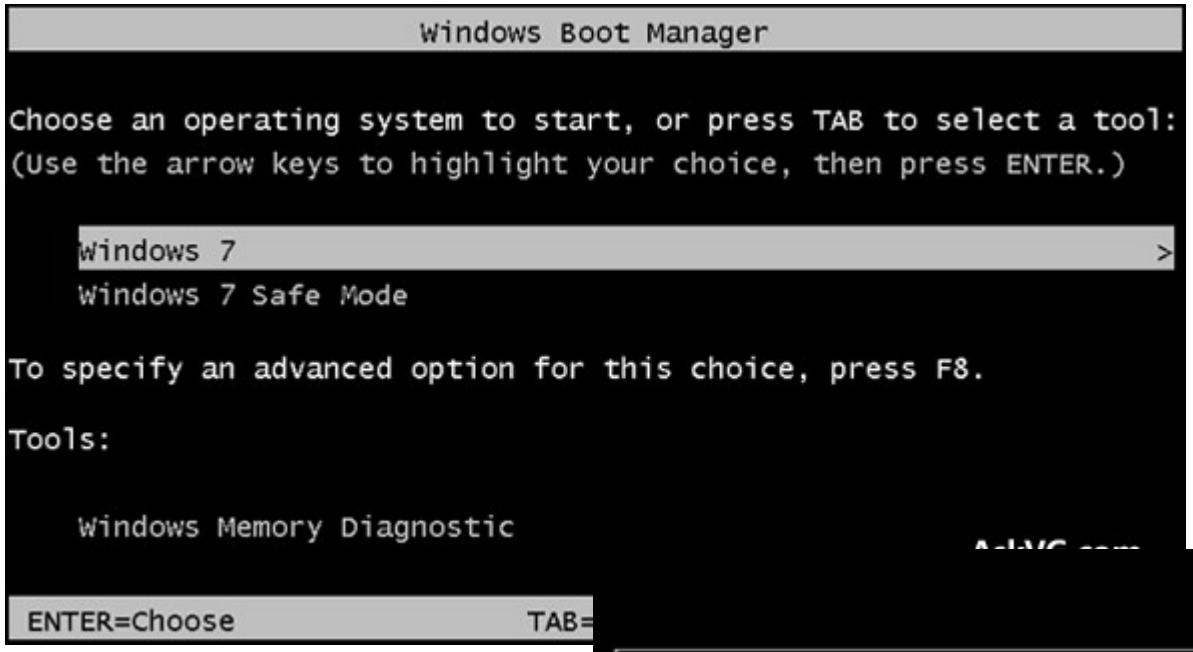


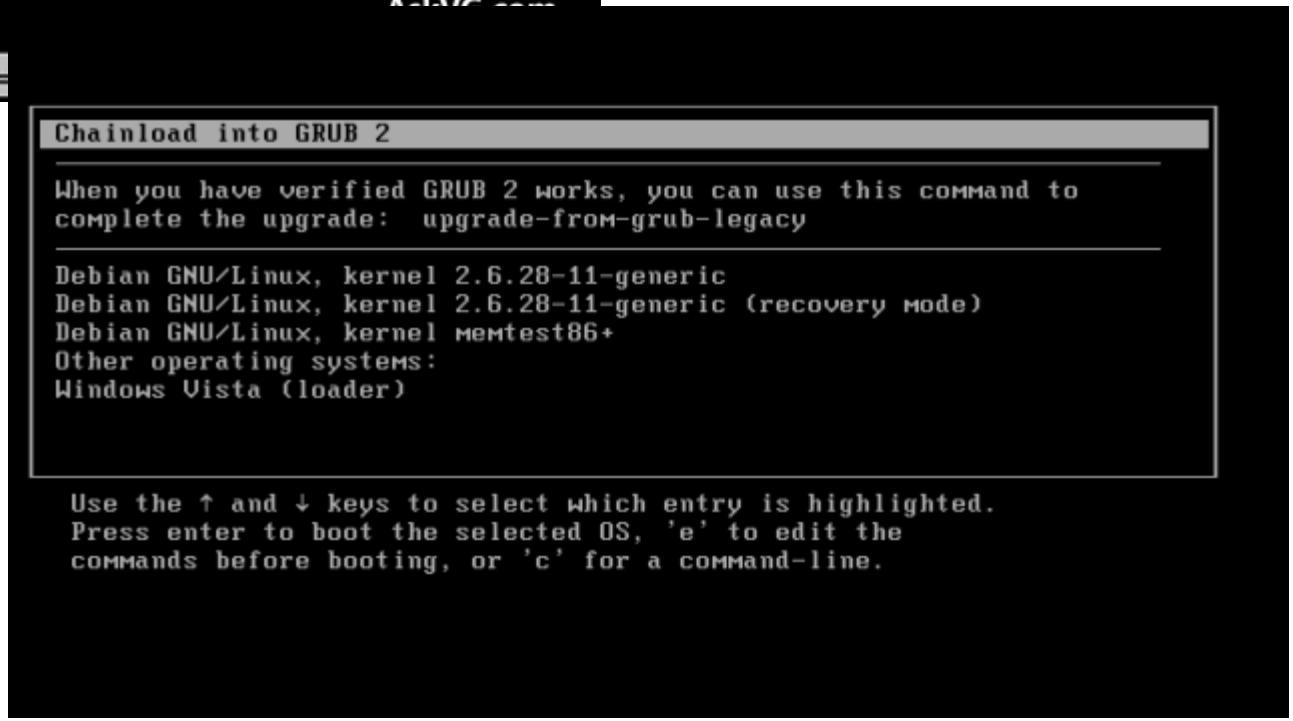
How an Operating System works

The OS boot procedure

The boot procedure

- The system clock signal is up and running
  - The CPU starts it's operation from a fixed ROM loader address
- Level 0: The ROM loader
  - This is a „flashed" system initiator program
  - It is in a fixed memory range, e.g.: ROM, EEPROM, flash, etc.
  - BIOS (in i386), bootROM (in Android)
  - It detects and initiates the first boot device (e.g.: HDD)
  - It loads the bootloader program to the RAM, then starts it
- 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 and loads the next stage
- 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


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)

- The kernel's self loading process
  - Starts it's operation in user mode (e.g. x86 real mode)
  - A small utility uncompresses the kernel's code
  - First initializations: memory manager, stack, interrupts, other descriptors…
  - Getting system parameters from the loader
  - Initializing basic devices (e.g. keyboard, video card, …)
  - Changing into kernel mode (and to 64 bit HW mode is applicable)
- First steps in kernel mode
  - Writing page descriptor tables (before this point the CPU can used a small partition of the memory, now it can use the whole RAM)
  - Setup IT vectors and handers
  - Starting protected memory management
  - Creating process description table and starting the first process (thread)
  - Loading the driver necessary for the system start (initrd: initial ram disk)
  - Architecture dependent init. Functions (drives, DMA, CPU, etc.)
  - Starting the scheduler
  - Setting up the data structures for parallel computing (context changes)
  - Loading and starting the code init
- The first user level program is running (init)

Booting the OS (Windows kernel)

- The VBR finds, loads and runs the Level 2. loader (Bootmgr)
  - It starts in 32-bit user mode
  - 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 – the kernel's loader
  - Runs in 32/64-bit kernel mode
  - Loads the Ntoskrnl.exe and it's dependencies and the device drivers necessary to the system start
  - Forwards the system init. parameters to the Ntoskrnl.exe
- 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 handers
    - 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
    - Initializing: video (progress bar), 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
  - Performs special user mode tasks
    - Using only low-level (core executive) system calls
  - Checking file systems integrity, attempts to repair (Autochk.exe)
  - Sets up the basic environmental variables
  - Sets up the pagefiles
  - Builds the whole registry database (up to this point, only part of it was loaded)
  - Starts the Wininit.exe program (S0InitialCommand)
  - Creates the default sessions (1 typically)
  - Starts the csrss.exe in every session
  - Starts the login manager (Winlogon.exe)
- Wininit.exe – further user mode init. Steps (session 0)
  - E.g. starts the service manager (services.exe)
- Csrss.exe – user mode part of the Win32 subsystem (session 1+)
  - E.g. starts the console manager and others
- 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
  - 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:
      - 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: telinit, init, shutdown, halt, reboot
    - Query the actual state: who -r

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

- The configured system state is set up by init to the corresponding runlevel
  - To do this it checks the files in /etc/rc?.d, where ? is the runlevel
  - The init performs the scripts in these folders
  - These are running in a predefined order (by special naming conventions)
  - These set up the system
    - Mounting drives and file systems
    - Starting services (user login, GUI, webserver, etc…
- The system is ready after init commands

- The system administrator can specify the active system services
  - The services can be managed manually:
    - service <service-name> <start|stop|restart|...>
    - The service names are listed in /etc/init.d/
  - The services assigned to a specific runlevel can be modified
    - ntsysv, tksysv, chkconfig, bum

Alternatives of sysinit

- Problems with init
  - Dependencies between scripts (order)
  - Cannot run in multiple threads -> slower
  - Error handling is not sophisticated

- Systemd (RedHat, CentOS, Ubuntu 15.04+, Arch Linux, Debian etc)
  - Declarative description of the services, precise dependency trees
  - Parallel and scheduled starting of the services (lowers booting time)
  - Detecting and managing errors
  - Changed command set:

```
systemctl <start|stop|restart|...> <service-name>
```

  - Instead of

```
service <service-name> <start|stop|restart|...>
```

  - 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

Today's OSes: modular and monolithic

- Modular design
  - avoids loading the entire kernel into the memory
  - compile time / configurable / runtime

Linux: vmlinux
Windows: ntoskrnl.exe

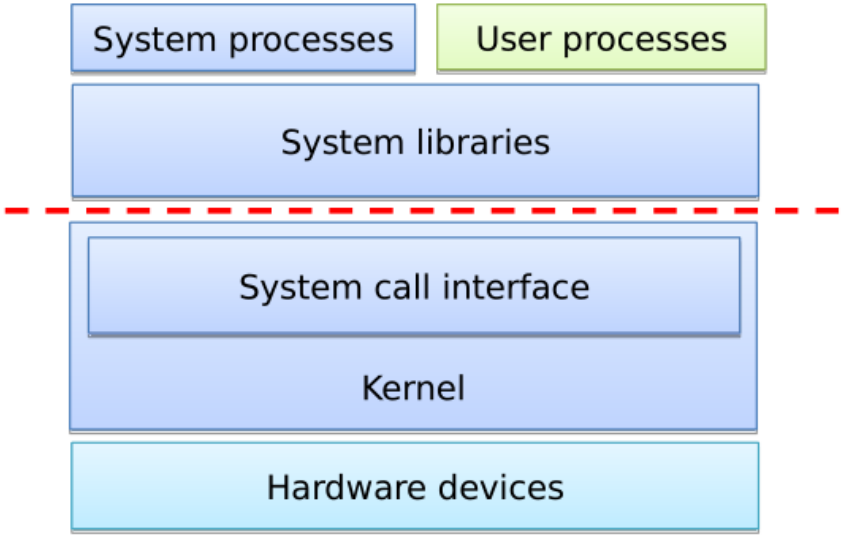
- Distributed
  - the kernel is composed of multiple, separated address spaces
  - it also implements a communication system between them

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**), that repares the arguments
- The libc performs the SYSCALL instruction (generating an IT)
  - the interrupt changes the CPU-mode (to protected)
  - the kernel's SYSCALL interrupt hander function is invoked
- The SYSCALL handler prepares the system call
  - 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 processing returns to the utility function of libc, which sets the return values
- The libc returns from the called function

Virtual system calls

- 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 (Linux)
  - 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()

Advanced topics

What's the problem with kernel structures?

- When did the TV said?
  - Don't turn me off, 220 important updates are pending
  - Needs reboot, because updates are underway – during watching a movie
  - Pay €400 or all of your channels will be encoded
- When did a vehicle control system crashed?
  - Because a dirty CD is inserted
  - One of the components are changed during a service
- Why do we accept such things from computers?

What is the main problem with today's OS kernels?

- A huge code base written by humans
  - 10 – 100 programming errors in every 1000 LOC (source)
  - 20 million LOC .....
  - ~~fault detection, isolation and correction~~ faults and malicious

Source: Linux.com (2016)

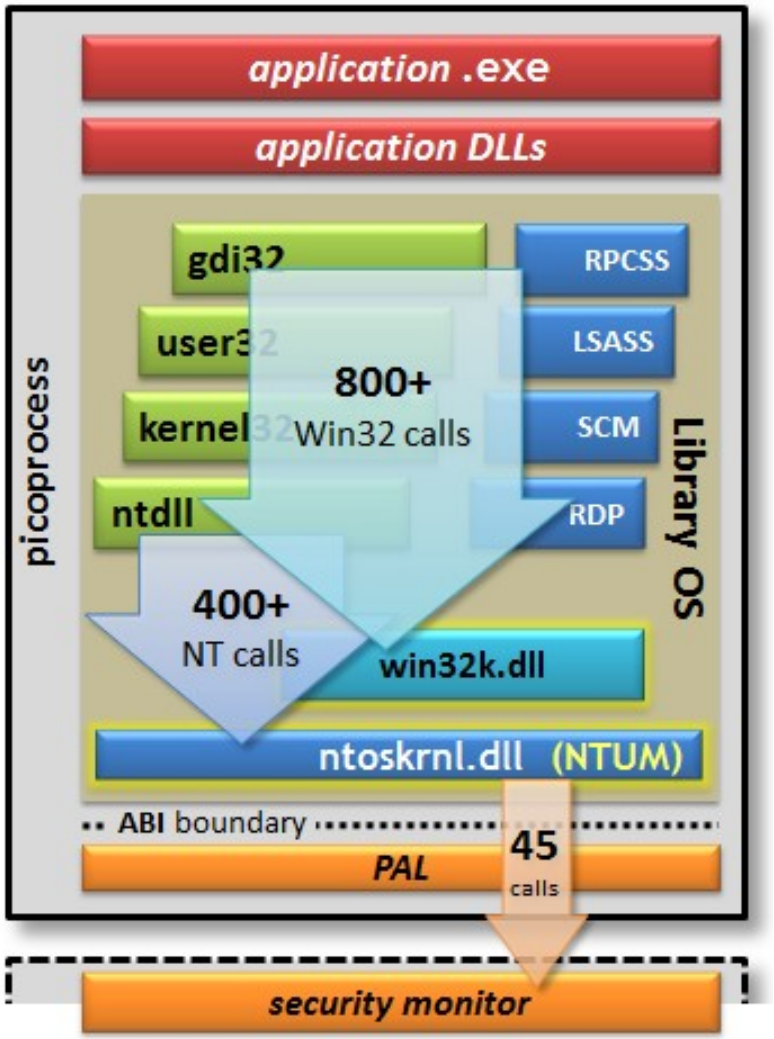
What can we do?

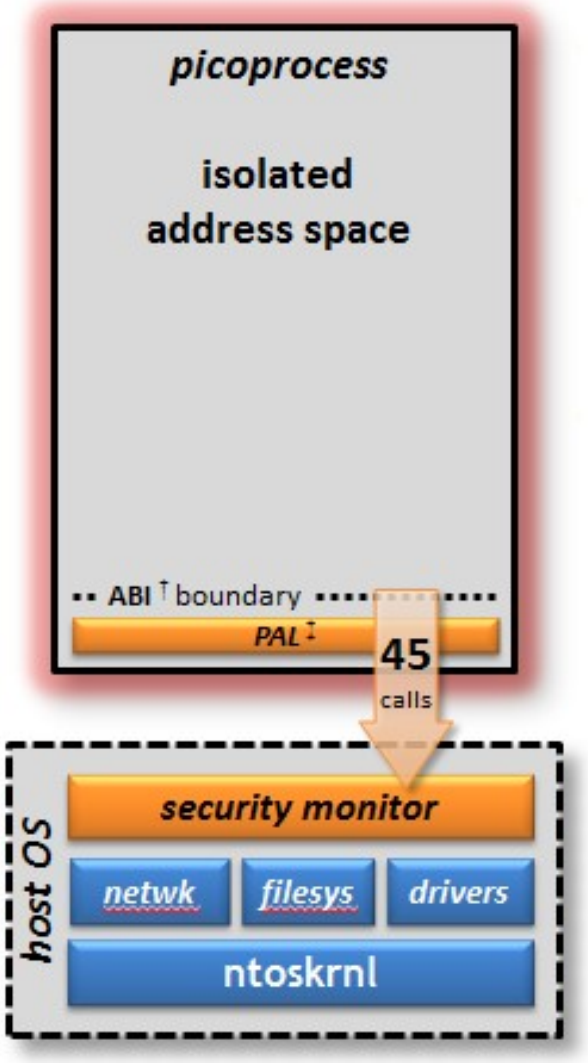
- Kernel sandboxing armored OS
  - create a wrapper around susceptible calls (error detection)
  - provide a kernel component to detect and recover from such errors

- OS/app sandboxing KVM/vmware, Docker, MirageOS, Drawbridge
  - smaller attack surfaces, more control and governance
    - virtualization: one more level of control
    - containers: completely separated subsystems on the same kernel
    - unikernel: mini kernel, application + library OS
      Critical review

- Change the kernel design from monolithic to distributed
  - distributed system
  - only necessary functions are implemented in protected mode

Windows Library OS and pProcess (Drawbridge)





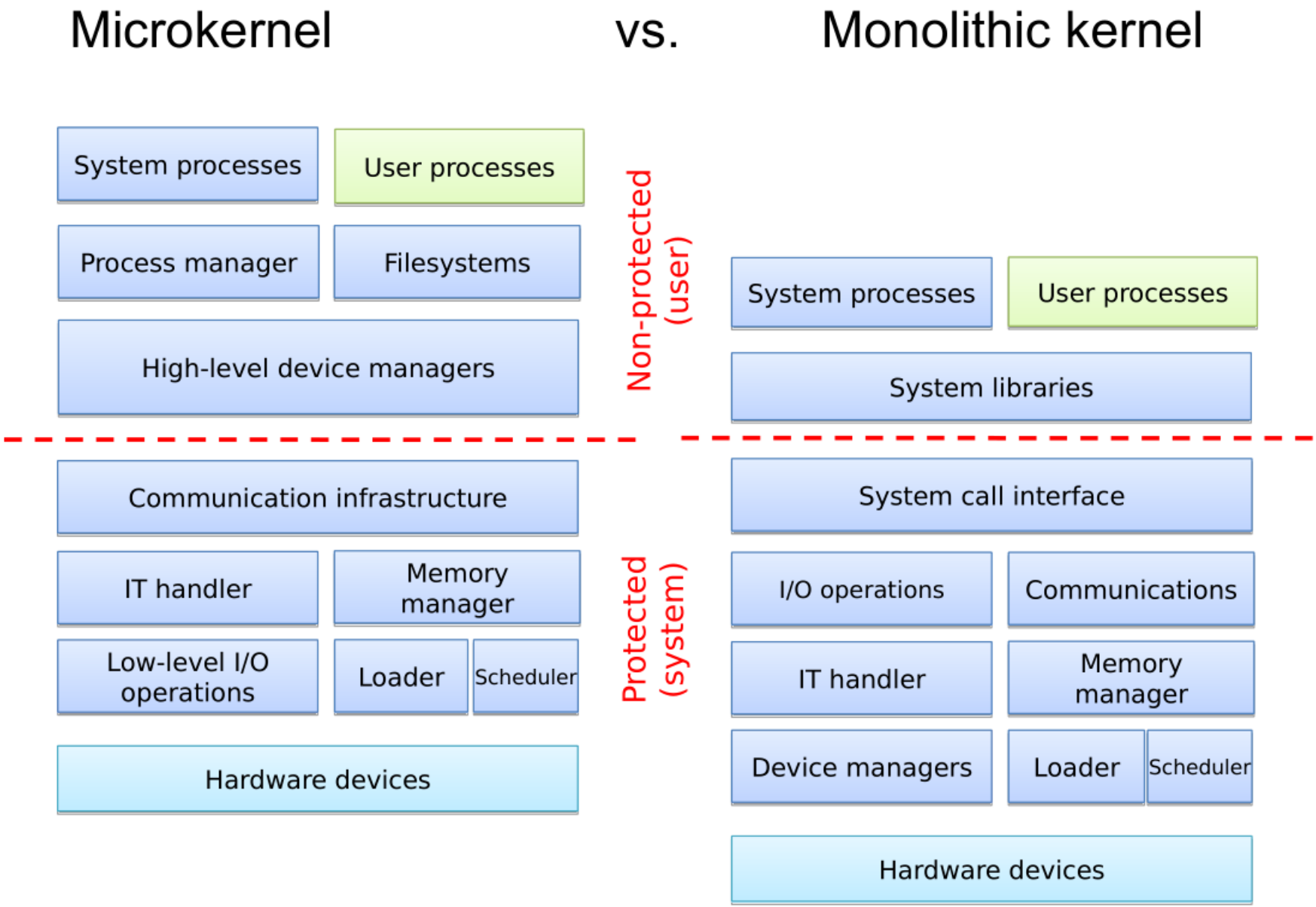
Forrás: microsoft.com

Microkernel

- *The **microkernel** is an OS kernel that contains only a minimally required control program and a communication infrastructure for loosely coupled user-mode tasks.*

- Distributed system
  - a minimal execution environment: memory management and scheduling
  - low-level hardware device management
  - communication infrastructure
  - everything else is in user mode

- Pros and cons (see Tanenbaum-Torvalds debate)
  - flexible
  - more secure and reliable (easier to handle user mode errors)
  - good programming concepts
  - **slooooooow**
  - harder to implement for most programmers


Microkernel
vs.
Monolithic kernel
System processes
User processes
Process manager
Filesystems
High-level device managers
Non-protected (user)
Communication infrastructure
IT handler
Memory manager
Low-level I/O operations
Loader
Scheduler
Hardware devices
Protected (system)
System processes
User processes
System libraries
System call interface
I/O operations
Communications
IT handler
Memory manager
Device managers
Loader
Scheduler
Hardware devices

Microkernel variants

- L4 microkernel: faster and more reliable
  - faster IPC (10-20 x)
  - only 7 API calls
  - 5 – 15 thousand LOC
  - **its operation can be described and verified formally**
- Hybrid kernels
  - mix micro and monolithic kernels
  - OS X XNU (Apple), Mach microkernel + BSD Unix hybrid kernel
  - there are similar L4 microkernel experiments, see

L4 family tree

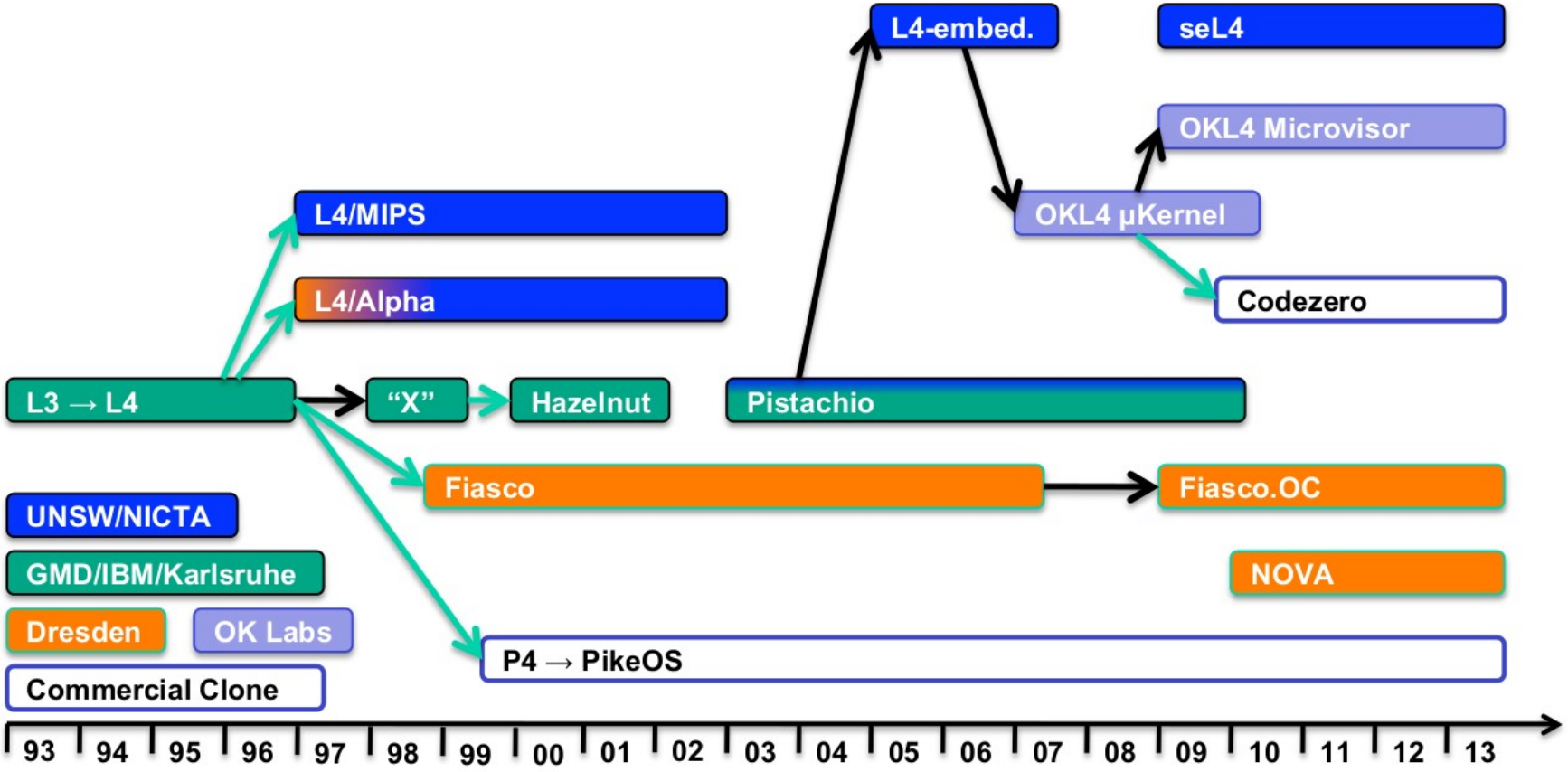


Figure 1: The L4 family tree (simplified). Black arrows indicate code, green arrows ABI inheritance. Box colours indicate origin as per key at the bottom left.

Forrás: Kevin Elphinstone , Gernot Heiser, From L3 to seL4 what have we learnt in 20 years of L4 microkernels?
Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, November 03-06, 2013, Farminton, Pennsylvania

Summary

- The kernel is a complex piece of software
  - layered, modular, monolithic or microkernel design
  - has many issues (a good place for improvement)

- The boot process is also quite complex
  - ROM, RAM, OS and kernel loaders

- The basic operation of an OS
  - system services and programs
  - user programs
  - system calls
  - kernel internals

- Try this at home
  - the boot process
  - system services (turn off and on)
  - init or systemd management
  - system call monitoring