Emery Berger

emeryberger.github.io / CompSCI-630

Systems Lunch @ Friday — Fall

random number

TA Abhinav Jangda

github!

Hot CRP

emery@cs.umass.edu

- SCRIBES!
- course load:
  - 1 core / semester for MS
  - " 1st semester MS/PhD
  - > 1 core = too much!
- schedule - yes, we have a midterm & final!
- reviews

FORTRAN (& COBOL)

1954

military - Calculating artillery trajectories
- decryption

first computers - people!

general-purpose computing machine

automatic computers

Turing machine

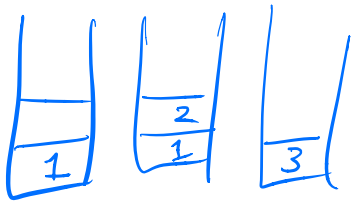
von Neumann - machine] data & code together

Harvard architecture

fetch-decode-execute cycle

assembly language

- lowlevel — direct access to memory
  - operations not abstract
- stack architecture
- register architecture



PUSH 1
PUSH 2
ADD
POP

A, B, C, D
AX BX CX DX

MOV 1, A
MOV 2, B
ADD A, B, C

"Intel" "ARM" (subset of instr.)

JVM, .NET, Python bytecodes, WASM

- portability

ASM ≠ portable

backward compatibility

- hard to read, hard to debug

---

1. PORTABLE
2. READABLE
3. WRITABLE

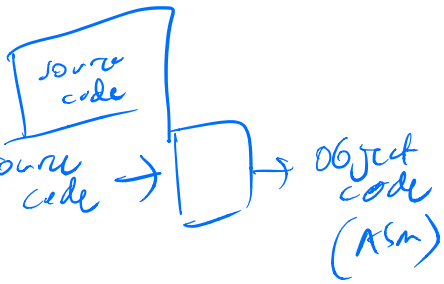
- COBOL – manages } domain-specific languages
  FORTRAN – scientific }

④ FAST

Python
Ruby

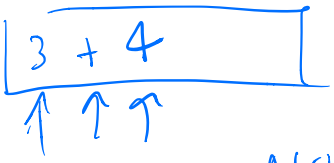
Matlab
LOGO
SQL — non Turing complete
CUDA
Excel
—

interpreted
vs.
compiled

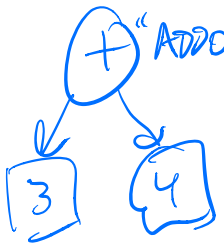


matmul
4k * 4k
~ 7 min.

numpy —
~ .4 s



Abstract syntax tree (internal representation IR)




```
switch (op) {
  :
case ADDOP:
  result = t->left +
           t->right;
  break;
  pc++;
  :
}
```

ADD A, B, C

programmer time MORE VALUABLE than cycles
— modern POV

- productivity
- debuggability
~ 100x OK !?)

interpreter → just-in-time compiler (JIT)

Python PyPy

JavaScript V8
Chrome

WASM

---

predates - parsing
- compiling
- program analysis
- typechecking
- scope
- object-orientation
- modularity
- structured prog.

FORTRAN
1st optimizing compiler

I, J, K integer

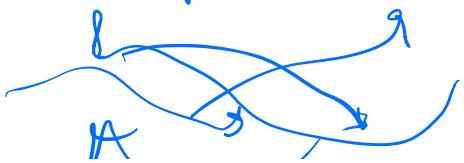
X, Y, Z float

```
DO  10  I = 1,10
10  CONTINUE
```

(was comma) line numbers

BASIC

"spaghetti code"



control flow

DO 10 I = 1.10

DO10I = 1.10

FORTRAN 77 , FORTRAN 90 — evolving

LEGACY

FAST The Fastest PL today! 2019 !!

optimization

static analysis + transformations

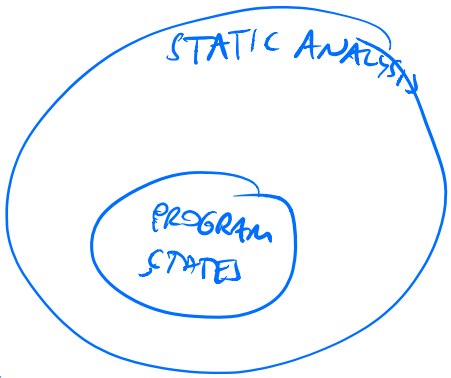
semantics-preserving

compile time × ~~runtime~~

halting(p)

if (false)

$X \in \{1, 2, 3\}$



conservative view

over-approximation

Halting problem

halts (program_text)

⇒ true if program always terminates

⇒ false otherwise

p' if halts(p) run forever

else terminate

(1) Constant folding

$x = 3 + 4$

⇓

$x = 7$

* PI * PI

(2) Constant propagation

$x = 7$ $z = 12$

$y = x + 3 + z$

⇓

$x = 7$ $z = 12$

$y = 22$

(3) ``strength reduction''

$x = x \; (**) \; 3$ — expensive

$\Downarrow$

$x = x * x \; (*) \; x$ — cheap

/ %

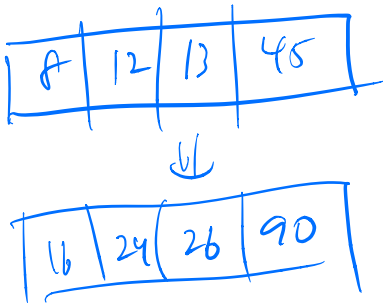
$x = x \;\%\; 8$

$\Downarrow$

$x = x \;\&\; 7$

111

④ vectorization




```
for (i = 0; i < 16; i++) {
    a[i] *= 2;
}
```

$\Downarrow\Downarrow$

vector ops (4x speedup)

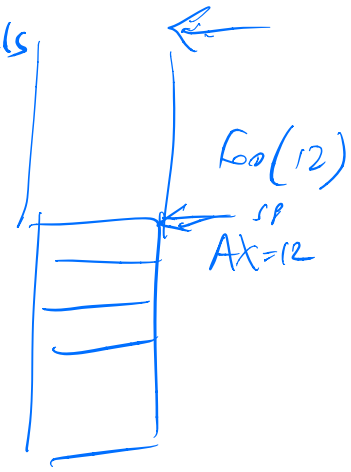
⑤ inlining

function prolog

copy arguments into locals
create stack frame

```
foo(int x) {
  ...
  return i1   jump
}
```

stack resident



epilog

copies out return value
restores registers
pop frame
returns (jump)

```
int add(int a, int b) {
  return a + b;
}

foo() {
  z = add(a, b)
}
```

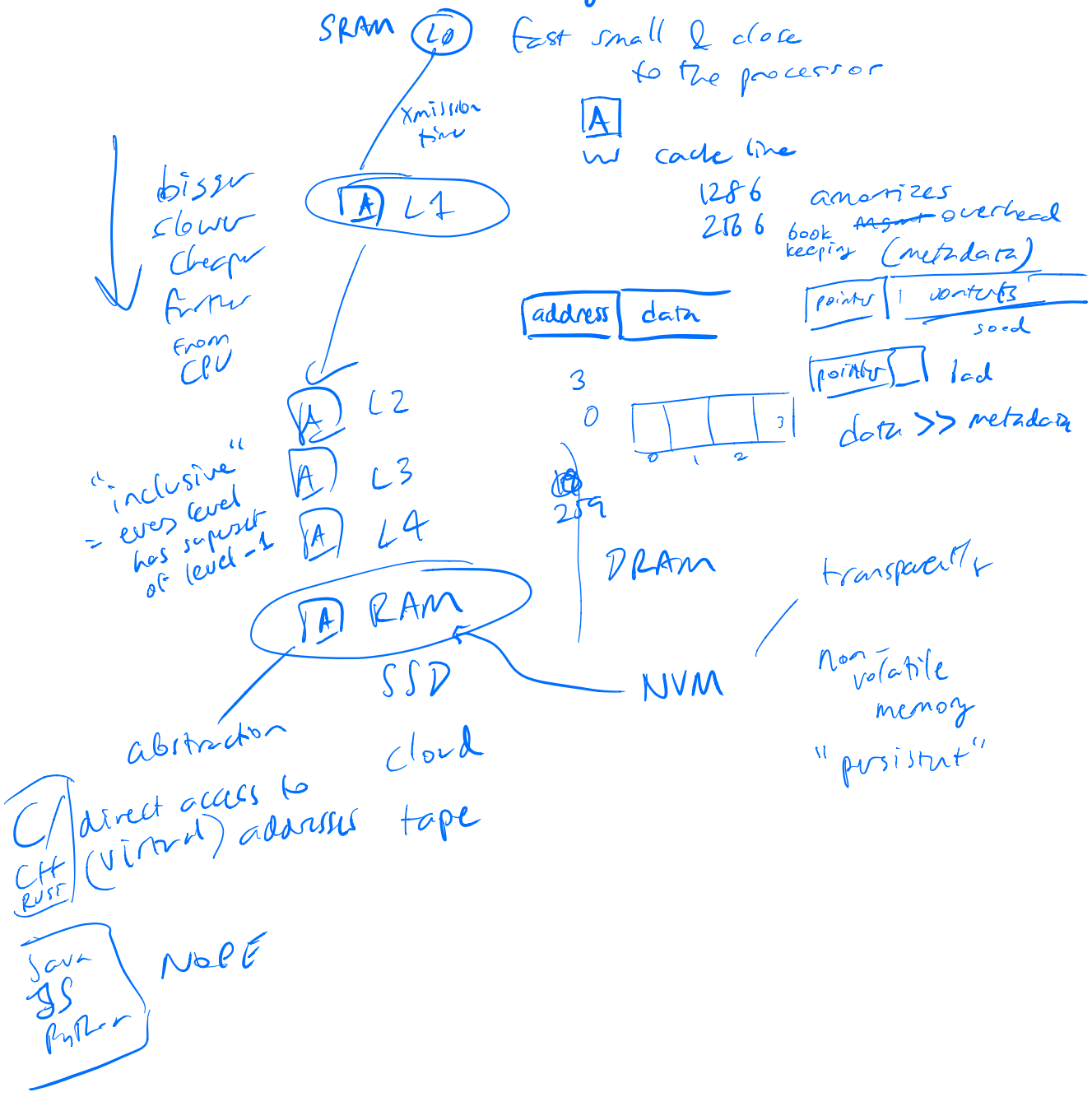
⇓

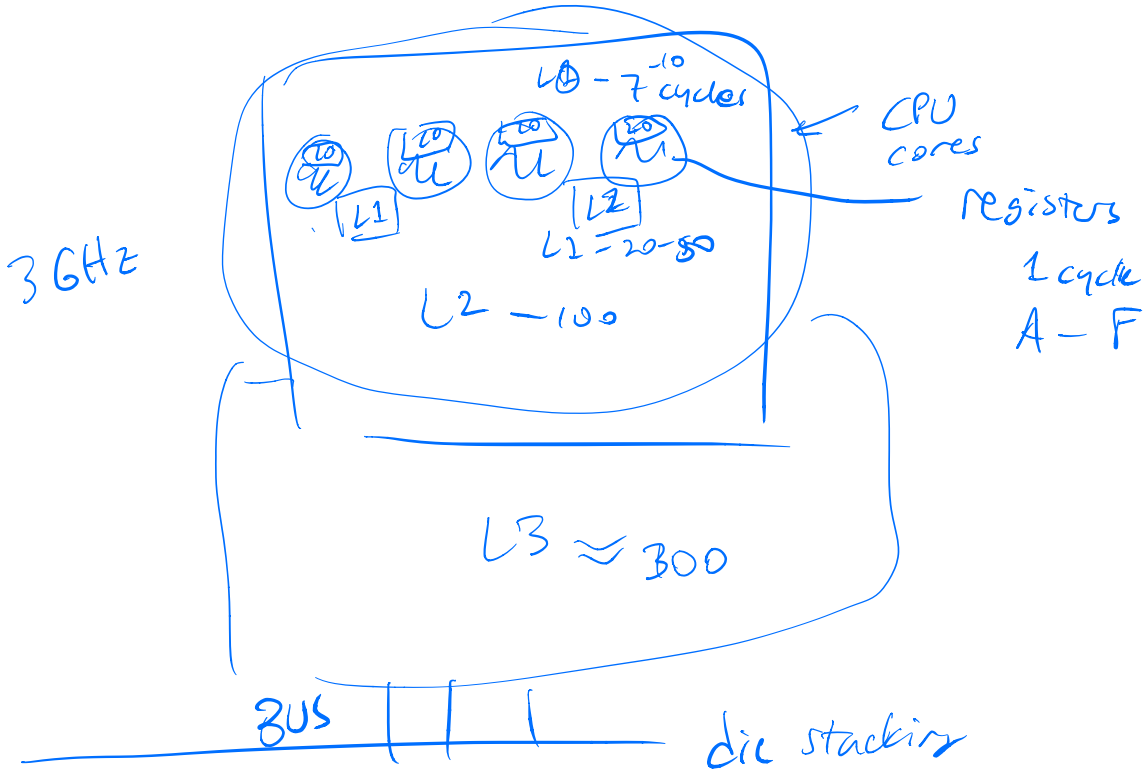
```
foo() { z = a + b; }
```

- code size
- I$ instruction cache

---

memory hierarchy





RAM "1 GB/s"

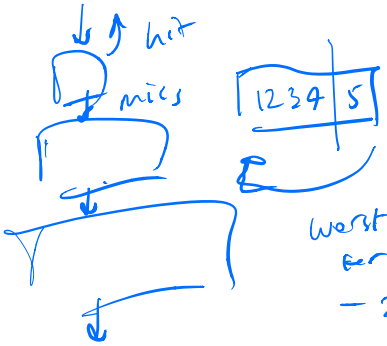
RAM bandwidth

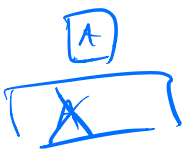
~~1000~~

10,000

caches — "replacement policy"

eviction "LRU"





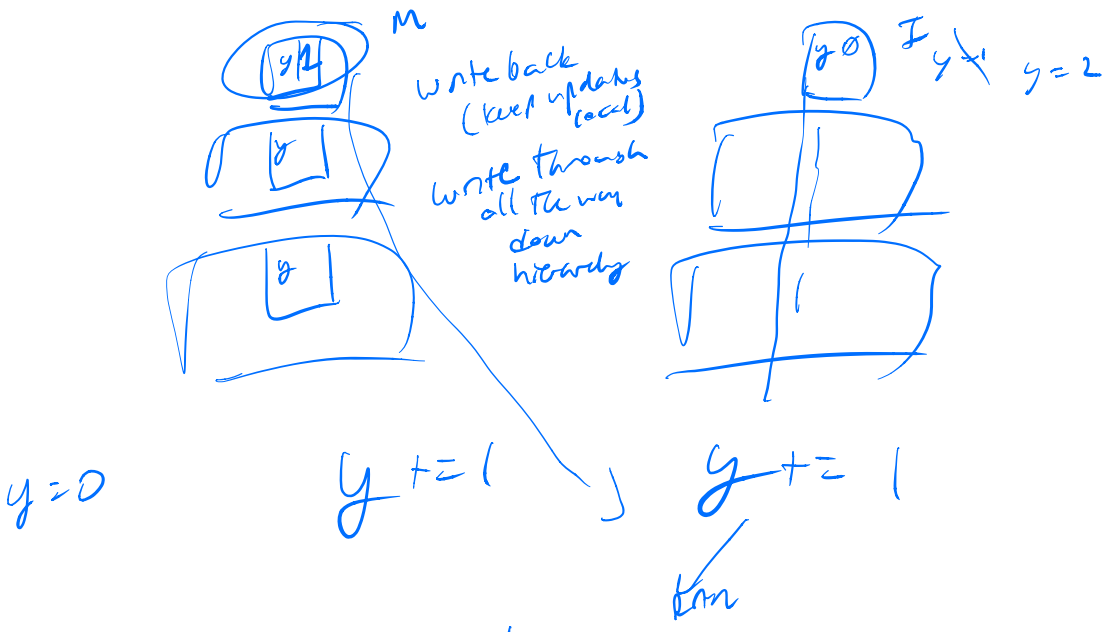
latency

wait time

caches hide latency

worst-case for LRU

- ≈ OK for inclusive
- catastrophe for exclusive



cache coherence

MESI protocol

- Modified — I changed it
- Exclusive — just me
- Shared — several CPUs have it (unmodified)
- Invalid — trash

approx LRU implemented

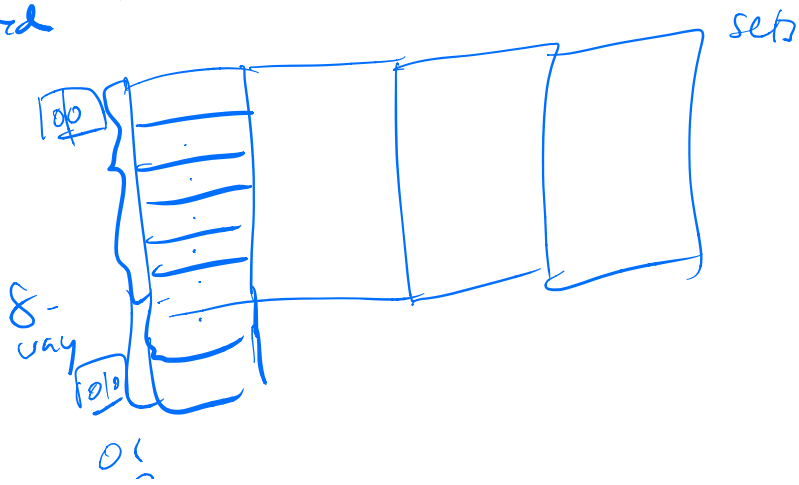
(addr % #cache line size) / cache line size

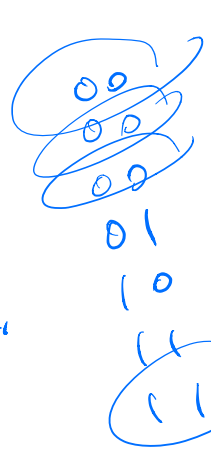
Time

3 addr

5 addr

addr



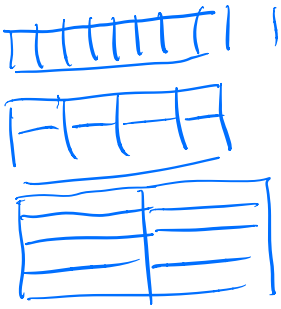


Cache misses
Can in different
types -

Capacity
Conflict — "hash collision"
Coherence 3C's model

saturated counter

direct mapped — 1-way
2-way
4-way
8-way
16-way
fully associative — ∞ way



TLB
translation lookaside buffer

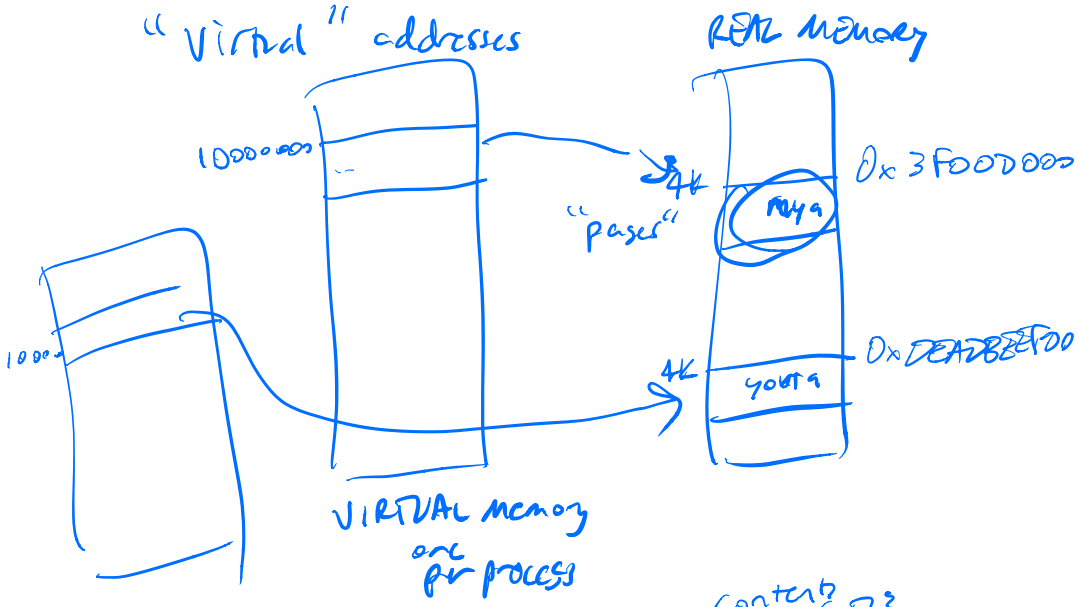
CPU cache

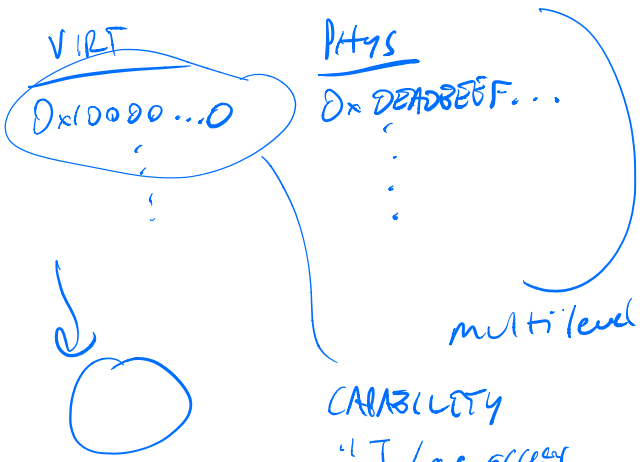
cache of virtual → physical addresses

```
#include <stdio.h>
int main() {
    int a;
    printf("&a = %p\n", &a);
    return 0;
}
```

0x10000000
address

process isolation





CAPABILITY
"I have access to this memory"

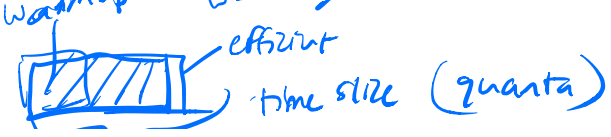
TLB flush
— discard all TLB entries

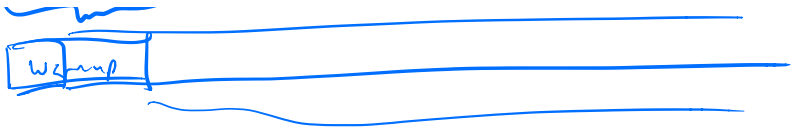
---

time slicing
scheduling

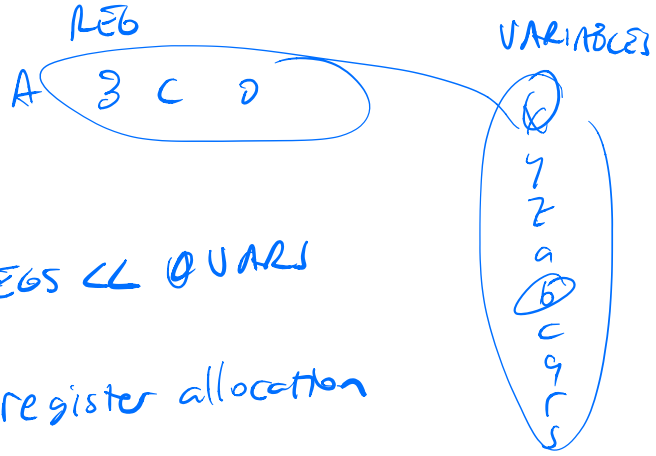
"cold" cache
warming up the cache

CONTEXT SWITCH overhead
processes ↑
threads ↑





amortize warmup
across the queries



REGS << VARS

register allocation

"spill" — copy contents of register to mem "eviction"!

straight-line code
— predictable!

"graph coloring" NP complete!

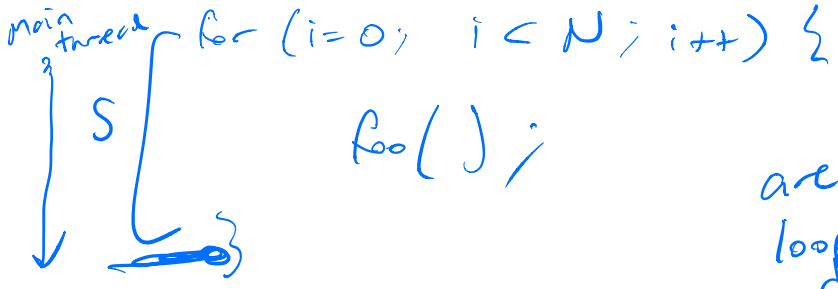
Why is FORTRAN fast?

~~OMP PARALLEL~~
DO I = 1, 100
DO J = 1, 100
A[I][J] = A[I][J-1]+1
ENDDO
ENDDO

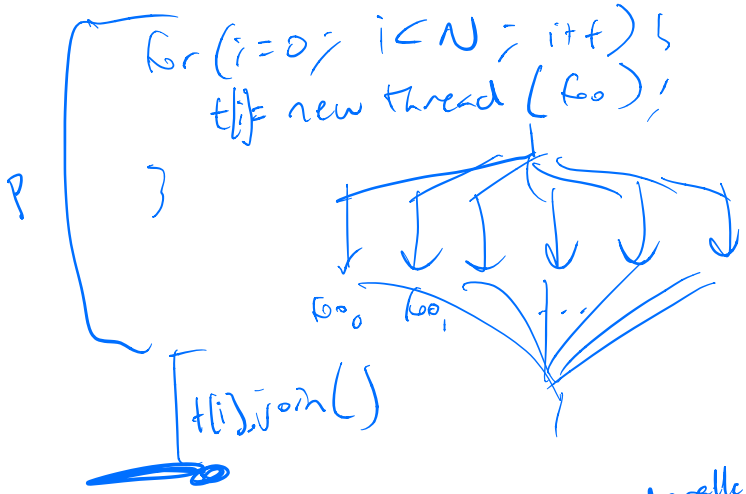
pragma omp
OpenMP
C
~~"hint"~~

directives

automatic parallelization
"holy grail"



are there loop carried dependencies?



result(S) ≡ result(P)

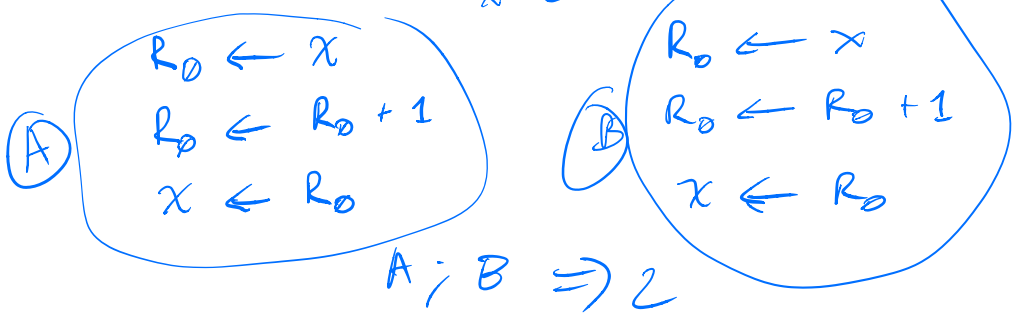
parallel prog semantically equivalent to seq prog.

does foo() have side effects?

```
foo() {
  x++;
}
```

(global)

read x into a register
increment it
write the result back
x = 0

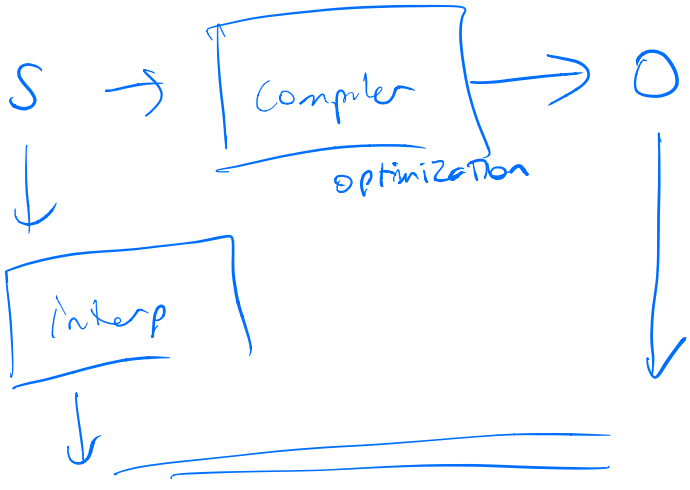


A ; B ⇒ 2

$X \in \{1, 2\}$

non-determinism!

race condition concurrency errors




```
for (i=0; ic
  x = 12
  y = x + 100
  z = y ** 100
  q = 2!
  cout << q;
```

dead code elimination

```
foo() {
~ x ~
}
```

```
foo (int i) {
  ~ i ~
  return ack(i, i+1);
}
```

Ackermann

```
foo(int *a, int i) {
  a[i] = ack(i, i+1);
}

foo(int *a, int *b, int i) {
  a[i] = ...
  b[i] = a[i] + 1;
}
```

```
int x[200];  int *y = &x[0];
v = foo(x, y, i)
```

aliasing

alias analysis
pointer analysis

p { }
q { }

$p \cap q = \emptyset$

```
int *p;   p ∈ {}
if (...) {
  p = q;      p ∈ {q}
} else {
  p = r;      p ∈ {r}
}
```

symbolic execution

abstract interpretation

$p \in \{q, r\}$

```
x = 12
x = x - 100
x . . . .
```

X O, N, P

```
if (x > 0) {
    |
}
```

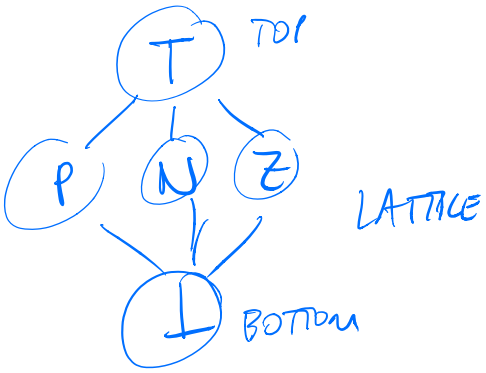
does this run ever?

```
x = ___
x = x - 1
```

$\{\emptyset, N, P\}$

$\emptyset \Rightarrow \emptyset$
$N \Rightarrow P$
$P \Rightarrow N$

CONSERVATIVE APPROX PROG BEHAVIOR



$\{x : T\}$

```
if (-) {
    x = -1
} else
    x = 1
}
```

$\{x : N\}$

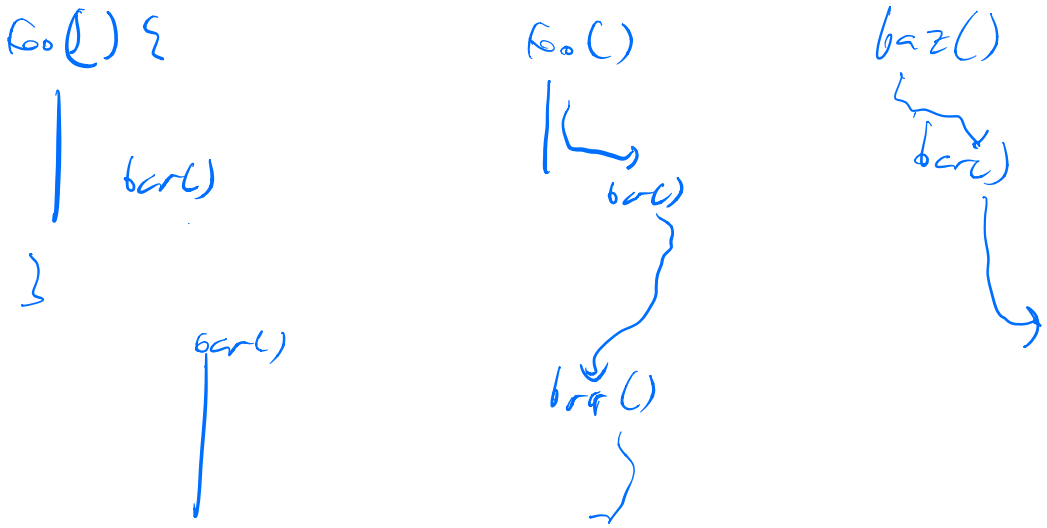
$\{x : P\}$

$\{x : \bot\}$

```
if
    x : P
else
    x : P
```

$x : P$

intraprocedural vs. interprocedural



Context sensitivity

Flow sensitivity

Steensgard's algorithm

RCR sensitivity

false positives

false negatives

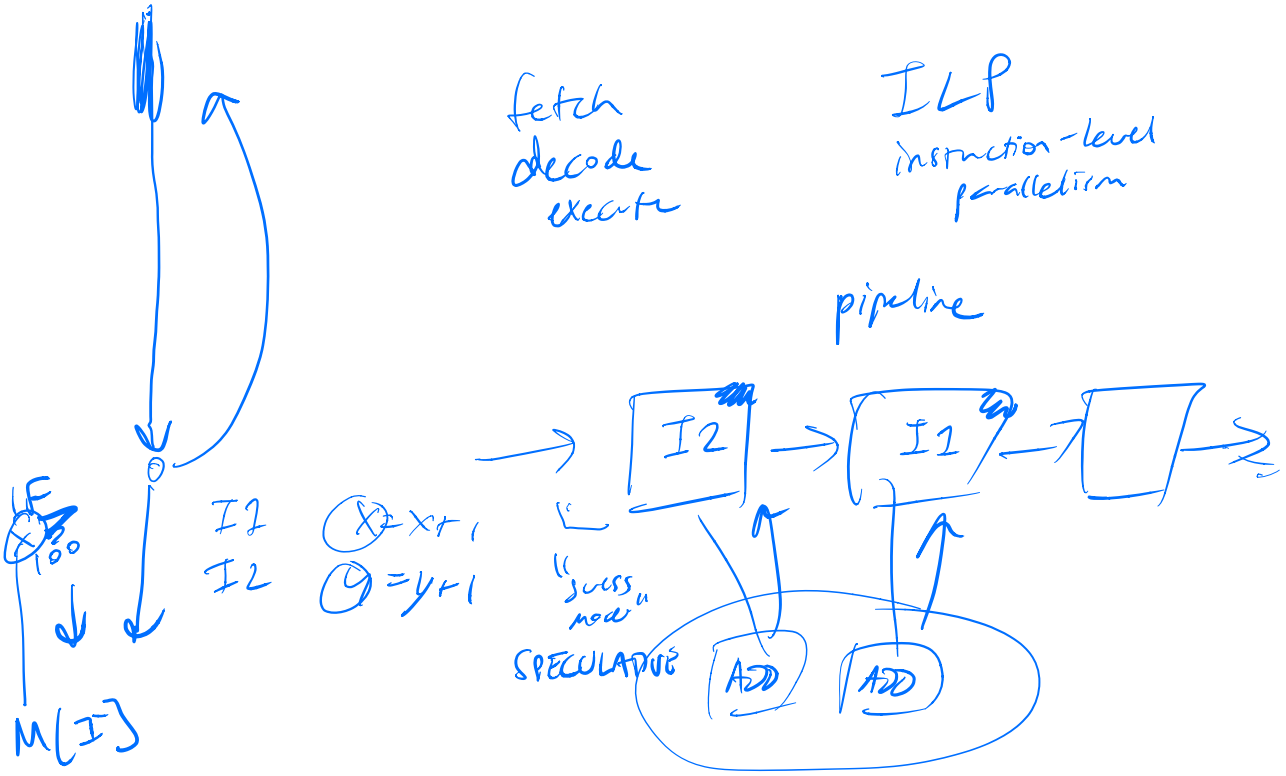
precision - recall tradeoff

$\top$ — { ⊘ }

if (x == ——)

y = x / ⊘ ⊤

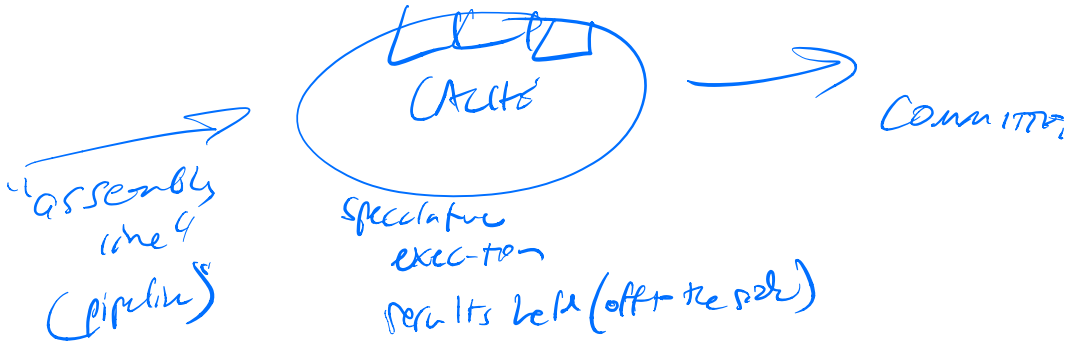
—W



PIPELINE STALL

"bubble"

"BACK EDGE TAKEN"



Speculative execution

hide memory latency

SPECTRE / Meltdown

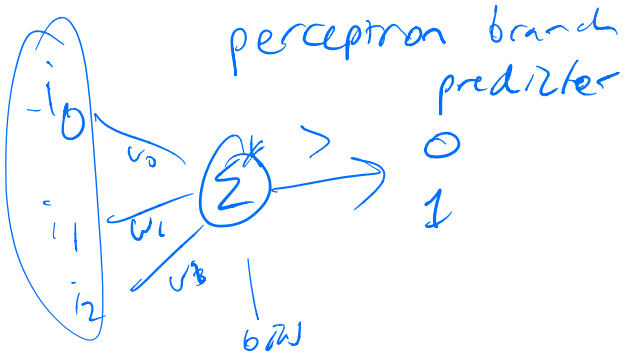
timing channel — covert channel

side channels

last branch taken — 85-90% effective

PHT

MODERN BRANCH PREDICTORS — 99.9% effective



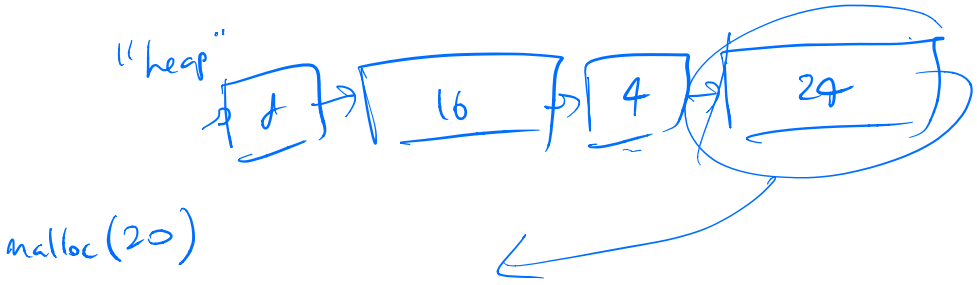
---

dynamic memory allocation

global

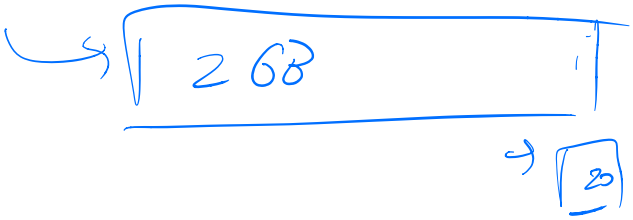
stack variables

dynamic memory — "the heap"



Project 2

First-fit — SPACE INEFF



best-fit

SPACE EFFICIENT

SLOW ....

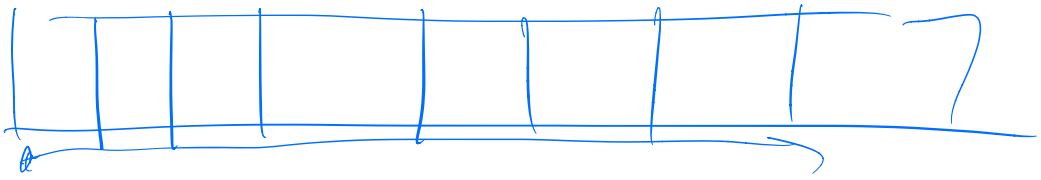
$n \; O\left(\log \frac{M}{m}\right) * n$

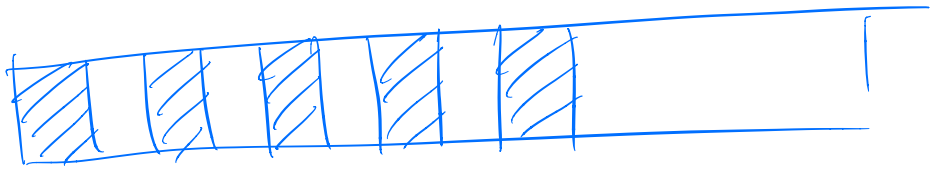
M — biggest

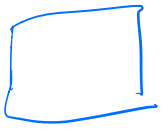
m — smallest

worst-case "fragmentation" = mem CONSUMED / mem requested

$O(M \times m)$ best fit







Compaction

NOT IN C/C++

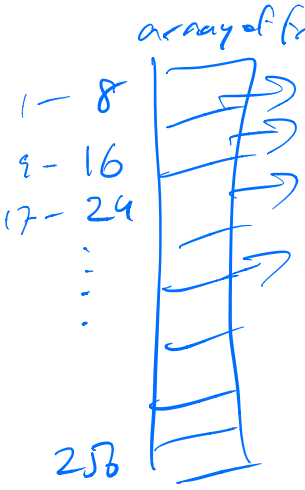
(LIST)

Python, Ruby

"custom allocators"
"ad hoc allocators"

malloc — "too slow"

Recognizing Custom Mem. Allocation



Py_Object_malloc

```
if (sz <= 256) {
    use small obj.
} else
    use malloc
```

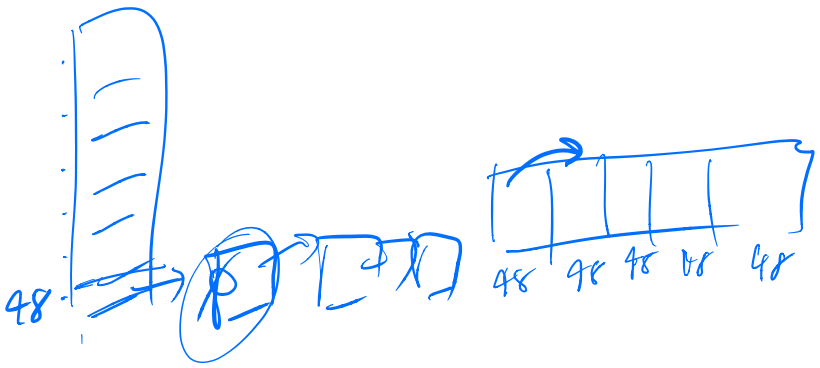
		C	Python
int	≈ 32 bits	4 b	24/28
long	≈ 64 bits	8 b	

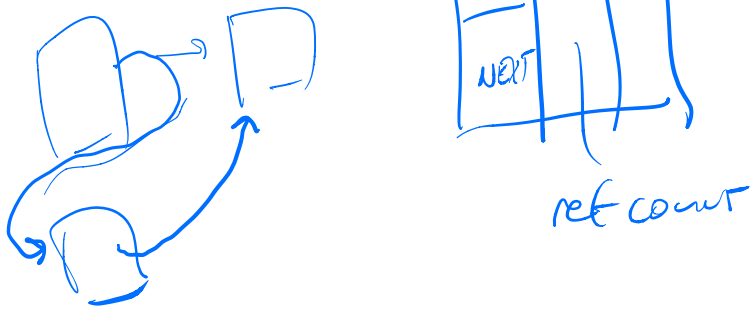
"a" [a][0] NUL 1/16 50

~~{"a": 2}~~

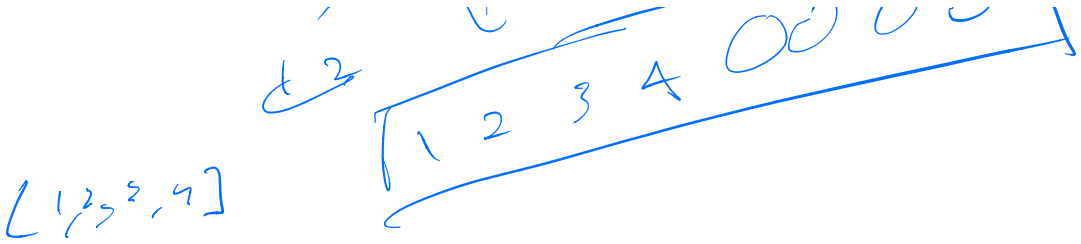
240

60 20 75 100 140





$x = [1, 2, 3]$

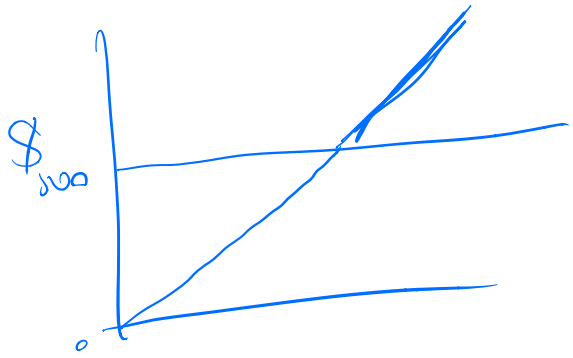


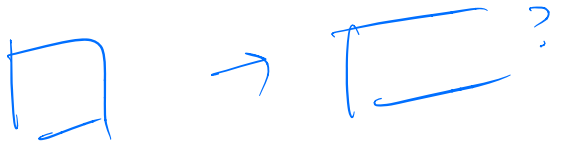
ski rental problem

$50 rent

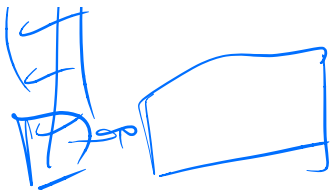
$500 buy

rent until you reach cost of buying



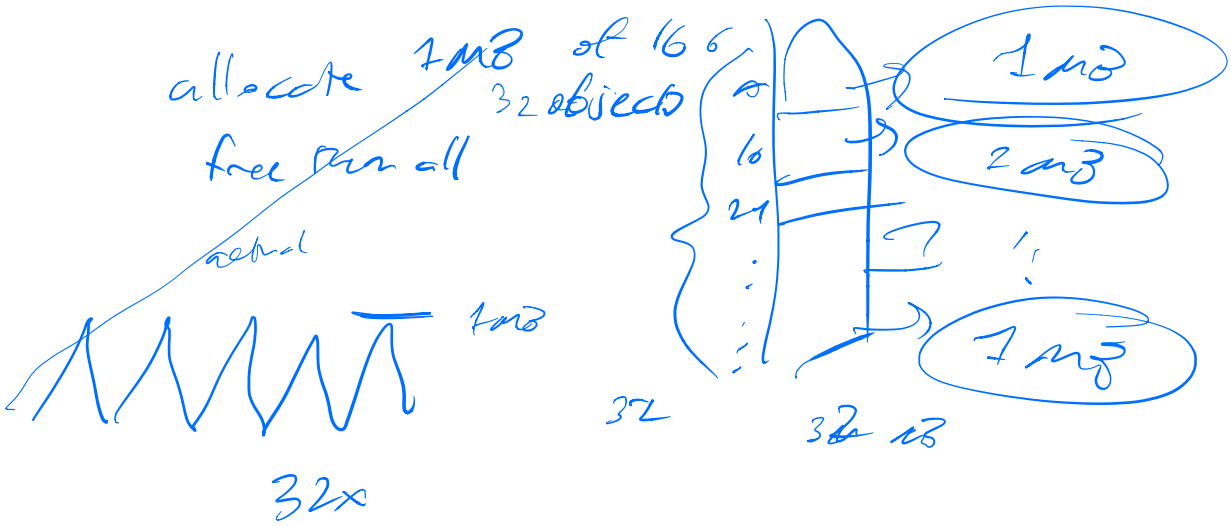


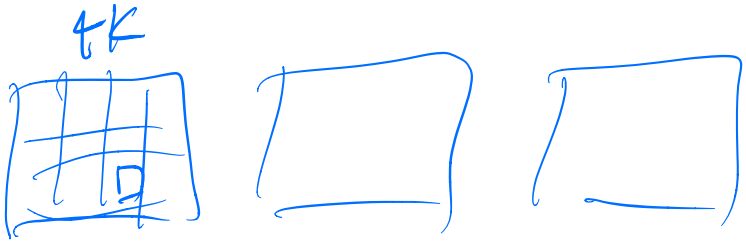
segregated size classes



allocate 1 MB of 8 byte objects
free them all

allocate 1MB of 16 6
32 objects
free them all



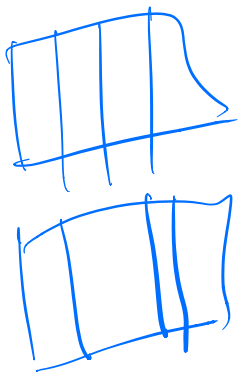


Space-time tradeoff

f(a) =

0	a
1	b
2	c

f(0) = a
f(1) = b
f(2) = c



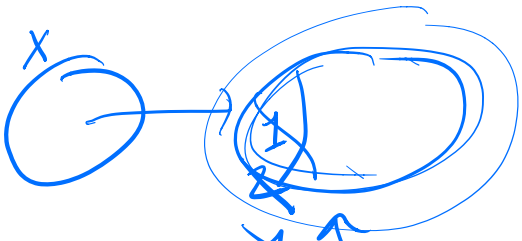
Reference counting

malloc
free

Python Ruby JS Java PHP

Lisp

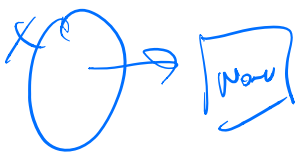
garbage-collected
(no "delete"/free)



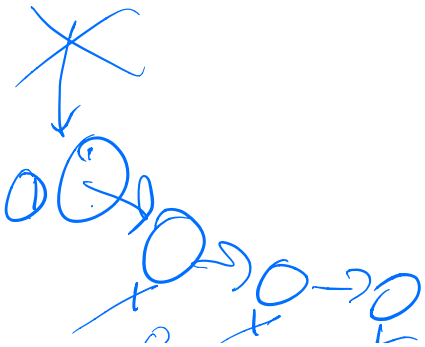
x = []

y = x

x = None



RC = 0 ⇒ safe to reclaim



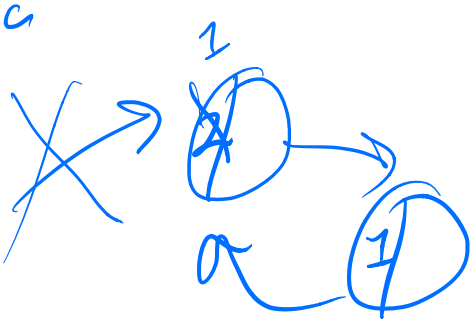
y = None

doubly linked list

a = {'key': 0}

a['key'] = a

CYCLE

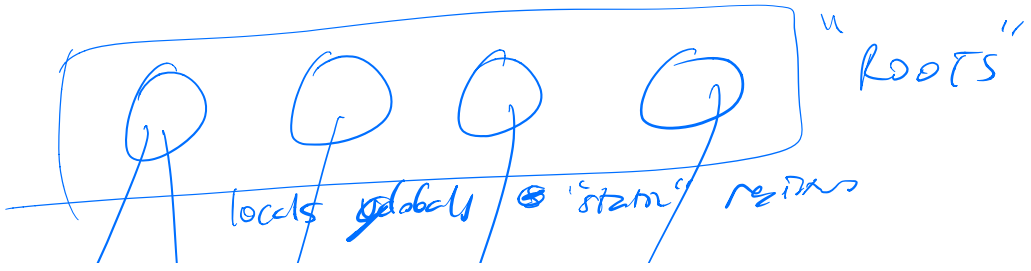


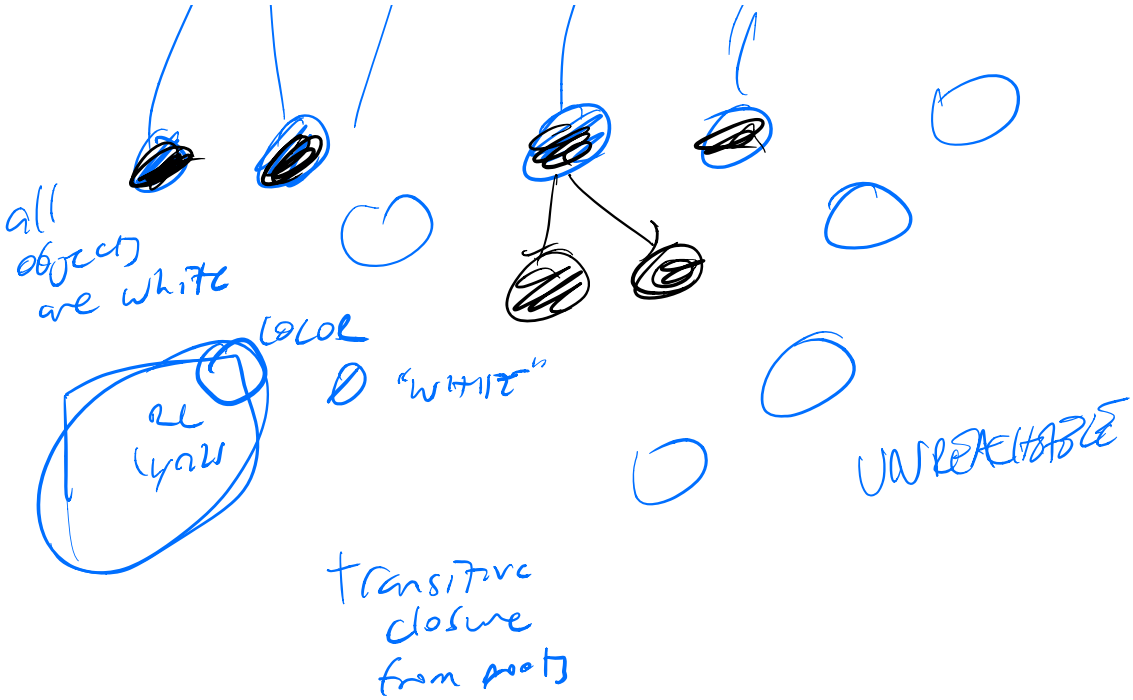
RC: (1958) incomplete GC algorithm

---

mark-sweep (1958)

"generational" GC (1983?)





David Ungar
Smalltalk

self

Ole Agesen

Urs Hölzle
Craig Chambers

prototype oriented

dynamic

GC algorithms

JIT compilation

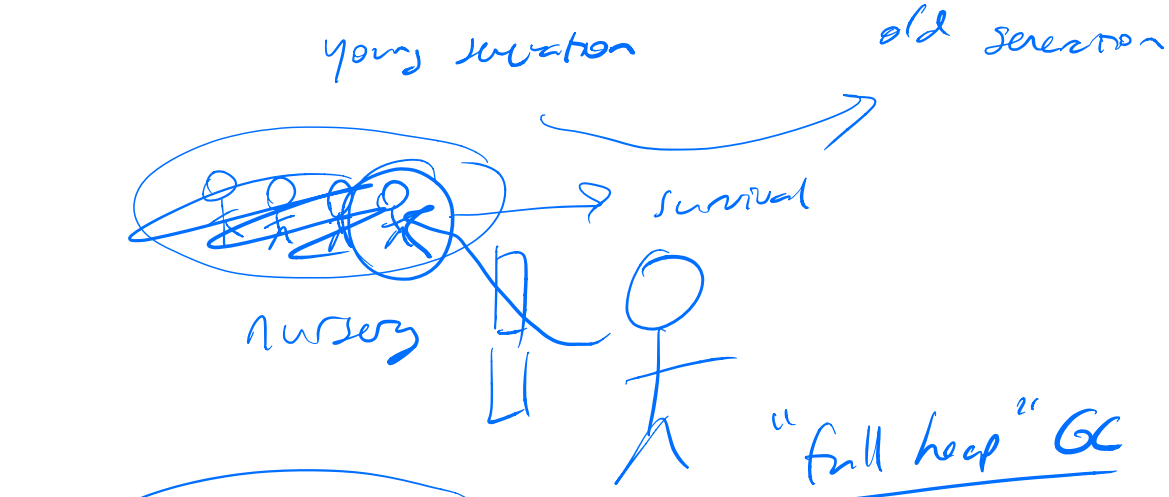
dynamic languages

Lars Bak

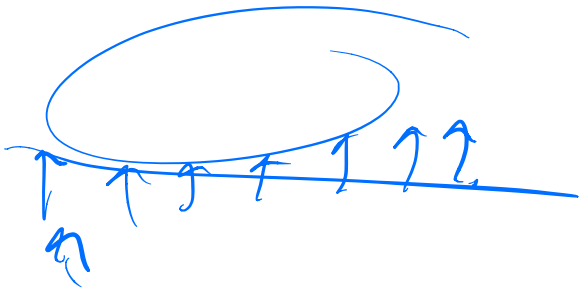
polymorphic inline cache
"PIC"

Google → V8

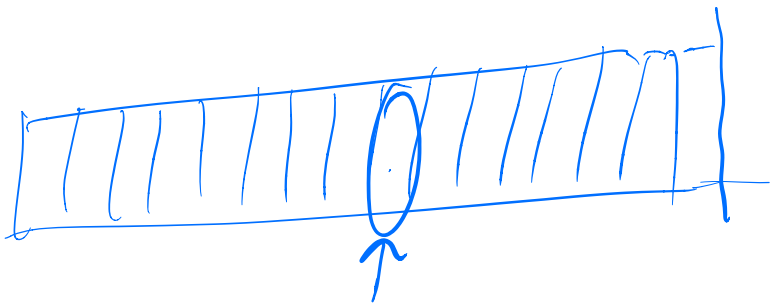
most objects die young

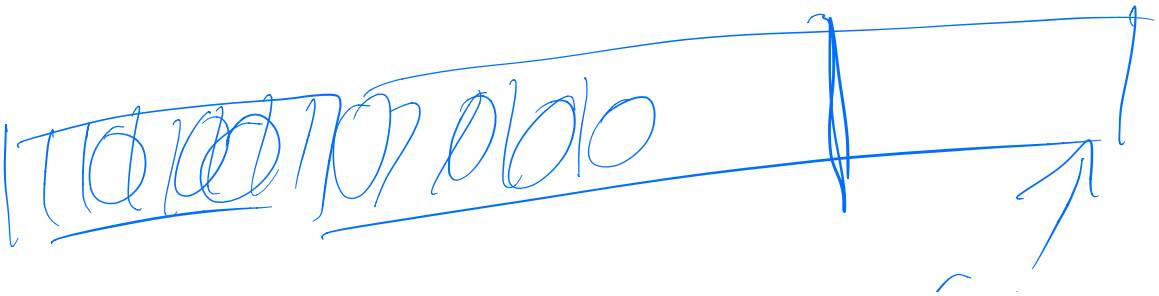


"full heap" GC
mark-sweep



gen GC →
takes pressure off the full heap GC



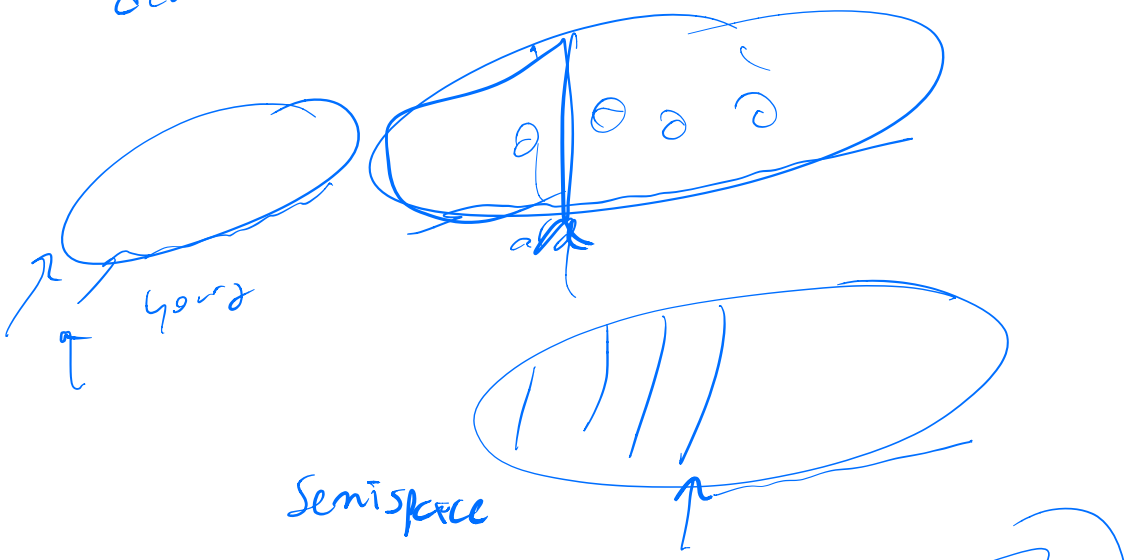


GC

~ 2 ⇒ 5x

Quantifying the cost of GC vs. Explicit dem Mgmt

"dead" objects



immortal

```
for i in range(n)
  for j in range(n) pretending
    for k in range(n)
      C[i][j] += A[i][k] * B[k][j]
```

- .pyc

PyPy

Ontogeny recapitulates phylogeny

SCRIBES
Lillian
& Sara

OS: single user, single process, no protection

multiple users, multiple processes
"time-sharing"
time slices
memory protection
file system protection
:

VM
"MMU"
memory management unit

PC — Apple
IBM PC
: 1981

Mac 1984
1 user
1 process
no protection

Motorola 6502 68000 macs
Intel 808x...

no MMU

MS-DOS
Microsoft

CP/M
8080
Z-80

Gary Kildall

Microsoft BASIC
Altair
Gates & Allen

8088

"security through obscurity"

ATET
Unix

BSD
↓
OS X

iPhone
stripped down Unix!

→ multiple processes
mem protection
security model!

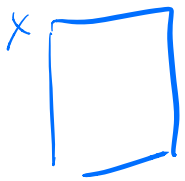
PL?

→ dynamically typed lang — values have types

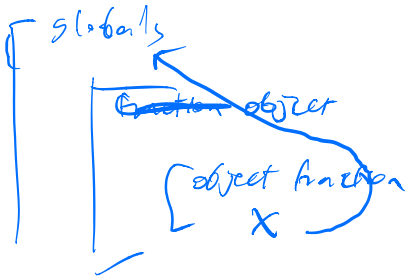
(statically typed lang) — values & variables have types
cannot change

x = [ ]
x = { }
x = " "

→ dynamically scoped
(static / lexically scoped

x 

"weird scope rules"



match the innermost scope

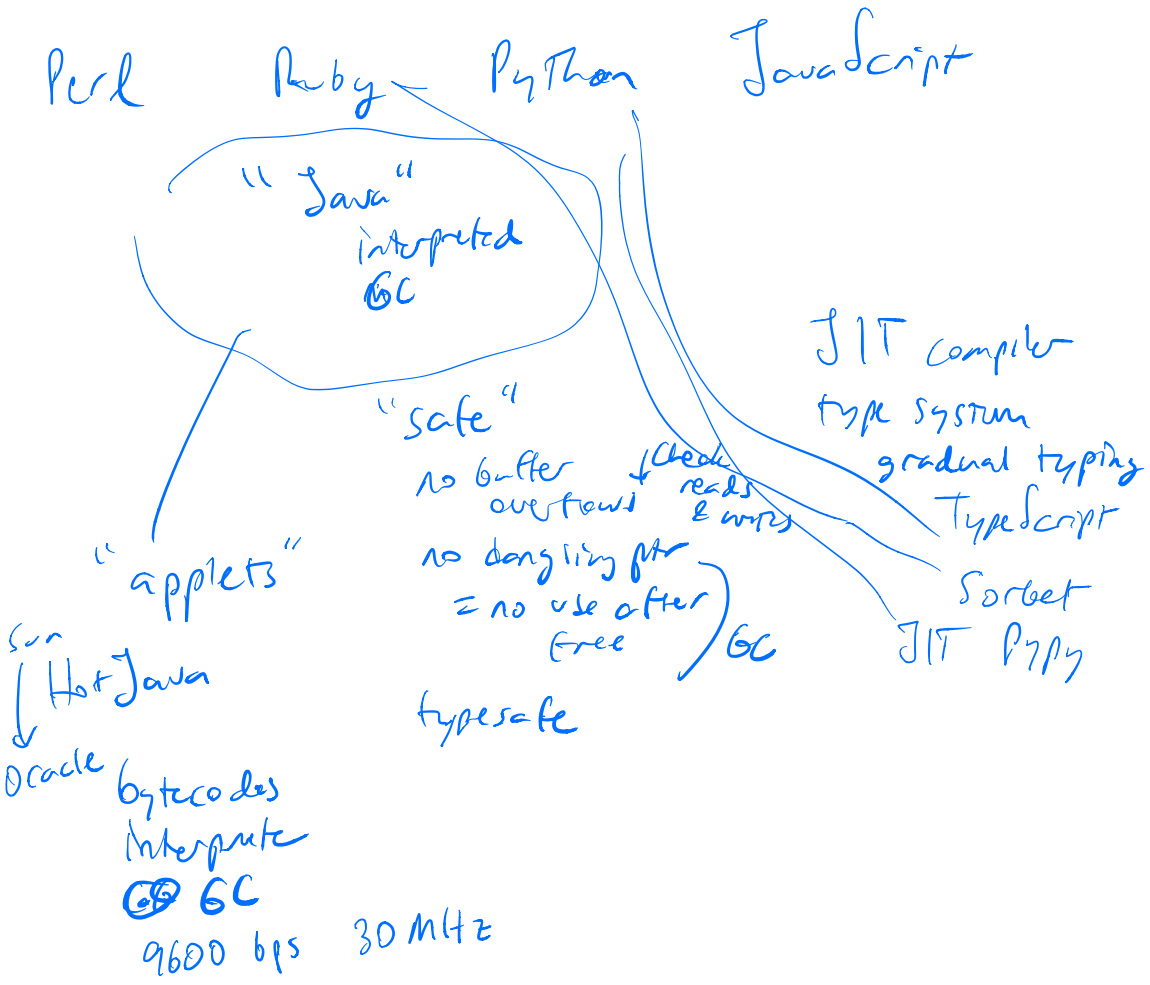
```
foo() {
  ~ x ~
  x = x+1
}

bar() {
  int x;
  foo();
}

baz() {
  int x;
  foo();
}
```

walking call stack

— Interpreter eval

hard to compile into efficient code

— reference counting GC



Netscape ← MOSAIC

Navigator

Mozilla

1995

Brendan Eich

LISP

Scheme

(((( ))))

{ }

"curly-brace languages"

LiveScript 10 days

Browsix

Doppio

asm.js

subset that is easy to compile

asm.js

```
x = (y & 0);
x = a + b;
```

Web Assembly

- safe
- efficient
- fast to parse!!

loading parsing compile

ISA Instruction set architecture

- arbitrary memory (module mem protection)
- arbitrary structure JMP GOTO
- hard to analyze

register-based
stack-based
architectures

x86
CD

NaCl
pNaCl

Compiler
Some "safe" subset of x86

ARM

PUSH
POP
ADD

ROP return oriented programming
return-to-libc

"gadgets"

CFI
Control-Flow Integrity

WASM - CISC or RISC?

simple small instruction set

JVM
Java virtual machines
Rockwell
WASM chip?

stack oriented

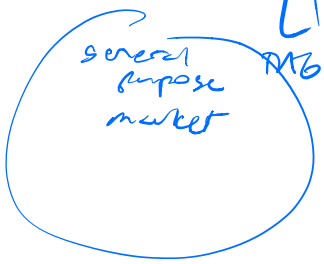
CISP machines
CAR
CDR
~ CONS
- how support for GC

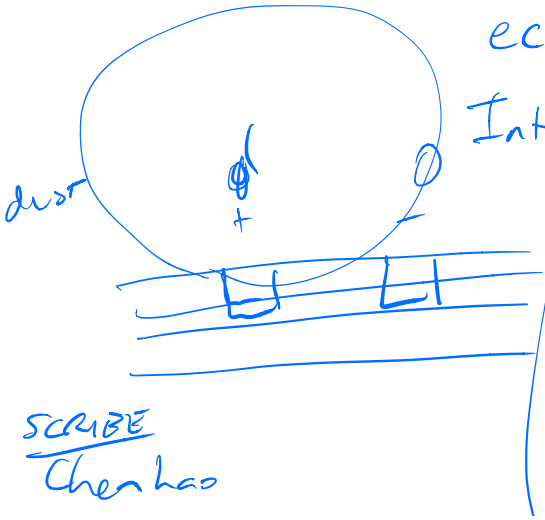
TI
Explorer

tagged architectures

niche market

special-purpose
vs.
several-purpose





economy of scale

Intel - fabs ≈ $1B

↳ very small transistors "feature size"

↳ very high "yield"

SCRIBE
Chenhao

Shaurya

microarchitecture designs

hardware process

AMD

- fabs - not as good

⇒ lower yield

→ lower profits

32-bit to 64-bit

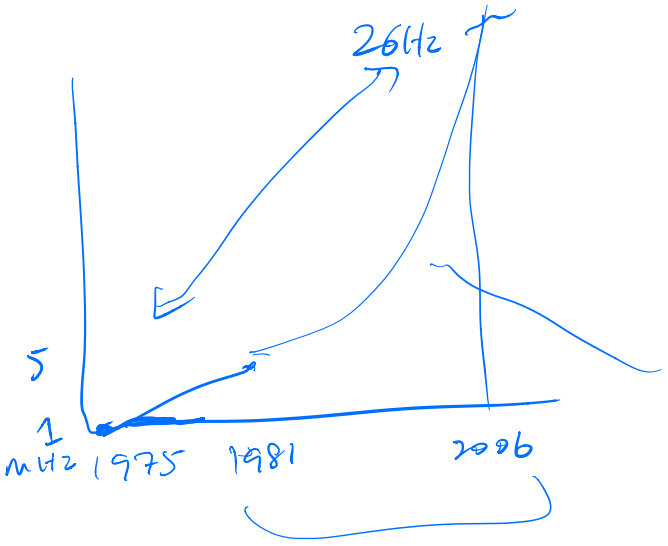
≈ multicore

Dennard scaling

increase clock speed w/o increasing voltage

↳ w/o increasing heat.

≈ 2006

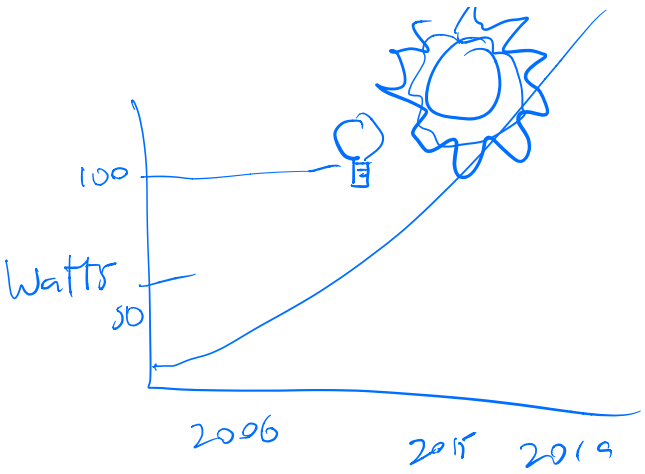


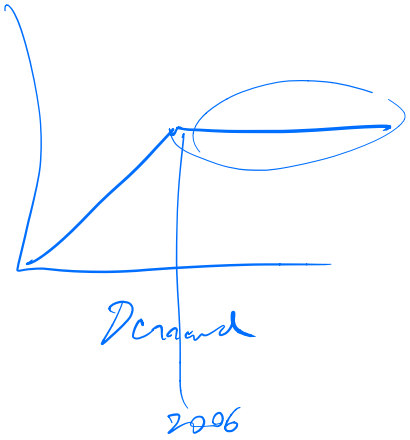
2× clock freq / 18 mos.

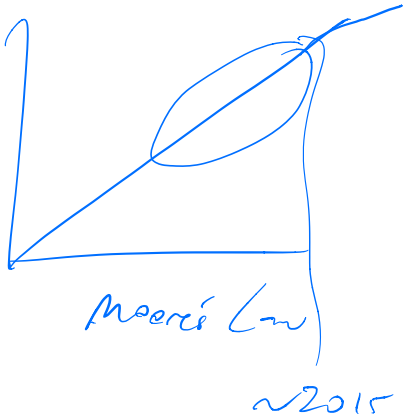
25 years

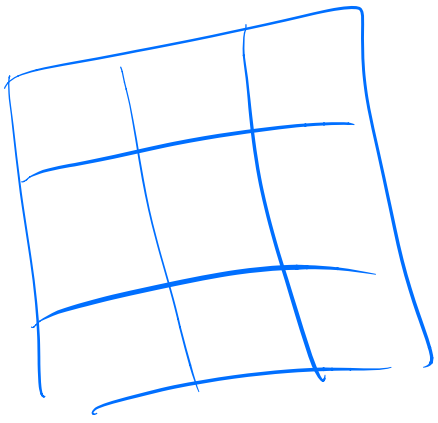
2000/5

400×


100
Watts
50
2006
2015
2019

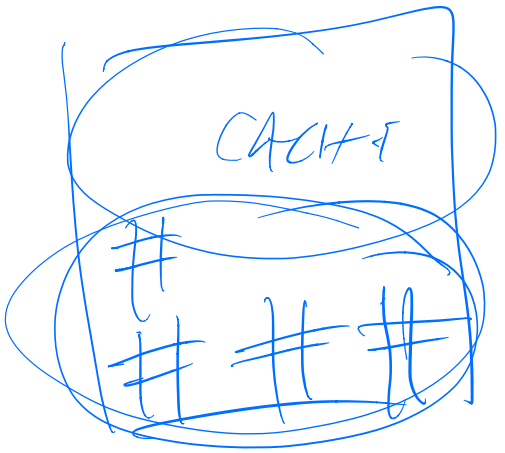

Demand
2006


Moore's Law
~2015

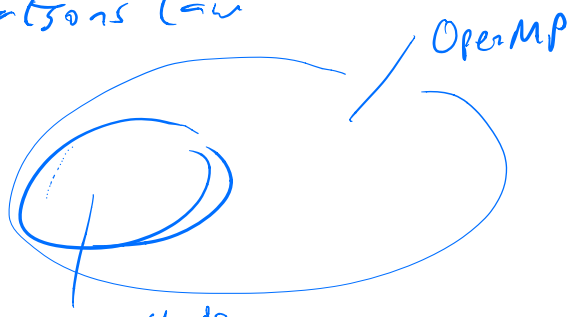


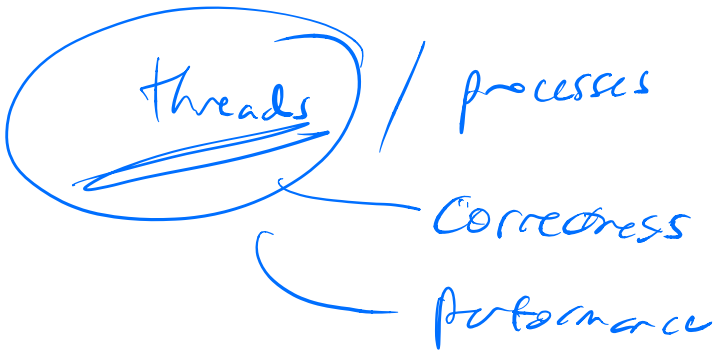

EMPTY
REAL
ESTATE

Cache

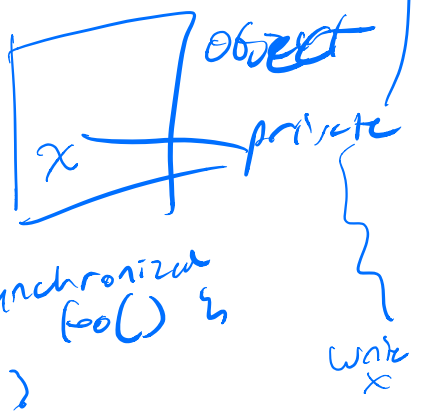


Amdahl's Law
Gustafson's Law





modularity



acquire a "lock"

pthread_mutex_acquire(&l);
X = X + 1
pt_mutex_release(&l);

"atomicity"

transactions: all or nothing

atomos — indivisible

atomic

Democritus

transaction

- commit
- abort
  rollsback

"conflict"
retry

spin lock

lock

while (l == 1) // while locked wait — sleep
;

(atomic) if (l == 0) { l = 1; } // acquire lock atomically

TST & LOCK —

(UNLOCK) $l = 0$;

"thundering herd"

fair locks
queuing locks

---

adding locks everywhere

neither necessary (for correctness)
nor sufficient (for performance)

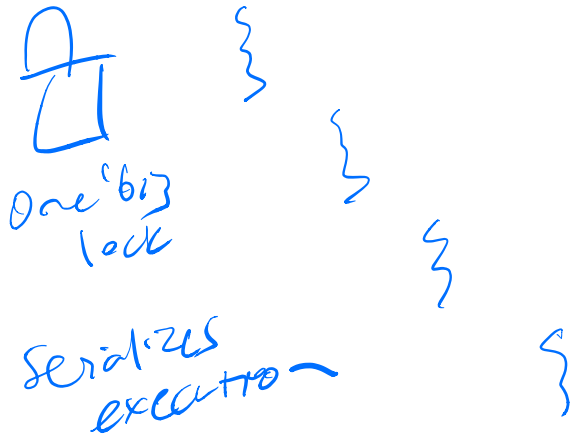
$x = 0, y = 0$ $\{ x = y \}$

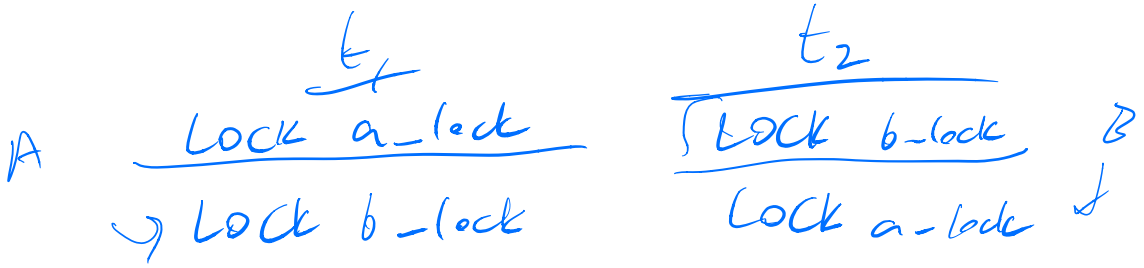
lock $x = x + 1$
$y = y + 1$

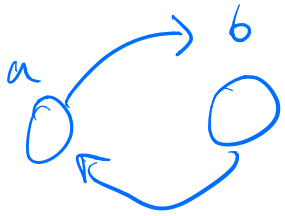
lock $x = x - 1$
$y = y - 1$

fine-grained locks → coarser
for atomicity

{ } { } { }





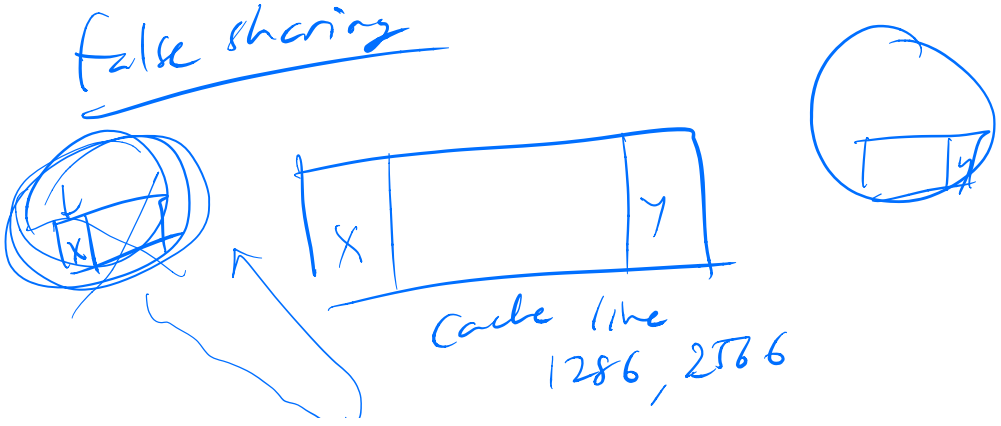


deadlock

"deadly embrace"

cycle ⇒ deadlock

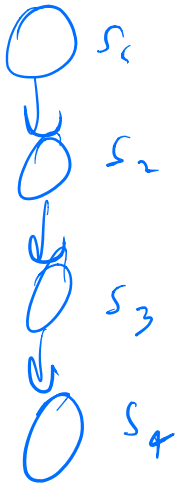
false sharing

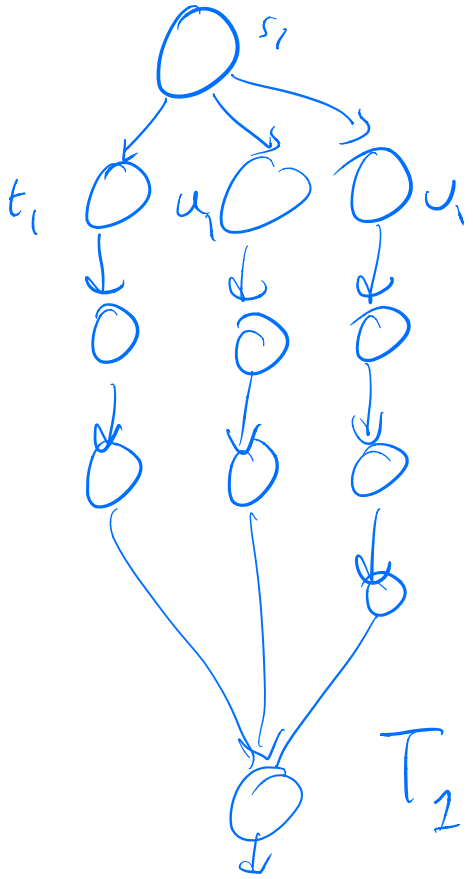


cache line
ping ponging

$\sim$ 5-10x slowdowns

Amdahl's Law





computation DAG

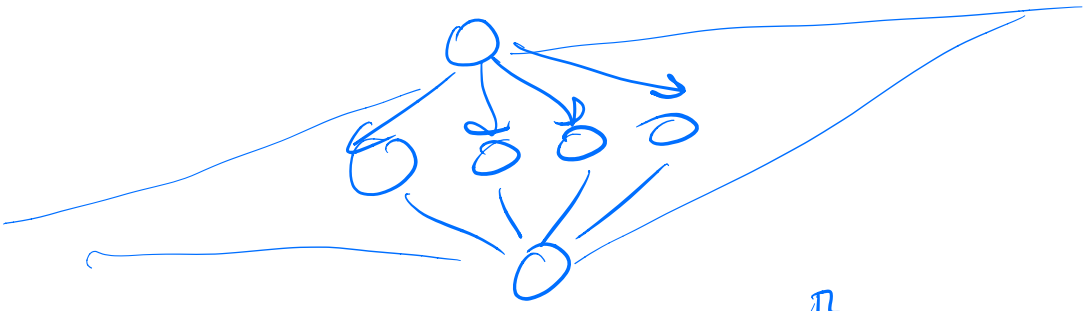
$T_1 = 4$

$T_1 = 12$

time on one processor

work = T_1

"$T_n \simeq \frac{T_1}{n}$" goal – perfect scalability



T_∞ = critical path

max from start to bottom

"height of DAG"

"serial part"

$$T_P \approx \frac{T_1}{P} + T_\infty$$

T_1: parallel work; P: parallel; T_∞: serial part

90% of program — perfectly parallelized

embarrassingly parallel

$$T_P = \frac{90\%}{P} + 10\%$$

$P \to \infty$

100s — 10s

max speedup — 10x

VM hosting ≈ embarrassingly ||

scales w/ # of customers

Amdahl's Law

— 1 workload fixed size

today's world

Netflix — indep running movies

emb. ||

→ # customers

Google

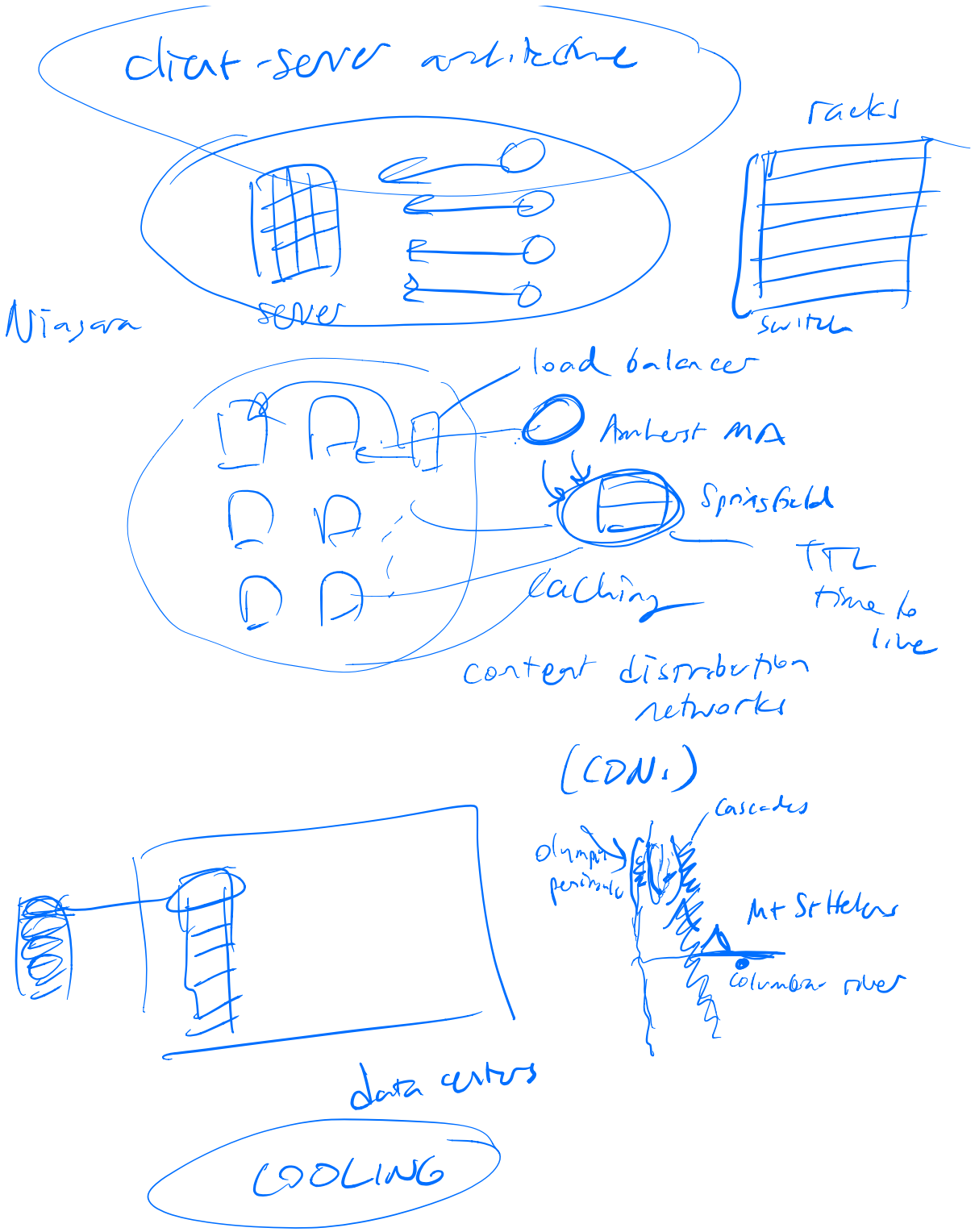
FB

Gustafson's Law

$\Delta S >> \Delta P$

↑ size of problem

↑ # of processors


client-server architecture
racks
Niagara
server
switch
load balancer
Amherst MA
Springfield
TTL
time to live
caching
content distribution networks
(CDN)
cascades
Olympic peninsula
Mt St Helens
Columbia river
data centers
COOLING
CPUs → cluster → GPUs → ASICs

multicore CPUs

graphics processor unit

GPGPUs

general-purpose

application specific ICs

cost $k/hour Amazon

E(budget) = $k+ε

OpenCL Cuda

VHDL

WebGL

"ASS"

accelerator

8087 FPU floating point unit

CPU

Intel 8088 IBM PC

SIN
TAN
EXP
LOG
.
SQRT

MMU

memory management unit

Newton iteration

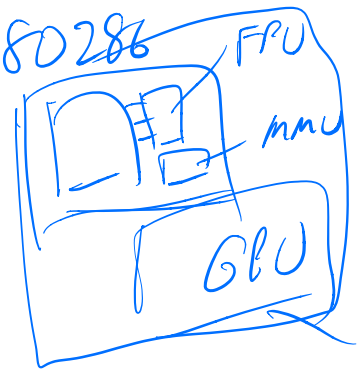
$\sqrt{156}$

$\frac{156}{2} = 78$

$\times 78$

> 156

≈ 80286



integrated GPU

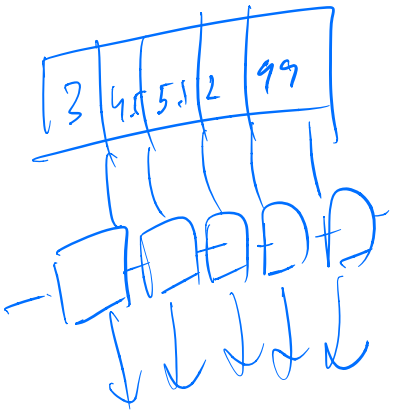
39

19

⋮

accelerators: CRYPTO

Vectors



CRAY

vector operation

highly AUX

128 bit

512

"neural" ← HYPE

s/neural/matrix multiply/g

"tensors"

TVM

NPU

TPU — Google

tensor processing unit

RISC vs. CISC

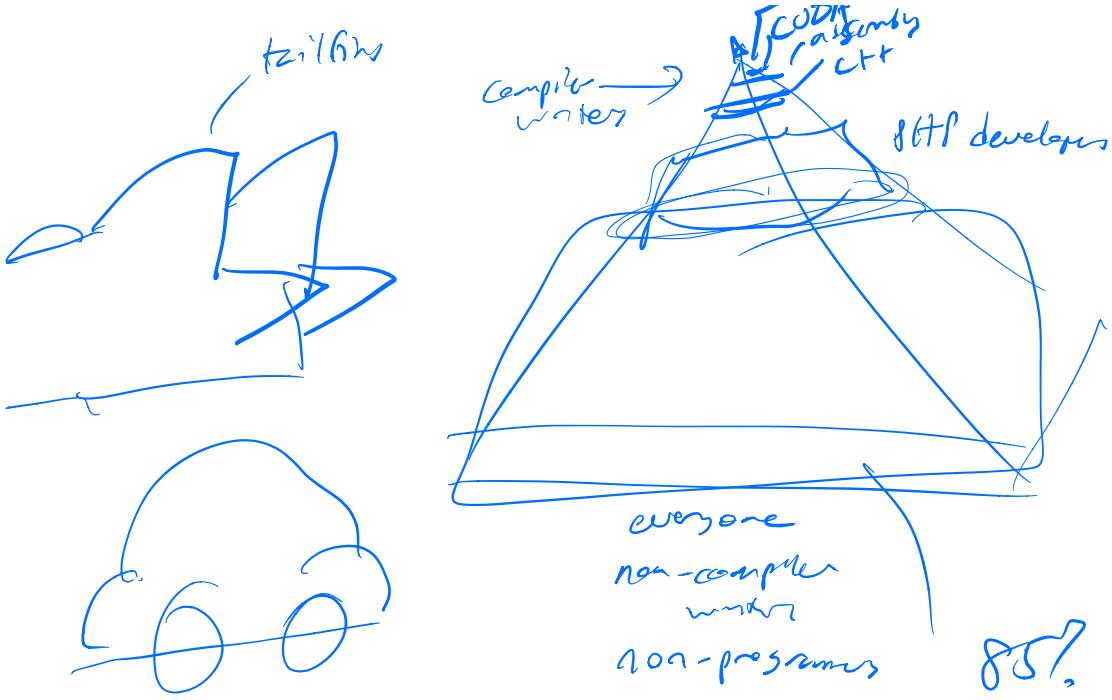
every Android device — ARM

x86 CISC

μops RISC

CHIPP

fast concurrent atomics



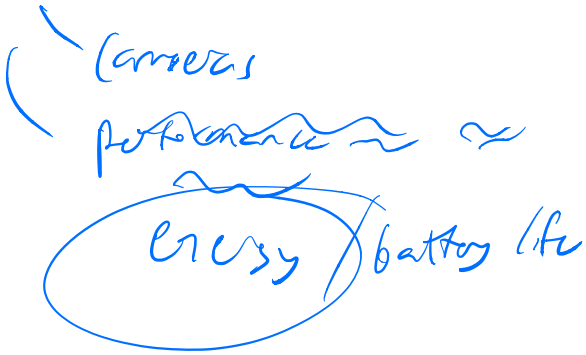
Mercedes
Volvo

rear facing
cameras
(backup camera)

Demand scaling

2 core
4 core
8 core

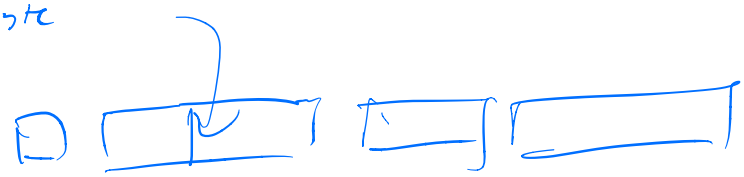
phones



standard size instructions

NOP → Lock ROP stacks

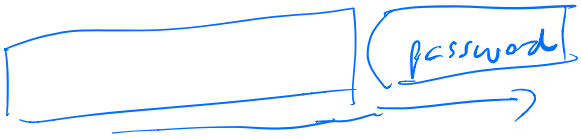
1 byte



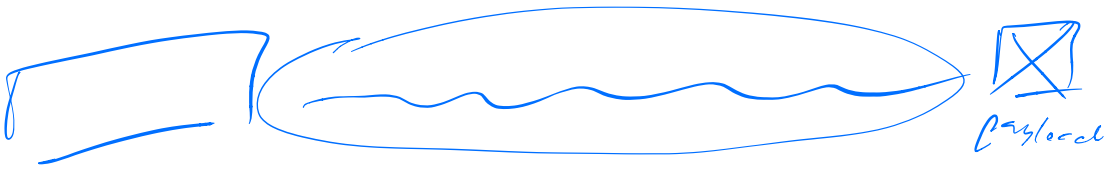
variable sized instructions

ROP

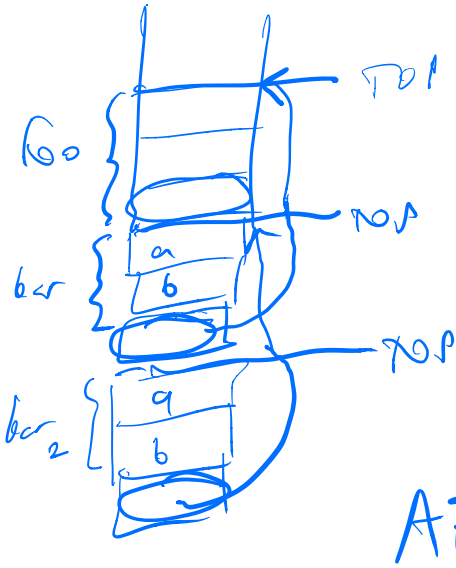
low level security exploits — memory unsafety



buffer overflow



stack smashing attack



- OLD_SP = SP
  SP = new SP (local vars + ...)
  ⋮
  SP = OLD_SP

ABI

application binary interface

```
int a;
int b;
```


```
*(b+1)=pc;
```

"root"
"pwn"

Morris worm 1988

DDoS distributed denial-of-service attack

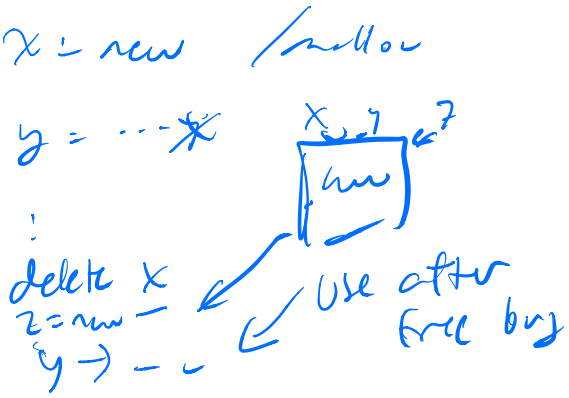
NOP NOP NOP NOP shellcode

polymorphic

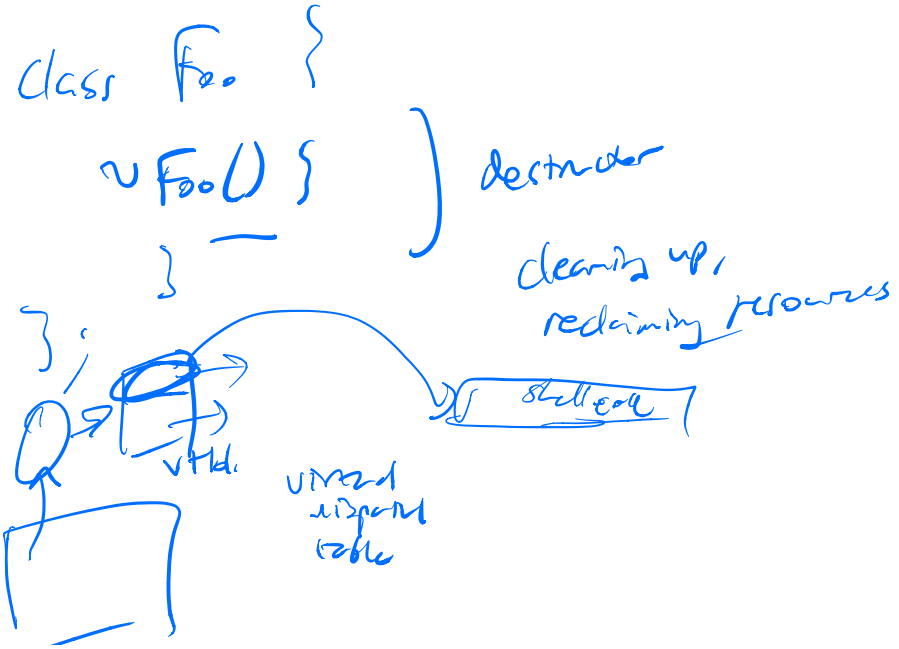
English

script kiddies

dangling ptr / use-after-free



"temporal safety"

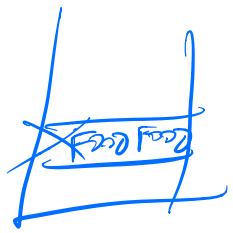


delete f;

stack smashing: shadow stacks

"stack canaries"

XOR "key"



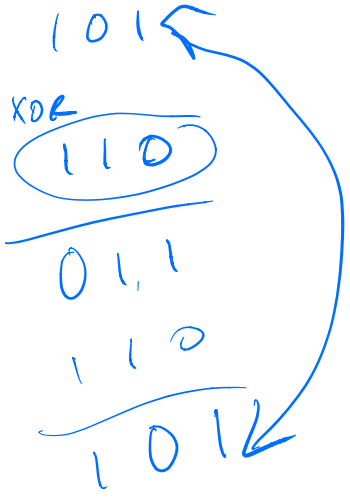
x ^ y

1 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

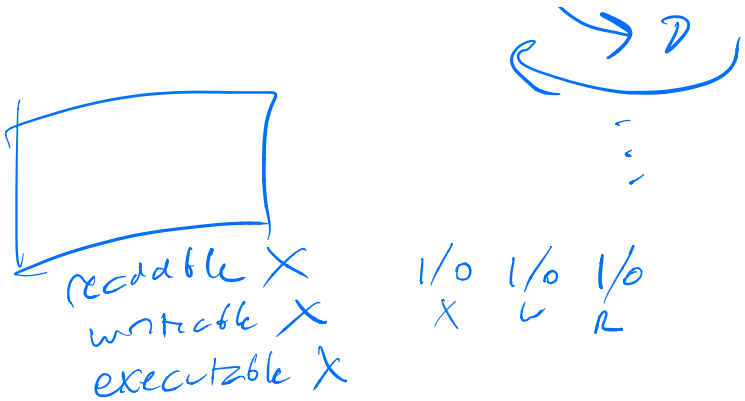
0 XOR 0 = 0



a XOR key XOR key = a

Defender

"gadgets"



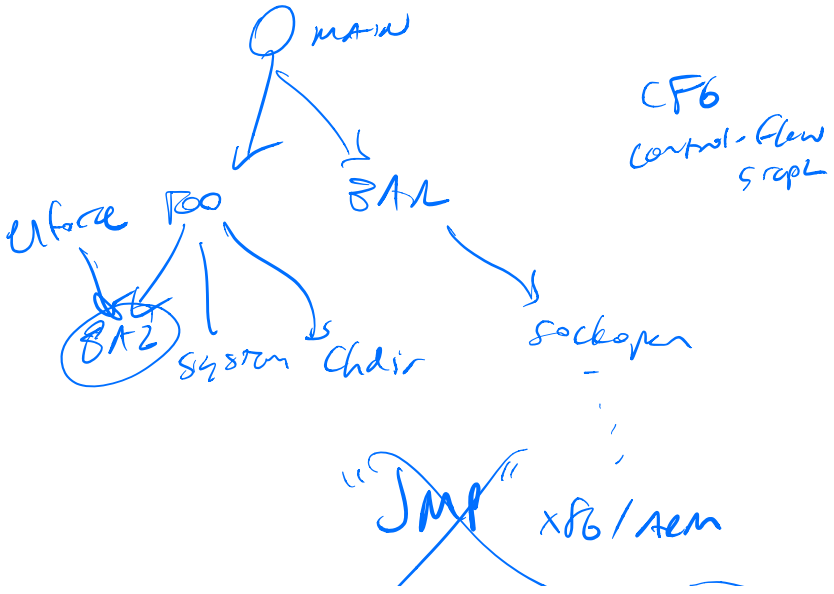
NX protection

return-to-libc attack

to "system"

"rm -rf /"

CFI control-flow integrity

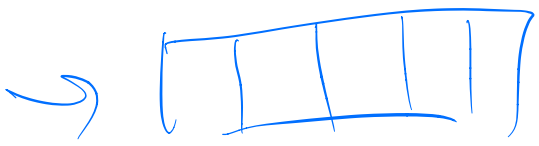


"one level of indirection"

"all problems in CS"

RISC smaller instruction set
"fixed-size instructions"

1. easier to parse "decode"
2. harder to exploit
3. faster → easier to design how to make fast



"YSX"

HTM

ARM - vector instructions

- crypto "

Intel MPX

- memory protection

Google "ASan"
Address Sanitizer

clang -fsanitize=address

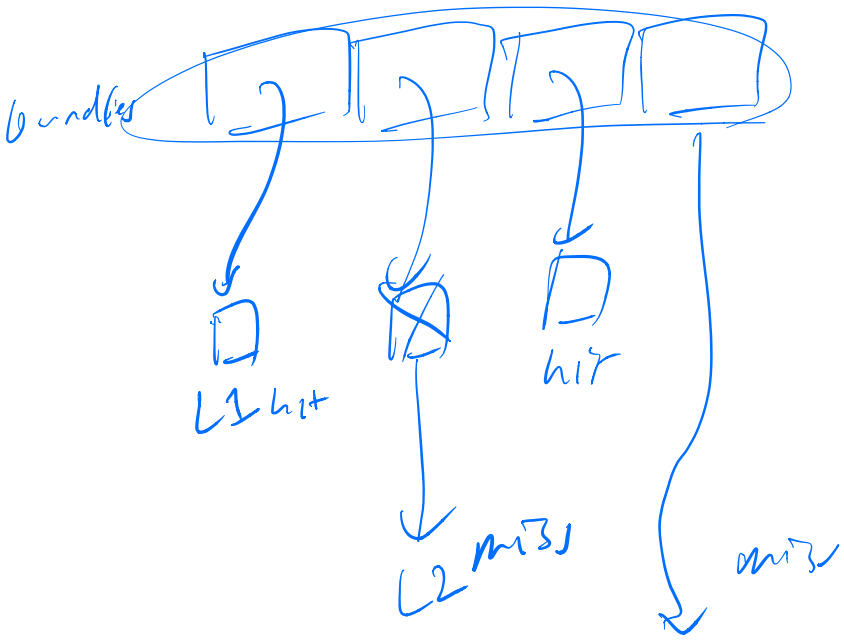
VLIW

very long instruction word

EPIC

explicitly parallel instructio

Co



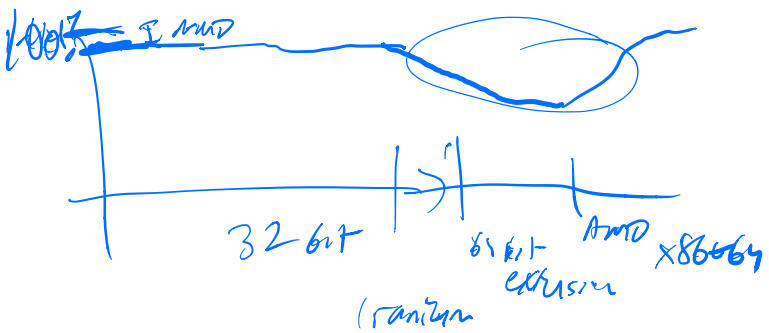
PREDICATION

IF branch

Itanium

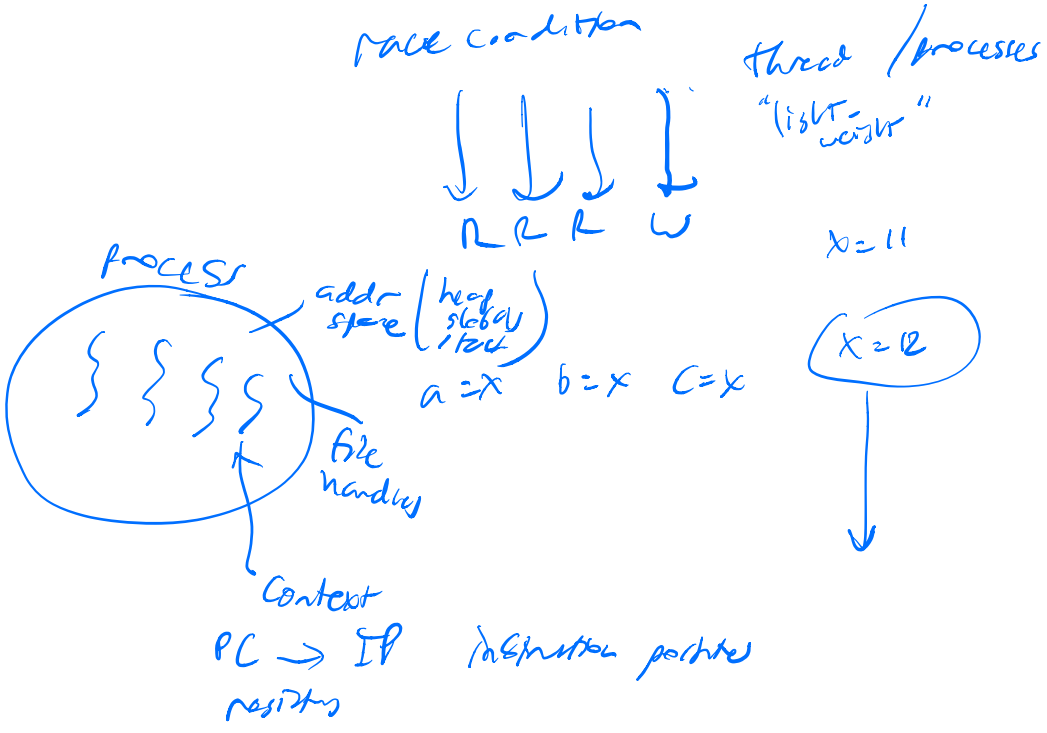
IA-64

Itanium



MOV Q
ADD Q

Concurrency



Concurrency

Errors

races

atomicity violation

Order violation } condition variables

deadlock — wake-up problem

livelock

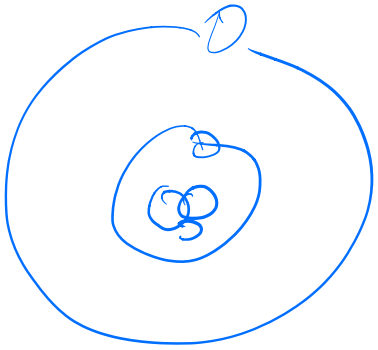
lock
while(true)
q -> wait()
check state
if(.) break

unlock

NONDETERMINISTIC

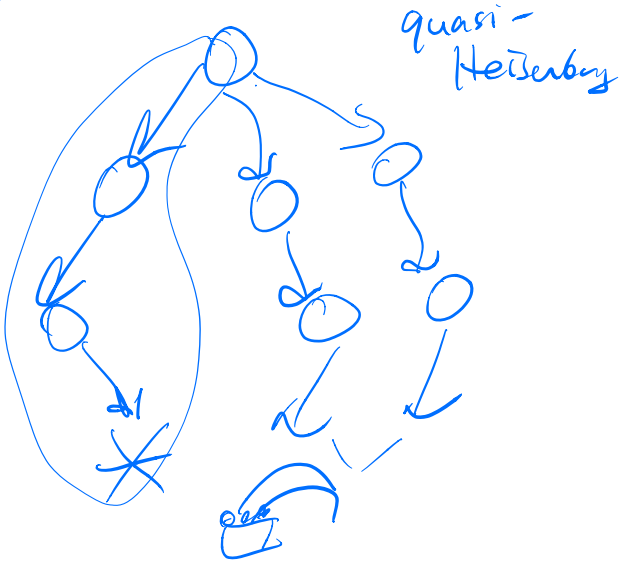
Heisenbugs

DETERMINISTIC bugs — Schr bugs



— severity?
— frequency?

Jim Gray



true
Heisenbugs
↳ Indeper failures

→ replicas
restart/retry
↳ increase odds of success

developers

debugging
determinism!
Bohr Bugs

Users

Heisenbugs
↳ retry
↳ fault tolerance
(↓ dep. failures)

D+E

100%

- race detectors
- replay based debugging

static analysis

dynamic analysis

- "knows the future"

"only knows the present & the past"

abstract values

int x = 12;

x = x+1;

x = 100;

exactly

concrete values

```
int x = 12;
f = .... read value from a socket...
if (f > 1) {
   x = x + 1;
} else {
   x = 100;
}
// x == 13 || x == 100
// x ∈ {13, 100}
```

f ↓

precise

maybe not generalizable

loss of precision

approximate

conservative

bug detection

partial specification

P never derefs NULL

"implicit specification"

segfault
NPE
races
buffer overflow
use-after-free
assertion failure

formal verification

prove program P ← code
implements spec S
↑ logic

"complete/total specification"

benign races / malignant races

"Hogwild!"

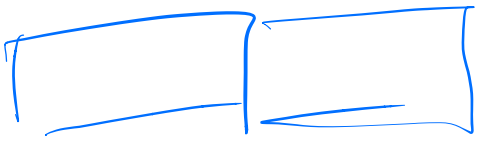
atomically
4 bytes

Word-sized read/write

X = 1
001

X = 4
100

101



double
d

d = 01

d = 2

d = ~~

" O.O

[...]

sort :

formal specification

postcondition

$\forall i : 0 \le i < N-1 :$
$a[i] \le a[i+1]$

"a is sorted in ascending order"

Bogosort

Quicksort

average case $O(n \log n)$

worst case $O(n^2)$

Sir Tony Hoare

{pre} — {post}

precondition

Mergesort

and

worst case $O(n \log n)$

quis custodes custodiet

trusted computing base (TCB)

spec + solver + code gen ⇒ generated code

TCB

≈ 1000 LOC

real : hundreds !

targets for formal methods

narrow APIs
well specified
really important

Project Everest
"https" TLS F*

Heartbleed

---

static analysis for concurrency errors
— false positives

precision

recall

print "NO BUGS HERE!"

print "BUG!"

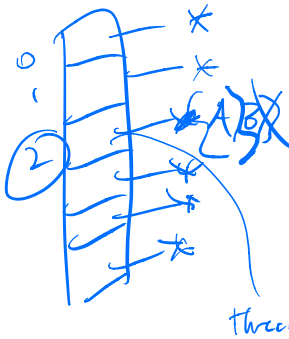
---

dyn analysis — no false positives
witness race!

— recall lower —
only reports races
in that execution

Thread Sanitizer

lockset analysis



current locks

lock A
lock B
access (2)
unlock B
unlock A

happens-before analysis

{A} ∩ ∅ = ∅

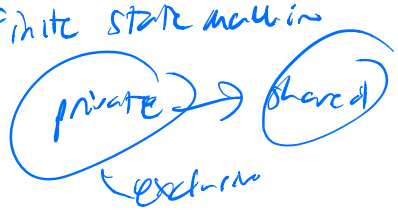
lock A
access (2)
unlock A

{B}

lock B
access (2)

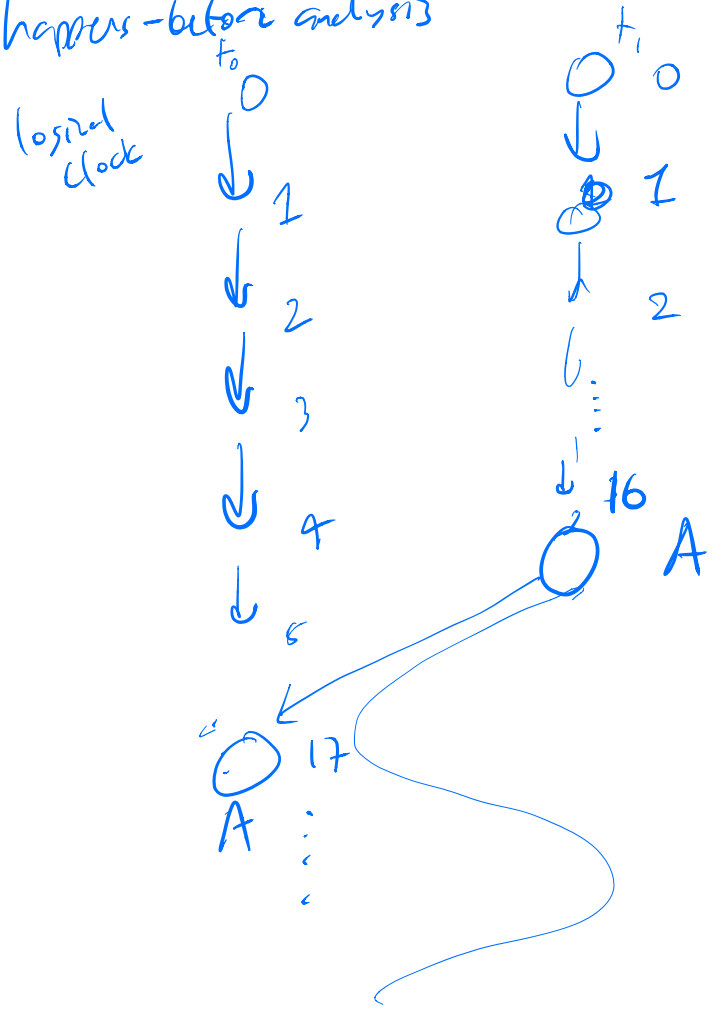
{B} ∩ {A} = ∅

Finite state machine



ERROR

happens-before analysis



Why Threads?

- parallelism
- coordination
- hiding latency ↑ performance
  - I/O
  - memory

└ average time per job
↳ time delay until "first byte"

throughput
"throughput" — rate of jobs getting done

throughput – latency tradeoff

batch

interactive

time sharing

no context switches

CPU util — 100%

/dev/tty

ed

MULTICS "Eunuchs"

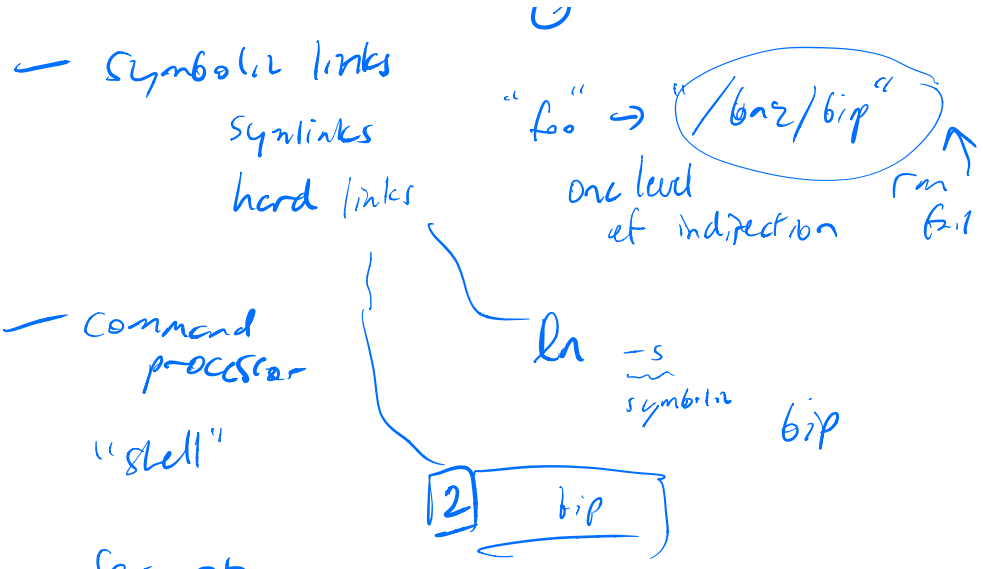
1964 – 1967

MIT + GE + Bell Labs

– hierarchical FS

all previous: "flat"

relative paths



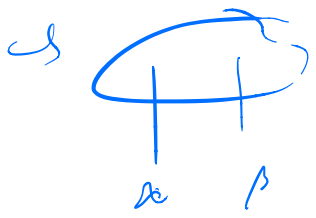
Why a choice of quantum size a tradeoff?

```
newObject
mark
clear
sweep
gc

for (auto it = allocated.begin();
     it != allocated.end();
     ++it) {
  (*it) -> clear();
}

clear();
mark(&root);
sweep();

Object *
newObject() {
```

```
Object * obj;
if (freed.empty()) {
    obj = new Object;
} else {
    obj = freed.front();
    freed.pop_front();
}
allocated.push_front(obj);
return obj;
}

void mark(Object* a) {
  if (a && !a->isMarked()) {
    a->mark();
    if (a->isPointer()) {
      mark(a->getPointer());
    }
  }
}
```

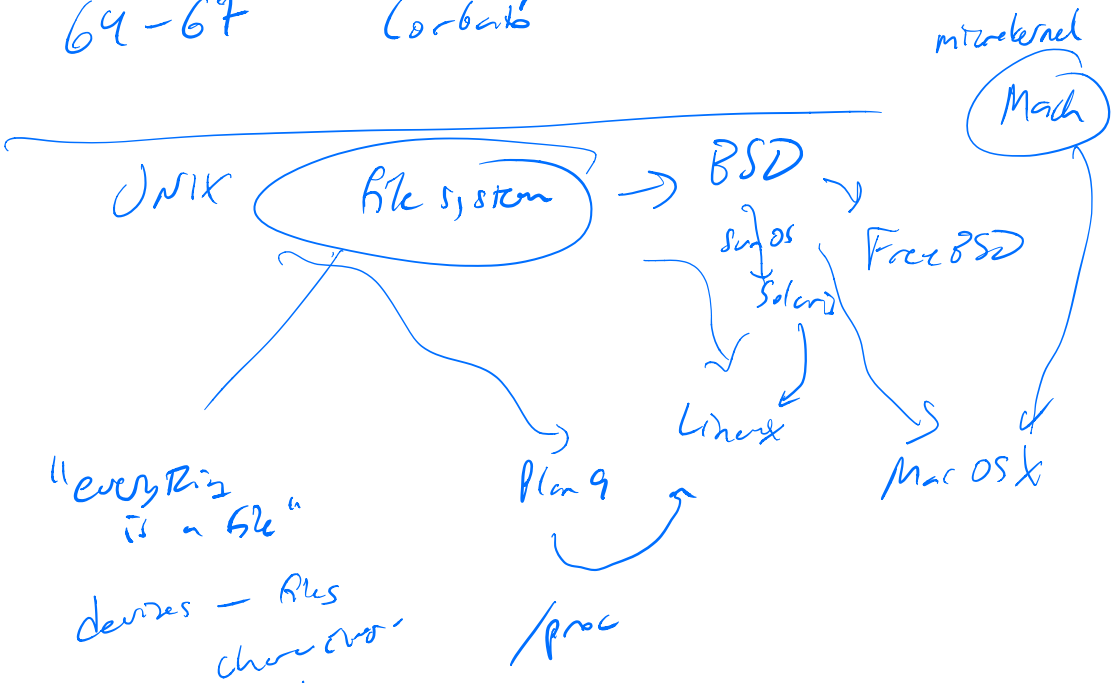
---

cons α β



hardware enforced security

64-67 Corbató



stdin stdout
stderr

"one level of indirection"

echo "hello world" > foo
> /dev/—

—— | ——— +

% (cut -f 1 -d\) | sort | uniq | wc -l

pipeline parallelism

"streaming"

data flow graph

awk — Aho Weinberg Kernighan

sed — s/——/——/g

grep — general regular expression parser

Perl — Larry Wall

Python — Guido van Rossum

Ruby — "Matz"

killer app

"Rails"

Ruby on Rails

Nyklaus Wirth

Pascal

Modula-2

Oberon

Scala — Martin Odersky

EPFL

Java servers

Spark

VisiCalc 1979

Microsoft Word

Excel

Powerpoint

Office

Anders Hejlsberg

Turbo Pascal

TypeScript

JavaScript

Stockholm Syndrome

Brendan Eich

David Turner

= KRC

↘ Miranda™

Miranda™ ... Research Software Ltd.

Simon Peyton-Jones — MSR Camb.

Phil Wadler — Edinburgh

John Hughes — Sweden

Paul Hudak — Yale

. . . .

Haskell Curry

$f(x, y, z)$

f x y z

"f 3" currying

---

pure lazy functional prog. language

int a = 12;

int f(x) {

return x * 2 + (a++);

}

1/0

f(1/0)

0

ones = 1 : ones

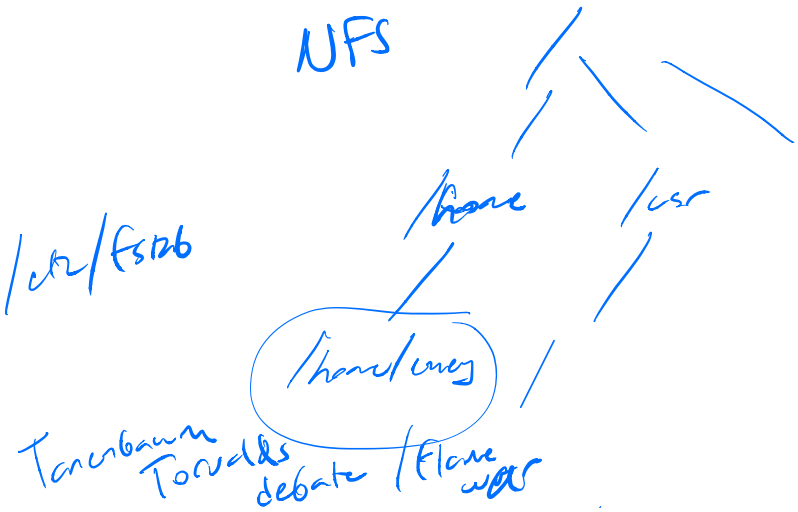
[a * 2 | a <- [1, 2, 3]]

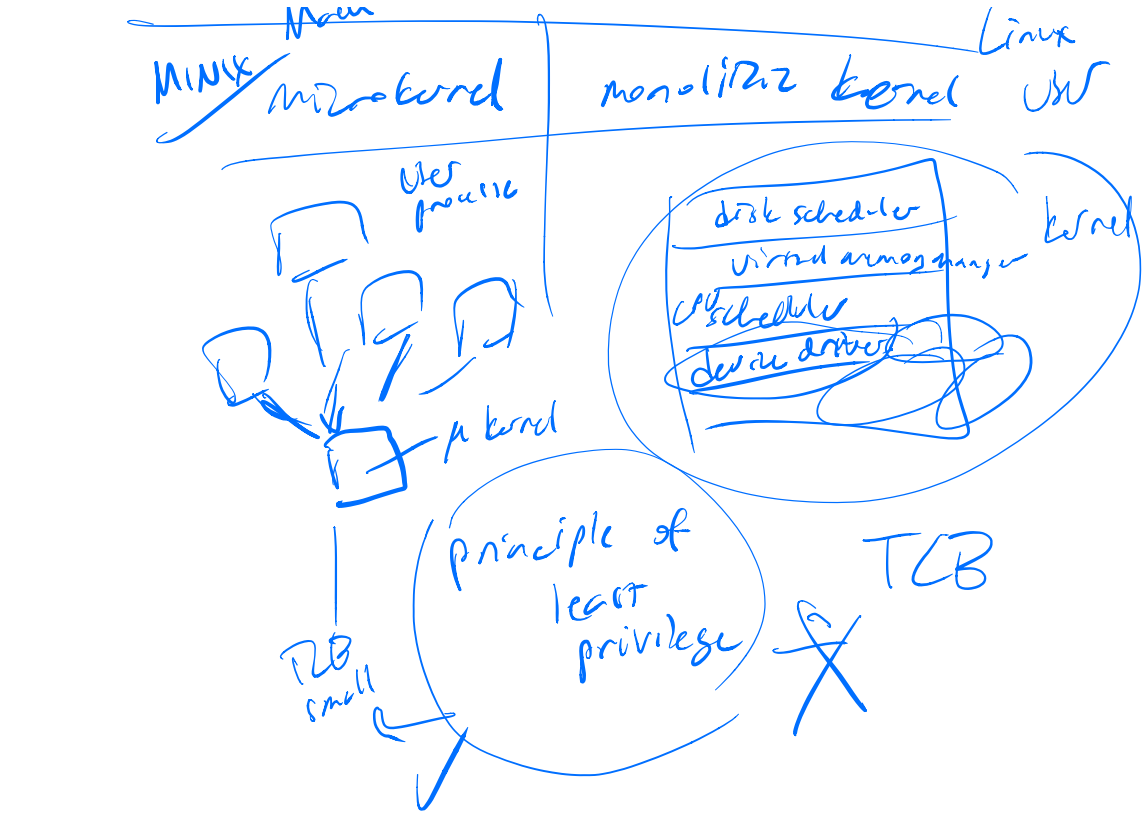
---

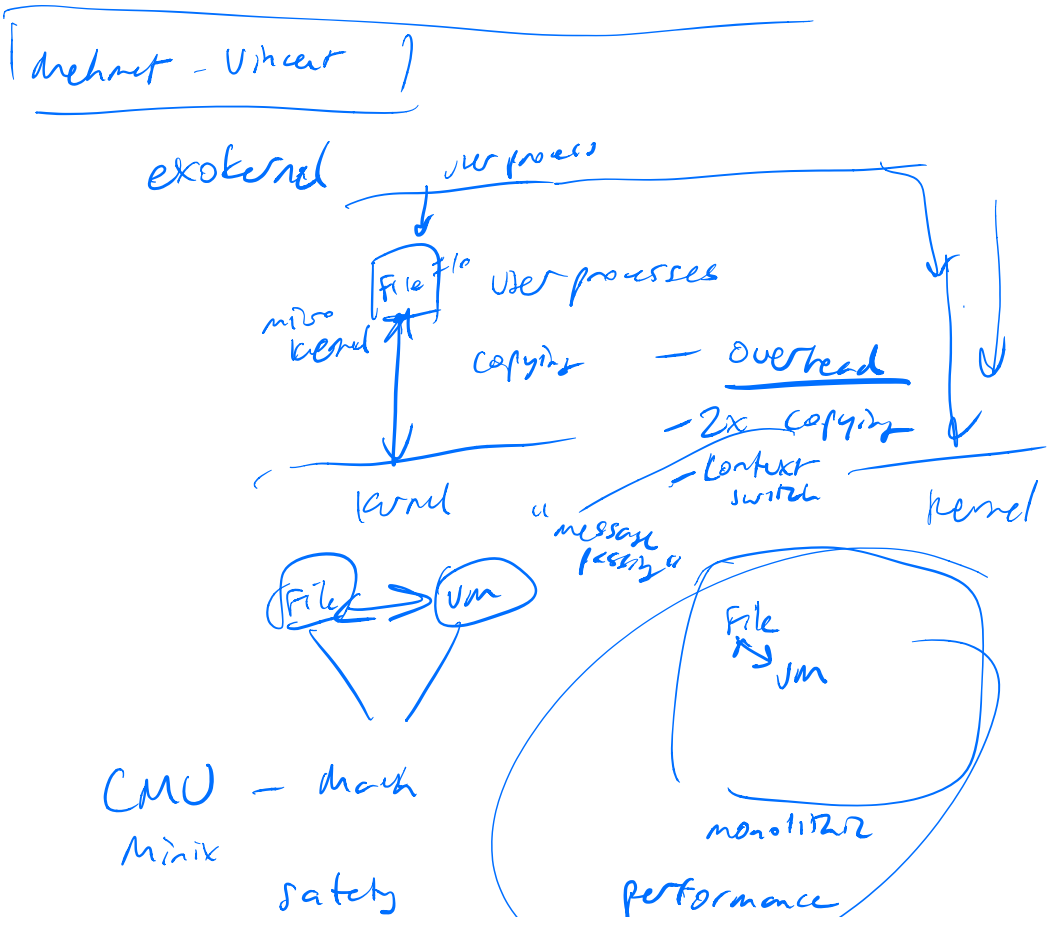
print ("foo")

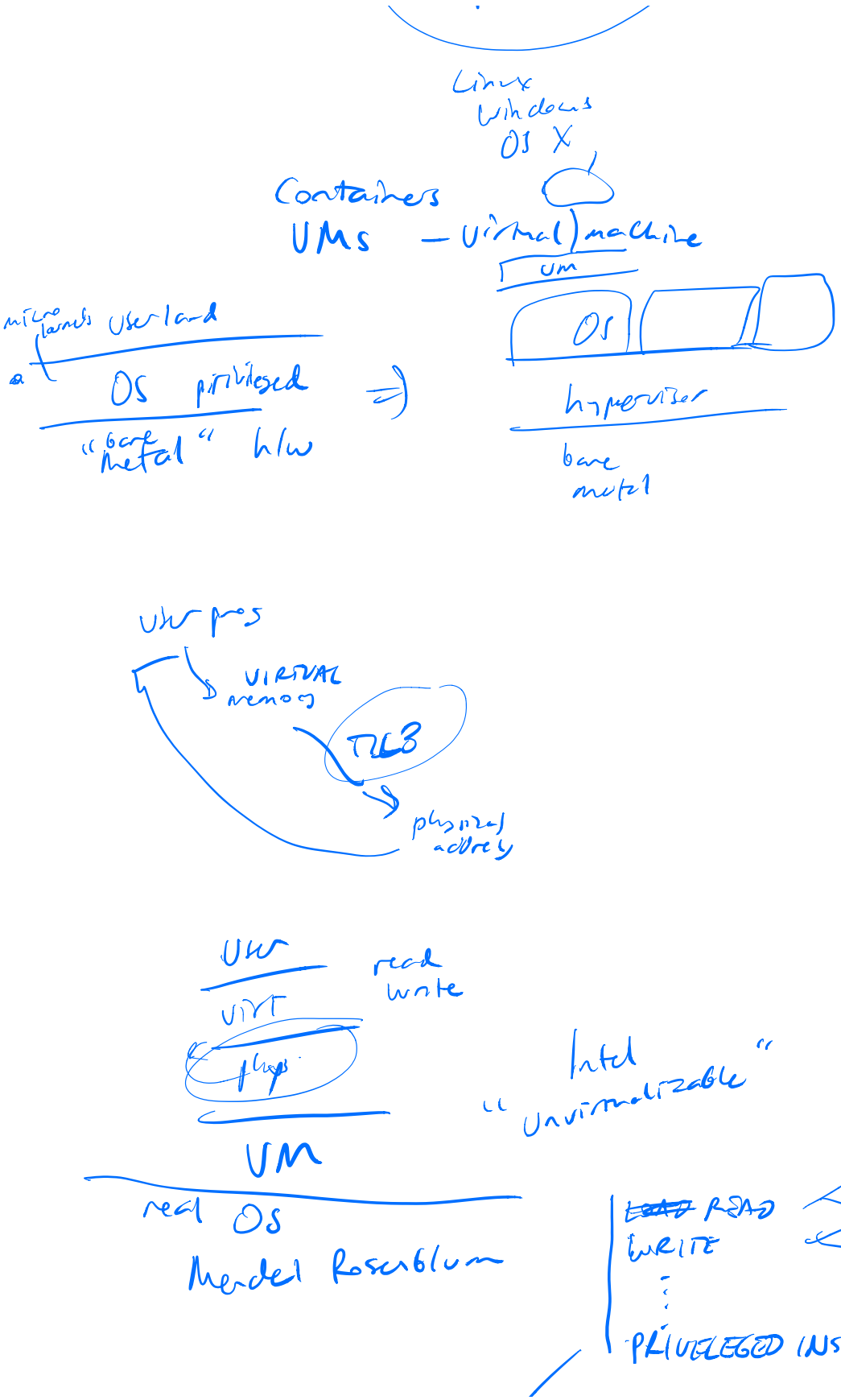
---

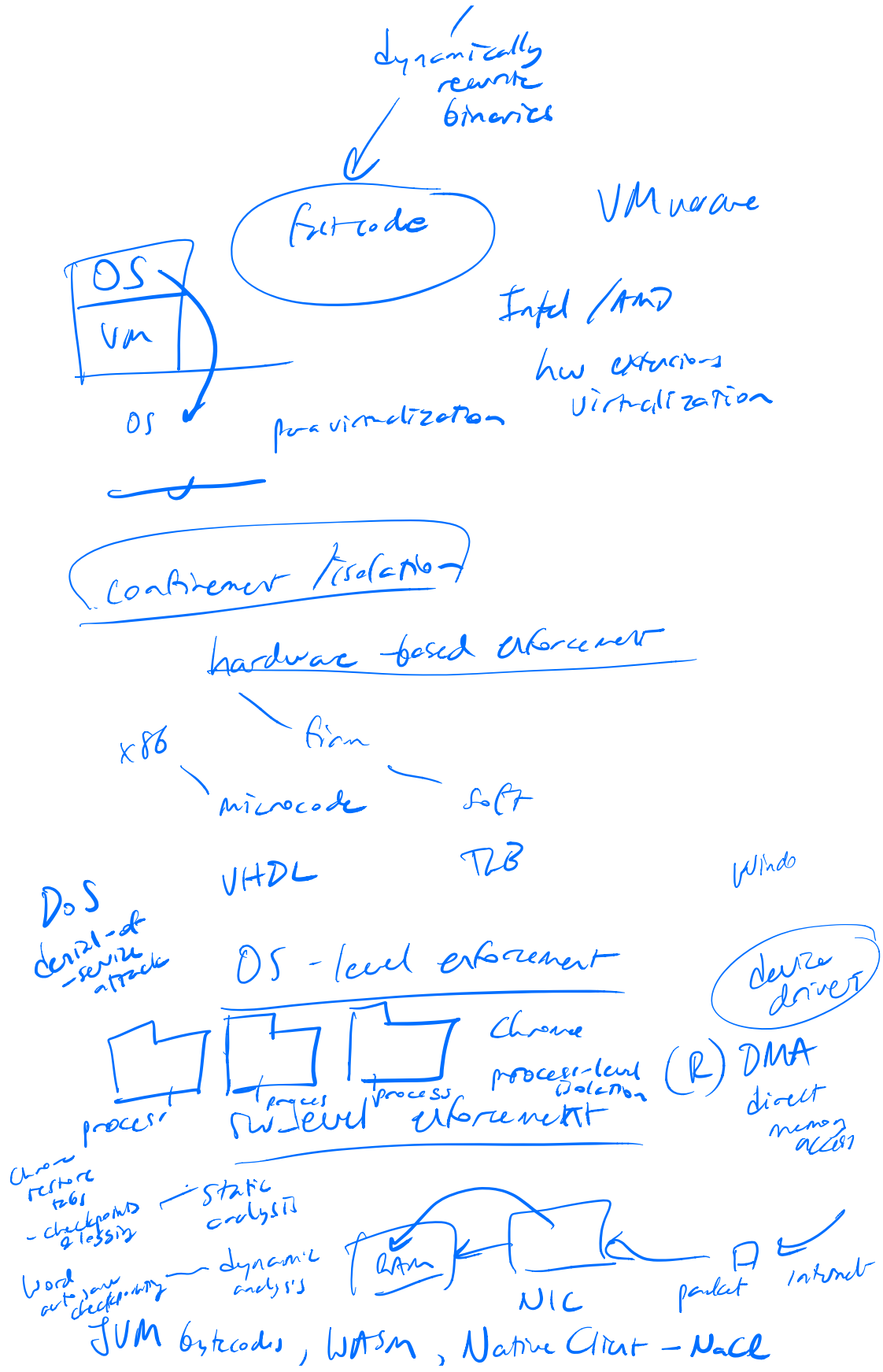
mountable file systems




Mach
Linux
MINIX
microkernel
monolithic kernel
user
user process
disk scheduler
kernel
virtual memory manager
CPU scheduler
device drivers
μ kernel
principle of
least
privilege
TCB
TCB
small


Mach - Unix
exokernel
user process
File I/O
user processes
micro
kernel
copying
— overhead
— 2x copying
— context
switch
kernel
kernel
message
passing
File
VM
File
VM
CMU — Mach
Minix
safety
monolithic
performance


Linux
Windows
OS X
Containers
VMs – virtual machine
VM
OS
hypervisor
bare metal
micro kernels
user land
OS privileged
"bare metal" h/w
user proc
VIRTUAL memory
TLB
physical address
User
read
write
VMM
phys
VM
real OS
Mendel Rosenblum
Intel
"Unvirtualizable"
READ
WRITE
PRIVILEGED INSTR.


dynamically rewrite binaries
bytecode
VMware
OS
VM
Intel / AMD
hw extensions virtualization
OS
paravirtualization
containment / isolation
hardware based enforcement
x86
ARM
microcode
soft
VHDL
TLB
DoS
denial-of-service attacks
OS-level enforcement
device drivers
Chrome
process-level isolation
(R) DMA
direct memory access
process
process
process
SW-level enforcement
Chrome restore tab - checkpoints & logging
static analysis
Word auto save checkpointing
dynamic analysis
RAM
NIC
packet
Internet
JVM bytecode, WASM, Native Client - NaCl

Windows failure → BSOD PNaCl

kernel panic DDK

- fails internal integrity check
- hardware error

Wisdom of the crowd Galton

VoxPL

"homo economicus"

rational agent — maximize gain

cost-benefit analysis

risk = P(break in)

cost = Σ costs of replacement

$30K

$\frac{1}{10,000} * \frac{30,000}{1} = \$3$

$1 ransomware $5

Fort Knox

social engineering
phishing

asymmetric attack

iOS / Android

walled garden
App Store

OS level isolation

chroot

user: something

cd /home/enes
rm -fff *

confinement
covert channels

cd /App/Foo/Something
chroot .
cd /

access control

Android

permissions at install time

SCRIBES: Jun, Kyle

"sandboxing"

- containment at app/OS level

JVM

JS
NaCl
WASM
web assembly

V8

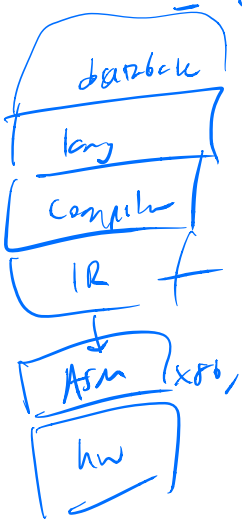
Containers
"VMs"

C++ — spatial memory error [→ buffer overflow

temporal " " use-after-free

LangSec language-level security

- Systems-level lang
  enforces spatial & temporal safety
- "memory safety"
- "type safety"
- control-flow integrity



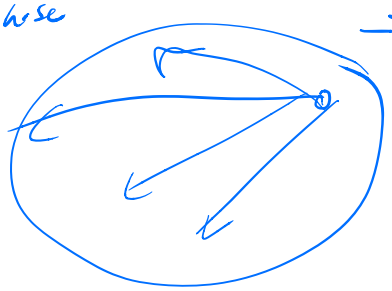
- high-level security ✗
  - SQL injection attack
- vuln low-level security
- "safe" PL?? ✗

Is mem safety (thr.) sufficient?

Rust
- memory safety
- concurrency

Ownership types
- no GC
"unsafe"
→ NOT MODULAR

"threat model"
- physical access
- cyber use



"cat and mouse" same "arms race"

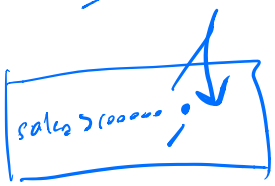
#define while if

query = "SELECT * FROM employees WHERE " + cond;

cond = "salary > 100000" position != 'CEO' &&

sqlserver.execute(query, results);
send (results);

SELECT —; DROP TABLES



XKCD Little Bobby DROP TABLES

---

drive-by attacks

zero-day vulnerability

heap spraying
JIT spraying

s = "shellcode" if

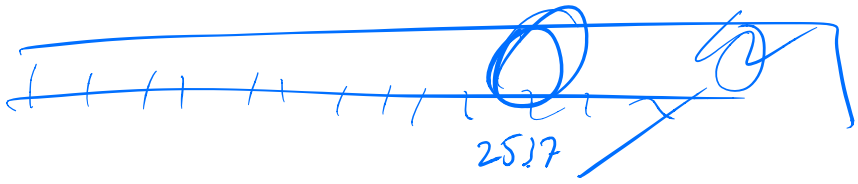
:

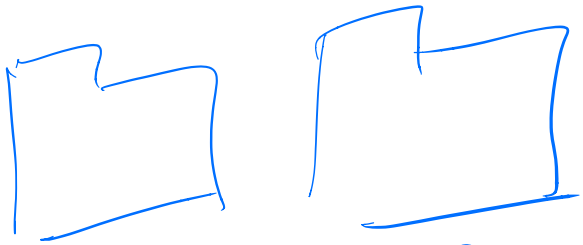
:

:

speculative execution

a[x]

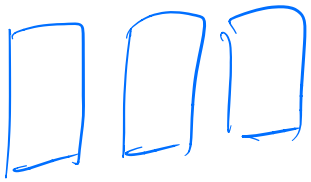


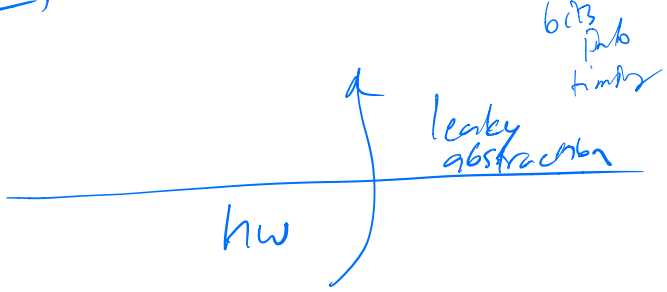


SharedArrayBuffers

high res timers

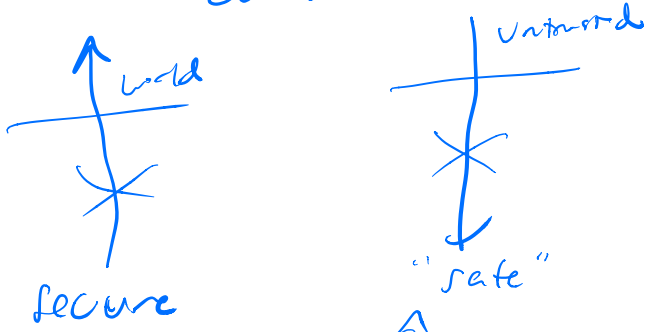
Rowhammer — Onur Mutlu





privacy integrity

duals

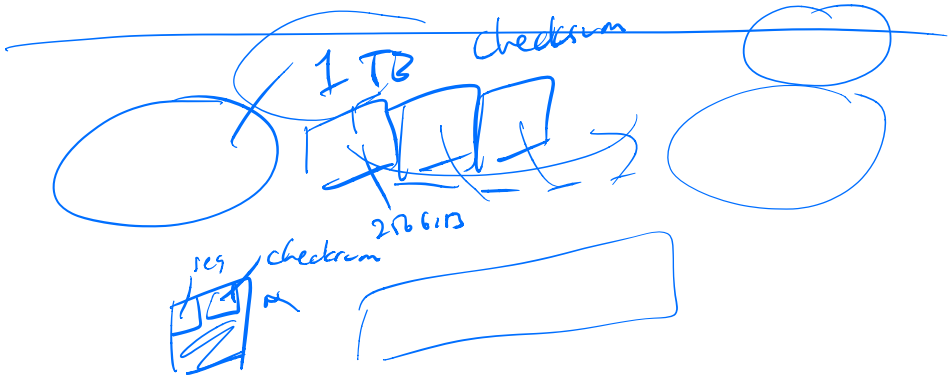


information flow

"tainting"

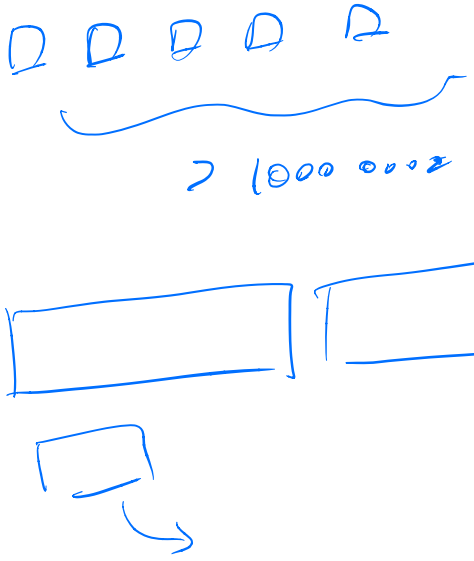
taint(n)
v = ——— socket
x = - — v
y += x ← sanitize function
send(y)

sanitize(untainted) = untainted
(tainted) = untainted



P(corrupt packet undetected) = 1/1,000,000

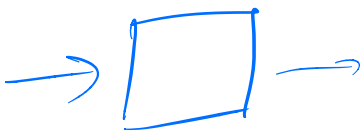
silent failure

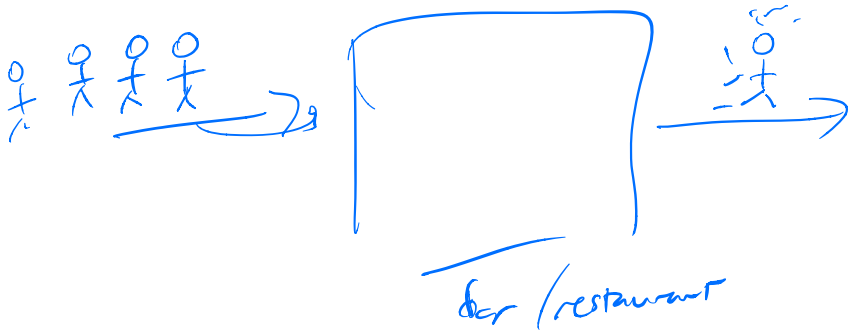


MOAR BITS

Shedding load

~ "admission control"





$L = \lambda W$ Little's Law

L: how many people waiting in service

λ: arrival rate

W: service time

100/hr 1 hr/customer

$E[L] = E[\lambda] \cdot E[W]$

backlogged

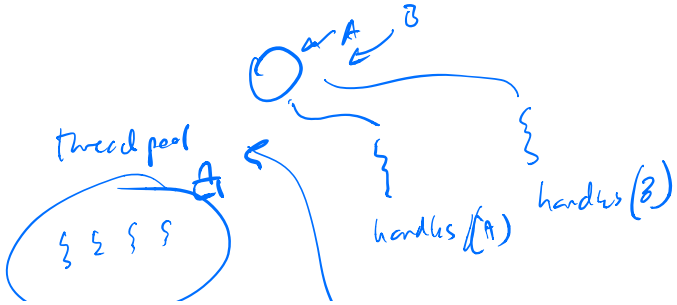
load shedding

admissions control — bouncers

QoS

DoS

"thread pool"



N = 30
N = 100

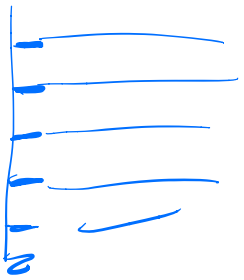
admissions ctrl mechanism

cryptography

"security through obscurity"

codewords

"stesonography"



"key"

Caesar cipher

key = 5

A B C D E F G ...
E F G H I J K

Symmetric crypto scheme

A B C D E F ...
Z Q G A F R ...

DEAD
AFZA

frequency analysis

Ciphertext

Q Z Q

E
T
A
O
I
N
O

0 — — — 0

THRDLU...

One-time pad

XOR

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

notating keys
codebooks

Enigma

known plaintext attack

- decryption!

Goal: encrypted message ≈ random noise

⇒ best you can do is brute force

— make key COMB

probability

$E(\text{decrypt the bank box}) = \frac{1}{2} * \text{key space}$

(e.g. 2^{1024})

2^{2048}

if you get my key, you can't: decrypt, encrypt) asymmetric keys

---

encrypt w/o decrypt

↖ vice versa

attestation / signing

Diffie Helman

one way function

$f(\alpha) \rightarrow \beta$ easy ✓

$f'(\beta) \rightarrow \alpha$ hard!

prime factoring is hard

$A \wedge \overline{BC} \wedge D/A$ ...

A : T

C : F
B : T
D : T

$n = pq$

Alice & Bob

Mallory Eve "Sybil attacks"

reputation based systems

Amazon reviews
Yelp
TripAdvisor
Netflix
⋮

$E_A(S) \rightarrow m \rightarrow D_A(m) \Rightarrow S$

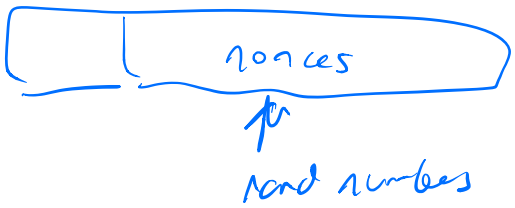
(e, d, n) e coprime to d if relatively prime to $(p-1, q-1)$

Signature = $D_B(M) = s$

encrypt $E_A(s) \rightarrow s\_alice$

Alice $D_A(s\_alice) \rightarrow s$

now $M = E_B(s)$



PKI

public key infrastructure

root certificate authorities

quis custodiet custodes?

$$\text{availability} = \frac{MTBF}{(MTBF + MTTR)}$$

mean time between failure

mean time to recover

ROC

recovery oriented computing

fail-safe vs fail stop

fail-safe → rollback ("undo") error

fail stop → stop or crash

fail-over — replicas

State replicated

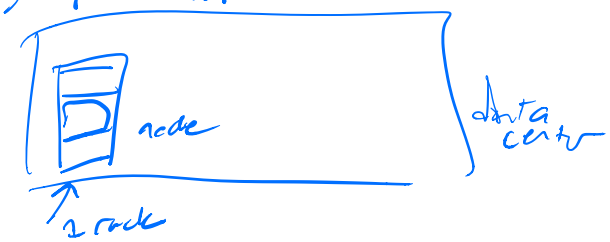
replicated state machines
Bressoud & Schneider

replication

Single point of failure

Boeing 737 Max

geographic distribution



Correlated failure = BAD

GOAL: independence

$P(\text{failure}_1) * P(\text{failure}_2)$

$\frac{1}{1000}$ $\frac{1}{1000}$

$= \frac{1}{1{,}000{,}000}$

fault tolerance
- hw — RAID
- sw — transactions (ACID) — replication

BASE

Heisenbug — non-determinism

Bohr — determinism

f(input) f(input)

Bohr: correlated

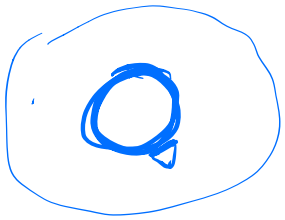
Heisenbug: independent!

SAT: $P(\text{failure})_{\text{Heisenbug}} = 1/100$

$E(\#\text{ of people sees error}) = 1 - P$

$= 99\%$

hw non-det failures



ECC RAM

error-correcting code

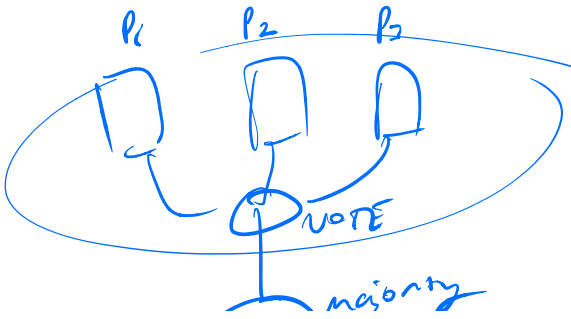
SECDED

Single error correction

double error detection

trinodular redundancy

3 things



$P = P_1 * P_2 * P_3$

Transaction-s

Atomic
all or nothing

Consistent
in face of concurrency

Isolation

Durable
- committed
- "stable" storage
(persistent)
- hard disk

log of updates
journal

WRITE ADDED class,
WRITE " "
WRITE SUBTRACT ...
DROPPED

ABOUT to COMMIT
COMMITTED

Checkpoint "snapshot"

Concurrency control:
locks - pessimistic
- optimistic
"conflict" - ABORT
rollback & restart
retry

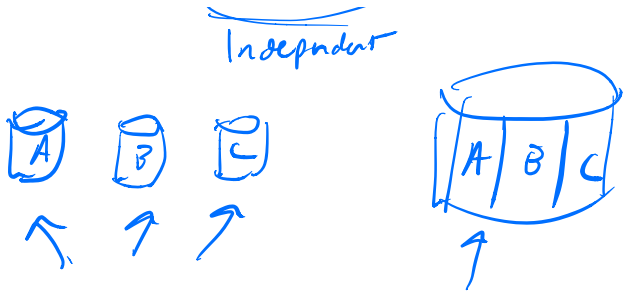
RAID

NOW
network
of
workstations

RISC

disks → bigger → more expensive!

Redundant
Array of (Inexpensive Disks)



f = P(failure of a disk) = 1/1000

Success: $1 - f = \frac{999}{1000}$

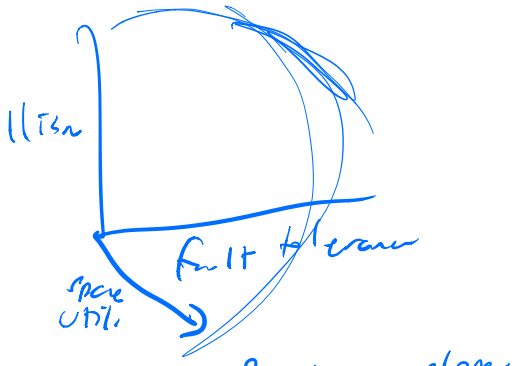
$(1-f)^2 = \frac{998}{1000}$

$\simeq .997$

10 = 99%

100 = 90%

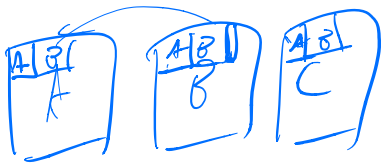
1000 = 37%



balance latency
2 fault tolerance
1 space

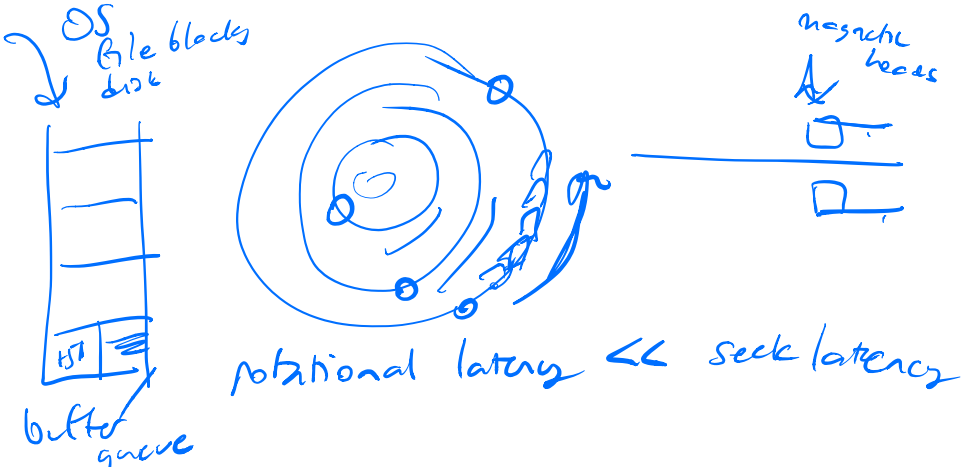
Pareto envelope

speed



Hamming code

RAID-5



rotational latency << seek latency

sw fault tolerance

- failure-oblivious
- probabilistic memory

approx computation

loop perforation

```
for(int i=0; i<N; i++){
}
```

Bolt

---

talk tomorrow at noon

Caroline Lemieux

CS 150/1

12:15 · Wed

---

failure-oblivious

- assert(0)
- throw exceptions
  - ^ unhandled

general

- x = 1/0
- x = *(0)

} later programmer intent

- These are bugs

"always bugs"

— x = (null) → foo

— exit(-1)

exit(0)
correct execution

— memory safety errors

buffer overflow (spatial)
dang ptr / use after free (temporal)

x = new char[9];
x[9] = 'z'
x[9...15] SAFE
[.....] "overflow"

CAVEAT!
— concurrency errors
race

"benign races"

C++ memory model
Sarita Adve
Hans Boehm

x=1 x=x+1
undefined behavior

if (x > 0) { depends on undef behavior
  dead
  ⋮
}

Hoguild!

higher performance
~ undefined behavior!

"integer"
logic
discrete!

"floating point"
bigint
bigint
exact math

image processing
NLP
speech understanding
SGD

SORT

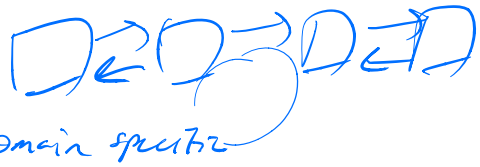
logical spec

fault detection

spec?

spec ≈ code

auto data structure repair



domain specific

C → safe C

checking read/write

x = new char[8]

x[9]

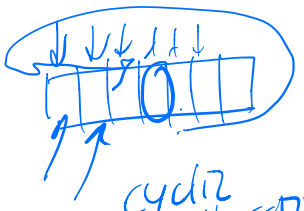
OOB

- drop write
- cache OOB writes

Fail stop

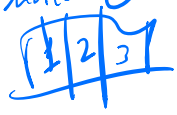
x[9] = 'z'

addr	value
&x[9]	z



cyclic allocator

malloc (1)
malloc (2)
malloc (3)



malloc (1) x = new

delete

x...

(use after free)

c = x[9]

c = 'z'

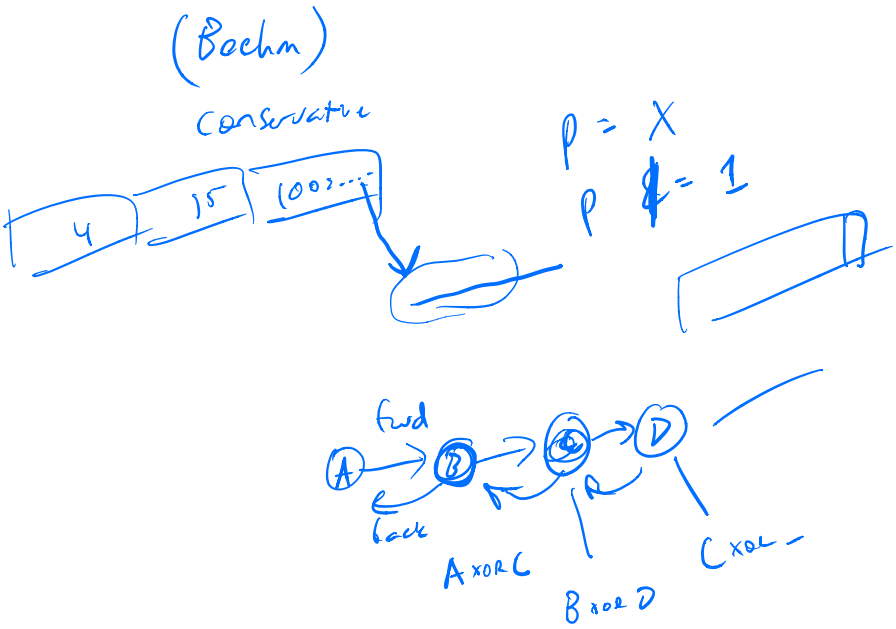
"infinite stacks"

x(10)

x[11]

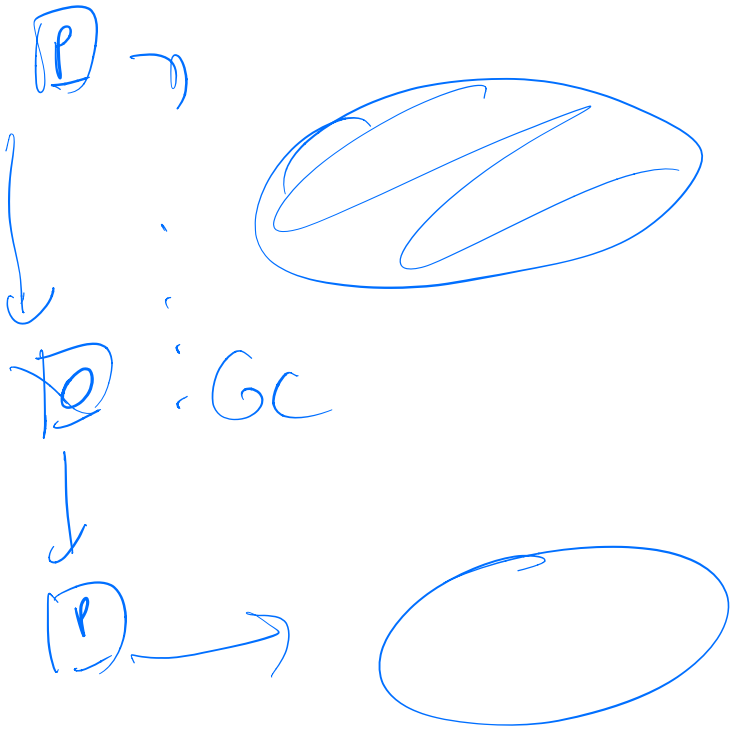
x[0000 0007]

use GC


(Boehm)
conservative
4
15
100...
p = X
p &= 1

fwd
A
B
C
D
back
A XOR C
B XOR D
C XOR -

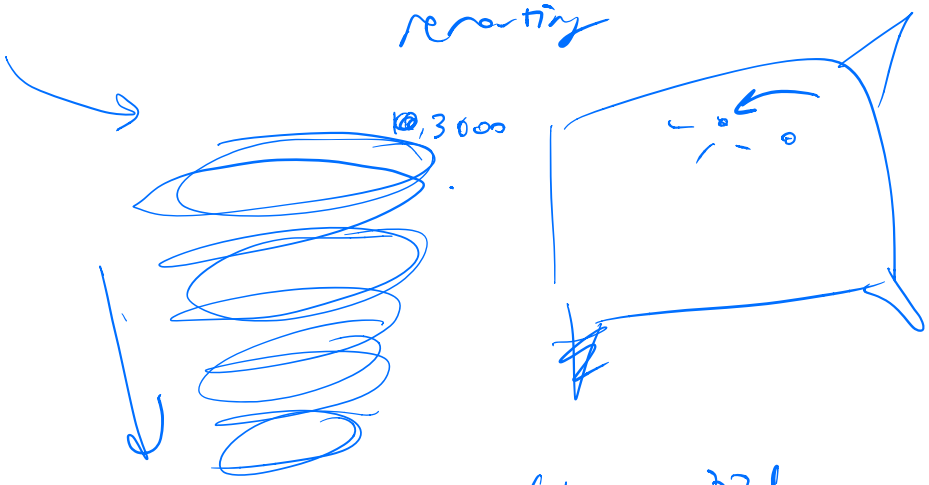
A XOR C XOR C ⇒ A


P
: GC
P

ATC

backoff protocol

rerouting



safety-critical

checkpointing

backups

phone apps

not safety critical

retry

checkpointing

backups

fsck

file system check

OOM killer

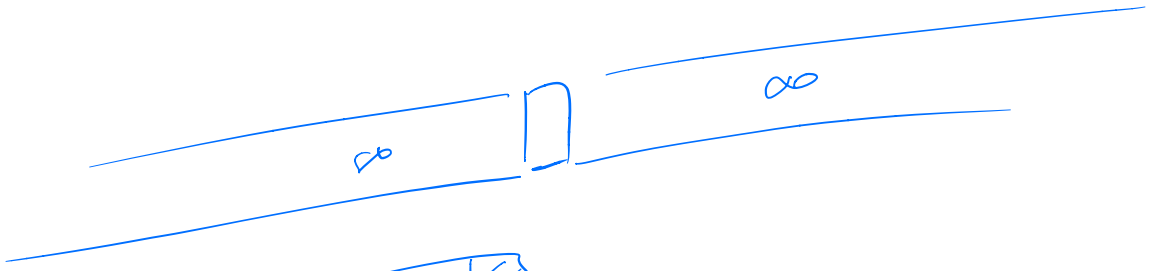
Jitter

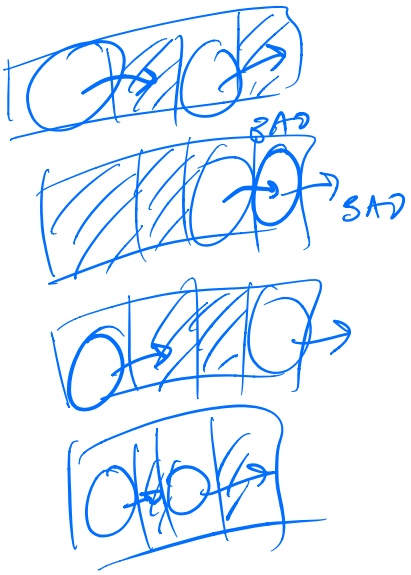
FD - slow

Archipelago

moral hazard

infinite heap semantics





Bohr → Heisenbug

p

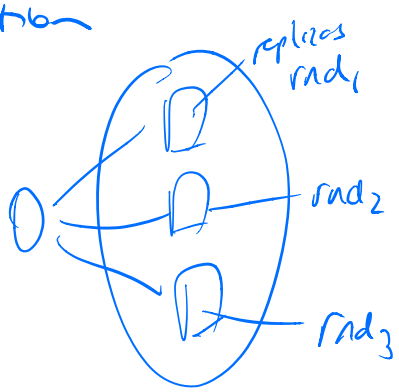
100% user fail

$E[\#\text{user fails}] = p \times N$

space for reliability

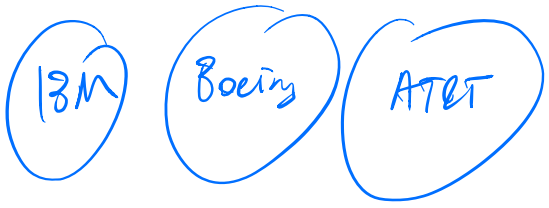
probably-ish ≠ probabilistic

replication



independence of failures

N-variant systems — not actually independent

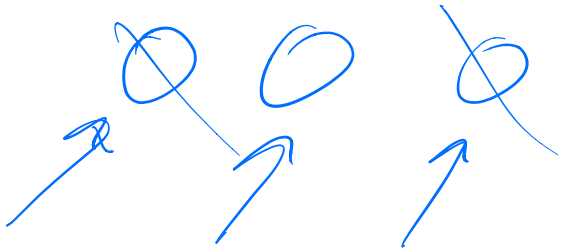


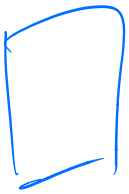
---

replicated state machines "RSM"

Bressoud & Schneider (Cornell)

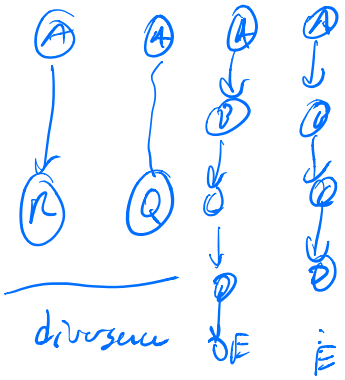
DFAs





deterministic

can't implement OS's w/o determinism (threads! async I/O (in order arrival of I/O) req. for det. user input scheduling



RSMs
+determinism — Bohr bugs

RSMs
+ Dieltard — Heisenbugs w/ probabilisms

time
\ RNG

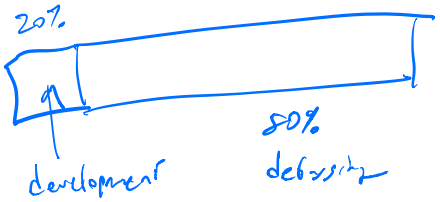
never roll your own crypto
hash fn
rng
binary search :)

GPL infection

Dieltard → Robust Heap → Fault-Tolerant Heap
Win 7

FUD
Win7

MSR Tech Fest



security
system

how secure is it against attack by an attacker who can do X) threat model

fault tolerance

independent hardware software correlated...

Dieltanker → Win8

profilers

throughput too low
latency too high

prof

linecount

```
main() {
   ↙ l1++
   ↙ l2++
   ↙
     foo ↙
   ↙
}
```

```
for( )
  for( )
    for( ) {
      — linecount++
    }
  }
}
```

end-to-end

"kernel" (not OS kernel)

↳ heavily compute-centric library function

MM, ...

deep learning

NOTE: SERIAL CODE ONLY!

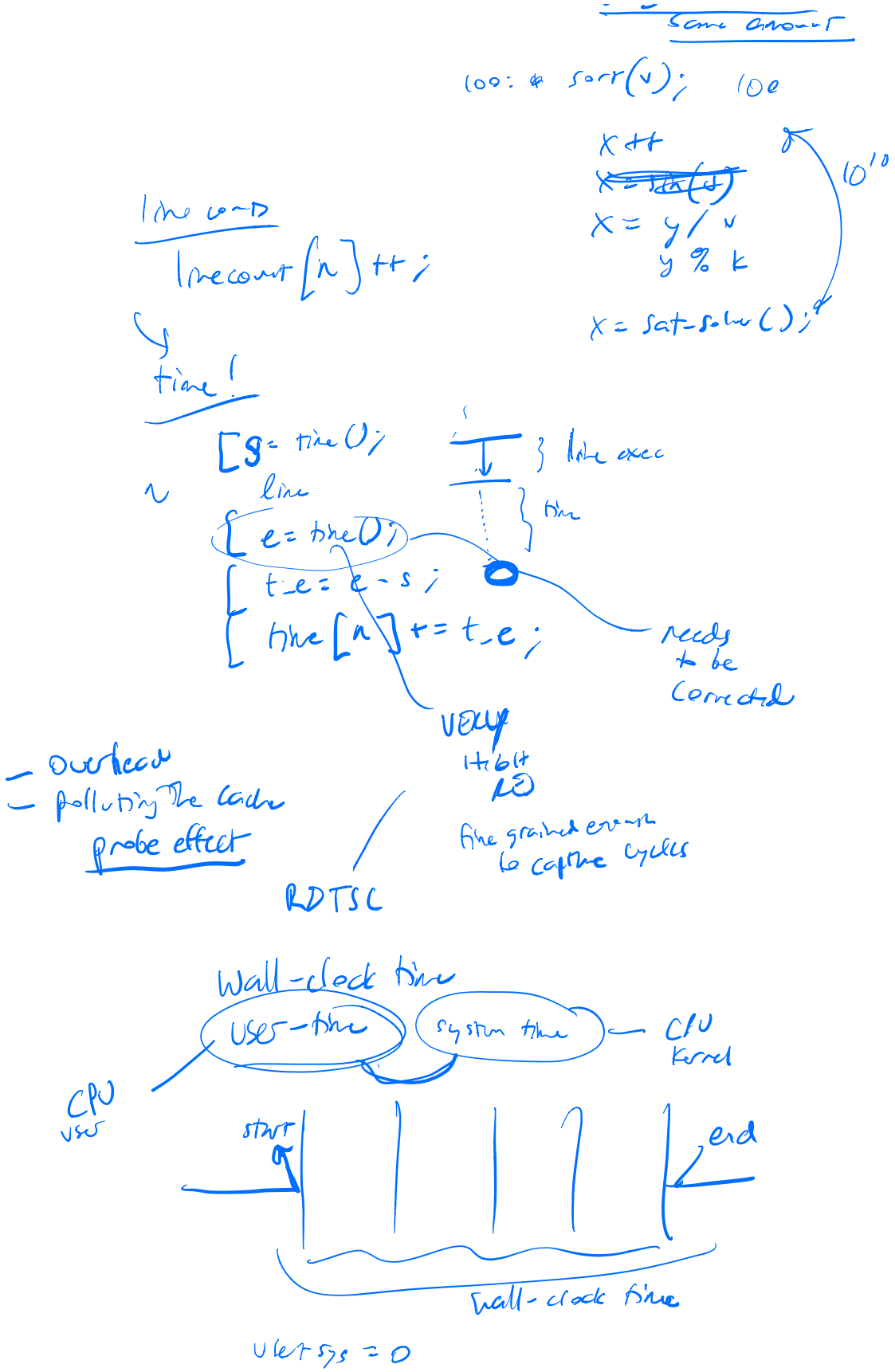
advantages (pros)

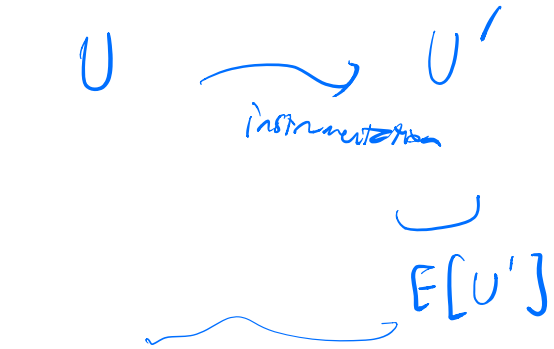
if linecount (trip count) high
→ bottleneck

disadvantages (cons)

— overhead
slow program down

— assumption - uses line costs


Same amount
100: * sqrt(v); 100
x++
x = y / v
y % k
10^10
x = sat_solve();
line cnt
linecount[n]++;
time!
s = time();
line
e = time();
t_e = e - s;
time[n] += t_e;
line exec
time
needs to be corrected
VERY
high
resolution
fine grained enough to capture cycles
- overhead
- polluting the cache
probe effect
RDTSC
Wall-clock time
user-time
system time
CPU kernel
CPU user
start
end
wall-clock time
user+sys = 0

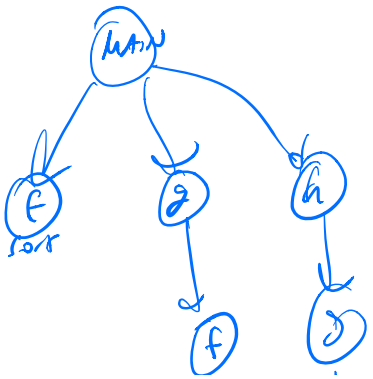


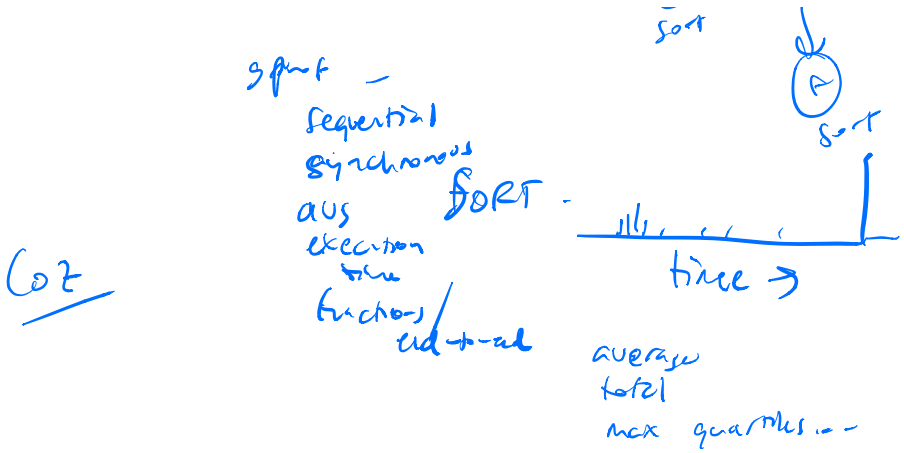
Gprof — low overhead ✓
— accuracy ✓
— unbiased (time not true count) ✓

Statistical profiling — Because of Sampling bias

50% line 12
10% line 11
5% line 15
⋮

Law of Large Numbers!





gprof —
sequential
synchronous
aus
execution time
functions
call-tree

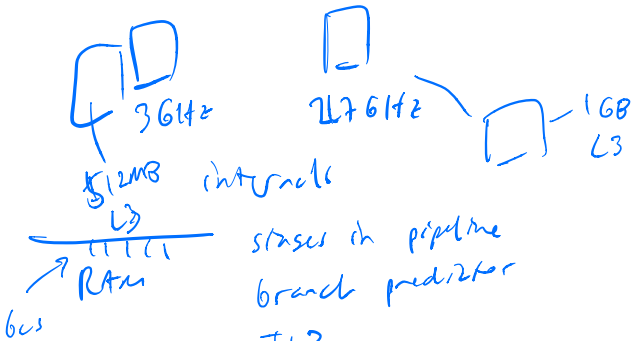
SORT

Coz

average
total
max quantiles ...

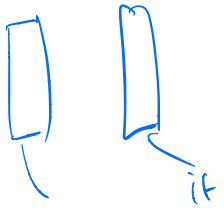
---

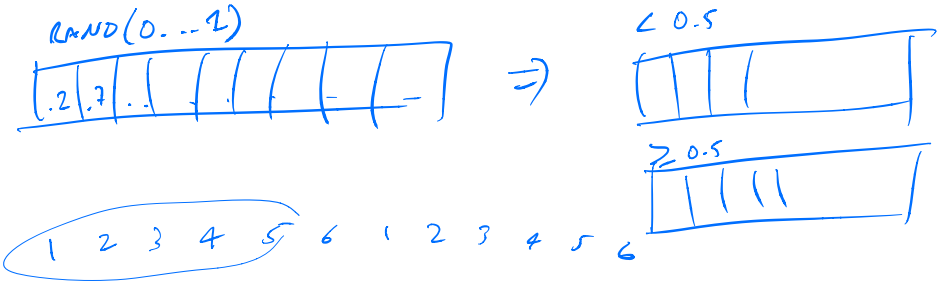
statistical sampling — periodically interrupt
→ where do you land
collect samples — distribution of execution times



interrupts
stages in pipeline
branch predictor
TLB
caches

performance prediction
performance portability

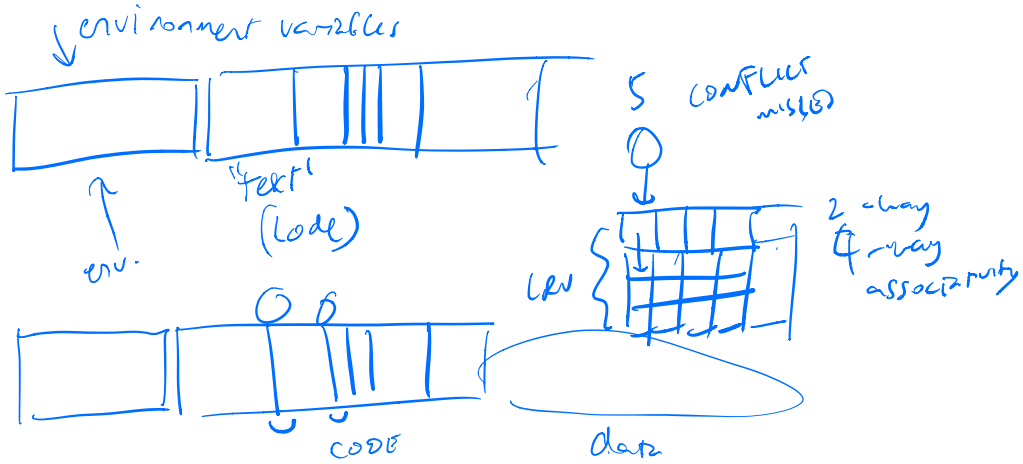




if ... == 6

Pattern Hash Table

Producing wrong data w/o doing anything obviously wrong!
Mytkowicz et al.



date
cwd
username
terminal id
home dir

cache!
I$ D$

on-chip hardware performance counters

perf

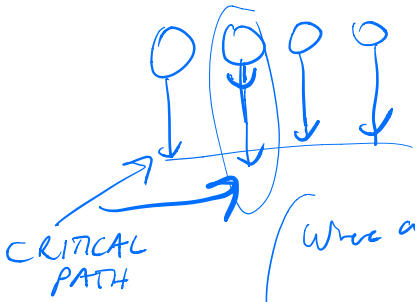
counts up
countdown timers — signal when perf counter hits zero

⇒ profiler for free

---

good old days

no threads
virtually no I/O
non-interactive
asynchrony



fallacy –

(where a program spends its time
⇒ where you should optimize)

not actionable

---

profiler wrap up – instrumentation based
sampling based
perf counter based profiler
causal profiling

"single node"

+ concurrency
+ asynchrony
+ parallelism
– distrib. sys

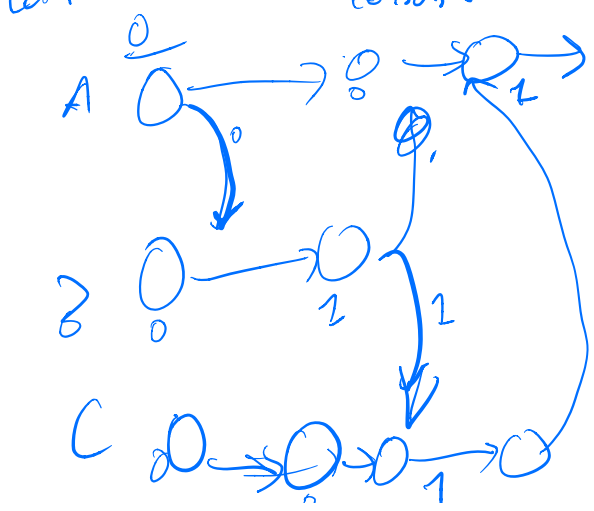
distrib sys?

no global synch clock

FLP

Lamport

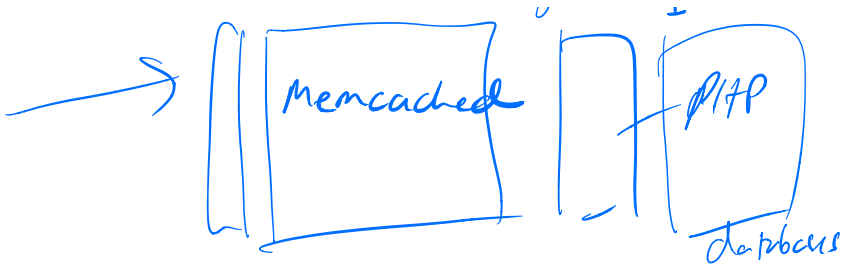
consensus



NTP

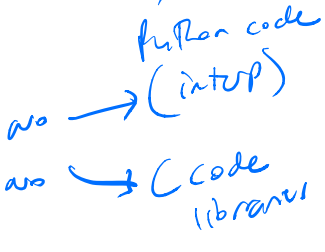
atomic clock

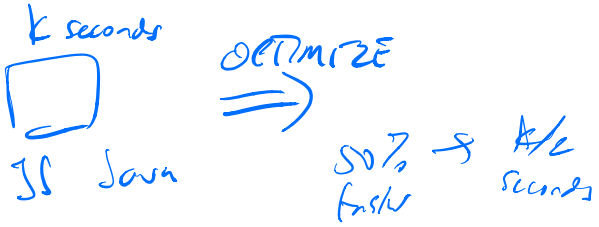
N particles / sec

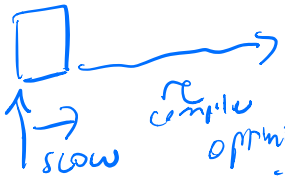


profiling "mixed" languages

Python PHP JavaScript







perf hard

Correctness hard
but monotonically increasing
- compositional

full correctness

program respects a full specification

"program refines a spec"

Word, Chrome, ...

partial correctness

doesn't deref null

verify a property of program
→ "bug checking"

"always safe"
- deref null
- mem safety error
- uncaught exception
- segfault

verification

- security related
- narrow APIs

device driver

L4 microkernel

OpenSSL

IMPORTANT "SMALL" SLICES
→ existing formal spec

Heartbleed

Everest

F*

Y

C

"high assurance software"

very $

"guix custodet custodes"

"specification problem"

[NOT EXHAUSTIVE] testing can prove the presence of bugs but not their absence — Dijkstra

Principia Mathematica

Bertrand Russell & Whitehead

axiomatize all the math!

Gödel

Complete or consistent

pick one!

dynamic analysis — precise (low false positives) — low recall — bug detection
static analysis — imprecise (high false positives)
testing

"heroic static analysis"

"sound" — if there's any possibility of error, report it!

Coverity

arises due to over approximation

Fundamental tension

Soundness — low false positive rates

$x \in \{1..10\}$ / $y \in \{-5 .. 0 .. 5\}$

numerator — denominator

increasing ↓ precision

ptr analysis (still round)

−, 0, +

x / y

sound ⇒ error here!

1/0 1/-0 ...

"Scalable" "Infer" FB

Valgrind

Google — Sanitizer
- Address
- Thread
- UndefinedBehavior

ASan
TSan
UBSan

Clang++ -fsanitize=address

dynamic analysis

Pin — binary instrumentation — dyn analysis

→ before read/write

→ after read/write

Valgrind — 10x-100x

X = malloc(32)

User	addr	len
X	=	32

ASan ~ 1.2x → 2x

VALGRIND — for ( ) — before / after — x

ASan — check(x) — for ( ) — x

GC — "dynamic analysis"

eliminates errors instead of reporting

Exterminator

— lgc