CS Basics
Operating Systems

Fall Term 2024/2025

Emmanuel Benoist | BFH-TI

Operating Systems

- **Last Week**
- **Operating Systems**
- **Program**
- **Virtualization CPU**
  - The Process
- **Virtualization : Memory**
- **Persistence**
- **Conclusion**

Last Week

Representing Float Numbers

IEEE 754 has three parts

- sign bit
- Exponent
- Mantisse

First: convert the number in binary

- $10,5 = 0b\,1010,1$

Then : Write the number in “binary-scientific” notation.

- $10,5 = 0b\,1,0101 \times 2^3$
- mantisse = 0b 1,0101 , exponent = 3, sign=+

Use IEEE 754 notation

- Sign : + is 0; - is 1
- Mantisse without the first 1 is 0b 010 1000 0000 0000 0000 0000
- Expenent +127 is 130=0b 1000 0010
- Representation 0b 0**100 0001 0** *010 1000 0000 0000 0000 0000*
  0x41280000

Operating Systems

Operating Systems

Which OS do you know?

What is an OS?

What functionalities does it offer?

Do we need one? Why?

Program

What is a Program?

Basically *just instructions* (billions!), in *memory*, executed step-by-step by the *processor (CPU)*.[1]

1. The processor fetches an instruction from memory
2. It then decodes it to know what to do...
3. ...and executes it afterwards
4. After this, the next instruction is fetched and executed (back to step 1)

Eventually, when all instructions have been processed, the program ends. (does it?)

[1] This is a simplification of what we call the *Von Neumann model of computing*

Program Instructions

```
1125:    55                push   rbp
1126:    48 89 e5          mov    rbp,rsp
1129:    b8 00 00 00 00    mov    eax,0x0
112e:    5d                pop    rbp
112f:    c3                ret
```

Resource Management

Now that we know what a program is, and how it is executed…

- Who decides *which* program is to be run?
- How can we have *multiple* programs running at the same time?
- If so – which one is the first?
- Which program may access a *resource* (e.g. the screen or a USB stick)?
- How is it ensured that all programs play nicely with each other?

The answer to these questions (and more of them) lies in the operating system…

Virtualization

One of the key concepts in operating systems is *virtualization*:

- A program only sees a *representation* of a (physical) resource, e.g. the CPU or memory
- Often, this representation is more general, powerful and easy-to-use
- Each running program gets its own set of virtualized resources
- The program does not notice (in general) that the accessed resources are shared

The OS then acts as a *resource manager*, ensuring availability and fair distribution.

Virtualization CPU

Virtualization: CPU

When virtualizing the CPU, each program has the impression to run on its own CPU:

- Enables running multiple programs at the “same” time
- Provides the illusion of having an infinite number of CPUs
- Instructions from different programs do not interfere with each other

Virtualization: Abstract Idea

The main concept behind virtualization is to provide access to a single resource multiple times “at once” (and probably for different parties).

In the physical world, this is difficult. However, in computing, we can resort to a trick:

- Each resource is available only once
- Full access is given for a resource…
- …but only during a *restricted* time frame
- This is known as *time sharing*
- Everyone requiring the resource accesses it in turns

How is this possible?

The Process

Definition

A *process*[2] is (informally) a *running instance* of a *program*.

- A program is a set of instructions (and possibly data) stored *on disk*
- Each program in general exists only once on a computer
- A *process* is an instantiation of a program
- There can be $0 \ldots N$ processes (from different programs) running at the same time

[2] Sometimes also called a *task*

Process as an Abstraction

A process is an *abstraction* of a running program: A representation of everything relevant being read or written while the program is running (its *machine state*). Most importantly:

- The *address space*: The whole memory belonging to the process
- *CPU registers*, especially
  - The *program counter (PC)*, also called *instruction pointer (IP)*
  - *Stack pointer (SP)* and corresponding *frame pointer (FP)*
- I/O information (e.g. open files)

Process API

In order to work with processes, an OS must provide an API, which supports in some form:

- The *creation* of processes, as well as their *destruction*
- Functionality for *waiting* and *controlling*
- Access process *status* information

All modern operating systems provide such an API, although they all look a bit different.

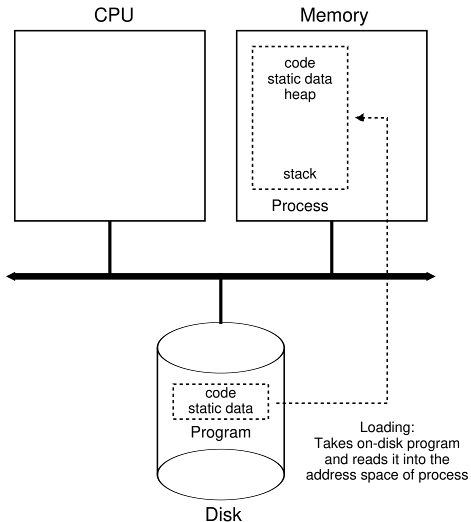
Process Creation

When creating a process, the OS performs a series of steps. In a nutshell:

- Load the program into memory:[3] executable *code* and *static data* (i.e. initialized variables)
    - This includes code from *shared libraries*
- Allocate the *stack*; often initialized with *program arguments* and *environment*
- Maybe preallocate some *heap* memory
- Initialize I/O
    - E.g. UNIX: open `stdin`, `stdout` and `stderr` *file descriptors*
- Run the `main()` function

[3]This is often done *lazily*, i.e. only when selected parts are required.

Process Creation Illustrated



Process States

A process can be in different states, simplified:

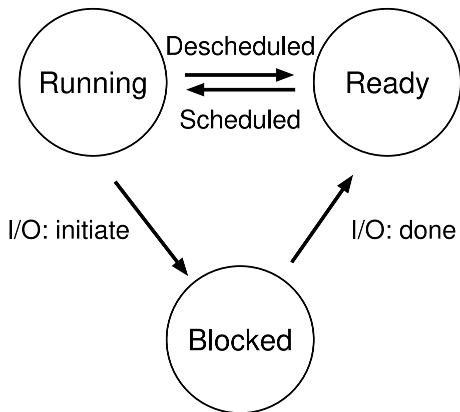
Running The process is running on a CPU, i.e. *it is executing instructions*

Ready The process is *ready to be run*. For some reasons, the OS has chosen not to run it at this given moment

Blocked (waiting) The process has performed some operation which makes it wait for an event to happen. Often, this is caused by an I/O request: reading data from disk or waiting for user input.

We say that a process is being *scheduled* if it moves from ready to running state; *descheduled* if it moves from running to ready.

Process States Illustrated



Process State Trace: CPU only

This is an example of two processes running, using only the CPU (no I/O):

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ done
5	-	Running	
6	-	Running	
7	-	Running	
8	-	Running	$Process_1$ done

Process State Trace: CPU and IO

This is an example with two processes; at some time, one of them is blocked due to an I/O request.

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ done
9	Running	-	
10	Running	-	$Process_0$ done

Virtualization : Memory

Virtualization: Memory

With memory virtualization, each program has the impression to access its own *address space*.

- The same (virtual) memory addresses can be used by different programs
- Size and layout of memory must not necessarily be the same as for the real, physical memory of the machine
- Memory of different programs is isolated from each other

What is a process in memory

When a process runs it needs to use memory:

- Memory for the program itself (the code that should be run)
- Statical variables that remain for the life of the process (placed within the program or nearby it)
- The stack where function parameters and functions local variables are used
- The heap for dynamic memory allocation.

The three areas are far away from each other to grow without risk.
Address space is large, for examples:

- Address of an instruction in a program 0x401000 (approx. $4 * 10^6$)
- Address of an element inside the stack 0x7fffffffdd68 (approx. $1.4 * 10^{14}$)
- Address of an element on the heap 0x5032309d

One big RAM for everybody

Many processes run simultaneously

- Each process has one's own address space
- Each process could use all the memory (we do not want to set the memory size in advance).
- The address space of each process is almost empty. Big empty spaces between static / heap / stack spaces.

Address space is large but real memory usage is small

- Each process need to have the illusion to use the entire memory
- But in reality it needs a small part of memory

Solution Paging

Virtual Memory is organized in pages

- Each process sees virtual memory
- There is a system (in the OS) to transform virtual memory into physical memory.
- Virtual memory is organized in pages, that are stored in physical memory frames.

Advantages

- Each process can use any address (it believes to be alone)
- A process can not see the other processes memory
- Security : Processes can not read or change memory of other processes.

Persistence

Persistence

The last major topic for Operating Systems is *persistence*:

- Data stored in memory is volatile and lost when the system is powered off
- I/O devices such as hard drives and solid-state drives (SSDs) are used for permanent storage
- Access to these devices is managed using directories and files in a *file system (FS)*
- In general, I/O devices are not virtualized but shared between applications

Conclusion

OS Design Goals

Having introduced virtualization, concurrency and persistence, we will now highlight some goals in *operating systems design*:

Abstraction The system should be easy and convenient to use; introducing abstractions enables this.

Performance Overhead, both in space (memory/disk) and time (CPU), needs to be kept small

Protection The OS, as well as individual programs require protection from each other. One of the main principles used here is *isolation*.

Reliability When the operating system fails, all applications fail as well. An OS must be capable of running 24h/day.

Security Protection against malicious programs, attackers. Can be considered as an extension of protection.

Efficiency Applies to various areas; with mobile and internet of things (IoT) devices, *energy efficiency* is critical.