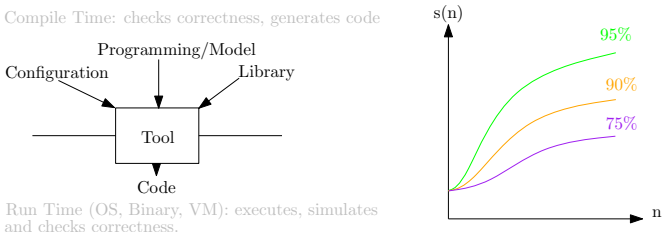
Intro to Operating Systems

1 Intro

Programming: establish functional correctness and adequate performance. Different languages provide different tools for automating this task.

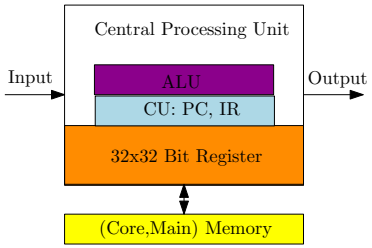


Multi-Core/Multithreading: Speedup $S(n) = \frac{1}{(1-p)+\frac{p}{n}}$ where n is the number of cores and p the degree of program parallelism. Sequentially is not a property of program but a property of program execution. Program execution is a highly non-linear property that depends on all aspects of a system (hardware and software). Real Time Programming needs predictability, not speed.

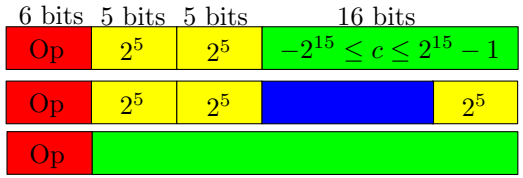
Hardware	Architecture	Computerability Algorithms Complexity	Semantics	Data Structures	Memory Management	Concurrency	Virtualization

2 Architecture

Defines functional and non-functional semantics of a software system. von Neumann Architecture: (1) Fetch, (2) Decode, (3) Execute.

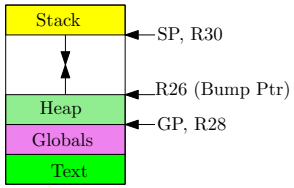


Current Instruction resides in the Instruction Record (IR) Register. The Program Counter (PC) points to the next instruction in main memory. Memory is byte-addressed and word-aligned. Syntax of instructions:



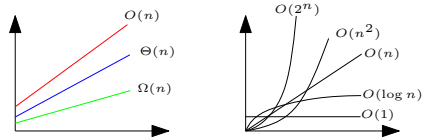
ADDI a,b,c	reg[a]=reg[b]+c, pc+=4
SUBI a,b,c	reg[a]=reg[b]-c,pc+=4
MULI a,b,c	reg[a]=reg[b]*c,pc+=4
MODI a,b,c	reg[a]=reg[b]%c,pc+=4
CMPI a,b,c	reg[a]=reg[b]-c,pc+=4
ADD a, b, c	reg[a]=reg[b]+reg[c],pc+=4
SUB a, b, c	reg[a]=reg[b]-reg[c],pc+=4
MUL a, b, c	reg[a]=reg[b]*reg[c],pc+=4
MOD a, b, c	reg[a]=reg[b]%reg[c],pc+=4
CMP a, b, c	reg[a]=reg[b]-reg[c],pc+=4
LDW a,b,c	reg[a]=mem[(reg[b]+c)/4],pc+=4
STW a,b,c	mem[(reg[b]+c)/4]=reg[a],pc+=4
POP a,b,c	reg[a]=mem[reg[b]/4],reg[b]=reg[b]+c
PSH a,b,c	reg[b]=reg[b]-c,mem[reg[b]/4]=reg[a]
BEQ a,c	if(reg[a]==0) pc=pc+c*4 else pc=+4
BR c	pc=pc+c*4
BSR c	reg[31]=pc+4,pc=c*4+pc
RET c	pc=reg[c]
JSR c	reg[31]=pc+4,pc=c

Memory Layout: R30 is the stack pointer, R26 the Pointer for dynamic allocated data (Heap Bump Pointer). R28 is the global Pointer. R27 is used as Return Register and R31 as Link Register.

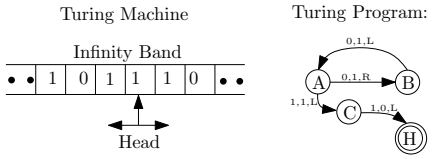


3 Computability & Complexity

Computability: Explores the boundaries of computability.
Complexity: Explores the cost of solving problems and executing algorithms.
Instruction Sets: CISC (x86/64, i386), RISC (ARM, Sparc), OISC.... 1 Instruction is enough to compute everything: $mem[b] = mem[b] - mem[a]$, if $mem[b] \leq 0$ goto c.
$f(n) \in O(g(n)) \Leftrightarrow \exists c > 0 \exists n_0 \forall n \geq n_0 \quad f(n) \leq cg(n).$
Other classes: $\Theta(n)$, $\Omega(n)$.



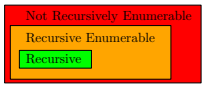
A numeric algorithm runs in pseudo-polynomial time if it runs in polynomial time in the numeric value of its input. A simple, abstract machine model is the *Turing Machine*. A Universal Turing Machine simulates a Turing Machine.



Church-Turing-These: If there is a program/algorithm that computes a function, then the same function can be computed with a Turing Machine Program.
Turing: There is no algorithm that computes whether an arbitrary program halts on a given input (Rice: generalization).
Proof-Sketch:

- Let h be $h(i,x) = \begin{cases} 1 & \text{program i halts on x} \\ 0 & \text{otherwise} \end{cases}$
- Define $g(i) = \begin{cases} 0 & \text{if f(i,i)=0} \\ \text{undefined} & \text{otherwise} \end{cases}$
- g is partial but may be computable
- if $f(e,e) = 0 \Rightarrow g(e) = 0$ then $h(e,e) = 1$.
  if $f(e,e) = 1 \Rightarrow g(e) = undefined$, then $h(e,e) = 0$.

The set of programs that do not halt is much larger then the set of programs that may halt or halt surly. **Recursive Enumerable**: accepted by a Turing Machine, but may not halt. **Recursive:** Accepted, and the Turing Machine halts.



Hardness: asymptotic complexity says something about some requirements (growth) but nothing about how difficult a problem is.
$\mathcal{P}$-class of problems is solvable in polynomial time. $\mathcal{NP}$ is solvable with Not Deterministic Turing Machine (NTM) or verifiable in polynomial time with a certificate (upper bound). $Q \in \mathcal{NP}$-hard if $\forall Q' \in \mathcal{NP}$, there is a reduction $Q' \leq Q$ (lower bound). $Q \in \mathcal{NP}$-complete if $Q \in \mathcal{NP}$. $SAT \in \mathcal{NP}$-complete. SAT is the problem of determining if a boolean expression is solvable.

4 Semantics

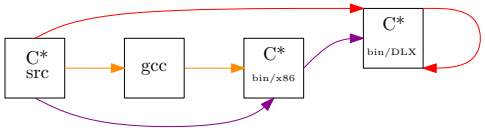
Semantics of a programming language defines the behavior of programs written in that language. Correct behavior only if fully understood. Semantic determined by architecture & tool to generate code.
Syntax provides structured information that is semantically relevant. Syntax for programming can be specified non-ambiguously and checked efficiently.

```
exp    = ['-'] term {('+'|'-')term}
term   = factor {('*'|'/')factor}
factor = identifier|integer|'('identifier')'
ident  = letter{letter|digit}
letter = 'a'|'b'|...|'z'
digit  = '0'|'1'|...|'9'
integer= digit{digit}
```

Code:

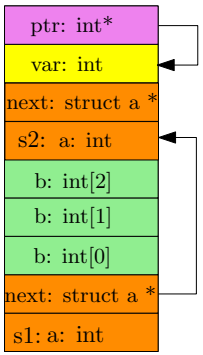
```
exp() {
  if (SYM == MINUS)
    getSymbol();
  term()
  while (SYM == MINUS || SYM == PLUS) {
    getSymbol();
    term();
  }
}
```

Bootstrapping: start with a tiny tiny language. Global Variables are located in a symbol table with the offset. Local variables (within functions) are located in another symbol table, with an offset on the stack (relative to the frame pointer).
If a symbol is not seen by now, put it into the fix-up-list and if the symbol is seen later, than fill in the correct address. If a symbol is not seen at the end, then throw an error. Recursive function calls are possible: $f(f(f(x)))$.



5 Data Structures

Provides structures in an otherwise unstructured linear address space. Goals: abstraction (tables, lists, graphs) and safety. Allows type checking which in turn helps to establish functional correctness (compile time: type checking, runtime: range checking). GC can replace virtual memory. A type system is the single most successful technology to ensure correctness.
Basic Data Types: `int var`, `int* ptr`.
Comp: `int b[3]`, `struct a{int a, struct a *next} s1,s2`.



Array	$O(1)$ access	fragmentation, runtime, bounded size, indexed access
Record	$O(1)$ access	memory fragmentation, compile time bounded, field access only
List	$O(1)$ head/tail, no fragmentation	$O(n)$ access time
Tree	$O(\log n)$ access, no fragmentation	$O(n)$ if unbalanced, 2x space of list
Hash	$O(1)$ access w/o collisions	memory fragmentation, worst case complexity

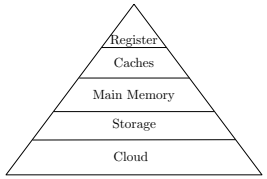
An Object is data (record) with functions/methods. For object oriented, you need

- polymorphism: class inheritance
- procedure polymorphism (method overwriting with dynamic linking during runtime, method overloading at compile or runtime)

Data Structures help for proper typechecking, memory safety and correctness. Java has a strong type system, compile and runtime checking and a Garbage Collector.

6 Memory Management

Volatility and Throughput is small at the top, capacity and latency high at the bottom:



Memory Management provides

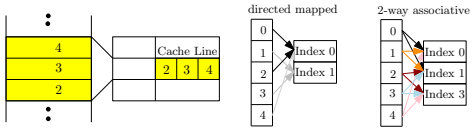
1. efficient access: low memory consumption
2. safe access: what can be freed (dangling pointers)
3. no leaks: what must be freed (never freeing is safe but leaks)

Most programs exhibit temporal & spatial locality.

temporal	access of a given location is likely followed by access of the same location in the near future (cache).
spatial	access of a given location is likely followed by access of a location nearby in the near future (ondemand loading).

6.1 Caches

Direct Mapped: Each memory address can only be in one cache location (fast search, high miss rate, needs more space).
Associative: Each memory address can be in any cache location (slow search, but high hit rate, saves space).



False Sharing: Thread A alters 2, Thread B alters 4 and so, the cache-line has to be updated/reread from main memory.

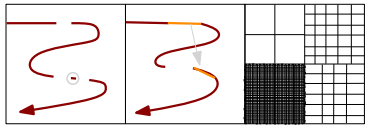
6.2 Fragmentation

Memory is allocated in contiguous chunks of different sizes and deallocated in an order different from the allocation order. Fragmentation exists because of

- allocation of different sizes
- order of allocation is not the order of deallocation
- contiguous allocation

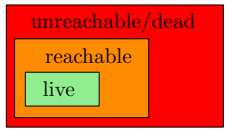
Fragmentation: the sum is greater than what is needed next but then maybe no chuck large enough to accommodate the next request.

- coalescing: non-moving but no bounds. Coalescing is the act of merging two adjacent free blocks.
- compaction: bounds but moving. Compaction is the process of moving live objects to eliminate dead space between them.
- partitioning: large bounds and upper limit.

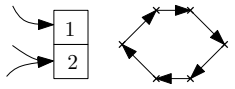


6.3 Garbage Collector

Automatically, decide what is used and what not:

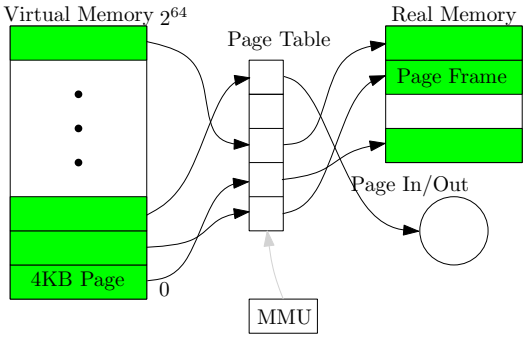


Tracing GC: compute reachable set.
Reference Counting: compute unreachable set.
Avoid unsafe deallocation and dangling pointers. GC suffers from reachable memory leaks. Spatial Isolation with Java: requires types, type checking and GC. Virtual Memory: requires hardware for fast access and OS/MMU.

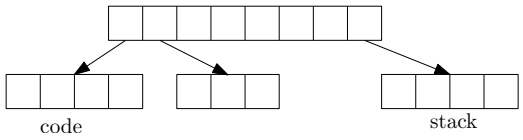


6.4 Virtual Memory

Virtual Address Space are just addresses which eventually map to real memory (address and value) through a page table. The CPU has a unit, the memory management unit, which points to a page table from a process. Every process has a page table, i.e. the process is a page table. The translation lookaside buffer (TLB) speeds up the translation from virtual to real addresses.



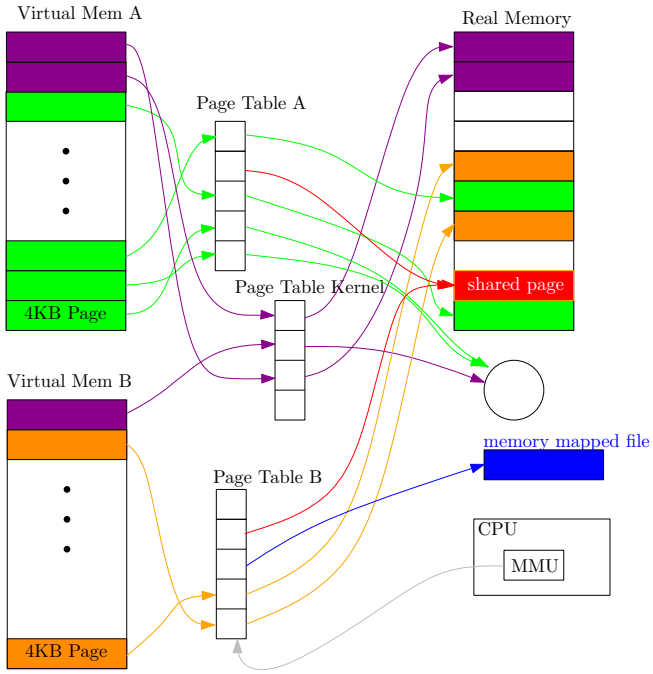
page-size is between 4KB-2MB. Bigger pages have faster address translation since the TLB is better utilized.
page-fault: access to a memory cell that is not mapped into real memory (prefetching & ondemand).
page-replacement: if all pages are full, replace 1 page. Find and evict page frame that will be not used furthest in the future (LRU, MRU, Aging, Clock).
(1) Put PC at the first instruction, (2) if page not available, page fault handler from the operating system, (3) jump back to the instruction. Every process has its own page table. The size of a page table is large, and therefore, a good data structure is needed. The MMU too, must understand this data structure: $2^{64}/\underbrace{2^{12}}_{pagesize} = 2^{52}$.



OS creates new page table for a process, then, at the beginning, a lot of page faults will occur because nothing is loaded into real memory.

7 Concurrency

With the help of virtual address spaces, it is possible to load more than one program at the same time. To switch between processes, the MMU has to point to the page table from the process.



7.1 Kernel vs Userspace

No interference between processes. This implies, that the MMU Ptr can't be modified from user processes. If a user process tries that, then the process must be killed from the OS. User Processes run in usermode and have no access to privileged instructions. Transition to the OS is needed, to modify MMU Ptr (kernelmode)
Cooperative: User process calls OS.
Preemptive: Clock & Interrupt to jump back to the operating system and to schedule processes.
Operating Systems are difficult because they have a bootstrapping problem (there is nothing, how can you do things like process management?) and are self referential.
Why multiprogramming ie more than one processes? Because our world is highly concurrent (networks, 3D games, on and on).

```
fork() {
  create page table
  initialize pc to first instruction
  start process
  many page faults ...
}
```

7.2 Threads vs Processes

A thread is a lightweight process and lives in the address space of a process.

Process	PC, Registers, VirtualMemory (Text, Stack, Heap)
Thread	PC, Register, Stack

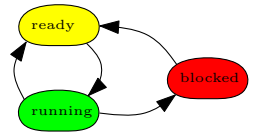
n processes, how to schedule them? Fairness with RoundRobin?

7.3 Synchronization

Consider the parallel execution of the addition in line 3:

`int x = 0;`
`fork();`
`x=x+1`
	`LDW 1,28,-4`
	`ADDI 2,0,1`
	`ADD 3,1,2`
	`STW 3,28,-4`

The result may be $x = 1$ or $x = 2$, depending on the scheduling of both threads/processes. If Thread a runs to the end, the result will be 2. What is needed is a form of atomizing. One possible solution is to deactivate interrupts (no other process can interrupt till the process turns interrupts on again). What, if interrupts won't be started again.
Another way is a *lock*. There exists assembly instructions, called Test and Set to check if a variable is 0 and set it if so. This can be used as lock.
Busy Waiting: Wait in a while-loop and do nothing (NOP).
Blocking: Put process into a kernel list and do not burn down cpu cycles during waiting. The OS wakes up if the lock is free again and puts the process into the ready queue:



If m processes wants a resource M, then $m-1$ tests are needed. If no process is in the ready queue, then a special process will be started: `IDLE`.
The *sleep* call puts a process into another queue and puts this process into ready, if the time is up. This may not match exactly the seconds given as parameter. Usually, the time slice is 10ms.

```
int pid = fork();
if (pid < 0) // error;
else if(pid == 0) // child
else // parent
```

Orphan: A child process whose parent died.
Zombie: A child process whose parent is sleeping/not read.

7.4 Bootstrapping