Transcript

Introduction to Operating Systems VO

Version v1.0

Daniel Kocher

Daniel.Kocher@stud.sbg.ac.at

Salzburg, March 2, 2015

Myth Busting

Myth: I prefer programming language X because it makes it more convenient to implement my application.

Answer: Wrong!

The fundamental problem of programming is to establish functional correctness and adequate performance. Different languages provide different tools of automating the process of establishing correctness and performance. The language should be chosen based on that insight.

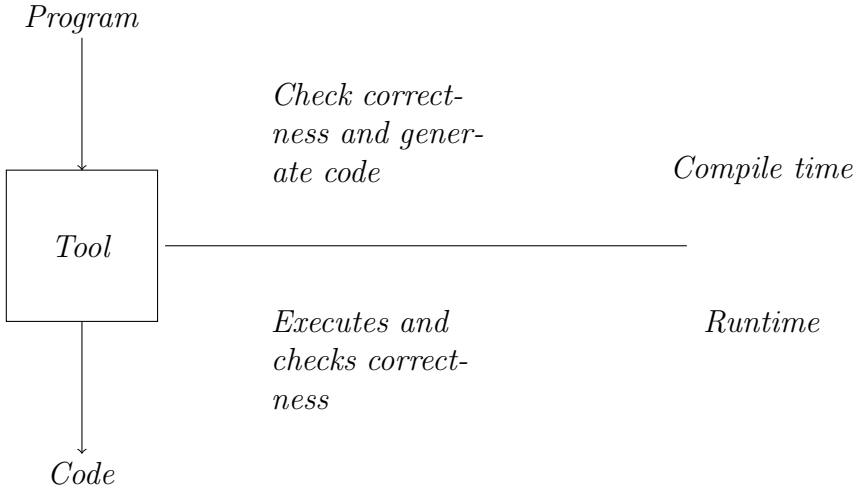


Figure 1: Process of establishing correctness.

Ultimate goal:
A Compiler checking correctness in such a sense that no exception is thrown while executing in any case. But this is infeasible (mathematical proof possible).

Hardware Exception: Interrupts, a mechanism to stop memory accesses in hardware.

Myth: I like garbage-collected languages like Java because they free me from memory management.

Answer: Wrong!

A garbage collector (GC) provides safe deallocation of unneeded memory but the programmer still needs to say what is unneeded, otherwise the system will run out of memory (memory leak).

General runtime complexity of a garbage collector: the size of the heap.

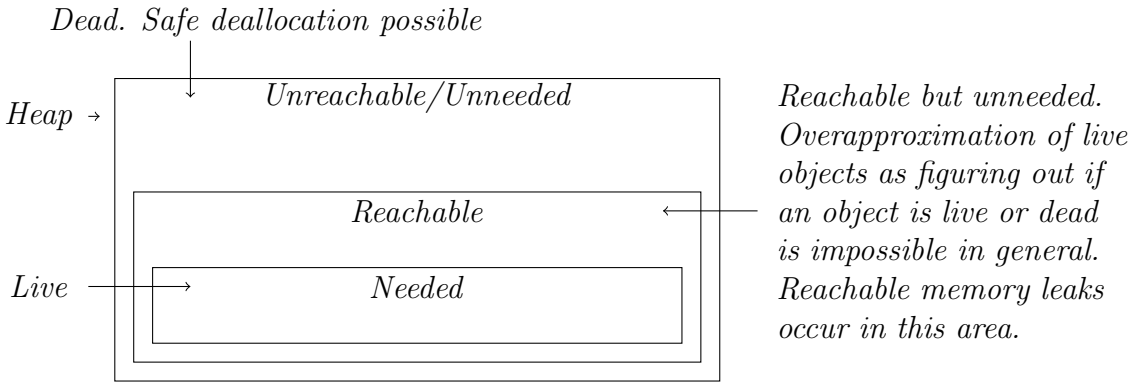


Figure 2: Unreachable, reachable and needed set of a program.

Multicore

Amdahl's Law: P represents program parallelism on N cores

- $S(N) = N$ if $P = 100\%$. This is ideal multicore scalability.
- In general: $S(N) = \frac{1}{(1-P)\cdot\frac{P}{N}}$

Sequential vs. Parallelized Code

The bottleneck of parallelized is the memory bus. It is limiting execution even if a problem can be perfectly parallelized without any side effects. Thus a parallelization factor of 100% is infeasible. Any shared resource at any level of an architecture creates a limitation.

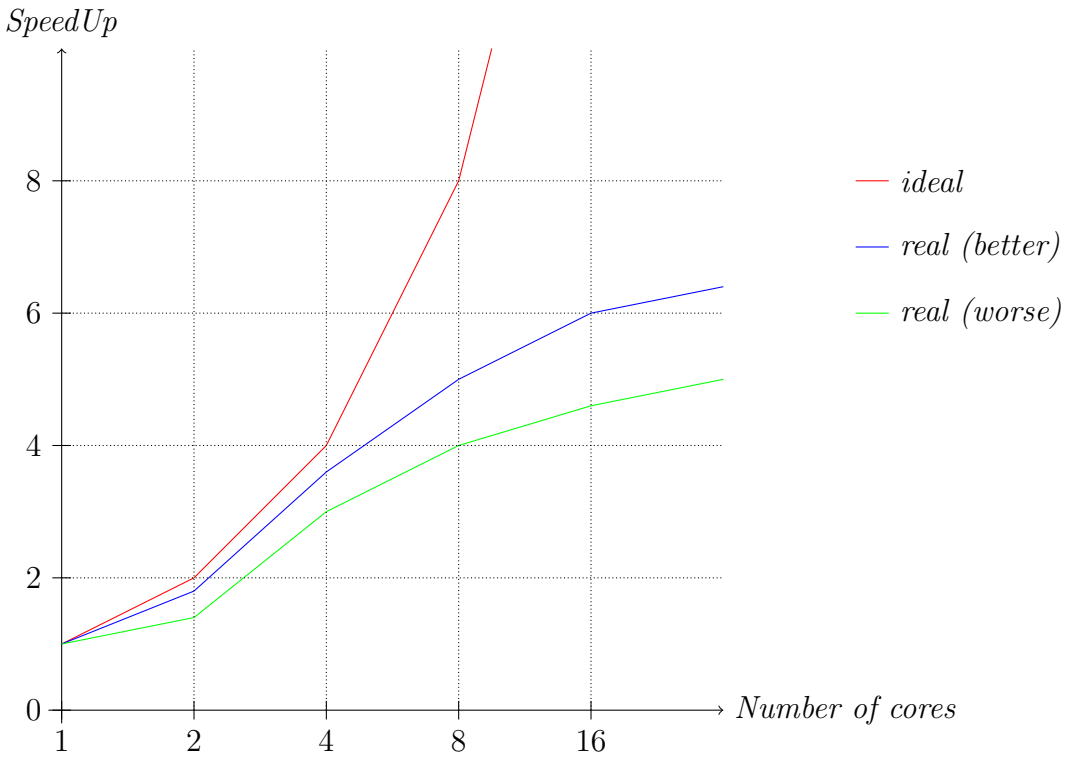


Figure 3: Amdahl's law plot.

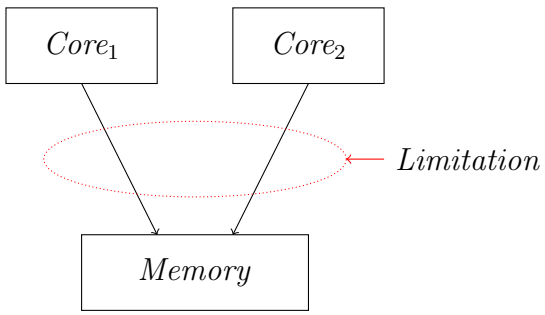


Figure 4: Bottleneck of parallelized code.

Architecture

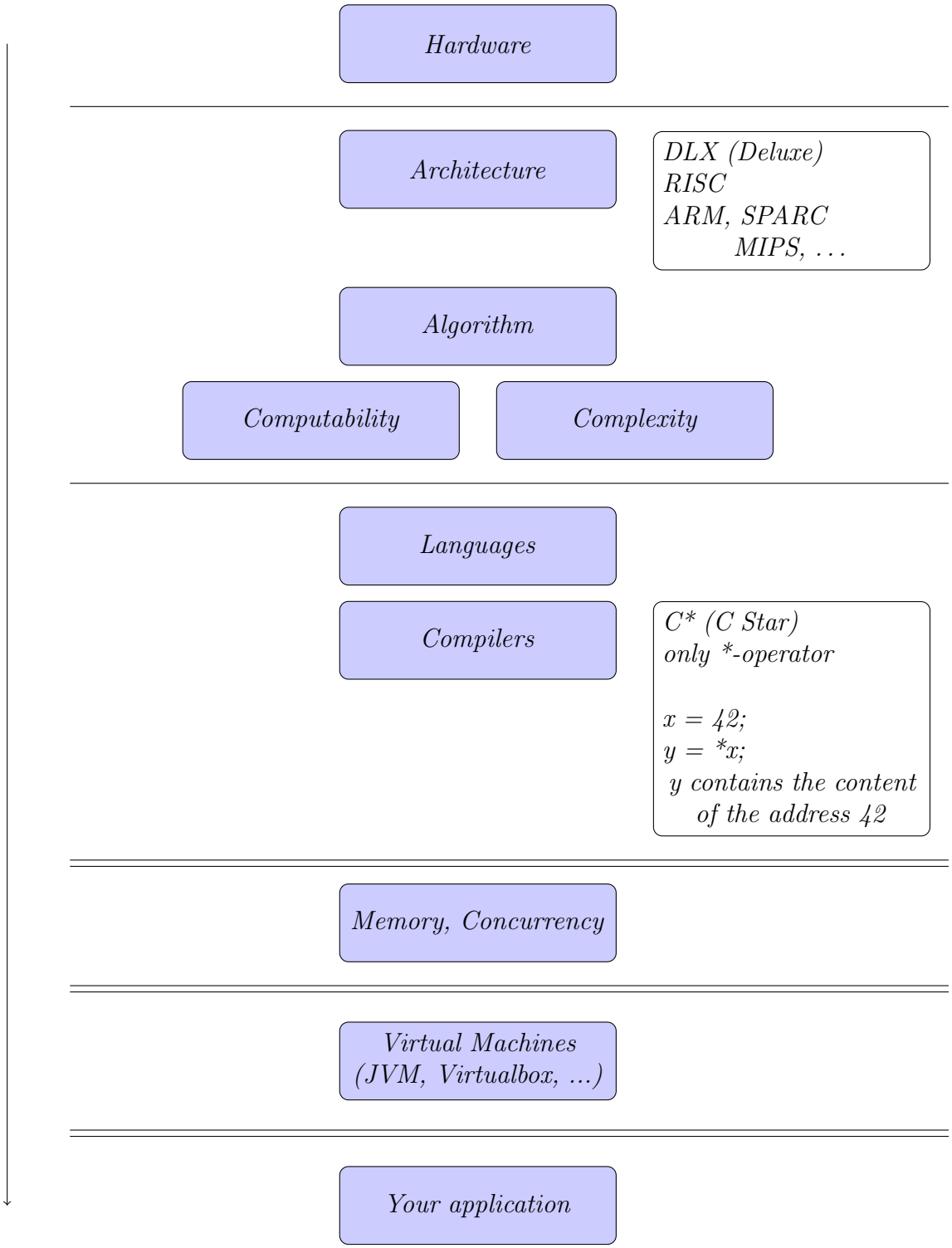


Figure 5: Architecture hierarchy (top-down.

Von Neumann Architecture

Introduced in 1945. This is a stored program computer: data = program. The Fetch-Decode-Execute cycle (see Figure 7) modifies the state of the machine.

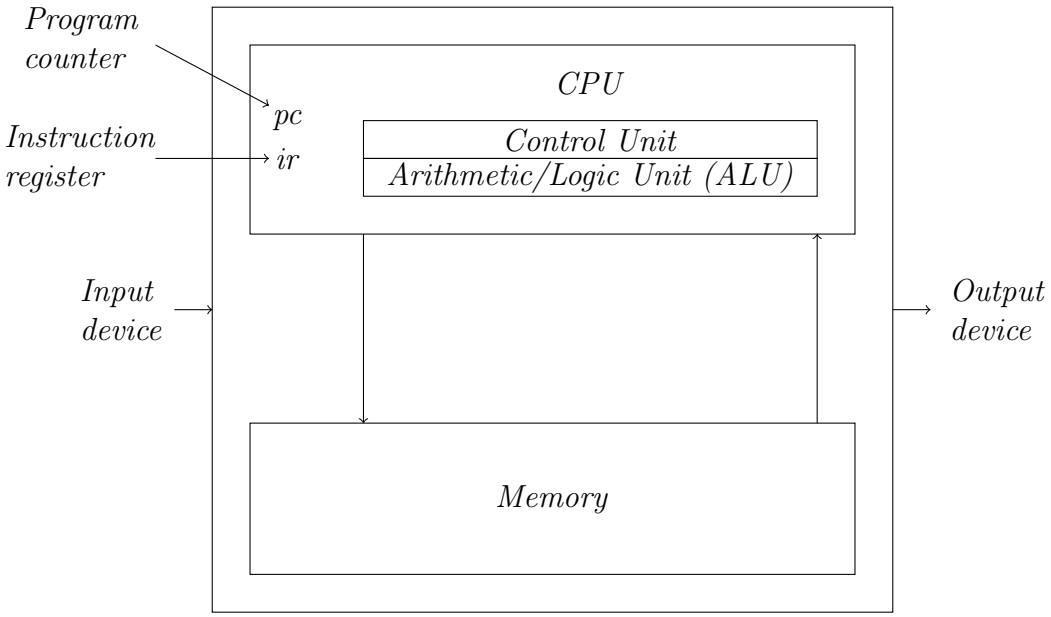


Figure 6: Von Neumann Architecture.

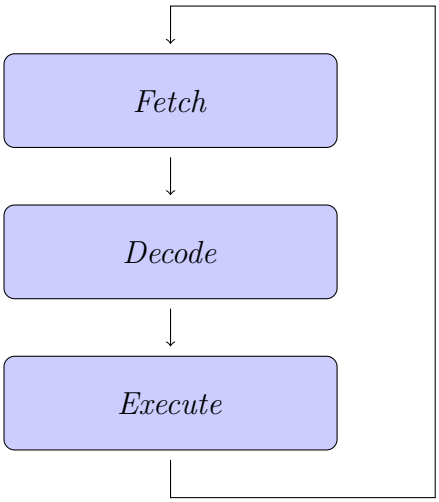


Figure 7: Fetch-Decode-Execute cycle.

DLX Machine

- Control unit:
  Instruction register (ir) and program counter (pc).
- Arithmetic unit:
  32x 32-bit registers; `reg[0]`, `reg[1]`, ..., `reg[31]`. `reg[0]` always contains the value `0` and `reg[31]` is the link register. Both are reserved by convention and must not be used for any other purpose.
- Memory:
  n 32-bit words, byte-addressed (see Figure 8), word-aligned;
  `mem[0]`, ..., `mem[n - 1]`.

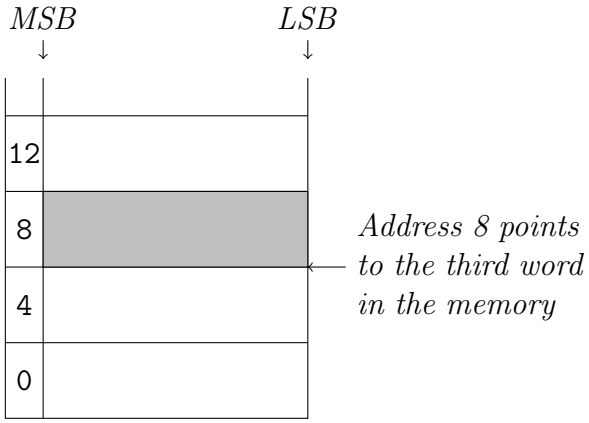


Figure 8: Visualization of a byte-addressed memory of 32-bit words.

Syntax Formats

General syntax of an instruction: `op a, b, c`.

F1

The length of `a` and `b` allow to address all 32 registers. The two's complement is used here because the implementation of arithmetics is easier (in contrast to the one's complement).

op	$0 \leq a \leq 31$	$0 \leq b \leq 31$	$-2^{15} \leq c \leq 2^{15} - 1$
6 bits	*5 bits*	*5 bits*	*16 bits sign-extended to 32 bits*
2^6 *ops*	$2^5 - 1$	$2^5 - 1$	

F2

E.g. `R1 = R2 + R3`

op	$0 \leq a \leq 31$	$0 \leq b \leq 31$	*unused*	$0 \leq c \leq 31$
6 bits	*5 bits*	*5 bits*	*11 bits*	*5 bits*
2^6 *ops*	$2^5 - 1$	$2^5 - 1$		$2^5 - 1$

F3

Absolute addressing in memory.

op	$0 \leq c \leq 2^{26} - 1$
6 bits	*26 bits*
2^6 *ops*	$2^{26} - 1$

Von Neumann Bottleneck: Memory read/write operations limit the performance.

Register Instructions

Arithmetic Instructions

F1

Instruction	Semantics	Additional information
`ADDI a, b, c`	`reg[a]:=reg[b]+c;`	Add immediate
	`pc:=pc+4;`	`c` is data (a constant)
`SUBI a, b, c`	`reg[a]:=reg[b]-c;`	Substract immediate
	`pc:=pc+4;`	`c` is data (a constant)
`MULI a, b, c`	`reg[a]:=reg[b]*c;`	Multiply immediate
	`pc:=pc+4;`	`c` is data (a constant)
`DIVI a, b, c`	`reg[a]:=reg[b]/c;`	Divide immediate
	`pc:=pc+4;`	`c` is data (a constant)
`MODI a, b, c`	`reg[a]:=reg[b]%c;`	Modulo immediate
	`pc:=pc+4;`	`c` is data (a constant)
`CMPI a, b, c`	`reg[a]:=reg[b]-c;`	Compare immediate
	`pc:=pc+4;`	`c` is data (a constant)
		`reg[a]==0 if reg[b]==c`
		`reg[a]>=0 if reg[b]>=c`
		`reg[a]>0 if reg[b]>c`
		`reg[a]=<0 if reg[b]=<c`
		`reg[a]<0 if reg[b]<c`
		`reg[a]!=0 if reg[b]!=c`

F2

Instruction	Semantics	Additional information
`ADD a, b, c`	`reg[a]:=reg[b]+reg[c];`	Add
	`pc:=pc+4;`	Register addressing `reg[c]`
`SUB a, b, c`	`reg[a]:=reg[b]-reg[c];`	Substract
	`pc:=pc+4;`	Register addressing `reg[c]`
`MUL a, b, c`	`reg[a]:=reg[b]*reg[c];`	Multiply
	`pc:=pc+4;`	Register addressing `reg[c]`
`DIV a, b, c`	`reg[a]:=reg[b]/reg[c];`	Divide
	`pc:=pc+4;`	Register addressing `reg[c]`
`MOD a, b, c`	`reg[a]:=reg[b]%reg[c];`	Modulo
	`pc:=pc+4;`	Register addressing `reg[c]`
`CMP a, b, c`	`reg[a]:=reg[b]-reg[c];`	Compare
	`pc:=pc+4;`	Register addressing `reg[c]`
		`reg[a]==0 if reg[b]==reg[c]`
		`reg[a]>=0 if reg[b]>=reg[c]`
		`reg[a]>0 if reg[b]>reg[c]`
		`reg[a]=<0 if reg[b]=<reg[c]`
		`reg[a]<0 if reg[b]<reg[c]`
		`reg[a]!=0 if reg[b]!=reg[c]`

Register Allocation Problem

Registers have to be used in order to execute an instruction. There are 29 registers (in theory) to use. In practice, at least in this course, less registers can be used because some registers are reserved for special purposes (stack pointer, heap pointer, globals pointer, frame pointer, ...). It must be guaranteed that these will not be used and furthermore already allocated registers must not be used for an instruction.

Examples:

C Code	Instructions	Additional information

`1 + 2;`	`ADDI 1, 0, 1`
	`ADDI 2, 0, 2`
	`ADD 1, 1, 2`
		First, naive solution

`1 + 2;`	`ADDI 1, 0, 1` `ADDI 1, 1, 2`	Second, better solution
`1 + 2;`	`ADDI 1, 0, 3`	Third, best solution Constant folding

`if(1 < 2)`	`ADDI 1, 0, 1`
	`ADDI 2, 0, 2`
	`CMP 1, 0, 2`
	`BGE 1, 0, <loc>`
		`<loc>` represents the line of code in the assembly code the CPU continues with

Memory Instructions

F1

Instruction	Semantics	Additional information
`LDW a, b, c`	`reg[a]:=mem[(reg[b]+c)/4];` `pc:=pc+4;`	Load word (from memory) Register-relative addressing
`STW a, b, c`	`mem[(reg[b]+c)/4]:=reg[a];` `pc:=pc+4;`	Store word (into memory) Register-relative addressing

`POP a, b, c`	`reg[a]:=mem[reg[b]/4];`
	`reg[b]:=reg[b]+c;`
	`pc:=pc+4;`
		Pop (from stack)
		`c`: size of the popped chunk
		`reg[b]`: contains stack pointer

`PSH a, b, c`	`reg[b]:=reg[b]-c;`
	`mem[reg[b]/4]:=reg[a];`
	`pc:=pc+4;`
		Push (onto stack)
		`c`: size of the popped chunk
		`reg[b]`: contains stack pointer

The stack grows from high to low addresses (top-down). In the `POP` and `PSH` instructions, `reg[b]:=reg[b]+c` and `reg[b]:=reg[b]-c` represent the actual removal of the element from the stack, respectively. In case of the C* language `c` will always be `4` because only a single type (`int`) exists in this particular language. In general `c` is the amount of data you want to remove from the stack.

Without register-relative addressing, which is a concrete form of indirect addressing, you could only talk about as many memory cells as your program uses. The only way to get the same expressivity as with register-relative addressing is to write self-modifying code.

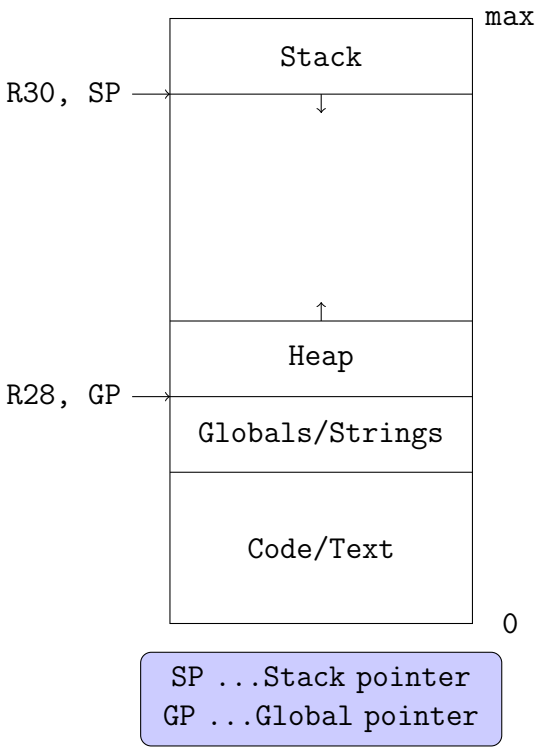


Figure 9: Memory layout when executing a program

Examples:

C Code	Instructions	Additional information
`int x;`	`LDW 1, 28, -4`	`R1 := x`
`x = x + 1;`	`ADDI 2, 0, 1`	`R2 := 1`
	`ADD 1, 1, 2`	`R1 := R1 + R2`
	`STW 1, 28, -4`	`mem[x] = R1`
`*x + 1;`	`LDW 1, 28, -4`	`R1 := x`
	`LDW 1, 1, 0`	`R1 := mem[R1 + 0]`
	`ADDI 2, 0, 1`	`R2 := 1`
	`ADD 1, 1, 2`	`R1 := R1 + R2`
`*(x + 1);`	`LDW 1, 28, -4`	`R1 := x`
	`ADDI 2, 0, 4`	`R2 := 0 + 4`, (4 because it's an address)
	`ADD 1, 1, 2`	`R1 := R1 + R2`
	`LDW 1, 1, 0`	`R1 := mem[R1 + 0]`
		`*(x + i)` is equivalent to `x[i]`

When dealing with `*(x + 1)`, the type of `x` has to be known as well as its size in the memory. The size of `x` in the memory determines the offset which is added to the address of `x`. In case of the C* language this value is always `4` because there is only one type and no support for composed types (`struct`). Nevertheless, the size of the type needs to be added in general.

Formula to compute `*(x + i) = x[i]`: `x` + `sizeof(x)` * `i`.

Control Instructions

F1

Instruction	Semantics	Additional information
`BEQ a, c`	`if(reg[a]==0) pc:=pc+c*4;`	Branch on equal to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BGE a, c`	`if(reg[a]>=0) pc:=pc+c*4;`	Branch on greater than or equal to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BGT a, c`	`if(reg[a]>0) pc:=pc+c*4;`	Branch on greater than to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BLE a, c`	`if(reg[a]<=0) pc:=pc+c*4;`	Branch on less than or equal to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BLT a, c`	`if(reg[a]<0) pc:=pc+c*4;`	Branch on less than to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BNE a, c`	`if(reg[a]!=0) pc:=pc+c*4;`	Branch on not equal to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BR c`	`pc:=pc+c*4;`	Branch (unconditional)
`BSR c`	`reg[31]:=pc+4;`	Branch to subroutine (unconditional)
	`pc:=pc+c*4;`	Tthe link register (`R31`) is saved to be able to return to the correct instruction

F2

Instruction	Semantics	Additional information
`RET c`	`pc:=reg[c];`	`c` = `R31`

F3

Instruction	Semantics	Additional information
`JSR c`	`reg[31]:=pc+4;`	`c` = `R31`
	`pc:=c;`	absolute addressing

All branches (conditional as well as unconditional) are pc-relative. The multiplication of `c` by `4` is because the words in the memory are byte-addressed. Branches are useful (especially because they operate pc-relative) because you generate so-called *relocatable code*. *Relocatable code* can be moved in memory anywhere and will not change its behavior because every address is computed relative to the program counter. The drawback of branches is the restriction of the addressable space: there may be more memory than you can use with branches. The solution is absolute addressing used by the F3 format. The F3 format allows you to address a range of 2^{26}.

Example:

C Code	Instructions	Additional information
`while(x<1) {`	`LDW 1, 28, -4`	R1 := x
`<body>`	`ADDI 2, 0, 1`	R2 := 1
`}`	`CMP 1, 1, 2`	R1 := R1 - R2
	`BGE 1, 0, 0`	`c` is set to zero, because at this point of time we don't know where to jump to (in general). A FixUp will later update this address.
	`<body>`	The body of the function
	`BR 0, 0, <top>`	`<top>` is the address offset to branch BEFORE the first generated instruction (`LDW 1, 28, -4`).

C* is a Turing-complete language:

- arithmetics or integer
- dereferencing operator
- assignment
- while loops

Functions are not needed to satisfy Turing-completeness but reduces the size of code by preventing code duplication.

In general, every if-else-construct can be replaced by while loops, but not the other way round. However, a while loop can be replaced by a combination of functions and if-else-constructs for Turing-completeness.

Functions

Why do we use a stack and no other data structure? Because a stack guarantees proper nesting of functions, constant time access and has no spatial drawback. These facts are a result of the LIFO (Last In First Out) property of a stack.

Example:

C Code	Instructions	Additional information
`f(x);`	`LDW 1, 28, -4`	R1 := x
	`PSH 1, 30, 4`	Push argument on stack
	`<body of f>`	
	`BSR 0, 0, <faddr>`	Branch to subroutine at address offset `<faddr>`
	`ADD 1, 0, 27`	`R27` is the return register (by convention)

`<faddr>` is often resolved by a part of the compiler called *Linker*. There may be two kinds of references: symbolic references (e.g. `f`, the name of the function) and direct references (e.g. `addr(f)`, the absolute address of the function in memory).

Examples:

C Code	Instructions	Additional information
`int f(int x) {`	`PSH 31, 30, 4`	Save link register (push `R31` onto stack)
`<body of f>`	`PSH 29, 30, 4`	Save frame pointer (push `R29` onto stack)
`}`	`ADD 29, 0, 30`	Set new frame pointer to current stack pointer See Figure 10
`return x;`	`LDW 1, 29, 12`	Load argument `x` from memory. The offset is 12 here because `x` is pushed onto the stack before link and frame register.
	`ADD 27, 0, 1`	Save `x` into return register
	`POP 29, 30, 4`	Restore caller's frame (register)
	`POP 31, 30, 8`	Pop link register from stack as well as `x`. This could also be done by two `POP` instructions each popping `4` bytes. In general the size to pop is `4 + #args * sizeof(arg`$_i$`)`.
	`RET 0, 0, 31`	Return to where function was invoked

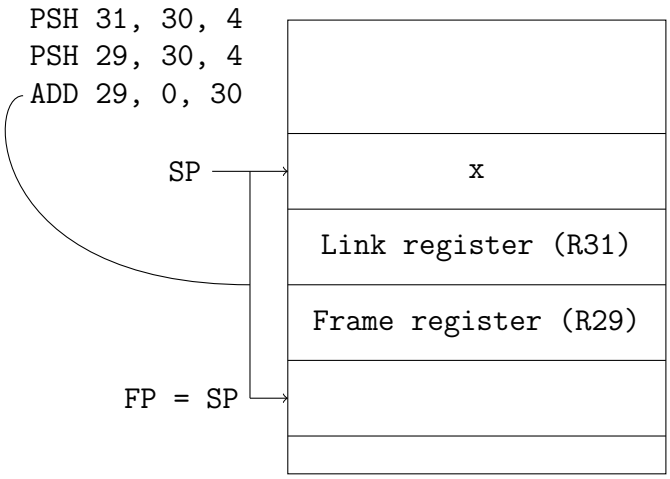


Figure 10: Stack snapshot of a function with one argument `x`.

Caller vs. Callee:

The caller of a function `f` invokes the function, e.g. `f(x);`.
The callee is the function itself, e.g. `int f(int x) { ... }`.
The callee can again be a caller if it invokes another function in its body, e.g. `int f(int x) { ... g(x); ... }`.

Declaration vs. Definition:

A declaration is inserted into the symbol table of a compiler, e.g. `int x;`. The declaration tells the compiler that a variable or function exists.
A definition is the actual value assignment, e.g. `x = 2;`. A definition can be done multiple times in a source code.

Example:

C Code	Instructions	Additional information
`y = x + f(x);`	`LDW 1, 28, -4`	Load `x`
	`PSH 1, 30, 4`	Save context of `f`
	`LDW 1, 28, -4`	Load `x`
	`PSH 1, 30, 4`	Push actual argument(s) for f onto stack
	`BSR 0, 0, <faddr>`	Branch to `f`
	`...`	Execute function `f`
	`POP 1, 30, 4`	Restore context of `f`
	`ADD 2, 0, 27`	Store return value into `R2`
	`ADD 1, 1, 2`	Actual addition of `x` and `f(x)`
	`STW 1, 28, -8`	Store result.
		`-8` because it's the second global variable

Heap

Dynamically allocated memory is located on the heap. In C there is the `malloc`/`free` combination to accomplish this, in Java there is the `new` keyword to allocate memory dynamically but no explicit mechanism to deallocate it. Java has a garbage collector which safely deallocates unneeded memory. However, Java provides implicit deallocation mechanisms. The first is to set a reference explicitely to null:

```
x = new A(); // dynamic allocation
...
...
x = null;
// if no other reference to x exists here
// x will be deallocated by the garbage collector
```

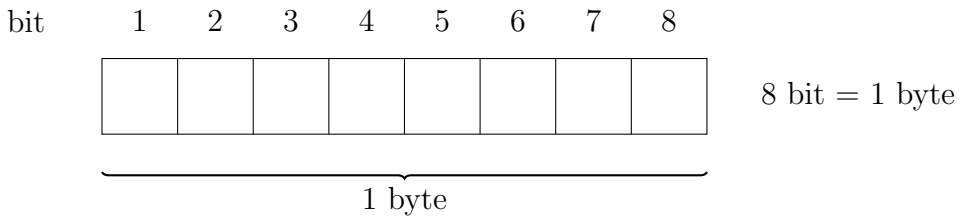
The second approach uses namespaces. When jumping out from a namespace, every namespace-local variables are unneeded if and only if there are no references from outside. If there are no such references the garbage collector will deallocate these variables:

```
{
    x = new A();
    ...
    ...
}
```

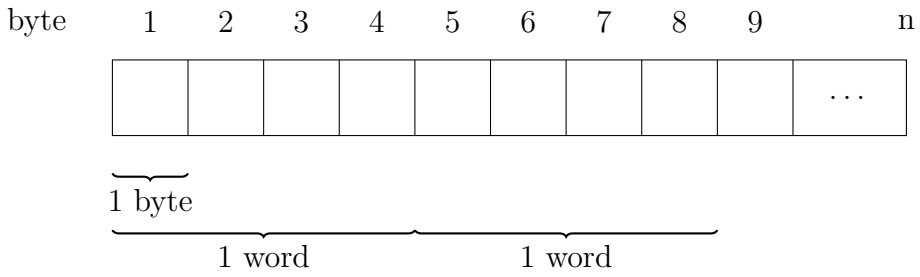
The simplest (but most unintelligent) implementation of a dynamic heap allocator is a so-called *bump pointer allocator*. A bump pointer is a pointer which only counts upwards, never downwards. In this case, there is no `free` to deallocate because it only grows upwards. However, the implementation of `malloc` is pretty simple and straigh-forward:

```
void *malloc(int s) {
    LDW 1, 29, 12
    ADD 27, 0, 26
    ADD 26, 26, 1 // R26 = Heap pointer
}
```

So every time `malloc` is invoked, these three DLX instructions are executed.



Byte-addressing and word-alignment:



Wie funktioniert der Zugriff auf den Stack?
Wieso wird `R1, R28, -4` gemacht? **Latency**.

Implementation von `void* malloc(int s)`:

```
LDW 1, 29, 12
ADD 27, 0, 26
ADD 26, 26, 1
```

`s: R1 = sum[R29 + 12]`
Da man in `C*` keinen Zugriff auf Register hat, muss obiges Verhalten per Assembler implementiert werden.
`malloc` ist in der (g)libc enthalten und ist heutzutage de facto ein Betriebssystem-feature.

Implementation von `int getchar()`:

```
RDC 0, 0, 27
```

Implementation von `void putchar(int c)`:

```
LDW 1, 29, 12
WRC 0, 0, 1
```

Liest den nächsten Character aus einem Stream.
`R27 = getchar();`
Streams: `stdin`, `stdout`, `stderr`

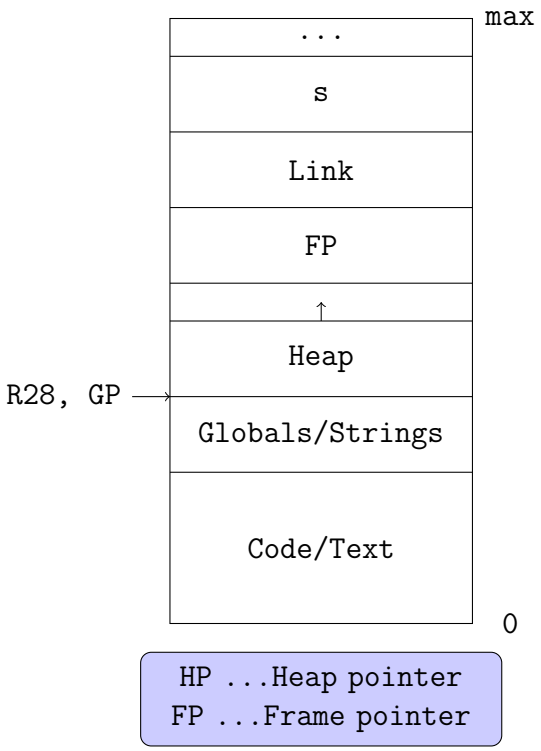


Figure 11: Memory layout revisited

F2

Instruction	Semantics	Additional information
`RDC c`	`reg[c] = getchar();` `pc:=pc+4;`	
`WRC c`	`putchar(reg[c]);` `pc:=pc+4;`	

Why is it so hard to implement `RDC` and `WRC` on application level? Because operating systems stuff is self-referential!

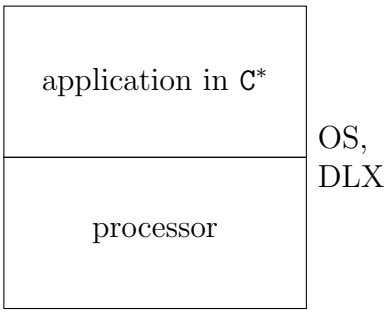


Figure 12: Application and processor

How does I/O work on a processor?

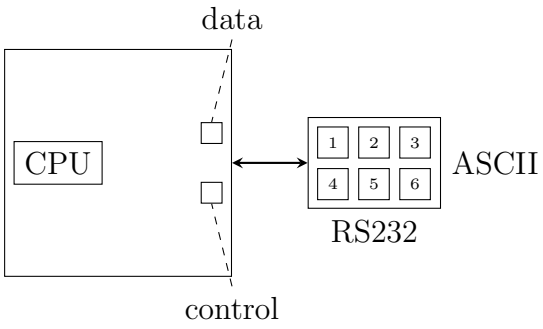


Figure 13: I/O on a processor

Computability & Complexity of algorithms

What is the minimal machine that is still universal?

- RISC: Reduced Instruction Set Computer
  Has separate instructions to load/store & compute.
  E.g. ARM, MIPS, SPARC, ...
- CISC: Complex Instruction Set Computer
  More complex instructions which load, compute & store in a single instruction.
  E.g. x86

RISC vs. CISC $\leftrightarrow$ Compiler vs. processor.

OISC/URISC: One/Ultimate Reduced Instruction Set Computer

`SUBLEQ a, b, c`: Subtraction less or equal

```
mem[b] := mem[b] - mem[a];
if mem[b] ≤ 0 goto c;
else pc := pc + 4;
```

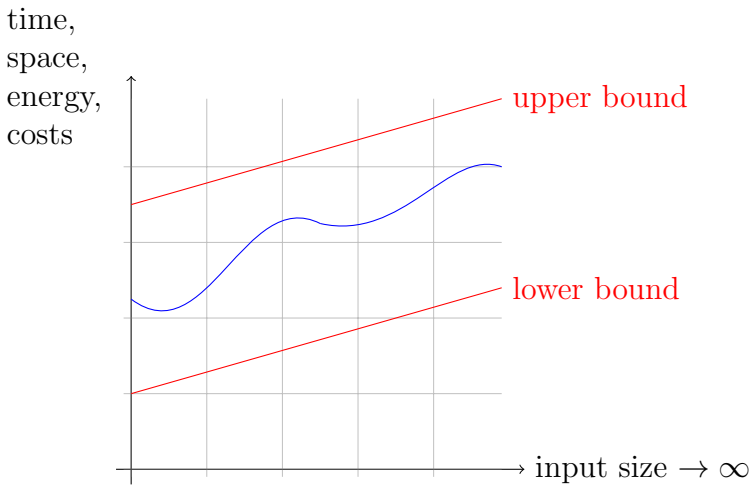


Figure 14: Asymptotic computational complexity [1]

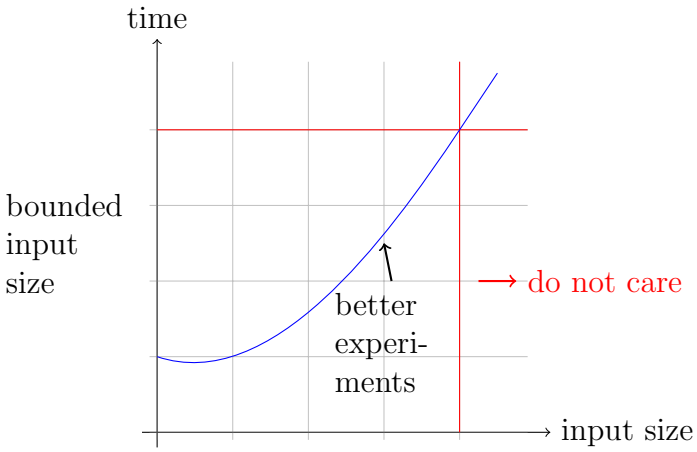


Figure 15: Asymptotic computational complexity [2] (for operating systems)

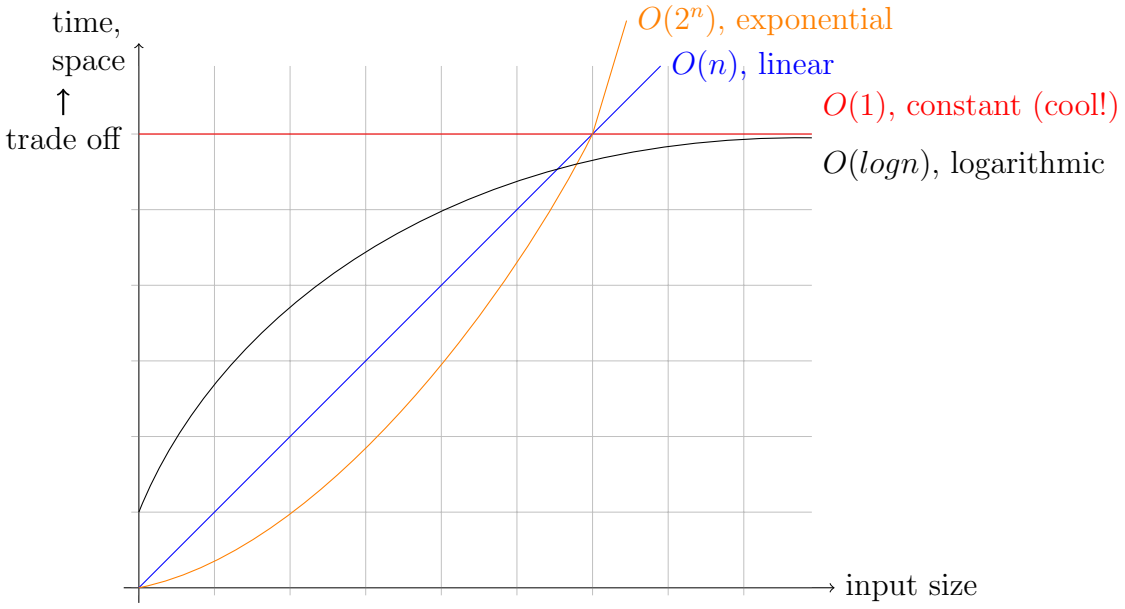


Figure 16: Big O notation (upper bound)

Big O notation (upper bound): $O(\#bits)$

$O(\#bits) = O(\#bytes) = O(\#char) = O(\#instructions) = O(\#objects)$

Numeric representation

unary: $\underbrace{||||||||||||||||}_{16}0$, binary: $\underbrace{10000}_{16}$, octal: $\underbrace{20}_{16}$, decimal: $\underbrace{16}_{16}$, hexadecimal: $\underbrace{10}_{16}$
Exponential reduction from unary to binary representation: $1 + n = \lfloor log_2 n \rfloor + 1$
The main difference ist the denseness of the representations.

```
fib(n) {
   if n ≤ 2
      return n;
   else
      return (fib(n − 1) + fib(n − 2));
}
```

This implementation has a worst-case time complexity of $O(2^n)$. Thus, `fib(7)` will terminate correctly but something like `fib(1024)` will not terminate correctly on most systems.
There exists another implementation of the Fibonacci numbers using dynamic programming. This implementation has a logarithmic worst-case time complexity.

$log(n) \cdot log_d(2) = log_d(n)$

From Turing to digital:

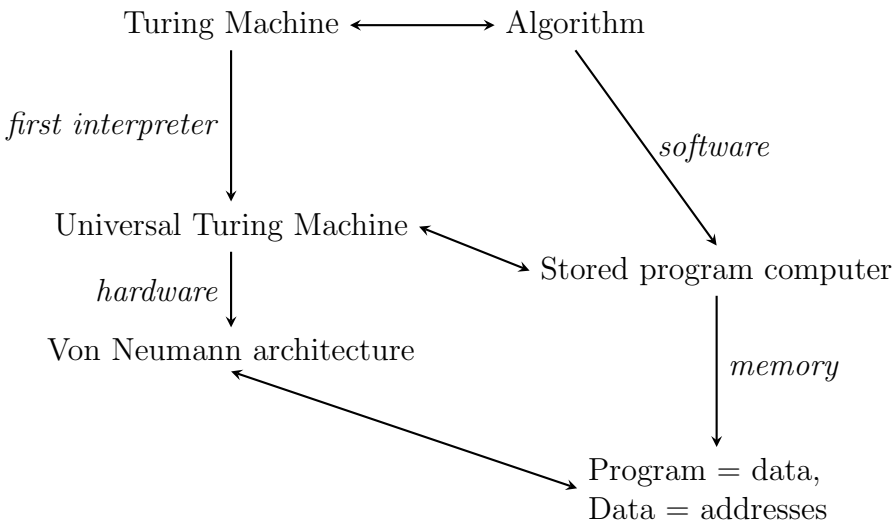


Figure 17: From Turing to digital

TM:

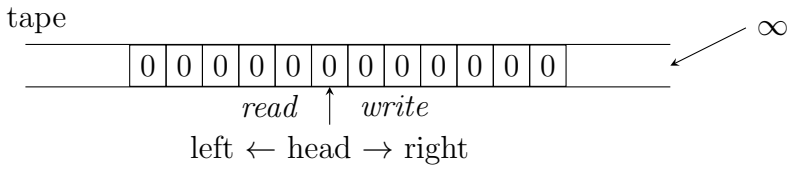


Figure 18: TM read/write

Program:

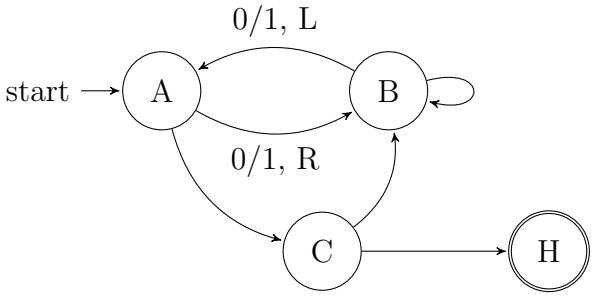


Figure 19: Busy beaver (3 states)

$$n! = \text{factorial(n)} = \begin{cases} 1 & \text{if n} = 0 \\ \text{factorial}(n-1) \cdot n & \text{otherwise} \end{cases} \Rightarrow O(n)$$

$$\underbrace{1024!}_{\text{just 4 digits but 1024 iterations/calls}} = O(2^d), d \ldots \text{number of digits}$$

$n!$ is pseudo-polynomial!

`1 0000 + 0 0001 = 1 0001` $\Rightarrow O(d)$
Addition is much better because it has linear time complexity in the number of digits. If the computation is done numerically (the numbers are represented numerically), the time complexity is logarithmic in the size of digits: $O(log(d))$.
If we represent the number binary, the time complexity is doubly-exponential: $O\left(2^{2^d}\right)$.

Considering the above recursion: There are at most $O(n)$ elements on the stack at any point of time (also in the worst case). Thus, this recursion has linear space complexity.
For some algorithm, e.g. sorting algorithms, the representation of the numbers has no effect because the numbers are only used in the comparison.

Computability:

Alan Turing:
There is no algorithm to decide whether an arbitrary program halts on a given input (1936). *Halting problem.*

Rice:
Generalization to any non-trivial property of partial functions (programs that may or may not terminate) computed by arbitrary programs. A compiler can not analyze a program for an arbitrary property.

Church:
1 month earlier than Turing using λ-calculus.

Proof sketch:
total function: defined on our inputs
partial function: will not terminate for inputs for which it is not defined

1. $h(i,x) = \begin{cases} 1 & \text{program } i \text{ halts on } x \\ 0 & \text{otherwise} \end{cases}$
   $h(i,x)$ is a total, computable function.
2. Let f be any total, computable function with two arguments (in fact it may even be $h(i,x)$).

3. $g(i) = \begin{cases} 0 & \text{if } f(i,i) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$

   Equivalent program code:
   ```
   if(f(i, i) == 0) return 0;
   else while(true) {};
   ```

   $g(i)$ is a partial, computable function.

4. $g(i)$ may be partial but is computable by a program e (as described in the step before).

5. If $f(e,e) = 0$: $g(e) = 0$ by definition $\Rightarrow h(e,e) = 1$.
   If $f(e,e) \neq 0$: $g(e) =$ undefined (does not terminate) $\Rightarrow h(e,e) = 0$.
   $\Rightarrow f$ is always different from $h \Rightarrow h$ can not be programmed.

This proof technique is called *Diagonalization* and was introduced by Cantor in 1891.

Given 2 infinite sets: which one is bigger (cardinality)?
E.g. $\mathbb{N}$ vs. $P(\mathbb{N})$: $\mathbb{N} < P(\mathbb{N})$ (strictly bigger!)

$\mathbb{N}$: There are enough natural numbers to encode any program you can ever think of.

Another view on the halting problem:

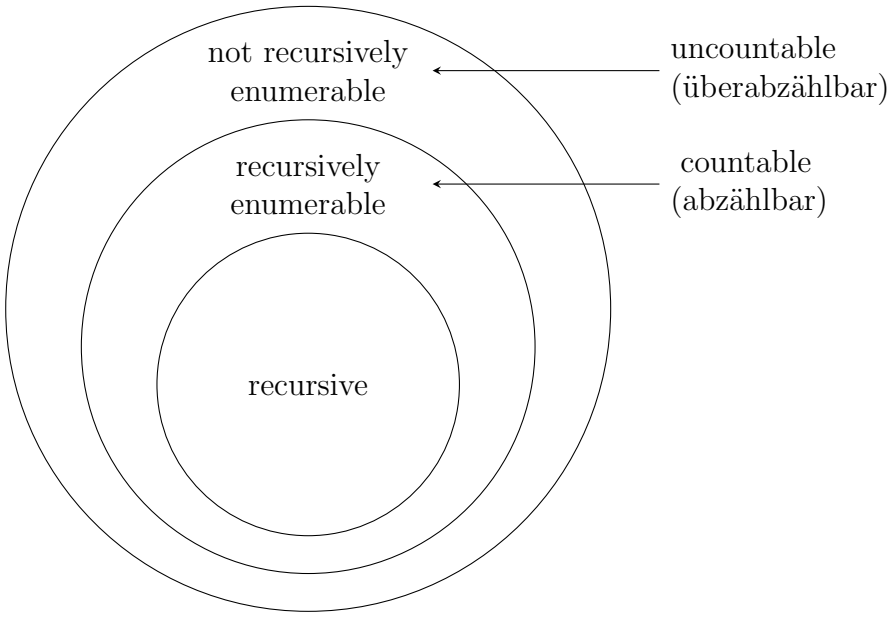


Figure 20: The halting problem in terms of languages.

- Recursive:
  A language (a set of strings) is recursive if it is accepted by a Turing machine which always terminates/halts.
- Recursively enumerable (r. e.):
  A language is recursively enumerable if it is accepted by a Turing machine which may terminate/halt on a string not in the language. $\{(i, x)|\text{program } i \text{ halts on } x\}$
- Not recursively enumerable (n. r. e.):
  Is actually not a superset of the r. e. set but n. r. e. $\cup$ r. e. results in the set of all reachable programs. This set is uncountable. $\{(i, x)|\text{program } i \text{ does not halt on } x\}$

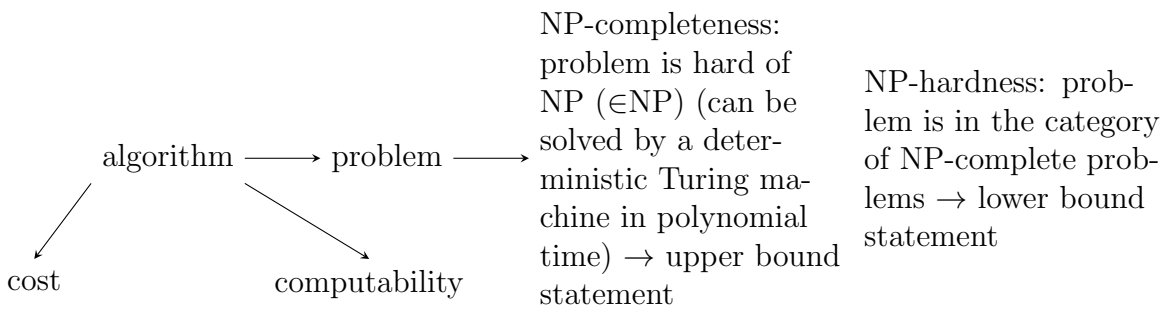


Figure 21: NP-completeness & NP-hardness

In full generality, a compiler cannot choose whether a code is dead or not
$\Rightarrow$ NP-hard.
In some trivial cases, the compiler can actually recognize dead code.

Syntex & Semantics:

Syntax: Parser, Syntax analysis, ...
Semantics: Code generator, semantics definition, ...
In combination: compiler

Syntax:

1. What is correct syntax? Specification
2. Parser

Example: `1 + (2 + y) * z`

Grammar:

Symbol	Description
`<string>` (e.g. expression)	non-terminal
`"<string>"` (e.g. "a")	terminal
`[<symbol>]` (e.g. ["-"])	optionality
`{<symbol>}` (e.g. {digit})	repetition
`|`	OR

Parser: context-free → counts symbols (e.g. matching parentheses); Recursive-decent parser

```
expression := ["-"] term {("+" | "-") term}.

term := factor {("*" | "/") factor}.

factor := identifier | integer | "(" expression ")".
```

Scanner (FSM): reads character-wise and decides; regular → if you can transform this into a single equation

```
identifier := letter {letter | digit}.

letter := "a" | "b" | ...| "z" | "A" | "B" | ...| "Z".

digit := "0" | "1" | ...| "9".

integer := digit {digit}.
```

Implementation of `expression()`:

```
expression() {
   if(symbol == MINUS)
      getSymbol(); // next symbol
   term();
   while((symbol == PLUS) || (symbol == MINUS)) {
      getSymbol();
      term();
   }
}
```

So far no semantics!

See

- CS4NonCSs Introduction.pdf
- CS4NonCSs Architecture.pdf

What we want:

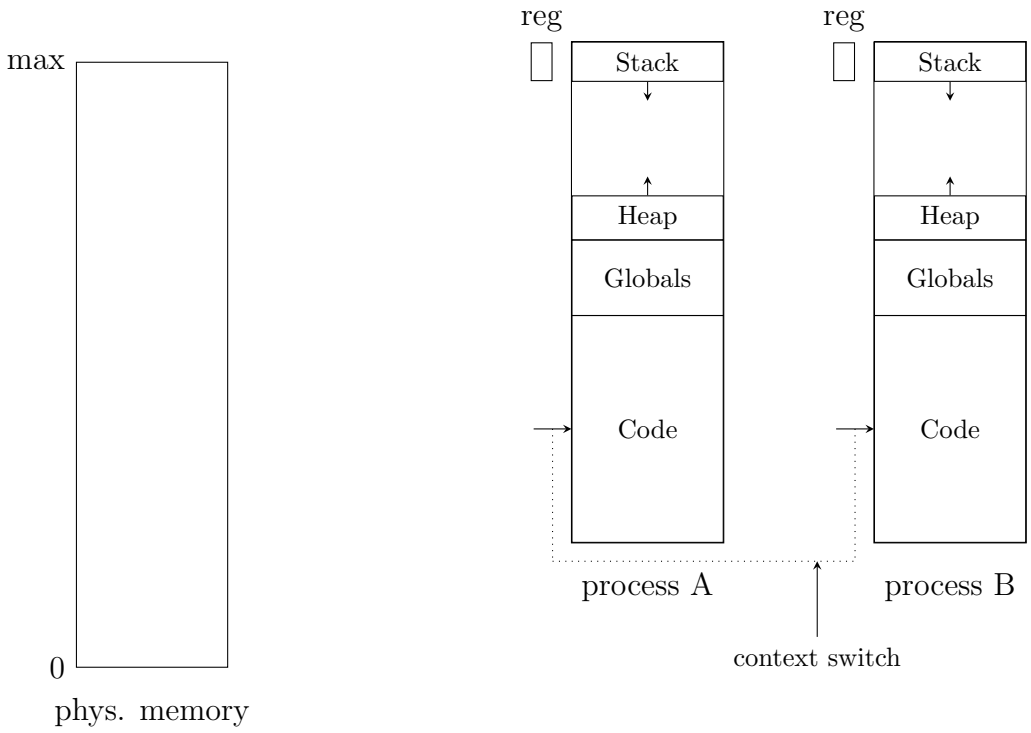


Figure 22: Each process thinks it owns the whole memory.

Each process has its own virtual memory which is loaded whenever a context switch is performed. *Concurrency* (competing for resources, time sharing) $\neq$ *Parallelism* (truely parallel, multiple processors). *Spatial isolation* is necessary to enable concurrency.

Data structures:

2 Goals:

1. Abstraction: table, list, tree, graph, ...
2. Safety: only accessing memory we have declared, memory access as specified.

2 levels where we can have safety:

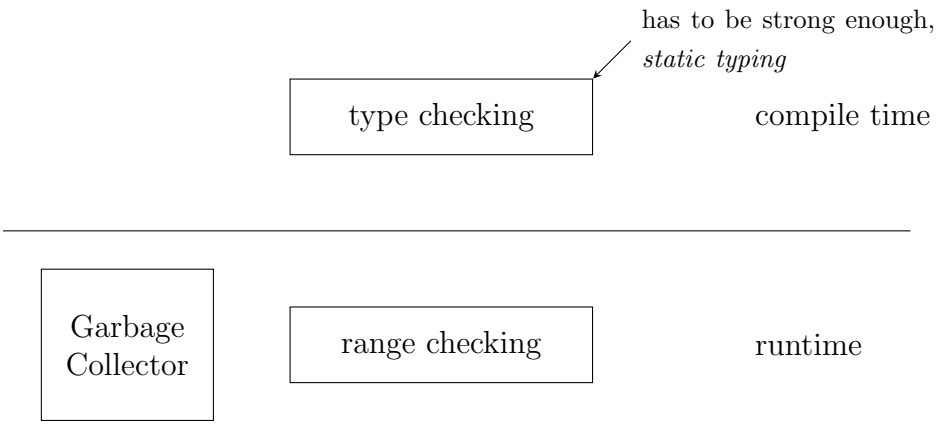


Figure 23: safety levels

Unless all accesses use constants as indices, in general no compiler can perform a range check at compile time. At some point there is a limit. E.g. `i = 10; a[i];` is recognized by some compilers.
If you do not allow dynamic allocation, your language does not allow to write a program which accesses memory outside (type and range checks required of course!). Suppose a `free` mechanism: you could free something and access it afterwards, or you could do `free(obj);` which creates a *dangling pointer*, and do `obj.field = 1;` afterwards.

Arrays:

```
int a[3];
```

The advantage of the contiguous layout in memory: constant time access.
The big drawback of the contiguous layout in memory: fragmentation.
3 ingredients of fragmentation:

1. contiguously allocated memory gives us constant time access (could be splitted as well)
2. blocks of different size
3. the order of deallocation is not equal to the order of allocation

In the worst case, there is a 50/50 allocation present. If we then want to allocate a big chunk of memory, it is not possible without some reordering.

Records:

```
struct rec {
  int i;
  struct rec *r;
};
```

The advantages of the contiguous layout in memory:

- constant time access (even faster than the array access because the offsets can be computed at compile time)
- no fragmentation (if the implementer knows what s/he is doing)

The disadvantage is the fact that the access complexity is not constant but linear in the size of the list.

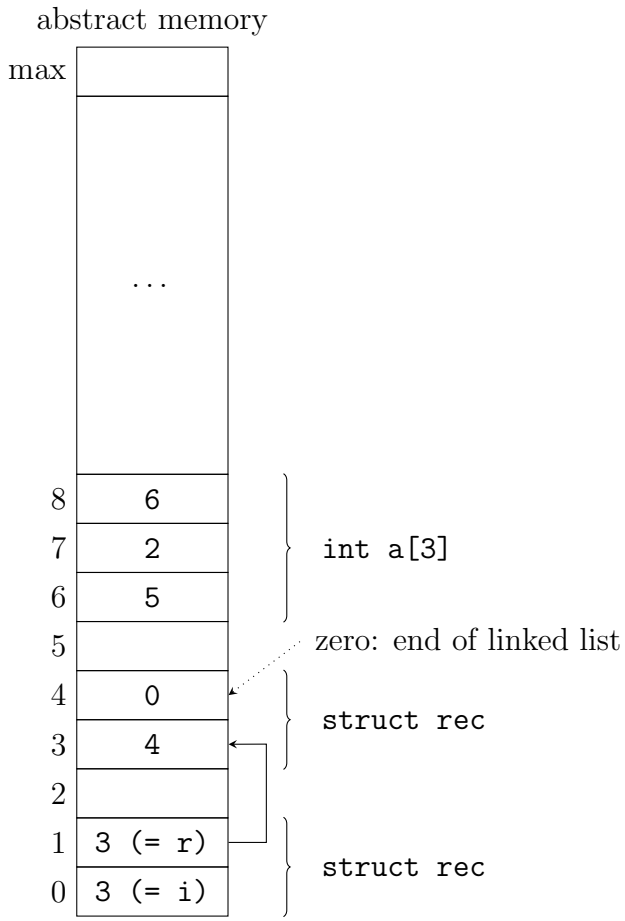


Figure 24: Arrays and records in the memory

The only other option (to use contiguous allocation):

Construct data structures out of pointers.

If we just allocate a billion of `struct rec` (which actually all have the same size), there is no fragmentation.

Memory Hierarchy:

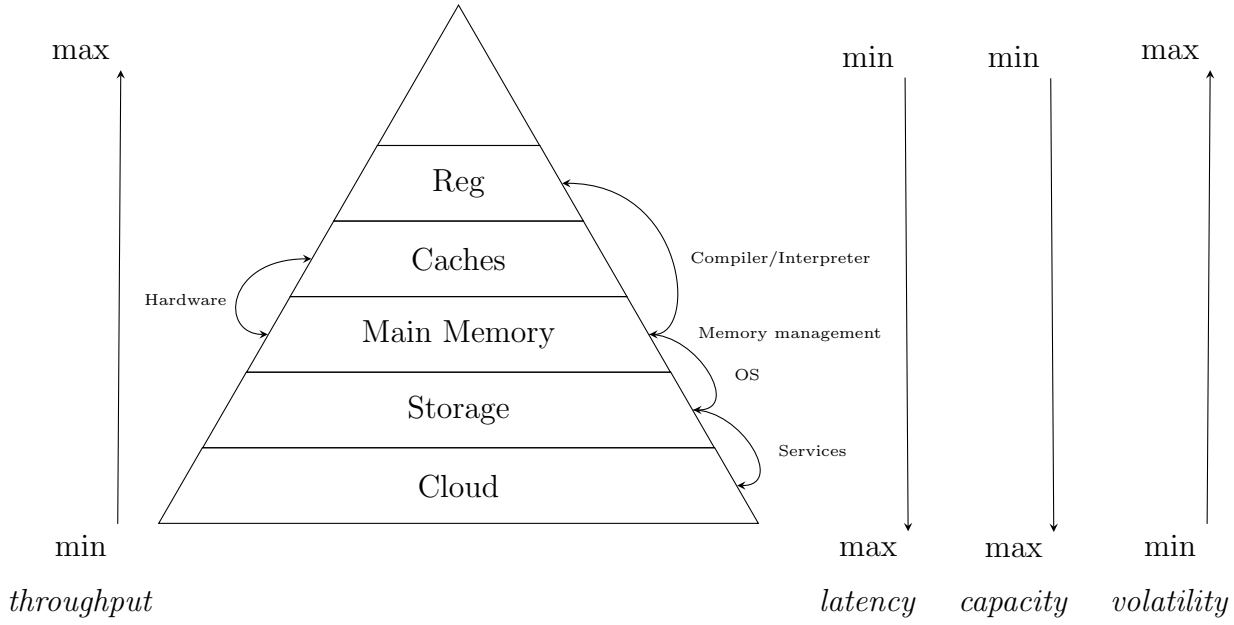


Figure 25: Memory hierarchy

Services: Dropbox, Google Cloud, ...

Caches:

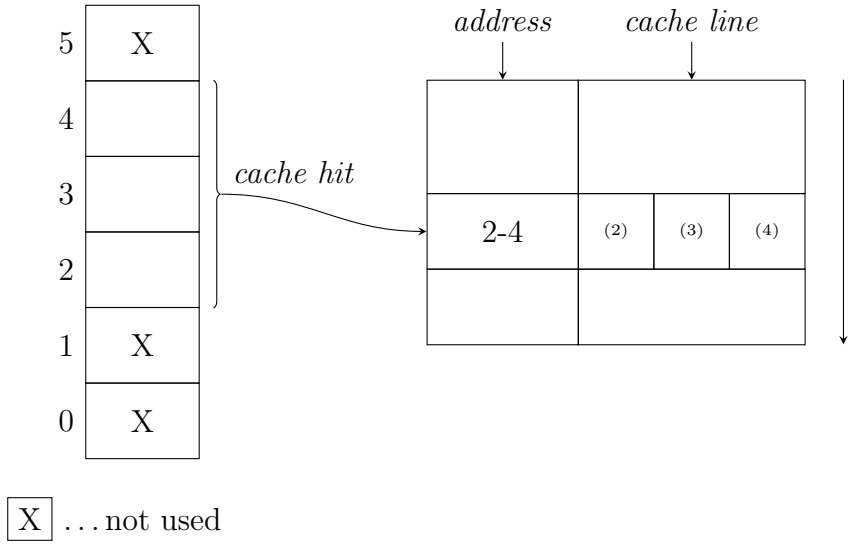


Figure 26: Caches

Why is caching beneficial? Checking and searching in the cache and not the main memory is much faster.

What motivates a cache? *Temporal locality*: access of a given memory location is likely to be followed by an access of the same location in the near future. Examples are looping und sequentiality.

What motivates a cache line? *Spatial locality*: access of a given memory location is likely to be followed by a location nearby (local memory neighbor location). This is especially beneficial for sequential code and may be bad for parallel code. An example is the linearity of data structures in the memory.

An example where cache lines are bad for parallel code: if 2 threads have two variables and align them in the cache line like in Figure 27. This is the worst case because the cache line has to be replaced every time.

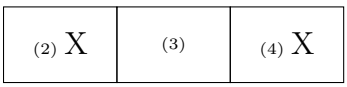


Figure 27: Example of *false sharing*

There are two extreme design choices when it comes to cache design:

1. fully-associative cache → each cell in memory can be in any cache location
   slow search, high hit rate

2. direct-mapped cache → each cell in the memory can only be in one cache location
   fast search, high miss rate (many different memory addresses mapped to the same cache line because the size of the cache is << the size of the memory → lower utilization.)

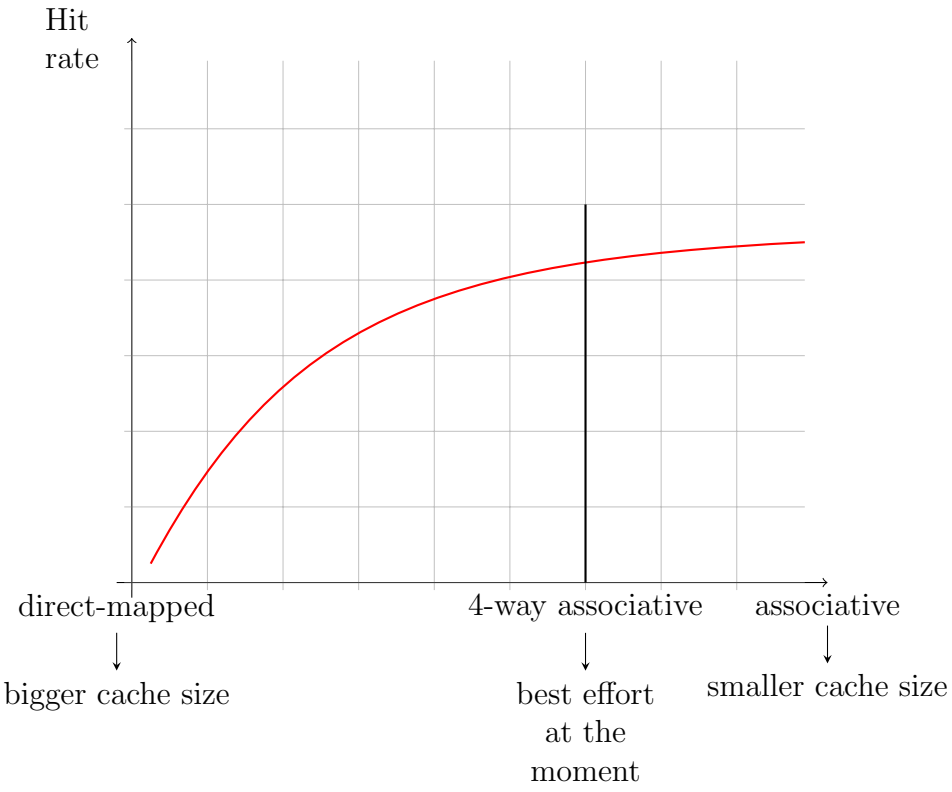


Figure 28: Hit rate of different cache designs

Fragmentation:

Caused by allocation and deallocation in different order. A bump pointer allocation is performed, the allocated memory is contiguous aligned.

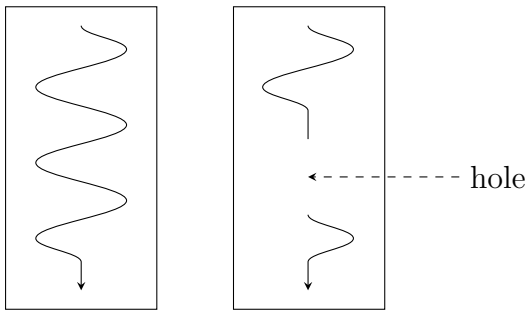


Figure 29: Memory hole

3 solutions:

1. Compaction:
   Make bigger holes and get a bound on fragmentation.
   Moving memory in space is slow ($O(n)$ in the size of the memory to move).
   Furthermore the addresses change ($O(n)$ in the size of the memory).

2. Coalescing:
   Non-moving, merge contiguous free blocks. No bound and causes an improvement on average but not in the worst case.

3. Partitioning (Prepartitioning):
   scalloc implementation from the University of Salzburg (scalable allocation).

scalloc:

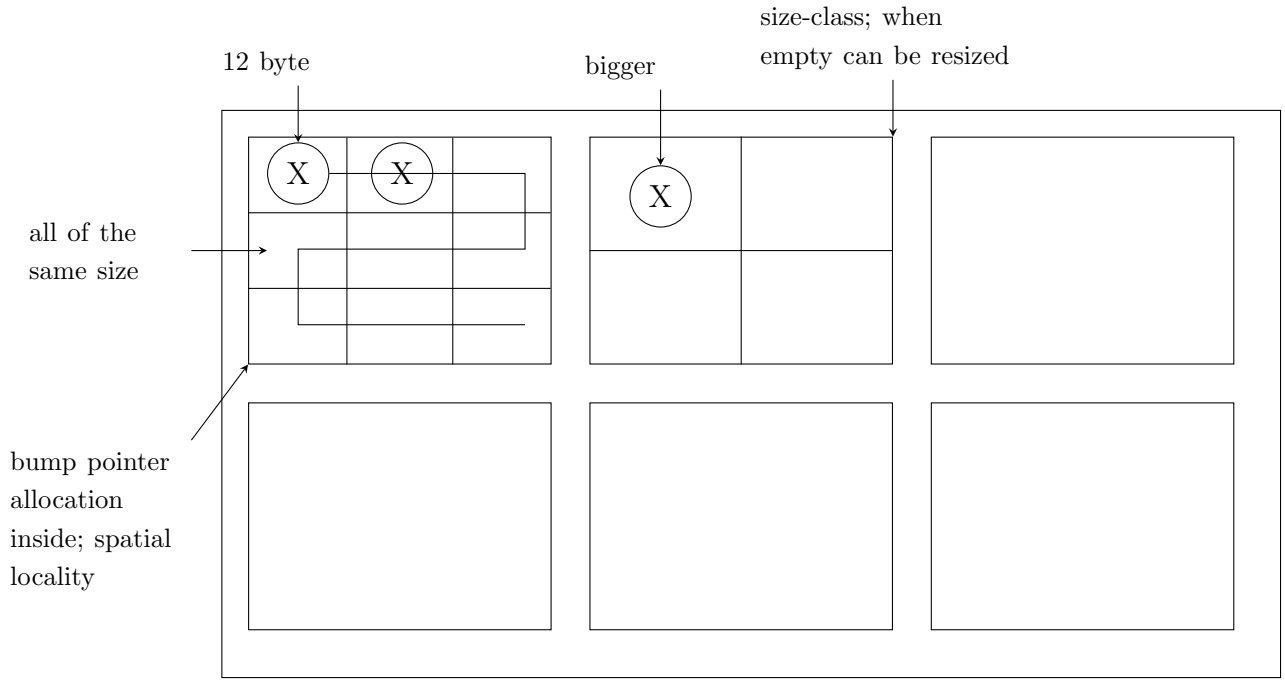


Figure 30: Visualization of the *scalloc* idea

Pros:

- no external fragmentation

Cons:

- internal fragmentation (whenever partitioning is done)
- anything bigger than the partition size cannot be allocated (bound on max. size)

Basic two techniques for Garbage Collection:

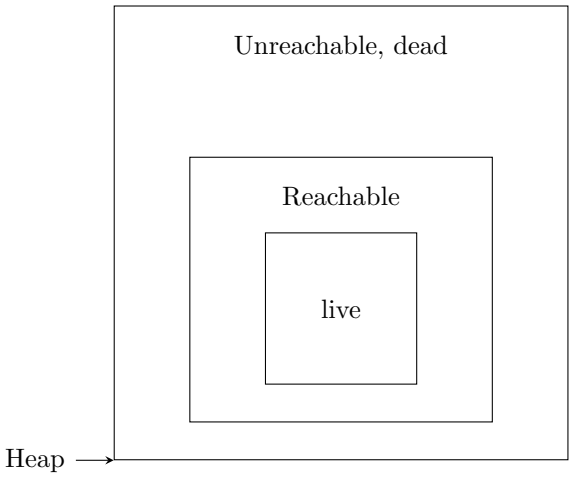


Figure 31: Sets of Garbage Collection

- live: will be used in the future
- dead: will not be used (computation of the dead set is impossible in general)

Tracing: Computes reachable set (and so knows the unreachable set). Semi-space allocator.

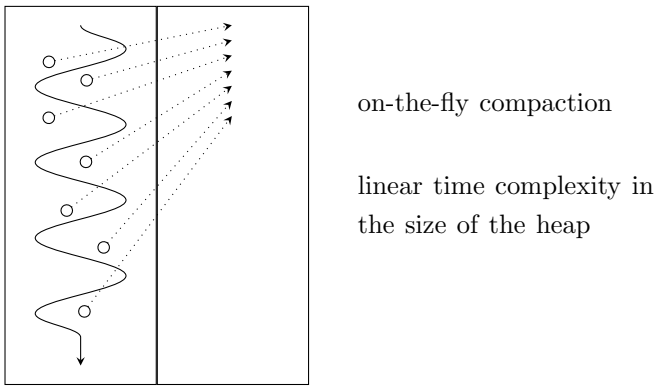
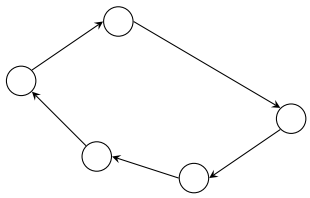


Figure 32: Semi-space allocator

Reference Counting: Computes the unreachable set: count references to an object and deallocate it when the count reaches zero.
Drawback: Incrementing/Decrementing every time.
Complexity: Needs cycle detection, $O(n)$ in the size of the heap.



would never be
removed because
all references
have count = 1
⇒ memory leak

Figure 33: Reference Counting: cycle detection needed

Reachable memory leak:

Occurs when the reachable set grows & grows because the programmer does not destroy references in garbage-collected languages.

In Multicore systems era:

Run Garbage Collector on an own CPU if one is free.

Real-time application:

You know how much time a specific operation takes and there is a guarantee on this max. time.

Thread Local Allocation Buffer (TLAB):

Each thread has its own TLAB to allow faster allocation. Whenever a new object is allocated on the heap, the object will first be placed in the TLAB. The performance improvement is because the threads can allocate additional memory within the TLAB without locking anything. The TLAB is pre allocated for each thread.

Virtual Memory:

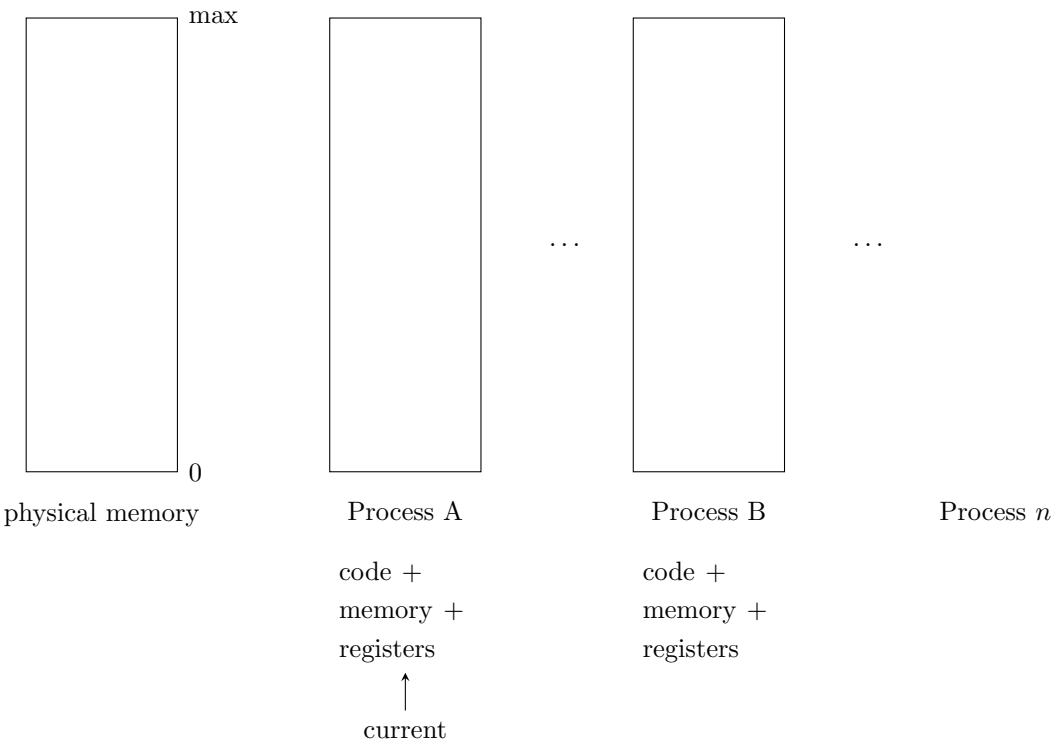


Figure 34: Virtual memory: basic idea

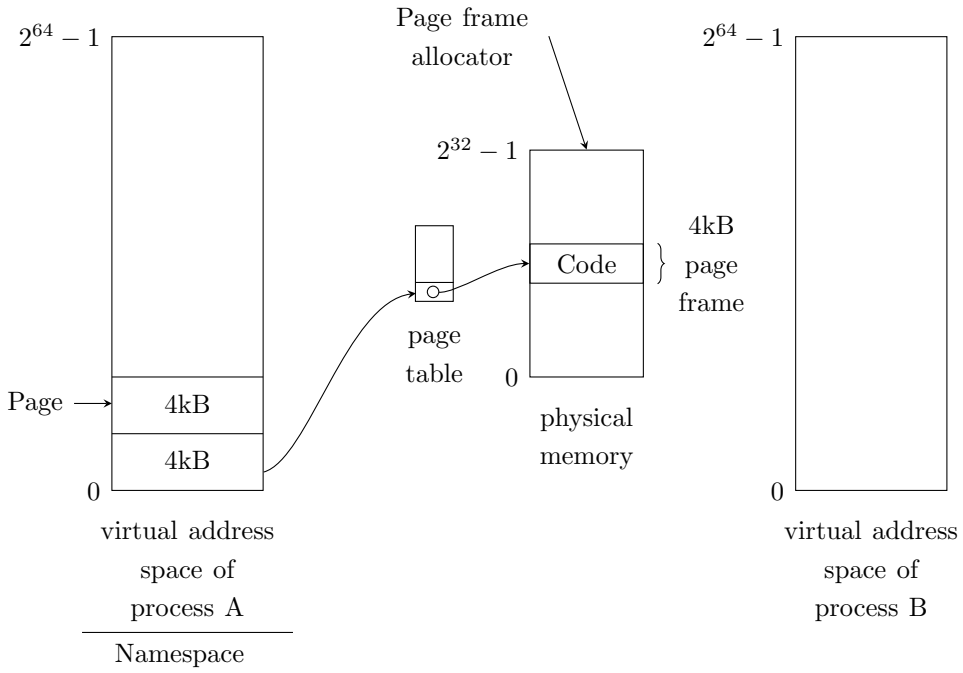


Figure 35: Virtual memory: virtual and physical address space

Logical the address space is from 0 to $2^{64}-1$. But current hardware supports up to 2^{48} and current software supports only up to 2^{45} addresses.

The page table resolves the mapping from virtual memory (only addresses) to physical memory (actual memory we can store data to).
Why do we use 4kB chunks? The smaller the pages the bigger the page table, e.g. if we would use 1B pages the page table would have a size of 2^{64} entries. The use of 4kB pages reduces this by a factor of 4.000. The fact that all pages have the same size (4kB) results in a big advantage: there is no fragmentation.

Memory Management Unit (MMU):

Very fast hardware to resolve the mapping virtual address $\leftrightarrow$ physical address. It also serves as a memory protector and tells the operating systems which virtual addresses are accessible. Because there are millions of requests, this mechanism has to be very fast and thus is implemented in hardware.

Page fault:

Page fault handler $\leftrightarrow$ Trap $\leftrightarrow$ MMU.

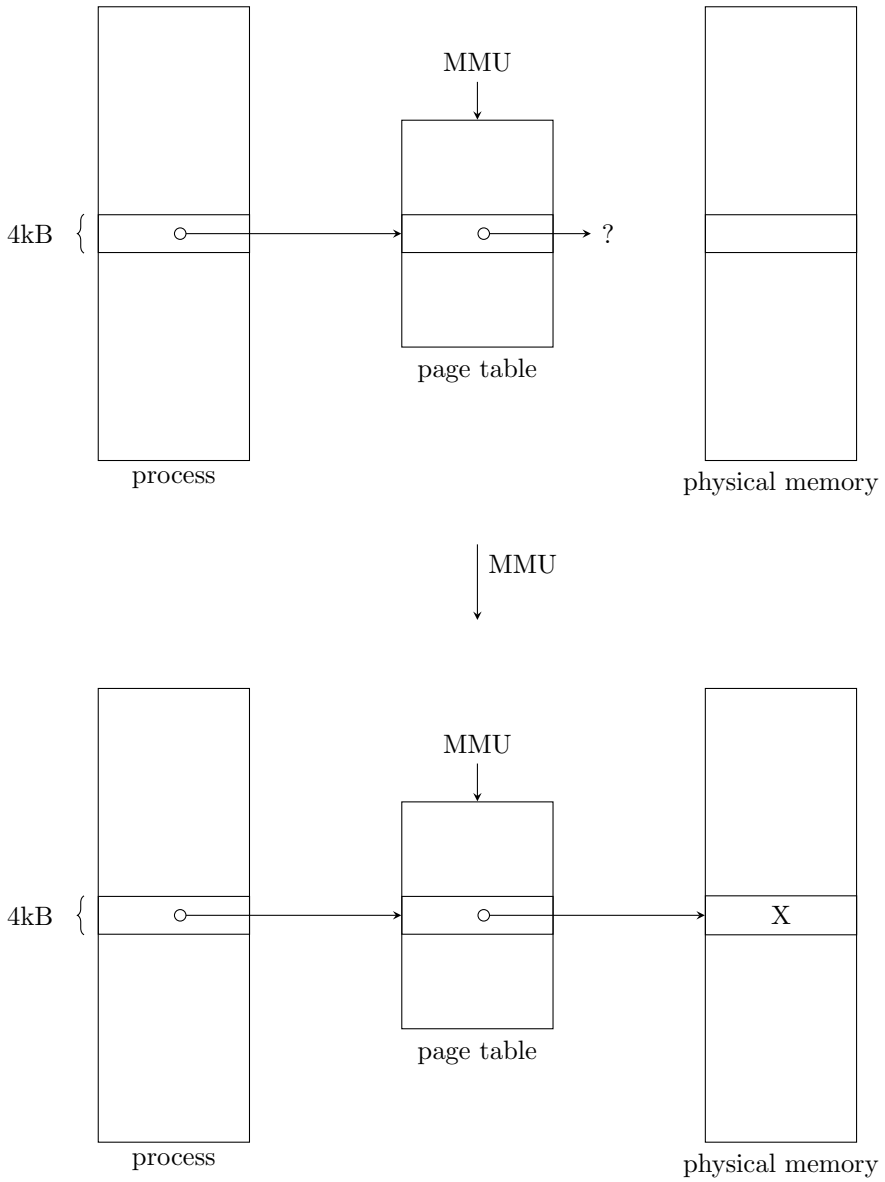


Figure 36: Resolving a page fault.

If a page fault occurs, the MMU resolves the corresponding mapping and the requested operation is executed again after the mapping was done by the MMU. This is called *on-demand paging*.

$\frac{2^{64}}{4kB} = \frac{2^{64}}{2^{12}} = 2^{52}$ is the size of the page table if 4kB pages are used. We would not be able to allocate such a big page table.
The address space is gigantic and the whole address space is (almost) never used completely by a single process (*sparsely/tensed populated*).
Thus, size of the page table $\ll 2^{52} \Rightarrow$ no array is used $\Rightarrow$ *2-level tree* is used.

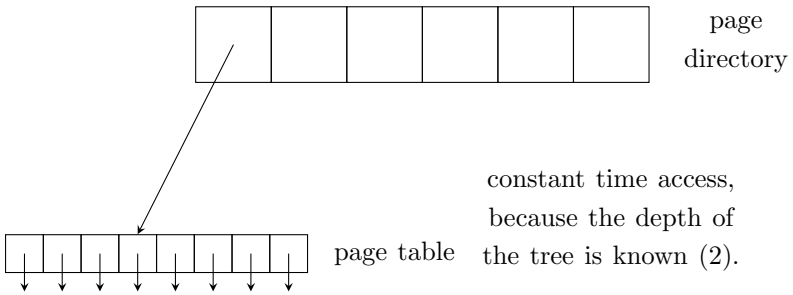


Figure 37: 2-level tree (or 2-phase tree).

Prefetching is even smarter. If a page is loaded, its neighbors (the page below/above it) are loaded within the same request (see spatial locality).

Physical memory full on page fault ⇒ Swap out page frames onto HDD/Flash/Cloud. Swap file entry tells the operating system that the page frame is located in the swap partition. Swapped out page frames are marked in the page table. Single page frames are swapped out (*frame granularity*).

Page replacement algorithms:

The best page frame to replace would be the one which is used the farthest in the future. But to determine this optimal page frame the operating system would need to look into the future. Thus there are several algorithms trying to approximate this behavior: clock, aging, LRU (least recently used).

Process switch:

When the operating system (the scheduler) switches the process, e.g. the currently executed process A is put to sleep and process B is scheduled next, the page table and the program counter (PC) are switched. The MMU then uses the page table of process B and the CPU uses the program counter of process B. Figure 38 and 39 show the process switch. So each process has the illusion that it owns the whole physical address space.

The process switch can be done in constant time because the operating system only needs to adjust the pointers of the MMU and the CPU, respectively. This results in a perfect isolation in memory.

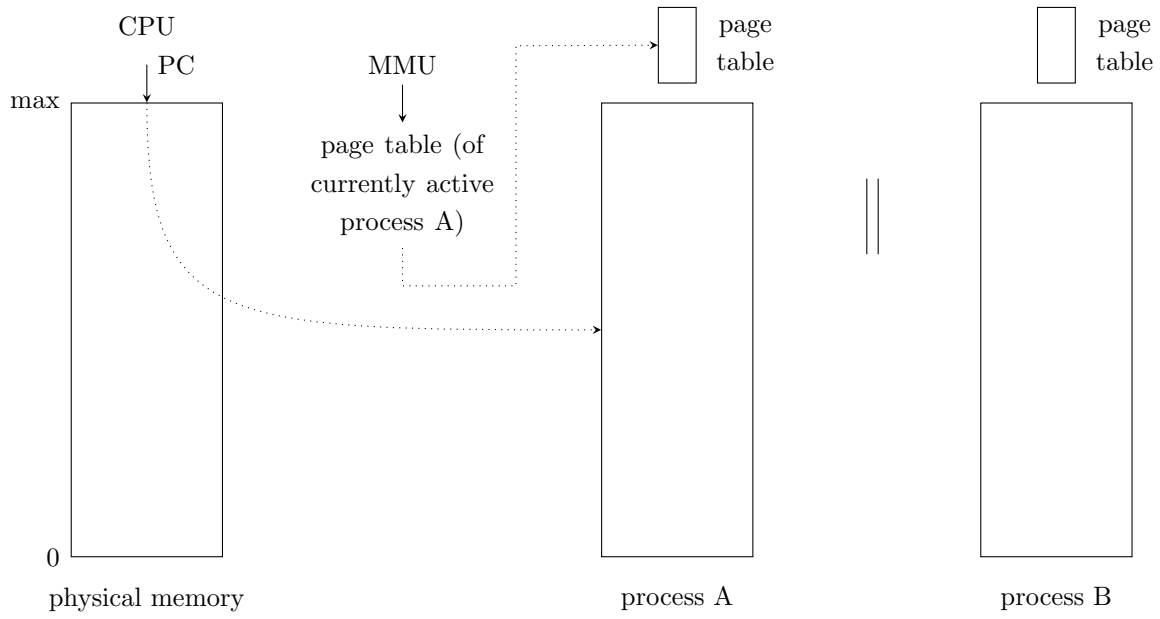


Figure 38: Page tables of concurrent processes: process A is active

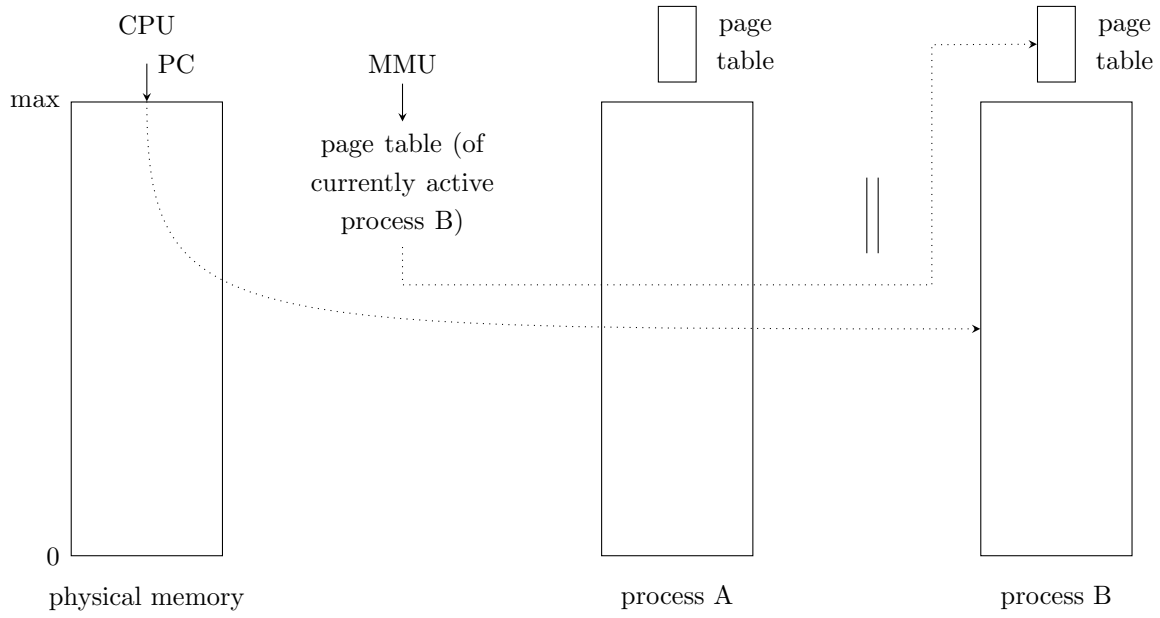


Figure 39: Page tables of concurrent processes: process B is active

Kernel vs. User space

A perfect operating system cannot crash because of a malicious user program. Is this possible? Yes, in 2009 *sel4* (*secure L4 μKernel*) proved this behavior.

The startup of an operating system is privileged. There exist two options to switch from user to privileged mode (in general):

1. User program is finished & executes a privileged instruction
2. A system call is executed by a user program

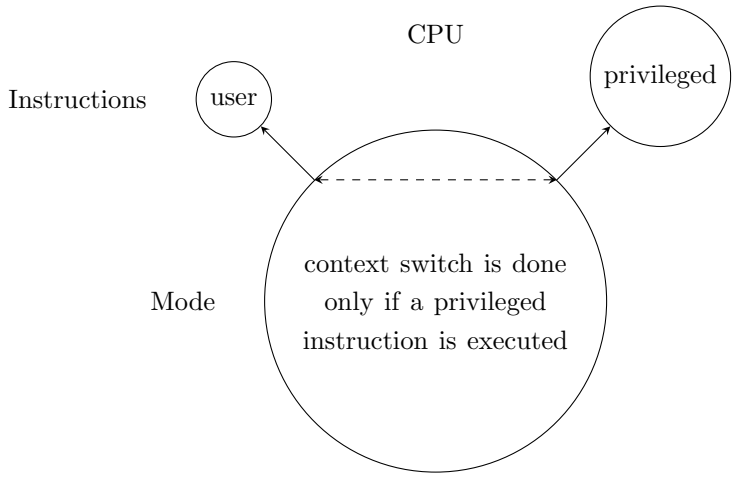


Figure 40: User and privileged mode

Implementing an operating system is a complex task because operating systems are self-referential (*bootstrapping problem* ⇒ kernel page table is appended on top of user page table).

Operating systems do not run concurrently when a user process is executed (a CPU executes instructions one-by-one). Thus the operating system has no control when an instruction of a user process is executed and has to get back the control somehow. There are two options to accomplish this:

1. Cooperative multitasking:
   E.g. Win 3.11 or Mac OS 9. Easier to implement but there is no guarantee that the control is ever given back to the operating system as the processes have to give back the control voluntarily.
2. Preemptive multitasking:
   Hardware clock parallel to CPU ⇒ this clock tells the CPU to switch the program counter. The operating system programs the clock, so it is guaranteed that the operating system gets back in control. Difficult to implement because there can be switches within two instructions ⇒ profoundly concurrent.

Recapitulation: memory of concurrent processes

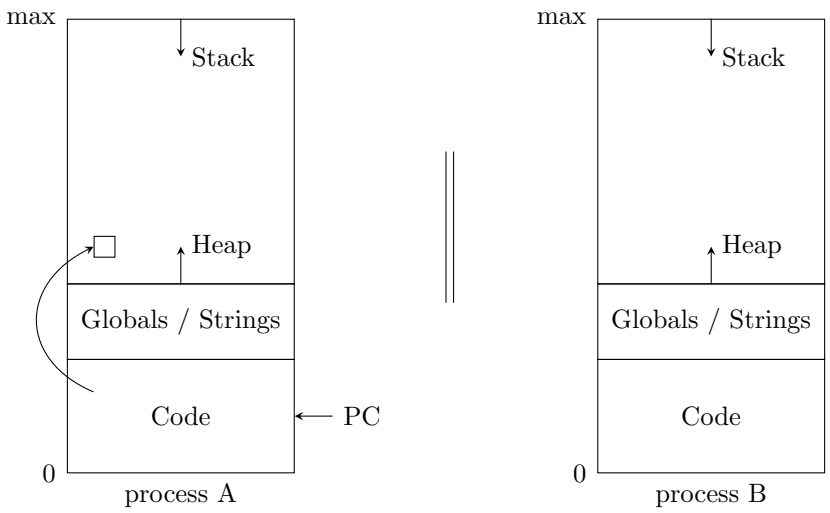


Figure 41: Concurrent processes and their memory layout

Fundamental reason for parallelism/concurrency (parallelism ≠ concurrency!):
Because the world we are living in consists of lots of parallel processes ⇒ real world picture; it is not about convenience! It is also about correctness proofs ⇒ to prove that one big program is correct is much more difficult to show than multiple concurrent programs.
The ultimate prerequisite to accomplish this: each process has to think he owns the whole machine & memory.

Physical memory is:

1. a set of addresses
2. content of these

Virtual memory is:

1. a set of addresses

Paging: Split virtual and physical memory into 4kB pages (of addresses).

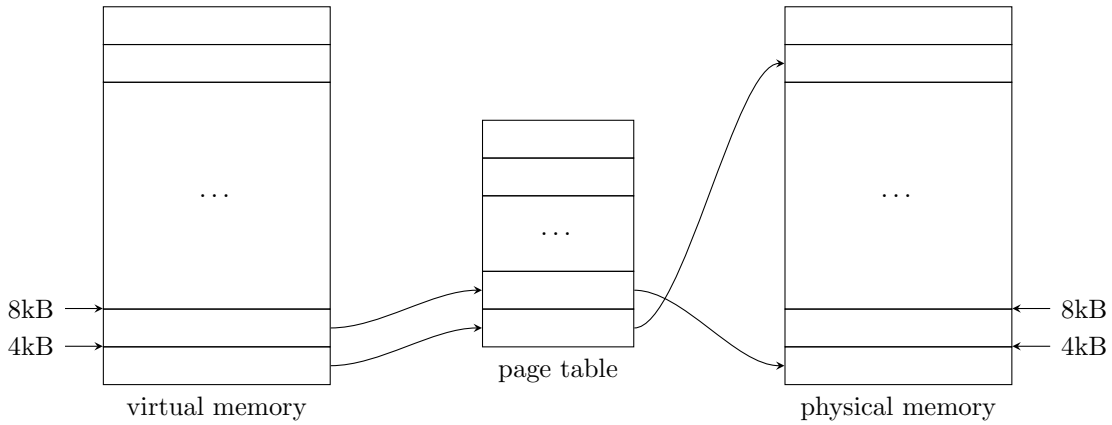


Figure 42: Mapping virtual ↔ physical memory

Page tables are process-local (each process has its own). On process creation the page table is empty (there is are no mappings of any virtual to any physical addresses). Page frames are organised by a free list ⇒ $O(1)$ access and no additional space needed (no overhead) because the pointers of the free list can be stored in the free page frames.

When a page fault occurs:

- create mapping → resolved by MMU → good performance needed because there are lots of requests → page table is cached into translation lookaside buffer (TLB) → is in RAM (not everything is split into 4kB chunks).
- if it is in the code segment: load 4kB from disk into frame.

CR3 register in MMU points to the currently used page table and is used when a process switch is done:

- set program counter
- restore state of new process (registers, et cetera)

Translation lookaside buffer (TLB):
Is a fully-associative memory: entry is split into two columns: content & address (see Figure 43). Has about 20 entries for $\frac{2^{32}}{4kB(\text{page size})} = \frac{2^{32}}{2^{12}} = 2^{20}$ page table entries. Thus, there is no one-to-one mapping because this would lead to as much virtual as physical addresses ⇒ no external fragmentation because of constant page size (4kB). Three ingredients of external fragmentation:

1. out of order deallocation
2. contiguity
3. different page sizes

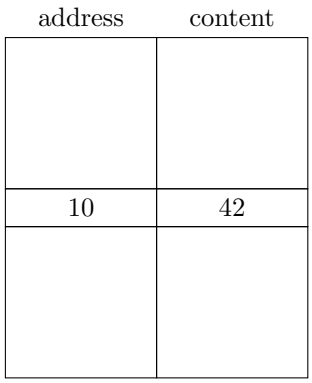


Figure 43: TLB organisation

The operating system asks the TLB if it got address 10. But why in MMU/page table?

- cache hit: mapping in TLB.
- cache miss: mapping is looked up in page table ⇒ slow and is stored in TLB after lookup.

Fully-associative vs. direct-mapped cache:
In fully-associative caches, when a request is made, the requested address is compared in a directory against all entries in the directory. If the requested address is found (*directory hit*), the corresponding location in the cache is fetched and returned to the processor. Otherwise, a *directory miss* occurs.
In direct-mapped caches, lower order line address bits are used to access the directory. Since multiple line addresses map into the same location in the cache directory, the upper line address bits must be compared with the directory address to ensure a directory hit. If a comparison is not valid, the result is a cache miss. The address given to the cache by the processor actually is subdivided into several pieces, each of which has a different role in accessing data.

On-demand paging:

The straight-forward approach to implement a page table would be a gigantic array. But this could not be stored in memory for every process. The solution have (more or less) constant time and solves the memory problem: a tree structure where the root has children of mini page tables (see Figure 37).

Prefetching:

Fetch neighbor entries in page table ⇒ exploits the *spatial locality* of code/applications.

Spatial locality in caches: a whole cache line is loaded. Because of loops and contiguous allocated data structures (arrays, . . .).

Temporal locality: it is likely that a cached address is needed in the near future again. This is the reason caches work. There are almost no programs without this property because of loops, variable sharing and functions/methods.

We have perfect spatial isolation of processes. How do we establish interprocess communication?
Shared memory: different page table entries of different page tables map to the same memory.

Process vs. Thread

- Process:
  - independent program counter
  - independent registers
  - independent virtual memory
- Thread:
  - independent program counter
  - independent registers
  - independent stack, everything else is shared

Communication between threads is very fast because everything is shared except for the stack. This bad isolation is the reason that most multi-threaded applications are incorrect. The correctness proof of multi-threaded applications is very difficult.

2 categories of page replacement algorithms:

1. frequency-based
2. based on the point of time of the access

Thrashing:

All processes operating on more memory than physically available ⇒ swap-out/-in all the time.

Scheduling problem:

n processes & all want to run. Fairness is important (no starvation of a single process), e.g. Round Robin, FIFO.

Scheduler does the process switch. Processes switch their state between *ready* and *blocked* (in the simplest model on a single core system). Ready processes are in a running pool (unsorted set of ready processes, in general).

Preemptive:

Triggered using hardware interrupts, otherwise the operating system would never get the CPU back from the process because the running process owns the CPU. Fundamental drawback: every code can be interrupted at any point of time ⇒ correctness reasoning is difficult because of the complexity.

Cooperative:

Easier to implement. Fundamental drawback: if one process is not cooperative, the whole concept is broken.

Multithreading:

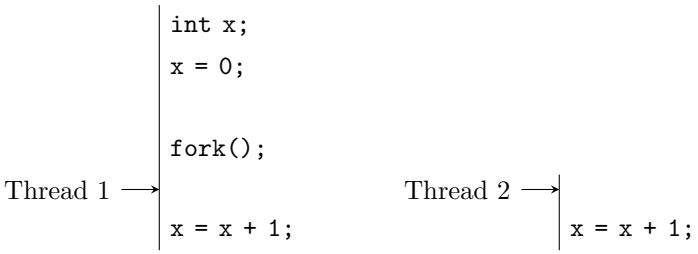


Figure 44: Possible interference when working with multiple threads.

Figure 44 shows two executing threads (with the same code of course). Which value has `x`? You cannot know because of possible race conditions.

Implementation of the incrementation of `x`:

```
LDW 1, 28, -4 // load x into r1, PCs of thread 1 and 2 are here!
ADD 2, 0, 1 // load value 1 into r2
ADD 3, 1, 2 // add r2 to r1 (increment x by 1) and store result in r3
STW 3, 28, -4 // store r3 (incremented x) back to memory
```

The program counter of thread 1, denoted by pc_{T1}, comes first, then a context switch occurs (hardware interrupt). Because the registers are stored (and restored after the process executes again), the value of `x` in thread 1 is not lost after a context switch.
Atomic instructions/code parts enable concurrency and the reasoning about the correctness of concurrent applications.
Disable interrupts globally before a code part is executed you do not want to be interrupted.
Drawbacks:

- This technique does not work in multicore systems.
- When the interrupt is not enabled again afterwards, the system is not cooperative anymore.

We would need a single instruction which reads and writes a variable `m` (mutex) dependend on the current value of `m`. ⇒ *Test & Set* paradigm (special machine instruction).

```
// these two lines have to be executed atomically!
while(m != 0);
m = 1;

// critical section

m = 0;
```

`m` . . . Lock acquire/release.

- *spin lock*
- busy waiting (until the interrupt occurs)

Mutex are not fault tolerant! If `m` is not reset by another thread, the busy waiting loop burns CPU time forever. So the number of instructions to mutual exclude determines how long the other threads wait busy.

Better solution (without a busy waiting loop): Traffic light model

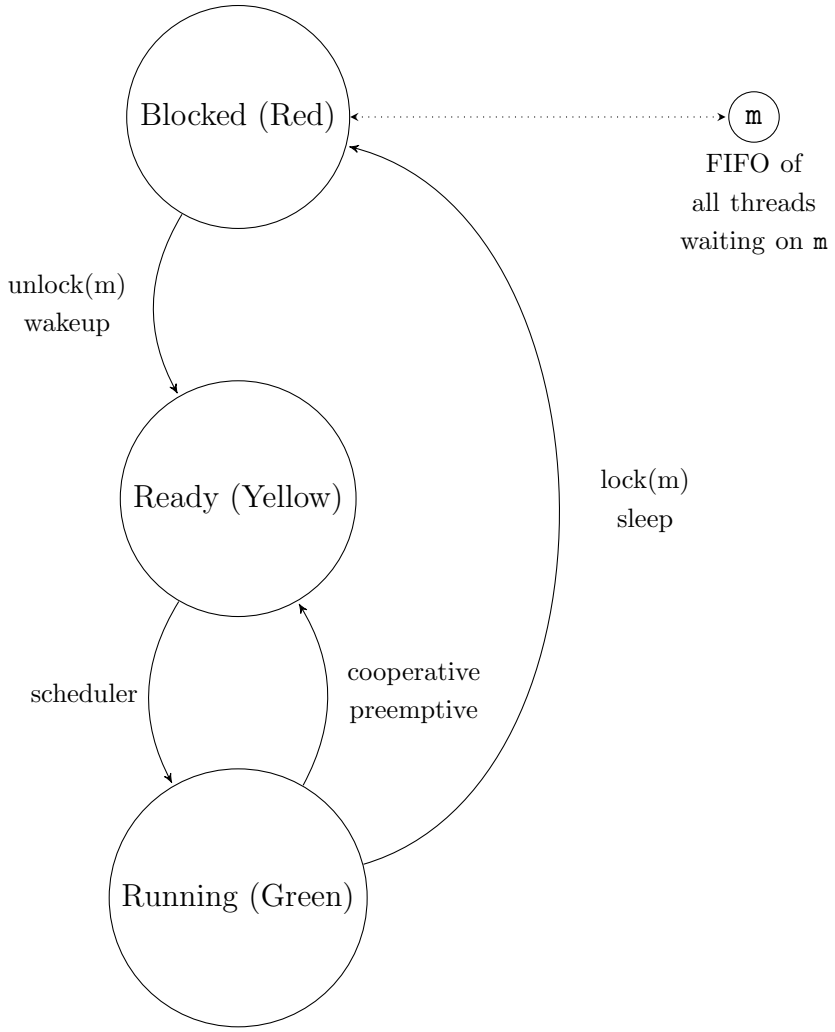


Figure 45: Traffic light model

All threads from the *blocked* list are stuffed into the *ready* queue. For n threads where 1 holds the lock, we have $(n-1)$ comparisons of `m`.

```
lock(m);

// critical section

unlock(m);
```

Thundering herd effect:

One of the ready threads gets the lock next, all other threads are again in the blocking

queue. So only a single thread is (in a fair manner) moved from the blocking to the ready queue.

If the ready queue is empty, there is a special instruction (*sleep*) which stops the CPU in hardware.

The *sleep* mechanism:

When `sleep(500)` is called, the current thread has to sleep for 500ms. Therefore the current CPU ticks + 500ms are stored in a thread-local variable (e.g. `int64_t wake_me_up_at`). Each time the timer interrupt is triggered, the current CPU ticks are compared to the value of this variable. If the current CPU ticks is greater or equal than the value of this variable, the corresponding thread is moved to the ready queue (woken up). All sleeping thread are organized in a sleep list, ordered ascending by the value of the `wake_me_up_at` variable. This reduces the comparison effort because the operating system does not need to check the whole list but only the first k threads in the sleep list (those where the value of the `wake_me_up_at` variable is less or equal the current ticks).

`sleep(500ms)` gives only a lower bound of 500ms but no upper bound. This means that the thread sleep at least 500ms but may sleep longer (see Figure 46).

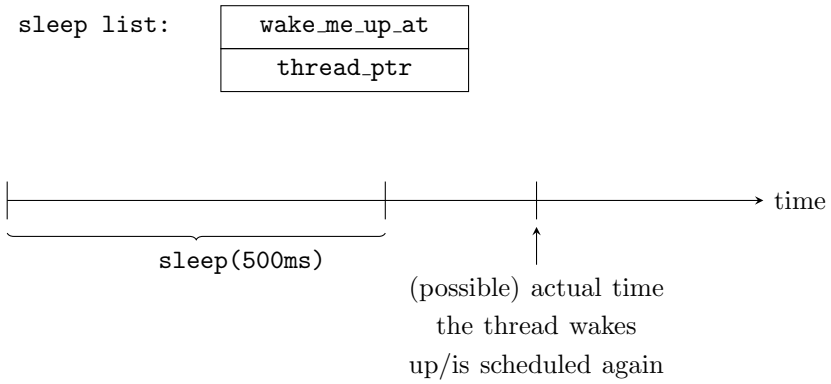


Figure 46: The *sleep* mechanism.

The *idle* process:

Needed because the ready set/queue should never be empty.

```
while(true)
  hlt; // special halt instruction to stop the CPU
```

Time slice dimensioning:

Nowadays time slices of about 10ms are normal. Tradeoff between bigger and smaller time slices:

- Bigger time slices: semantics of sleep call gets absurd.
- Smaller time slices: precision of sleep call better but the check needs to be done more often (energy inefficient). If the time slice is too short the operating system may even interrupt itself while checking the `wake_me_up_at` variable.

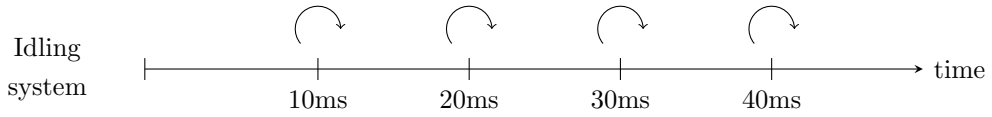


Figure 47: 10ms time slices.

Atomic instructions:

Increment (`x = x + 1;`)/Decrement (`x = x - 1;`) have to be atomic. Three ways to accomplish this atomicity:

1. *compare & swap* (CAS)
2. *load & link* (LL)
3. *store condition* (SC)

The memory bus gets locked (implemented in hardware).

ABA problem:

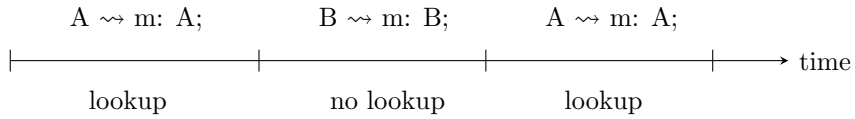


Figure 48: The ABA problem.

Because of the ABA problem, lots of algorithms of the literature were not used any longer.

How to avoid the ABA problem:

32-bit addresses; arbitrary 8 bits as version number; remaining 24 bits contain the data.
This solution does a good job but does not solve the problem completely because the 8 bit used as version number can overflow (already happened in the computational systems group).

Spawn a process (C-like syntax & semantics):

```
int pid;

// make a new process; pid is the process identifier
pid = fork();

if(pid < 0)
  // error
else if(pid > 0)
  // I am the parent process
else
  // I am the child process
```

Wait for a process (C-like syntax & semantics):

```
int pid;
int status;

// wait only returns when a child process terminates
pid = wait(&status);

if(pid <= 0)
  // error or there are not awaited child processes
else
  // the child process pid has terminated with exit status
```

The pendant to `wait` is the `exit` system call:

```
int status = 0;
exit(status);
```

The consequence of using `wait`, `exit` and `fork` is the possible generation of *zombie* processes.
If the parent processes exit, these are called *orphans*. The root node adopts all orphans in the hierarchy.

Booting the operating system:

The BIOS defines how the operating system is loaded into the main memory (RAM). The CPU is a stupid machine which only fetches, decodes and executes instructions one-by-one. At some point of time, `fork` has to be called but there is not infrastructure available.
In the bootstraping process, everything is set up by hand (ready queue, et cetera). The *init* process has pid 1, the *idle* process has pid 0.

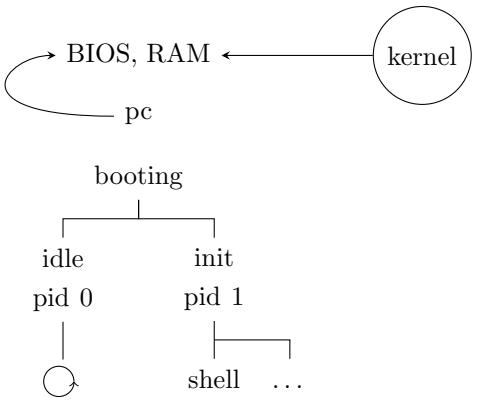


Figure 49: The boot process.

The `init` process is the parent of all other processes (except the `idle` process which loops infinitely). When an application is called using the shell, e.g. >`ls`, the call is parsed.
How can the first process be spawned? >`ls&` (this is the first fork).

Virtualization:

- system virtualization, e.g. XEN, KVM, ...
- process virtualization, e.g. JVM, ...

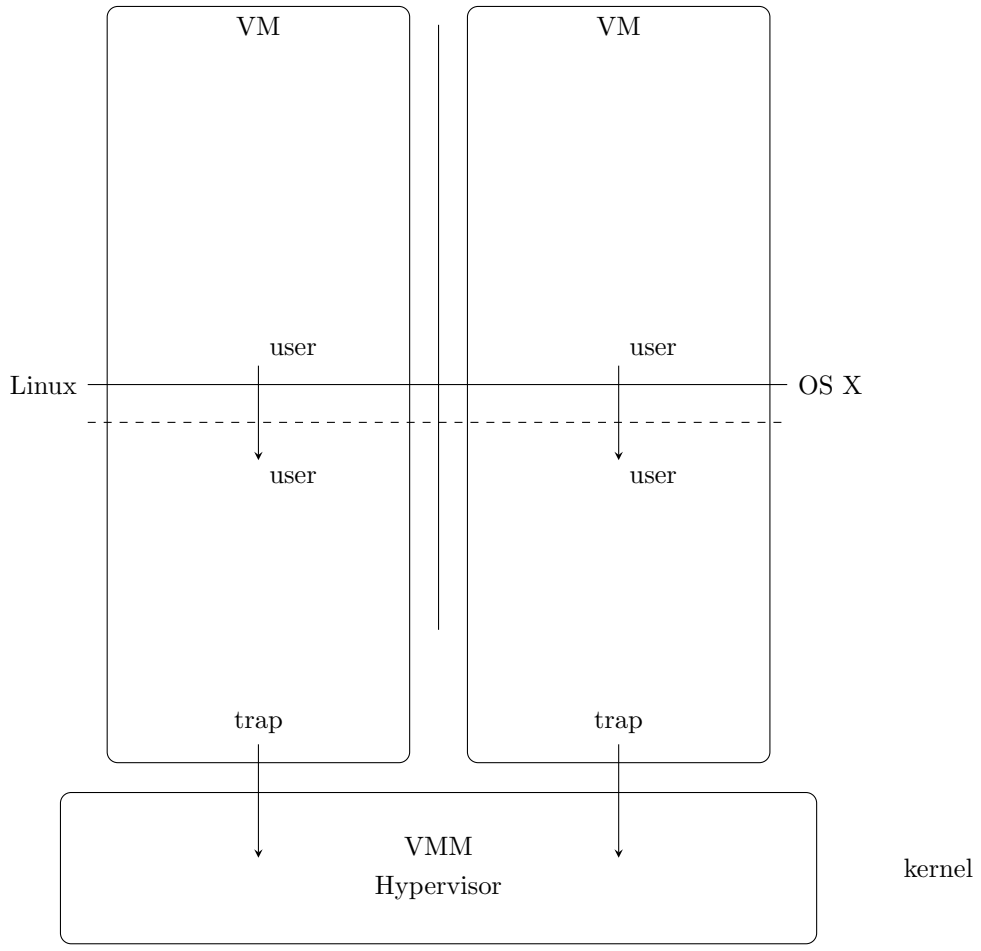


Figure 50: Virtualization.

VMM ... Virtual Machine Monitor

Self-compiling compiler:

C^* compiler written in C^*: `cstar.c`.

>`gcc cstar.c` $\rightsquigarrow$ `cstar` (for the x86 architecture)
>`cstar cstar.c` $\rightsquigarrow$ `cstar1` (DLX, `cstar` $\neq$ `cstar1`)
>`cstar1 cstar.c` $\rightsquigarrow$ `cstar2` (DLX, `cstar1` $=$ `cstar2`)

Accessing an array:

$addr(a[i]) = addr(a) + i \cdot sizeof(elem)$.

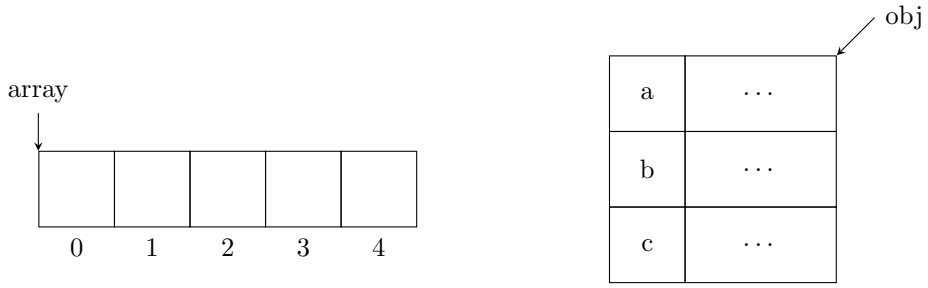


Figure 51: Accessing an element of an array.