TOC

04–10–18

Meta

- **To Do**
    - Learn base 2 arithmetic - powers of 2 up to 2^{32}
- keine Hausuebungen - Material regelmaessig anschauen
- fuer heutigen Termin nur Organisation und Motivation

Folgender Text auf Englisch - VO wird mal auf Englisch, mal auf Deutsch gehalten.

Motivation

- mentions article on a microchip (https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies) [12–10–18 Update: This story appears to have some credibility issues]
- brings in 19" server - might contain above chip
- Def.: OS - “Software that runs on a simple machine and creates the illusion of having 1000s of these machines”
- Goal: Developing “thinking tools” that allow contemplating complex processes
    - Understanding how computers execute parralel processes
        - requires understanding hardware; we’re focusing on this for now
- **Von Neumann Architecture** - key idea: both program code and program data sit in main memory
    - CPU and RAM communicate via memory bus
    - GPUs are not built according to Von Neumann
        - work with another execution model
        - have same capabilites as CPU, only difference is speed - GPUs are not needed to be complete/universal, only needed for speed
    - All modern machines work this way fundamentally
- Will see machines with persistent RAM in the future - flash and RAM will merge
    - Has huge effects on OSs
    - Problem: Cannot reset RAM. Things will age, problems never go away - this is one problem in OSs rn
    - Hardware changes have huge consequences on software, problems never end
- What happens when the machine is turned on and RAM is empty?
    - CPU is “completely stupid”, only knows to fetch an instruction from somewhere and execute it
        - So is memory - just “sits there and contains bits”
        - Individual steps are incredibly simple, raw amount of power is the reason computers can do anything interesting
        - “0 intelligence in this”
        - CPU can never stop executing instructions - **synchronous circuit**; there is never no “next instruction” until “turn off power” instruction is hit

- But how to get first instructions into main memory on bootup?
    - Bootloader. Sits in BIOS chip, instructs CPU to get OS code from SSD to RAM; then connection to BIOS chip is cut and OS code takes over
        - Can update bootloader/firmware once OS loaded
    - This process is called **bootstrapping**
        - Term comes from some story about a guy pulling himself out of mud with his bootstraps - Baron Muenchhausen

- Bootstrapping happens often in CS, this is because systems are **incomplete**
- CPU components:
    - Registers - tiny memory just like RAM, but by far the fastest
    - Arithmetic Logic Units implement integer arithmetic and bitwise operations
        - they're hard to make, other ways of doing the same thing exist; however consensus is that integer arithmetic is the best way atm
    - ALUs perform these operations on the values stored in the registers
    - separate Program Counter register tells CPU where to look for code in memory.
        - Memory is **byte addressed** (that means in 8 bit steps)
            - important concept: stuff is not only stored in binary but it is also used to address memory - can use data not only as data but also as *pointers* to data
            - this is because out memory is **word aligned** meaning we can only copy full words to registers, not individual bytes/bits - 32 or 64 bit chunks are transferred each cycle
                - Length of (Machine) Words depends on CPU - 32bit or 64bit
                - Memory bus has according number of lanes
                - this is because reading a 64 bit word takes 64x longer on a single lane
    - instruction register holds currently loaded instructions
        - instructions are 32 bit, hence w/ byte addresses we load 2 instructions at a time
- Class uses 4GB RAM ($4*2^{30}$), 32 64bit registers model
- **Big Message:** loads the 2 instructions from PC address, executes it, modifies state of the machine in some way, at the end PC changes to some other address (usually next address).
- and… Big Question. Since both code and data is in RAM, how do you differentiate code and data?
    - the OS does. The machine knows nothing.
        - PC is the only way for the machine to know where the code is.
        - PC points to a machine word (half a word in memory) and OS fetches the bits that are in here into the CPU
        - CPU interprets these bits as instruction, when it is done OS tells the CPU where to put the program counter next
        - this could be the next instruction but it could also be somewhere else.
        - Only in the moment when the PC loads the bits it is code. Only in that moment the meaning as a code is given to the bits, after PC moves on the machine may never look back there.
    - Data only becomes data because you look at it and interpret it in certain ways
    - Code you write assigns meaning to data
    - When you add two integers the bits become numbers, after you're done the bits can become anything else.
    - The machine is mindless - where does the meaning of the code come from? - Hardwired into the CPU.

Intro und/oder random Kram

- **To Do**
    - Paper in Slack und Selfie-System auf Maschine goennen
        - git clone … auf Kirsch's Homepage
        - C compiler wird gebraucht (gcc, clang, …) und cmake
        - mit make compilen
        - Shit ausprobieren der im Paper steht
    - Docker anschauen
    - Hex, Dec, Bin, Oct lernen
        - Hex Notation: 0x…
        - Oct Notation: 00…
        - Bin Notation: 0b…

- Notation fuer `usage` output von selfie: **extended Backus Nauv Form** (Backus und Nauv waren Wissenschaftler, Extended durch Niklas (?) Wirth - der Dude hat Pascal erfunden)
    - Parameter innerhalb {} braces - darf unendlich oft auftauchen (*repetition operators*)
    - Parameter innerhalb [] brackets - 1x oder 0x (*optional*)
    - | bedeutet oder
    - ' ' (space) bedeutet und
    - parenthesis gruppieren
- ./ vor selfie ist noetig aus Sicherheitsgrunden, man kann auch selfie's directory zu $PATH hinzufuegen, wird dann nicht mehr gebraucht
- Sinn von selfie - kann man komplett verstehen, anders als gcc etc
    - Selfie ist turingvollstaendig
    - Hilft, alle moeglichen Instruktionen zu verstehen, die es auf modernen Maschinen gibt
    - wird in dieser VO genutzt um Beispiele zu illustrieren
    - alles in 1 File geschrieben weil's cool ist
        - Simplicity ist ein Weg, hervorzustehen
- CPU kennt die Semantik der Instruktionen nur durch Hardware, d.h. komplettes Verhalten ist in Konstruktion der Prozessors (in den Schaltkreisen) kodiert
- Begriffserklaerung Vorlesung: kommt aus einer Zeit, wo sich nur der Prof ein Buch leisten konnte, weil die zu teuer waren - Prof hat dann einfach vorgelesen
- **Kirsch's Toolchain.**
    - iTerm2
    - zsh (super cool) - hat themes etc
        - Terminal vs. Shell - Terminal zeigt den ganzen Shit bloss an - commands werden von Shell interpretiert
    - Sublime Text

Computerarchitektur Fortsetzung

- Nachteil von Von Neumann: Security - Daten koennen versehentlich als Code gelesen werden. Weiters - alles haengt von Datenbus zw. RAM und CPU ab. Der Datenbus ist also **Von Neumann Bottleneck** - bestimmt ultimativ Maximalgeschwindigkeit von dem ganzen Ding
- CPUs haben immer dieselbe Loop:
    1. `fetch` - naechste Instruktion holen

2. `decode` - Instruktion dekodieren: was soll tatsaechlich getan werden?
3. `execute` - Instruktion ausfuehren
4. `GOTO 1`, mit ein paar GHz Takt ad infinitum bis der Strom weg ist

- Selfie's Maschinenmodell ist zsm mit Instruktionen in riscu.md auf Github zusammengefasst - e.g. `lui $rd,imm: $rd = imm * 2^12; $pc = $pc + 4 with -2^19 <= imm < 2^19` - nach dem `:` ist die Bedeuting erklaert
- in unserem Maschinenmodell wird Overflow weggeschmissen

Selfie Instruktionen

Instruktion	Bedeutung	Effekt
lui	load upper immediate	20 bit immediate int wird mit 12-bit left shift geladen (wir haben 64bit register, 20 bit int steht zw. Bit 12 und 32, davod und danach nur Nullen)
addi	add immediate	nimmt 12 bits und schreibt sie in die erste 12 Bits
	Arithmetikoperationen (register addressing)	
add		
sub		
mul		
divu		
remu		
sltu		Testet, ob `$rs1 < $rs2` und setzt `$rd` auf 1 falls true
	Control Flow	
jal	jump and link	Kern von Neumann - Code und Data im gleichen Memory. Code koennte sich selbst modifizieren - Viren (ein Virus ist eine Autofabrik, die Autofabriken baut)
beq	branch if equal	erlaubt Abzweigungen (PC relative addressing) - kann PC direkt manipulieren
	Memory	
load		identifiziert Bereich in Memory-Addressraum ueber ein Register und laedt den Inhalt ins Register
ecall		wird spaeter erklaert, ist die Instruktion, die fuer diese Vorlesung die relevante Instruktion ist - erlaubt, ein OS zu konstruieren

- lui und addi initialisieren zunaechst bits 12–32, dann 0–12 mit irgendwelchen Werten. Wir haben 64 bit: restliche Bits koennen reingeshifted werden
    - erste 2 asm-Befehle, die selfie generiert, sind lui und addi

18–10–2018

Today: How do you link a high level programming language to the hardware world?

Review and introduction

Last Time: Von Neumann Model w/ RISC-V (Reduced Instruction Set Computer). Modern x86 computers use CISC (Complex ISC), phones/tablets (ARM) use RISC

- Differences

    - what makes CISC *complex* and RISC *reduced*?
        - in RISC - fewer instructions that are *semantically atomic* - they do one thing only
        - CISC has 1–2K different instructions, some of which make the CPU do many things (e.g. loading data, performing some action, putting it back to memory in one instruction)
    - RISC has bigger executables - need more instructions to do the same thing
    - RISC compiler is simpler

- CISC came first. First RISC computer in the early 80s or something
- Why did people come up with RISC?
    - Lots of leverage in optimizing compilers - advanced compilers make stuff possible
    - Breaking up instructions allows compilers to optimize the code
    - Responsibility for improvement is moved from hardware to software

High-level languages vs. machine code

The key difference: We consider a technical view; convenience/flexibility are not the main points. May get a different answer depending on who you ask. Prof's ideas listed here:

High level programming language	Low level machine code
Few accessible states	All the states
Structure (of data and control)	no structure

Aim for today is understanding **state**.

State

Machine has memory (RAM, cache, …). We understand a state to be a particular configuration of all the bits the machine can store (a "snapshot"). A program can be written to bring a machine into any state.

This is huge but still a finite number - can be modeled like an automaton. We have some initial state and switch between states with each execution. Is totally deterministic.

We imagine some *state space* holding all possible states. Our aim is to keep the machine in a "good" state and out of "bad" states. Writing code is figuring out a way to stay in the realms of the good states.

High level languages reduce size of the state space dramatically. Code example:

`uint64_t* foo;` declares the name foo, allocates 64 bits for a pointer. We now have a state space of 2^{64} (came down to that from 2^{4GB}). Machine code that implements this has dozens of lines.

For the machine high level code is just a sequence of characters. The compiler reads these characters one at a time and disregards those that are not actual code (whitespace and comments - usually a high percentage of the code, all of this is for convenience). Only *symbols* (actual code) are considered.

e.g.: `foo = foo + 1` Compiler reads this word for word. Sees `foo`, but doesn't know what to do with it - proceeds to = and only then realizes it's an assignment. Reads right hand side to figure out operands etc. Then generates according intructions and proceeds to the next line of code. Selfie translates this assignment into 6 machine instructions.

String vs. string literal: the literal is the actual string. The string is the name of the string/the pointer. e.g. `foo = "hello world"` - the string literal is `hello world`. Only the string literal is relevant in the "machine world".

Selfie

Selfie gives some statistics about the compiled/executed code (most executed instructions etc).

Selfie is a C* compiler that compiles C*. Also implements a simple RISC-V virtual machine that can run itself.

Notes

Assignment: Look at Hello World example and generated machine code. Try to understand how the machine code implements the high-level commands.

Play around with selfie. Look at draft textbook. Try to understand output etc.

May also want to look at Python. Can implement things quickly.

Some comments

- Computers are the most complex machinery humans ever made
- Dozens of computers in cars - hard to program, so weird bugs happen (e.g. trunk opens randomly)
- Rocket science etc knows much better how to produce correct things - CS is young, most people in the field have no clue how to write *correct* code
- Producing the first "real" compiler took ~17 years of manhours in the 50s
- New ideas in science are often discredited, only later recognized as valuable
- *Minified* code removes all of the whitespace and comments. [Obviously useless for Python etc]. Demonstrates that there is no "magic" behind code. Just a string of characters

25–10–2018

Disclaimer: ~10m late and hungover.

Misc

Last time: looked at some code compiled by Selfie.

Memory organization in Von Neumann: can be done any way basically, but there is an agreement on how to do this and Selfie follows this exactly. This agreement appears to be consistent across most programming languages

The "standard"

- Divide memory into 2 parts: *static* and *dynamic*
    - *Program break* is the point at which memory is divided
- Static vs dynamic
    - static part is usually tiny and holds the code which is not going to change
    - dynamic part holds global variables, string literals, integer literals,

Notions of time

compile time and runtime.

- machine code comes out of runtime "one way or another"
- runtime is execution

Live and dead memory

A certain memory address is *live* because at some point in the future it will be either read or written

Dead memory is the opposite. These types only apply to dynamic memory.

Can we differentiate bewteen live and dead memory?

Think about this problem: Your apartment has a bunch of stuff. You want to clean up. You need to discriminate between "live" and "dead" stuff - project into the future: do you need this later or nah?

This is hard. People usually err on the "live" side. You might also have a *lot* of stuff and have to make that decision a lot of times. The more space/stuff you have, the more time you need.

So this translates to and is a key issue in memory management. CPU time is used to manage memory, so machines with more memory can actually become faster

Differentiating between live and dead memory

We divide the dynamic part into a stack and a heap. The stack grows from the top down and the heap from the bottom up. That means stack is upside down and grows from high addresses to low addresses

The stack is the most widely used structure for memory mamangement.

Memory allocation problem: Finding an address that is not used. With a stack free memory is just below where the stack ends. Stacks are also useful for implementing recursion - recursion is just a stack order.

We also utilize a heap. Is cleaned up by the garbage collector.

Stack holds local variables - when you enter a procedure the vars are pushed onto the stack and on leaving the procedure they're popped

Cannot free up memory in selfie. This is ok for any program that always terminates before running out of memory, which is the case for most programs we will write (code that takes input and calculates some output, i.e. implements a function).

There is code that just keeps running (game, browser, server, …). This requires garbage collection

Memory leaks: cases where your program just keeps requiring more and more memory

Very few programs are free of memory leaks. Can prove for some programs that they are.

Computer models

Turing Completeness (a program that can be written in a turing complete language can be written in any other turing complete language, almost all languages today are turing complete). That means all programming languages have the same expressivity. RISC-V is turing complete.

Finite State Machines: Can represent your computer. But finitely many states means it forgets. In finitely many steps it will forget what it did. It has Alzheimers (it is *forgetful*) but it doesn't require any memory management. This means if you can model your program with a FSM you should do it.

Much of the code of Selfie is based on a FSM for reading source code. That means no memory management is needed to do that.

Figure out what the minimum amount of information needed to do a job is. Then you can simplify your model and reason about correctness, maintain code etc. and it will also be more efficient.

15–11–2018

Intro

- Prof visited Boston last week. Went to *SPLASH* conference; ~600ppl attended, it's on programming languages, compilers, virtualization, …
- Says it'll be worth it to look at the webpage - is bleeding edge in programming language research. Python, OOP etc came out of this and similar conferences
- Prof gave a recorded talk on Selfie, ~1hr; should be online soon/now
    - Is a bit more advanced though, apparently
- Today: Explaining how, given some source code written in a programming language such as C or Java, to go to machine code. How to bridge the gap from fancy stuff to simple machine instructions.
    - Understanding this will make one a much better programmer

Review

- Code is just a textfile that is assumed to be written in a given language
- Output is machine code for a RISC-V machine (RISC-U subset)
    - Prof assumes we understand RISC-U now
- Compiler is basically a function translating bits to bits
- If semantics of machine code is understood, semantics of C* is understood
    - Like in correctly translating German to English, both must be correctly understood
- self-compiling compiler must be bootstrapped

Compiler stages

We have frontend and backend. This distinction is based on research by Noam Chomsky who tried to formalize natural languages. Resulted in programming languages and formal languages, even though that's not the original goal

The Frontend

There's a "fuzzy" area between frontend and backend. Frontend is syntax analysis. It does not *understand* anything, only sees characters and tries to recognize syntactic structure

Syntax is specified in a regular expression using EBNF: keywords, braces, …

The definition of legal symbols in C* is on Github TODO ADD LINK

We might have to write down a single EBNF rule to define e.g. integer, character, string, identifier: By substituting the given rules into one somehow (trivial)

Instead of `integer = digit { digit }`:

```
symbols = "w" "h" "i" "l" "e" | "=" | ...
integer = ( "0" | "1" | "2" | ... | "9" ) { ( "0" | ... | "9" ) }
```

If all EBNF rules of a language can be packed into one, then a scanner exists for that language. That scanner employs a finite state machine

E.g. a comment `//` would be parsed by reading a `/`, checking if it is followed by another one and if so ignoring all characters until the next `\n`

Integers are read digit by digit and must be converted to numerical values.

We have regular grammars and context-free grammars. C* cannot be written in a single EBNF notation because we have recursion. This is the distinguishing factor between a regular grammar and a context-free grammar.

Syntax checking is just done via checking for correctness of input according to EBNF grammar. This is done using a *recursive-descent parser*.

Most compilers are self compiling and are written in the language they compile.

Scanner

Simply takes one character after the other from input stream, transforms them into tokens.

Parser

Supposed to recognize structure of the code

The Stack

To parse a context-free grammar, a Finite State Machine plus a stack is needed (called a *push-down automaton* - for a regular grammar a FSM suffices).

The Backend

Basically a code generator. Actually produces the machine code. This is usually a much bigger part of the system since it has to figure out semantics, not just notation.

This is **part of the parser** because code is generated as soon as possible. That makes Selfie a *single pass compiler*. That means it goes over the input and generates the code without ever looking back. Most modern compilers are multi-pass compilers, this is usually for performance reasons: optimization etc. Single-pass is much simpler, however there are some complications that don't occur in multi-pass compilers.

Random notes

Semantics of the while statement can be seen quite clearly in Selfie's code.

Likely questions: substitute EBNF, how would parser/scanner implement given EBNF, …

22–11–2018

Names + Heap

Understanding names on an abstract level is critical to understanding the compilation process

static <-> dynamic memory: refers to “layout”

Stack goes downwards and back up as code is executed. Below is the heap (“haufen”) - a lot more complicated, efficient heap management is complicated.

How are names managed?

Example Hello World Program introduces two names: `foo` and `main`. System must somehow figure out how many bytes the corresponding values are going to need. Types define how operators behave. E.g. for a pointer in C, adding `1` to it will raise it by 8 bytes (points to the next word in memory).

Declaring vs. defining: For methods, the definition is the code. -> Tells us way more about the “thing”. The definition produces a connection between a name and some value

Symbol Table: Creates a list data structure for each entry in it. Each node holds a name and whatever the names are supposed to be (type, …). Takes O(n) for lookup. Now changed to a hash table because lookup was 30% of compilation time. The hash function looks at the first 8 characters of a string.

Types

Types regulate behavior of operators. What type will `10` be in `x < 10`, if `x` is an integer, a pointer, …?

In selfie comparing an unsigned integer with –1 returns “fuzzy” results: `1 < -1` returns `True`, since –1 as an unsigned integer is the largest number you can get as `uint`

Now: Done with this for now. That was needed foundations in order to understand what OSs do: they are software that run arbitrary amounts of programs in parallel along with themselves… how is this done? That’s the actual goal of this course

29–11–2018

Irgendwas

Wir abstrahieren Rechner ab der Ebene, wo Register etc. auftauchen

Bis jetzt - *Syntax* von Compilern gemacht. Regex/Context Free Grammar, Automaten, Encoding & Symbols. Was noch kommt: **Concurrency** - Software, die mehrere Sachen quasi-gleichzeitig laufen lassen kann. Das braucht **Virtualization**. Sollten bis jetzt wissen: was ist eine Von Neumann Maschine, ein RISC Prozessor, Scanner, Parser, was ein Compiler macht, wie eine while-Schleife in Maschinencode aussieht.

Stack Wiederholung. Selfie haelt sich an die Unix-Spec anscheinend. Parameter von Funktionen liegen auch im Call-Stack. Parameter sind im Code zwar Namen, beim Prozeduraufruf aber “actual parameters” oder tatsaechliche Werte. Die tatsaechlichen Werte landen auf dem Stack.

Wichtig: Unterscheidung *Compiletime* vs. *Runtime*. Was passiert bei Uebersetzung von Code und bei Ausfuehrung? Compiler haben beides parallel: Runtime, weil der Compiler selber laeuft, Compiletime weil der Compiler irgendwas kompiliert. Self compilation - alles ist irgendwie moeglich.

Welcher Code wird generiert, wenn wir eine Prozedur parsen? Wir haben eine *local symbol table*.

Muessen ausrechnen, welcher Parameter wo liegt, um sie spaeter noch referenzieren zu koennen. Alle Parameter sind gleich lang in C*. Local symbol table haelt alle Parameter "umgekehrt" - letzter zuerst, sagt uns: "skippe soviele Words vom Frame Pointer aus, um an den Param zu kommen". D.h. auf den Parametern liegt noch was drauf… die lokalen Variablen. Die haben negatives Offest vom Frame Pointer. Jede Prozedur hat einen Frame.

Methoden haben Prolog und Epilog.

Prolog baut Frame auf

Epilog entfernt den Frame etc. Am Ende: Return zu der Funktion, die mich aufgerufen hat. Dann muss der Stack gecleaned werden: wieder so sein, wie er vorher war.

Sollen den Shit zuhause probieren. `-S $OUTFILE` angeben, um Namen des output files anzugeben. Mipster benutzen: code laden und Mipster-Instanz mit 1meg RAM machen: `selfie -l test.c -m 1` oder direkt bei mKompilieren `-m` benutzen. `-d` macht dasselbe aber in Debug mode: printet alle ausgefuehrten Instruktionen.

Debug-Output Syntax: `$INSTRUCTION read |- update -> update

Globale Vars sliegen gabz oben auf dem `data`-Bereich.

Slides. https://www.icloud.com/keynote/0J_SKB-ofwiuxg-lCag-s-gOA#selfie

13–12–2018

Concurrency

Concurrency bzw. Betriebssysteme konstruiert die Illusion von vielen identischen Maschinen, die parallel unterschiedliche Programme ausfuehren. Ist ein schwieriges Thema - sollten Code angucken

Arbeiten unter der Annahme, dass sie selbst auf einer solchen Maschine laufen - konstruieren etwas, was wir gleichzeitig brauchen. Dieser Zyklus kann aber aufgeloest werden. Wie folgt:

Praesentieren einfache Form von Betriebssystem, was funktional gleichwertig ist

Soll auf einer Von Neumann Maschine laufen und die Moeglichkeit geben, beliebig viele weitere Von Neumann Maschinen (Prozesse) darauf zu "simulieren". Diese sind insbesondere nicht im Speicher beschraenkt, sollen alle 4GB haben zB, wenn die "echte" Maschine auch 4GB hat.

Bei `selfie` macht das `mipster`: `selfie -c $SOME_CODE -m $MEM_MB` startet Mipster mit `$SOME_CODE`

Kann selfie auf mipster ausfuehren! Via `selfie -c selfie.c -m 1`. Ist dann funktional identisch zu `selfie.c`

Weitere Parameter nach `-m 1` werden dann an das auf Mipster simulierte Programm weitergegeben

Mipster kann dann auf Mipster laufen und dabei Programme ausfuehren. In diesem Modell ist der unterste Mipster "Hardware" und der darauf laufende Mipster der Betriebssystem-Kernel. Dieser Kernel kann ein Programm ausfuehren.

Wie kommen wir von da aus zu beliebig vielen Prozessen? Muessen der simulierten Mipster-Instanz nur sagen, sie soll jedem Prozess nacheinander 50 Instruktionen geben. Das waere ein `round-robin scheduler`.

Ist exponentiell langsam - komplette Maschine zu simulieren ist cancerous. Moderne Betriebssysteme koennen context switchen, d.h. Code dirket auf der Hardware ausfuehren, mehr oder weniger.

`Hypster` - identisch zu Mipster, aber verwendet context switching im Gegensatz zu mipster, was interpretiert. Switcht sich selbst aus dem unterliegenden Mipster raus, indem er Mipster Bescheid gibt und einen Timer setzt, der irgendwann einen Interrupt sendet. Auszufuehrendes Programm ist damit recht gut gekapselt.

Frueher gab es cooperative OS - OS ist davon ausgegangen, dass das Programm die Kontrolle an OS zureuckgibt.

Hardware heute hat nicht mehr nur einen Kern. Wie bootet man sowas? Ein Kern fuehrt BIOS aus, was OS Code in Speicher laedt. Irgendwann waehrend OS Boot kommt Maschineninstruktion, die den 2. core einschaltet, welcher denselben Code auch ausfuehrt (jeder hat seinen eigene Program Counter), welcher den 3. Core einschaltet, usw usw … –> gleicher Code wird 6x ausgefuehrt. Irgendwann dann - jeder core baut sich einen eigenen Prozess (oder mehr). Wir haben #cores Program Counters, die alle im gleichen Code herumwurschteln koennen. OS muss dann aufpassen, dass sich der ganze Kram nicht in die Quere kommt.

Logisches Modell ist aber immer noch nicht anders als das, was wir heute kennen gelernt haben.

10–01–2019

Virtual Memory?

Zum Erzeugen der Illusion, dass 100e Instanzen dieser Maschine existieren, wird im Grunde die Hardware virtualisiert.

Naechster Schritt waere Cloud Computing mit dem Ziel “Computing as a utility” (i.e. sowas wie Wasser oder Strom).

Memory: Hat immer einen address space. Heute byte-addressiert, ganz frueher bit-addresiert. Sinn von byte-addressierung: 8x soviel Speicher ist addressierbar und es kommen immer 8 Bits pro Zyklus zur CPU durch. Aber Overhead, wenn man nur ein Bit braucht und mit Bitmasken/bitwise operators drauf zugreifen will.

In Selfie werden bitwise operations zu arithmetischen Operationen umgewandelt - d.h. sind mit Integer-Arithmetik simulierbar. Fuer Performance sidn bitwise operators aber deutlich besser.

Metriken, um Memory zu messen: *throughput* und *latency*. Analogine: DL Speed und Ping, oder: Glasfaserleitung vs LKW von NY nach SF. LKW ist langsamer, wenn es um einen Skype Call geht, aber schneller, wenn er mit Petabytes von HDDs vollgeladen wird. Ping time Sbg->SF: ~150ms.

Throughput kann erhoeht werden, wenn man mehr parallele Leitungen legt.

In selfie: byte-addressed und double word aligned.

Storage vs. address space:

Pointer sind sick: Der Speicher ist linear und kann Speicheraddressen speichern. Linear bedeutet: man kann von *contiguous* blocks reden zwischen Addressen a und b, wo a strikt kleiner ist als b. Was macht linearen Addressraum gut? constant time access. Waere das nicht so (d.h. Daten waeren scattered), muesste man eine Adresse fuer jedes Byte, was die Daten belegen, verfolgen. Die Linearitaet ist “implizite Addressierung” - man muss einfach nur weitergehen. Ist dann mehr oder weniger konstant, weil nonlinearer Speicher linear in der Anzahl Pointer waere.

Speicherfragmentierung: kann passieren, sobald man unterschiedlich grosse Blocks speichern will, contiguous blocks will und der Speicher nicht kompakt gehalten wird. Der Speicher ist kompakt, wenn keine “Luecken” da sind. Stacks fragmentieren auch nicht, da immer nur oben alloziiert/dealloziiert wird.

Ein fragmentierter Speicherraum kann effektiv defragmentiert werden, wenn der address space umprogrammiert wird, statt den Speicher physisch umzusortieren. das ist *nonmoving defragmentation*, den Speicher physisch umzusortieren waere *moving defragmentation*".

-> der physische und virtuelle Raum koennen unabhaengig voneinander kompakt bzw. nicht kompakt sein.

17–01–2019

Speicher

- address space (linear)
- addressiert storage
- in modernen Programmiersprachen wird address space von konkreten Speicheradressen zu Variablennamen wegabstrahiert
- in Selfie: byte adressed, word aligned.
- *memory management problem*
    - Adressraum muss verwaltet werden: welche Adressen werden gerade verwendet und welche nicht?
    - Unterscheidung used/free
    - Memory management: diese Unterscheidung laufend irgendwo irgendwie festhalten
    - Speicher ist ja in static/dynamic unterteil (static haelt Code und Daten), static ist also immer used, dynamic teil ist im Stack und Heap besetzt
    - Stack fragmentiert nicht
        - Speicher im Stack wird bei Funktionscalls alloziiert
    - Heap-Speicher wird explizit alloziiert - in C `malloc`, in Java `new`
        - `malloc` etc geben prinzipiell nur Speicheradresse zurueck, fuer die gelten soll, dass diese Adresse vorher ungenutzt war
        - `malloc` ist also ein "Zahlengenerator", der free mem-Adressen findet
    - in selfie: gibt kein `free`, Speicher wird einfach fortlaufend alloziiert. Ist simpel
        - `malloc`/`free` bzw. wie verwaltet man used/free memory ist insane kompliziert
    - Java hat auch kein `free` - Speicher wird freigegeben, wenn kein Pointer mehr auf eine Adresse zeigt

Verknuepfung von Address Space und Storage - ist nicht unbedingt 1:1 und ist Kirschs Lieblingsthema

- Wir haben zB 4GB RAM und wollen darauf mehrere Sachen ausfuehren (irgendwelche Binaries von irgendwelchen Leuten, wo sich jeder Entwickler dachte dass er den ganzen Speicher fuer sich hat)
- Loesung: Abstrahieren Adressraum zu einem virtuellen Adressraum
    - **Virtual Memory** ist *nur* ein Adressraum, der kann groesser oder kleiner sein als das was wir tatsaechlich zur Verfuegung haben
    - Virtual Memory Adressraeume sind mit 48bit implementiert, auch auf 64bit-Maschinen
    - jeder Prozess kriegt einen virtuellen Adressraum
    - **Segmentation**: einfachste Methode, virtual mem zu implementieren
        - zB jede Anwendung kriegt ein x MB grosses Segment im tatsaechlichen Speicher.
        - Das Segment kann dann von der Anwendung wie komplett eigener Speicher verwendet werden.
        - `load`-Instruktionen innerhalb der Anwendung muessen mit virtuellen Adressen arbeiten
            - sind dann zu physischen Adressen konvertierbar, indem man einfach die Startadresse des Speicherblocks der Anwendung draufaddiert
            - Das Konvertieren macht Hardware - Memory Management Unit - weil so oft solche *address translations* stattfinden (bei jedem Befehl)
                - Bei Versuch, auf Adresse ausserhalb des Segments zuzugreifen, kommt Segmentation Fault -

ob das auftritt, wird auch in Hardware ueberprueft

- Perfekte Virtualisierung des Speichers: als haette man die ganze Maschine fuer sich, obwohl das nicht so ist
- Wenn man zwischen Anwendungen switcht, muss nicht nur der Program Counter veraendert werden, sondern auch das aktuelle Segment etc.
- vor 20–30 Jahren war Segmentation der Stand der Technik

- Heute: wollen groessere virtuelle Adressraeume (nicht nur 100MB, sondern [viel] groesser als physischer Speicher sogar)
- wie kriegen wir das hin? - **Paging** oder Paged Memory

- machen uns virtuellen 2^{48} bit Adressraum, unterteilen in 4KB *Pages*.
- unterteilen physischen Speicher in eine bestimmte Anzahl 4KB *page frames*
- haben eine *Page Table* fuer jeden virtuellen Adressraum - prinzipiell eine Tabelle, implementiert als Array indiziert ueber die Pages: mappt 0-te Page zu irgendeinem page frame
- Page Table ist praktisch leer. Nur verwendeter Speicher hat ein Mapping - unbenutzte Eintraege haben nichtmal einen Eintrag. Oben im virtuellen Adressraum ist Stack etc, unten Code und Daten -> Page Table ist unten voll und oben und hat dazwischen Terabytes freien Speicher
- Welche Datenstruktur fuer Page Table? Intuitiv: Baum mit einer Ebene - ein Leaf fuer unteren Teil der Tabelle, ein Leaf fuer oberen Teil, Root die beide verbindet.
    - Leaves sind dann "Subtabellen" - auch als 4KB Page Frames implementiert. -> keine Fragmentierung
    - in der Praxis ist der Baum tiefer (3–4 Ebenen) und Konstruktion wird vom OS uebernommen, waehrend Lookup in Hardware implementiert ist

in der Praxis wird beim Starten eines Programms gar keine Page Table angelegt - nur beim Fetch der ersten Instruktion wird angefangen, die Table zu bauen. d.h. der eigentliche Start kostet nichts, nur wenn tatsaechlich angefangen wird etwas zu machen - das ist weil oft viele Teile des Codes gar nicht ausgefuehrt wurden etc.

Page Replacement Problem - welcher Frame kann rausgeschmissen werden und mit dem ersetzt, was man gerade braucht? Wir wollen den Frame austauschen, den wir am Spaetesten erst wieder brauchen.

- Austauschen setzt Page Replacement Algorithm voraus
- *Swapping*: Frame wird in persistenten Speicher geswappt
- wie finden wir einen sinnvollen Frame zum Swappen? - least recently used. Ist nur Heuristik anhand einer Eigenschaft typischer Software: *locality*. Die hat 2 Aspekte: *temporal* und *spatial*.

- Temporal sagt das VErhalten von Software ist temporally local gdw die Wslkeit dass auf eine Adresse, auf die gerade zugegriffen wurde, in naher Zukunft wieder zugegriffen wird, hoch ist.
- Spatial sagt hohe Wahrscheinlichkeit fuer Zugriffe auf Adressen, die nahe beieinander sind
- sagt intuitiv, dass Code sich irgendwie an einem Fleck/Hotspot bewegt.
- LRU Strategie funktioniert nicht immer gut - MRU macht Sinn bei Prozessen, die zB nur Speicheradressen lesen und danach nie wieder anfassen. Ist nur beste Heuristik nach forensischer analyse
- das alles passiert im Kernel

Shared Memory ist zB, wenn zwei Pages von unterschiedlichen Prozessen auf denselben Page Frame gelegt werden. So kommunizieren Prozesse im Endeffekt, fuehrt dann zu Concurrency Problems etc etc.

Copy On Write: Prozesse, die sich forken, teilen anfangs ihr Memory komplett (page tables sind 1:1 Kopien), nur wenn einer in "seinen" Speicher schreibt, kriegt er seine "eigene" Page

Switching frequency von Prozessen auf irgendeinem Laptop: ~10ms - das ist niedrig. Umschaltkosten werden zu teuer, wenn man oefer switcht