Operating Systems

Introduction & Process Loading

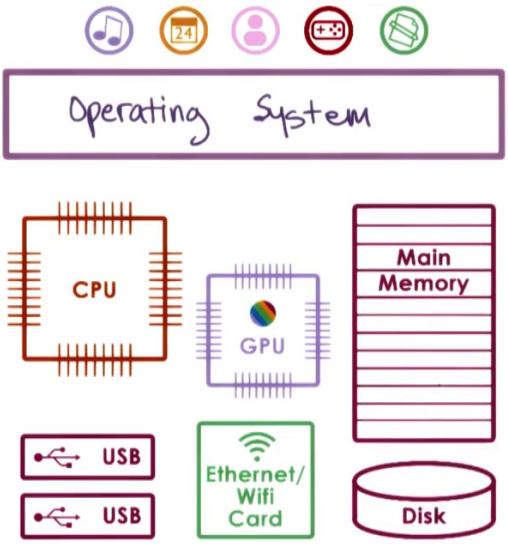
Sven Dziadek

October/November 2022

EPITA

What's an Operating System?

What is an OS?



- hide hardware complexity
- resource management
- provide isolation & protection

Definition

Definition not exact because of various applications:

- **Personal computers**:
  Desktop, Laptop, Smartphones, ...
- **Servers**:
  Web server, File server, Firewall, Network router, ...
- **Embedded Systems**:
  Home, Cars, Toasters, Internet of Things, ...
- **Industrial control systems**:
  Factories, Power Plants, Ships, ...

Definition (2)

Definition not exact because of broad spectrum of goals:

- Embedded systems with very optimized operating systems ($\leq$ megabytes)

  $\vdots$

- Graphical operating systems with lots of features ($\geq$ gigabytes)

Definition Kernel

Definition (Operating System Kernel)

The process always resident in memory/always running.

The remainder of the operating system is called **system programs**.

OS as a Control Program

Privilege Levels

Hardware supports **security** mechanisms.

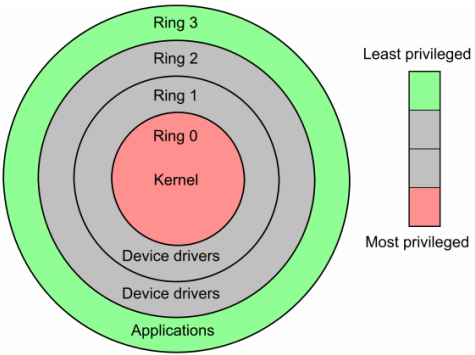
Kernel Mode (Ring 0)

Everything is allowed

User Mode (Ring 3)

Instructions and memory space is restricted

Other rings mostly not used.



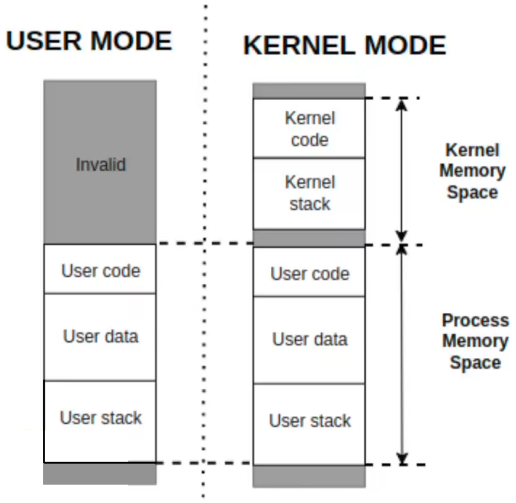
User Mode - Kernel Mode

User Mode

- No direct access to hardware (some instruction forbidden)
- Memory access only to user space

Benefits:

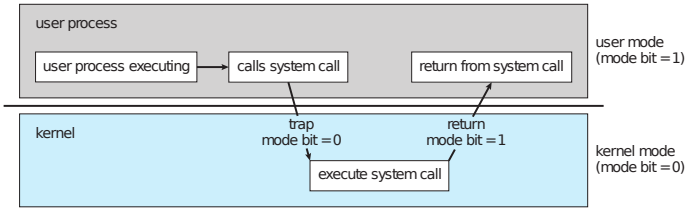
- Protection (against programming errors)
- Security (against attackers)



System Calls

Endpoints of the kernel API

- User processes call them to interact with the kernel
- Kernel checks parameters validity
- Kernel performs restricted actions for the user process
- Eventually, kernel returns to user mode



Linux System Call Table

%rax	System call	%rdi	%rsi	%rdx	...
0	sys_read	unsigned int fd	char *buf	size_t count	
1	sys_write	unsigned int fd	const char *buf	size_t count	
2	sys_open	const char *filename	int flags	int mode	
3	sys_close	unsigned int fd			
4	sys_stat	const char *filename	struct stat *statbuf		
5	sys_fstat	unsigned int fd	struct stat *statbuf		
⋮	⋮				

Interrupts

Exceptions/Traps (Software interrupts)

Synchronous events (also called *fault* or *abort*)

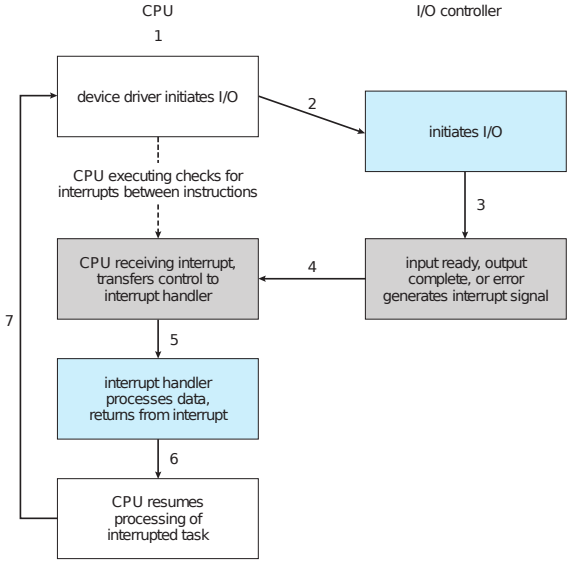
like "divide-by-zero"-error, page faults, illegal instructions, etc.

(Asynchronous/Hardware) Interrupts

Emitted by hardware, timer, I/O devices, etc.

Caution: can happen at any time

Interrupt Handler



- Interrupt vector stores addresses of **interrupt handlers**
- Some interrupts can be **masked**

Timer Interrupt

Timer regularly raises an interrupt
(e.g., 250 interrupts per second)

Timer interrupt is necessary for the OS
to reclaim control from an errant/malicious user process

Signals

Signals

- Inter-Process Communication (IPC)
- Communication from kernel to user process

Example

- SIGINT (`Ctrl-C`), SIGKILL, SIGQUIT (`Ctrl-\`),
- SIGTSTP (`Ctrl-Z`), SIGCONT,
- SIGSEGV, ...

User can define *signal handler*.

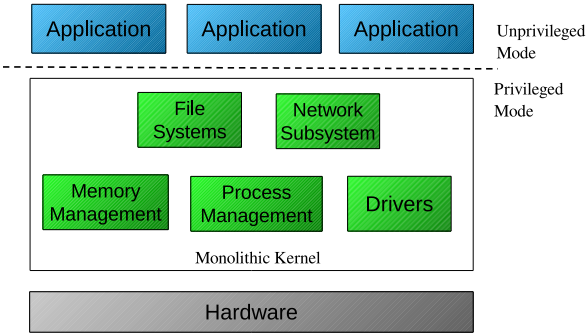
Demo

Linux command `time` measures time spend in kernel and user mode

`/proc/interrupts` stores all interrupts since last system boot

OS Kernel Types

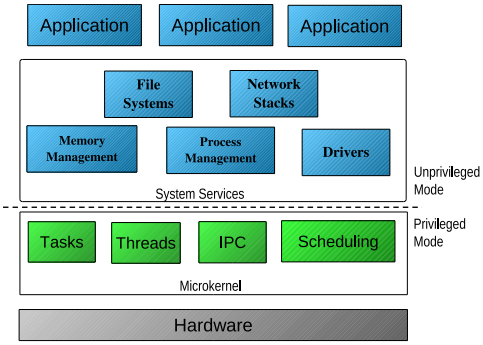
Monolithic Kernel



Kernel services and drivers are part of the kernel

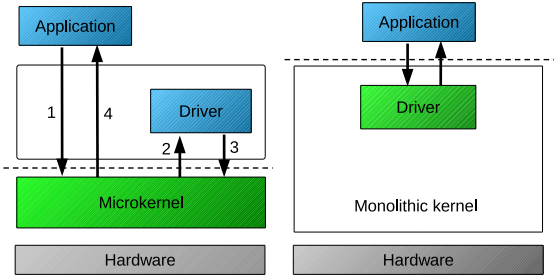
- faster
  - less messaging
  - less context switching
- but
  - bugs effect whole kernel
  - adding services needs to modify whole kernel

Micro Kernel



system services run in different memory spaces

- advantage:
  - adding service is easy
  - errors in services do not affect whole system
- but performance loss due to
  - more messages
  - more context switching



Other Kernel Types

Modular Kernel

Modules can be added and removed during runtime

- adding services becomes easy (even for monolithic kernels)

Example: Linux

Hybrid Kernel

Combination between micro- and monolithic kernel

Example: Windows, Mac OS X

System Architecture

Definition (Process)

Program in execution

Remember: Process is active – program is passive

Process consists of

- **program counter**
- data and code in memory
- access to file
- access to I/O devices

Scheduling

Scheduler manages CPU time

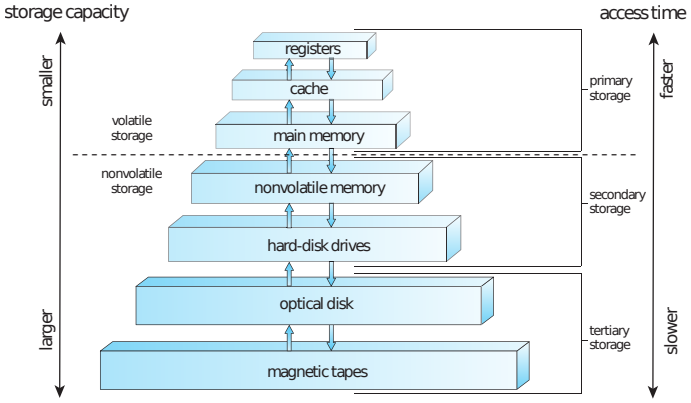
Multitasking

CPU executes multiple processes concurrently

Multitasking leads to fast response times

- switches occur frequently
- processes waiting for I/O yield the CPU for other processes
- concurrency creates impression that processes run in parallel

Storage



Three types:

- primary storage (can be used directly by CPU)
- secondary storage (is permanent – has to be kept synchronous)
- tertiary storage (only for backups)

Interfaces

ABI/API

API: Application Programming Interface

Specifications needed for programming, source code based

ABI: Application Binary Interface

Specifications of *calling conventions* and file formats, binary based

Calling conventions:

- where are parameters and return values placed? (registers, stack, memory, etc.)
- how to prepare function call and restore afterwards

POSIX

Portable Operating System Interface

Family of standards for maintaining compatibility between OSs

- C API
- Command Line Interface utilities
- Shell language
- Environment variables
- Program exit status
- Regular expressions
- Directory structure
- Filenames

Mostly compatible with all major OSs

For Windows see:
Windows Subsystem for Linux

ELF

Shebang

#!

`#!/path/to/interpreter` at the first line of text files

tells the kernel/shell what interpreter to use to run as script

Can be seen as a "magic number" to define file formats.

ELF

Executable and Linkable Format (ELF)

Common file format for executable files

Similar for object files and shared libraries

Not portable across different ABI and CPU architecture, etc.

Widely used on Linux, OpenBSD, Solaris, Android, ...

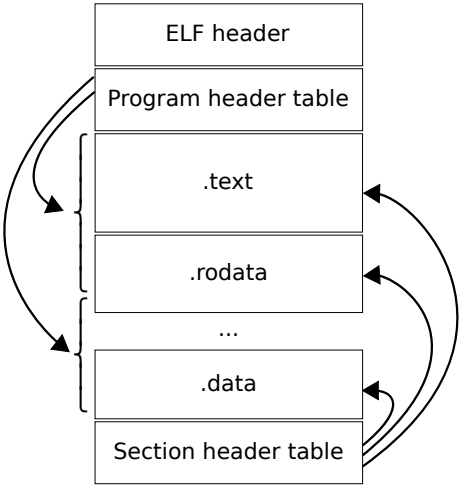
ELF Format

ELF header

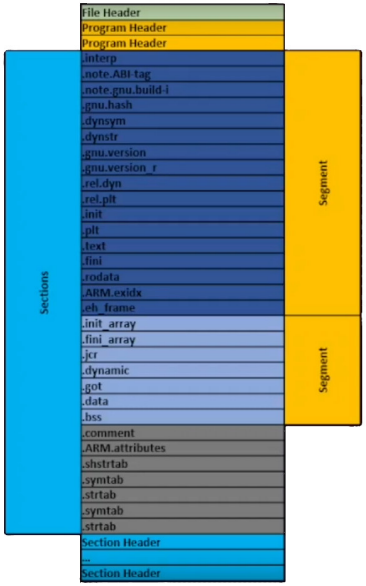
"magic number" + format configurations

File data

- Program header table
- Section header table
- Data



ELF Details



Segment

Used for **running** the program
(e.g. with correct access flags)

Each program header points to one **segment**.

Section

Used for **linking**

Each section header points to one **section**.

Demo

```
include <stdio.h>

int main()
{
   printf("Hello world!\n");
   return 0;
}
```

Use `xxd` to show ELF header.

`readelf` analyses ELF executable

ELF Header

Offset	Size	Purpose
0x00	4	0x7F454c46 or 0x7F"ELF" (in ASCII)
0x04	1	1 or 2 for 32- or 64-bit, respectively
0x05	1	1 or 2 for little or big endianness, respectively
0x06	1	1 for current version of ELF
0x07	1	OS ABI, e.g., 0x00 for System V
0x08	8	unused
0x10	2	object file type, e.g., 0x03 for shared object file
0x12	2	architecture, e.g., 0x3E for AMD x86-64
⋮		

View `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

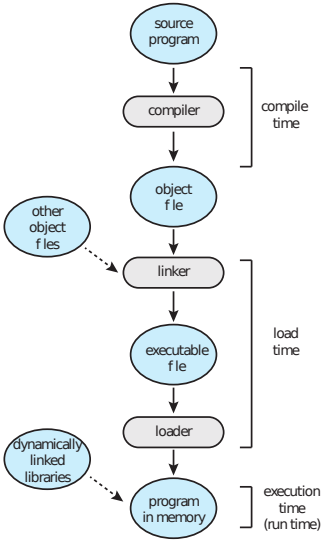
ELF Sections

Typical sections in ELF executables include:

- `.text` executable instructions (code)
- `.bss` uninitialized read-write data (will be filled with zeros)
- `.rodata` initialized read-only data
- `.data` initialized read-write data
- ⋮

Process Loading

Linker & Loader



Addresses in executables can be fixed at

- **compile time** (absolute code)
  - relocation needs recompilation
- **load time** (relocatable code)
  - all addresses are shifted by the starting address
- **execution time** (position-independent code)
  - addresses in same segments are relative
  - uses **GOT/PLT** for addresses in other segments:
    - *global offset table* (.got)
    - *procedure linkage table* (.plt)
  - allows *dynamically linked libraries*
  - allows *address layout randomization*

Dynamic Linker & C runtime

Dynamic linker/loader

`.interp` section, typically `ld.so`

- searches shared libraries
- load library (if not already loaded)
- relocate code (change **GOT/PLT**)
- return control to original process

crt0 (C runtime 0)

Object file to call `main` and `exit` afterwards

- automatically linked into every executable

Demo

Compile `hello_world` from above with `-static` and compare with

- `ls -lh`
- `ldd`
- run with `strace -C`