grammars / parsing / compiling

optimization

Chomsky hierarchy

recursively enumerable

↑

context-sensitive

|

context-~~insensitive~~ "free" CFG

|

regexps

(((3)))

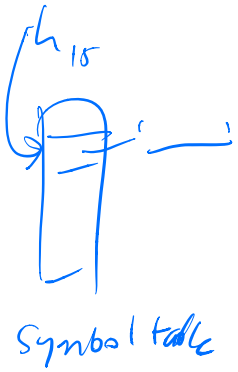
ALGOL-60

Backus-Naur form BNF

expr ::= '(' expr ')' |
number |
expr op expr |
unaryop expr

"var" "variable"

42 h_{15}



Symbol table

backtracking ~ exponential time

N tokens 2^N

text

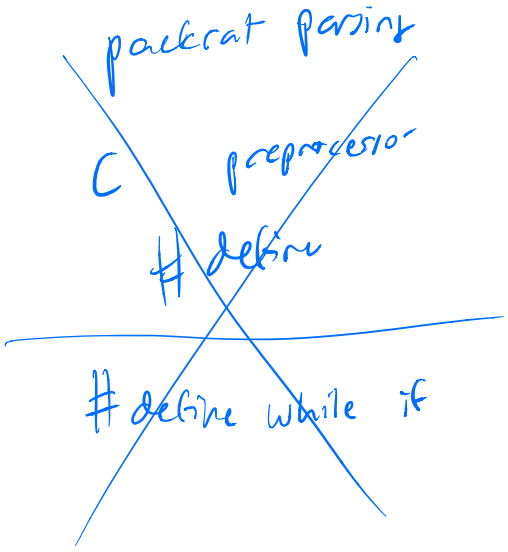
lexer → tokens → parse

lex
flex

grammar → yacc / bison → parse

Java... Antlr

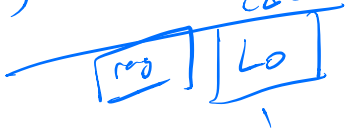
LL(1)
LALR(1)



optimization

semantics-preserving

$\forall$ inputs : $P(i) = P'(i)$

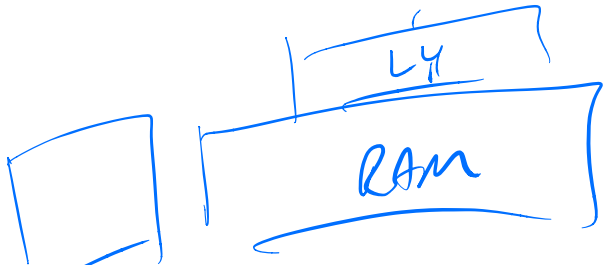


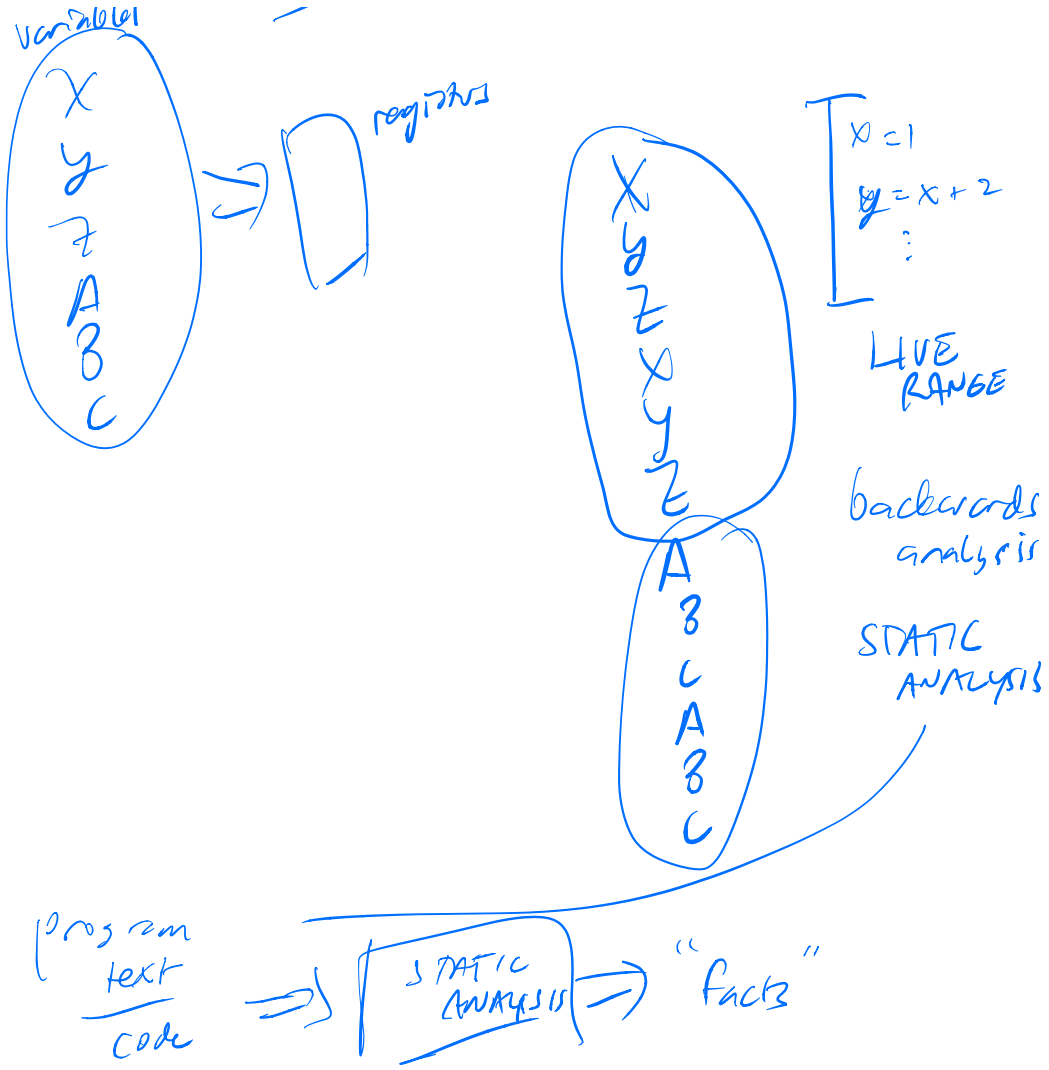
— caches

registers

memory

memory hierarchy

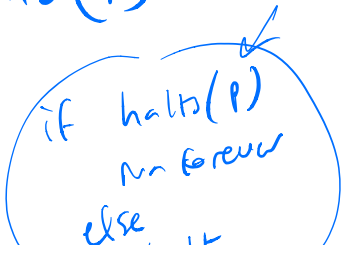




Halting problem

P : ∀ i : inputs : P(i) terminates

halts(P)



hell

STRAIGHT LINE CODE

LOOPS

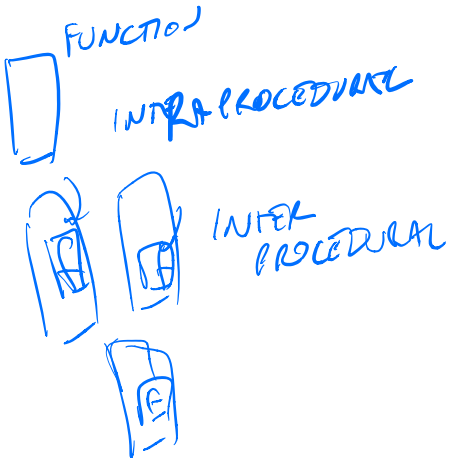
RECURSION

FUNCTION CALLS

FIRST-CLASS FUNCTIONS

EVAL EVIL

POINTERS



$x = [1, 2, 3]$ INT * x ;

$y = [1, 2, 3]$ INT * y

ALIAS ANALYSIS

POINTER ANALYSIS

for : x[i] y[i]

x + i y + i

f(int* x, int* y)

{

x = A

y = B

if ( )

x =

} else

x = y

}

$x: \{A\}$

$y: \{B\}$

$x_1: \{A, z\}$

$x_2: \{A, y\}$

register allocation
= graph coloring
NP-complete

strength reduction

X = pow(3, 2)

X = 3 * 3

X = 9

3^2

y = 12

z = y + 1 → 13

x = 2 → 13

Constant propagation

26

inlining — exposes optimization opportunities

```
int add(int x, int y) {
  return x + y;
}
```

reduces fn call overhead

```
foo()
  int a = 12;
  int b = 13;
  add(a, b);
```

closed-world

Whole-program

int c = a + b 25

analysis

Modular analyses

FORTRAN - OPTIMIZING COMPILATION

LISP - INTERPRETERS

switch (-) {

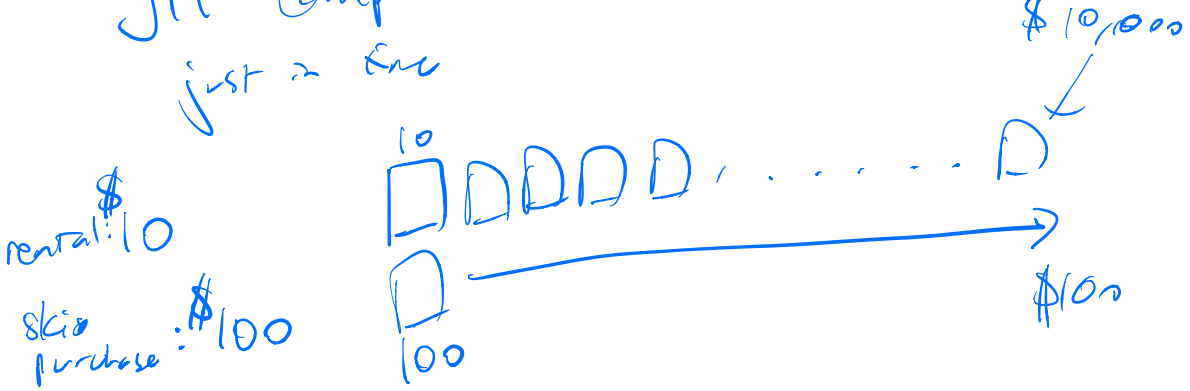
case ADDONE:

var(p) = var(p) + 1

break;

:

JIT compiles just in time

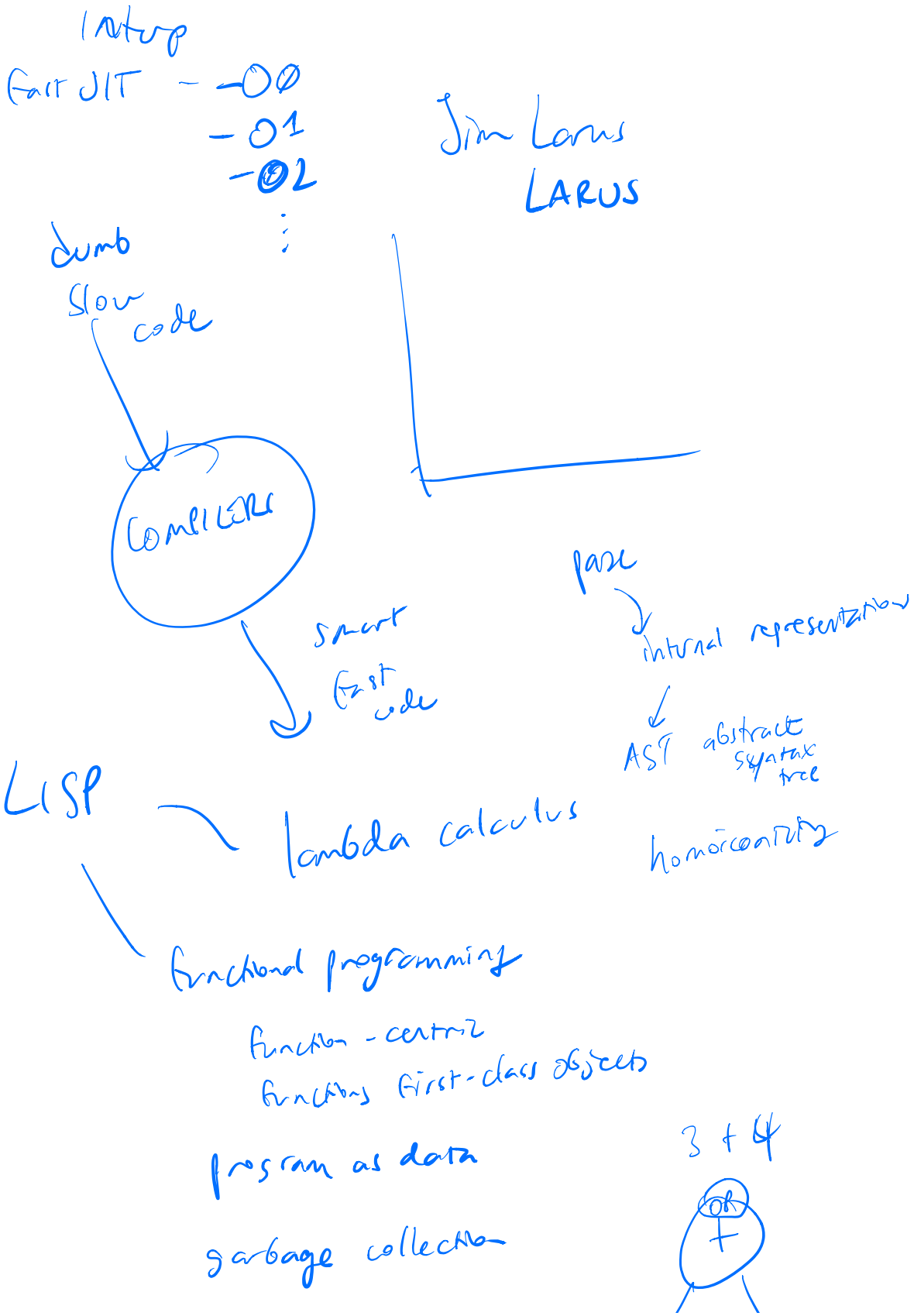


Ski rental problem

rent until you spend $100

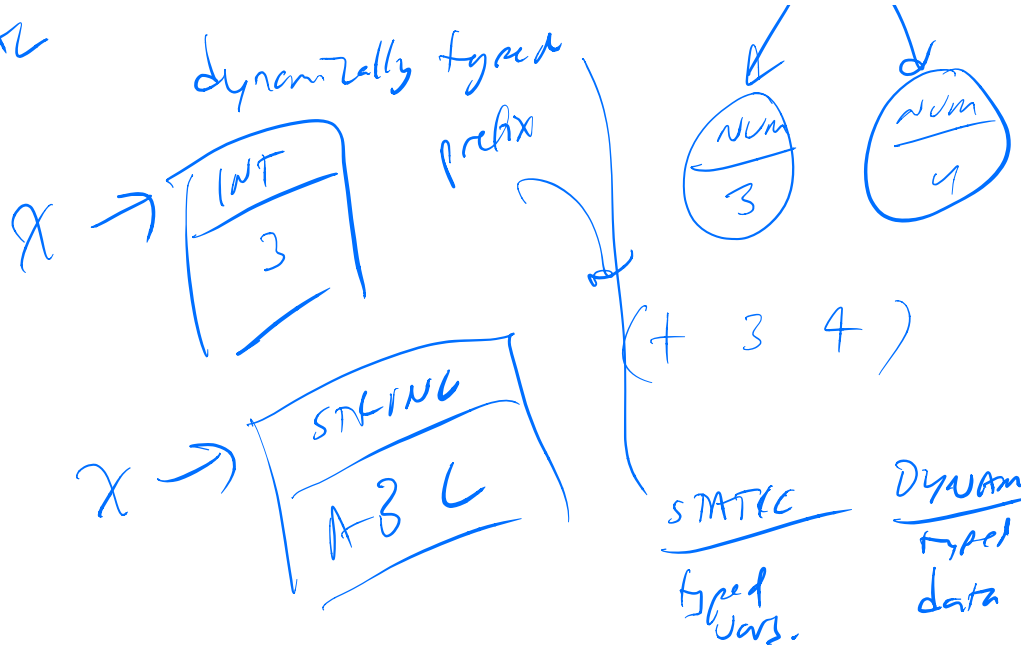
Then you buy

Worst-case: Spend $200 could have spent $100


Interp
Fast JIT - -O0
-O1
-O2
:
Jim Larus
LARUS
dumb
slow code
compiler
smart
fast code
parse
internal representation
AST abstract syntax tree
homoiconicity
LISP
lambda calculus
functional programming
function-centric
functions first-class objects
program as data
garbage collection
3 + 4
OR
+

EVAL

dynamically typed prefix



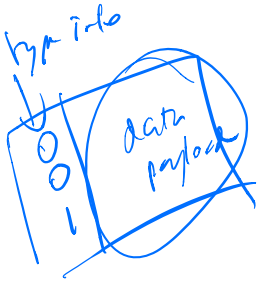
STATIC typed vars.

DYNAMIC typed data

(+ x 1)

boxed values

tagged



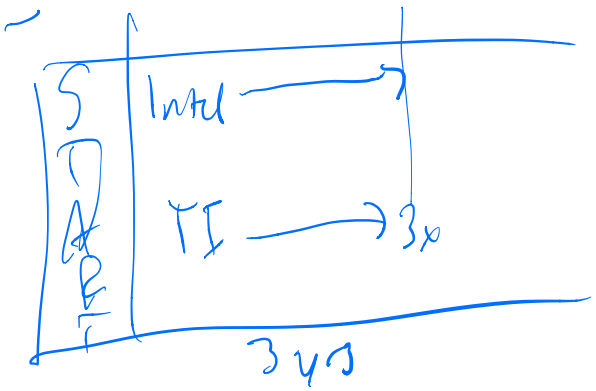
LISP machines 70s–80s

TI Explorer

DB machines

computer vision

neural net



Moore's Law
gates in area
doubles
~ 18 mos

Dennard scaling
↑ density ⇒ ↑ clock speed

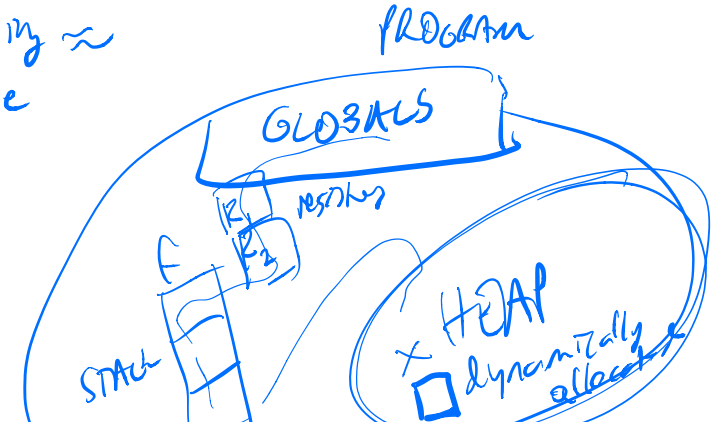
Proebsting's Law
compiler opts.
double perf
every 18 YEARS

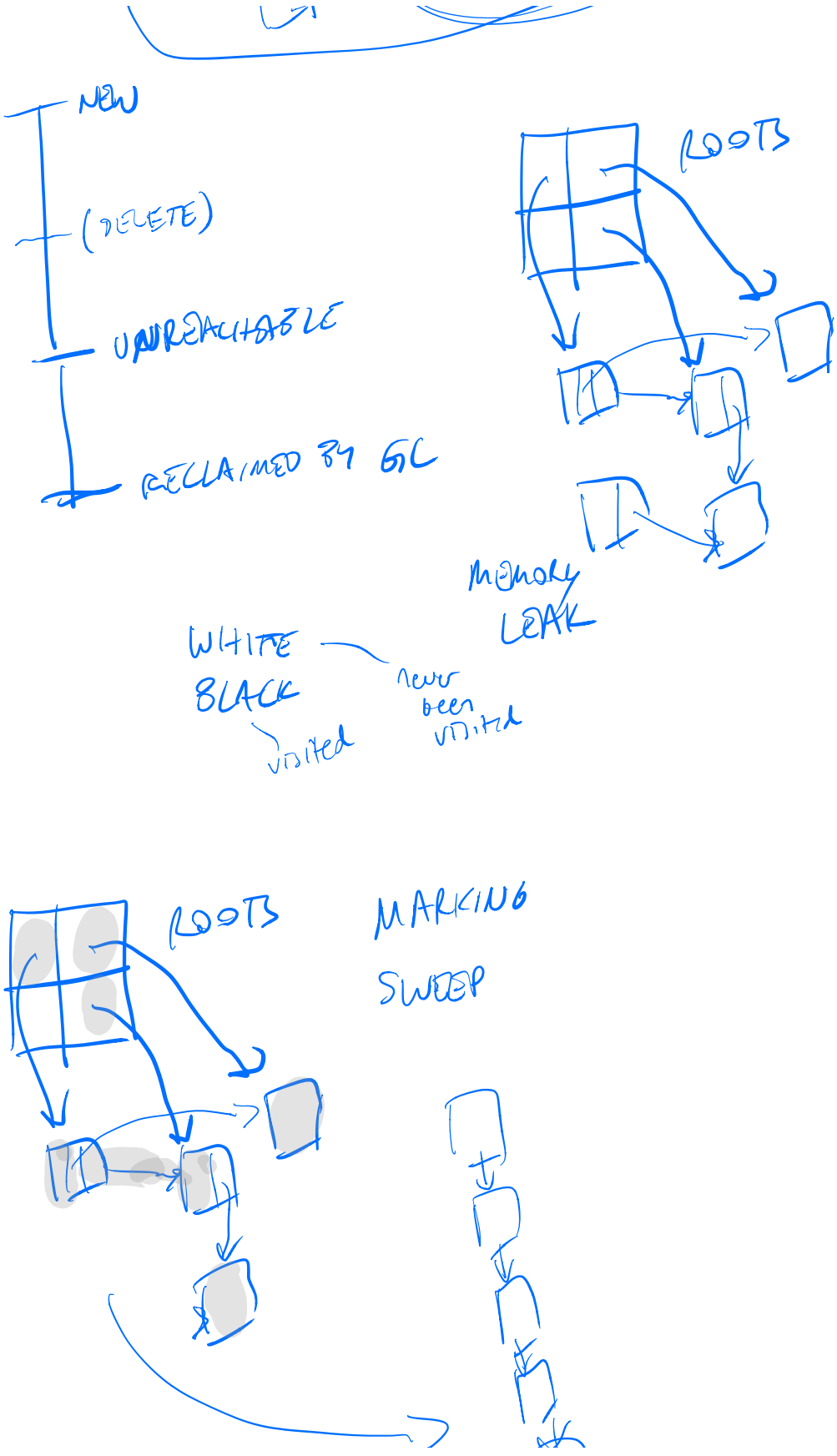
Self
Chambers Ungar
Hölzle
Jeff Dean

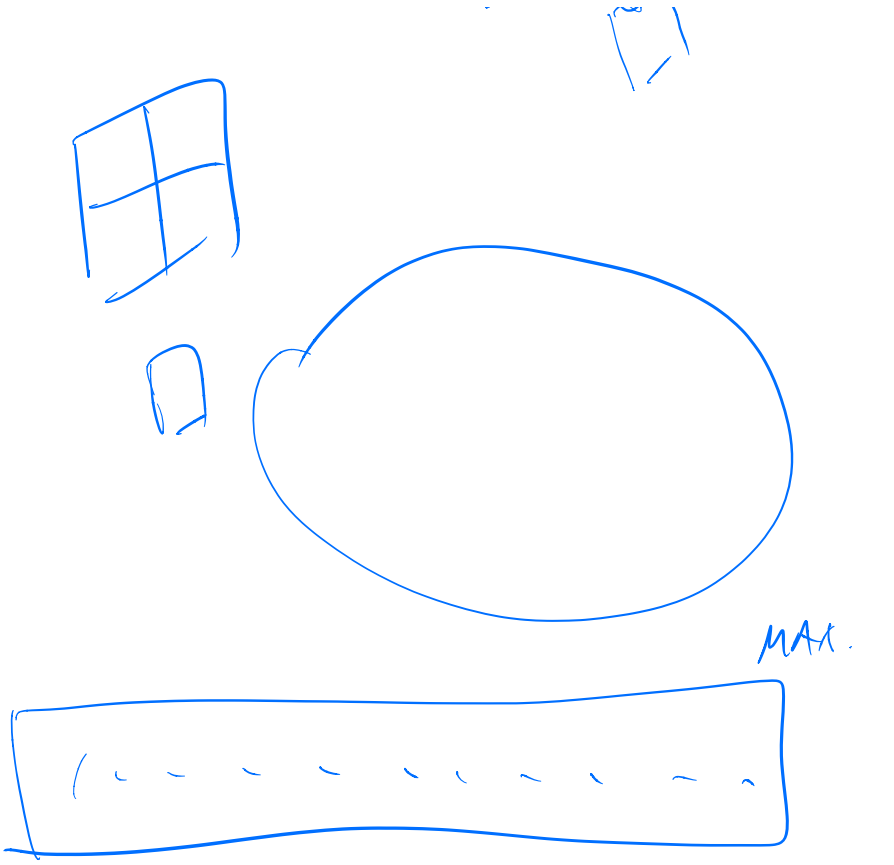
GC
mark-sweep
reachability

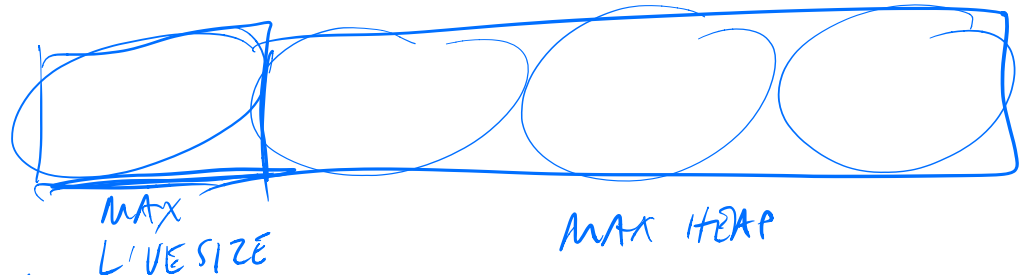
explicit mem mgmt
malloc/new
free/delete ∈ object "dead"
live
liveness

reachability ≈
Oracle




NEW
(DELETE)
UNREACHABLE
RECLAIMED BY GC
ROOTS
MEMORY LEAK
WHITE
BLACK
never been visited
visited
ROOTS
MARKING
SWEEP


MAX.
GC: STOP THE WORLD
pause time
GC latency
63.9999 GB



Overprovisioning

Quantifying the Perf. Impact of GC vs. Explicit Mem Mgmt

~ 3x - 5x more RAM ≃ malloc/free

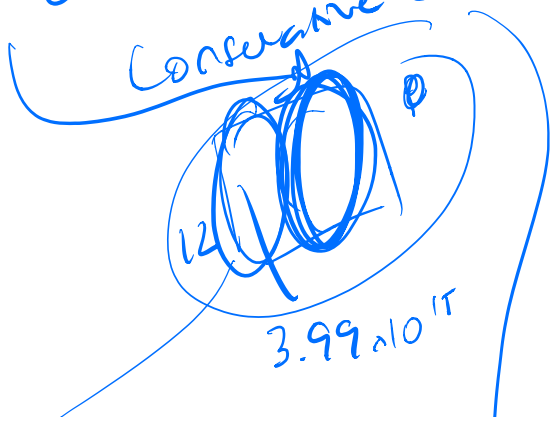
UNCOOPERATIVE GC

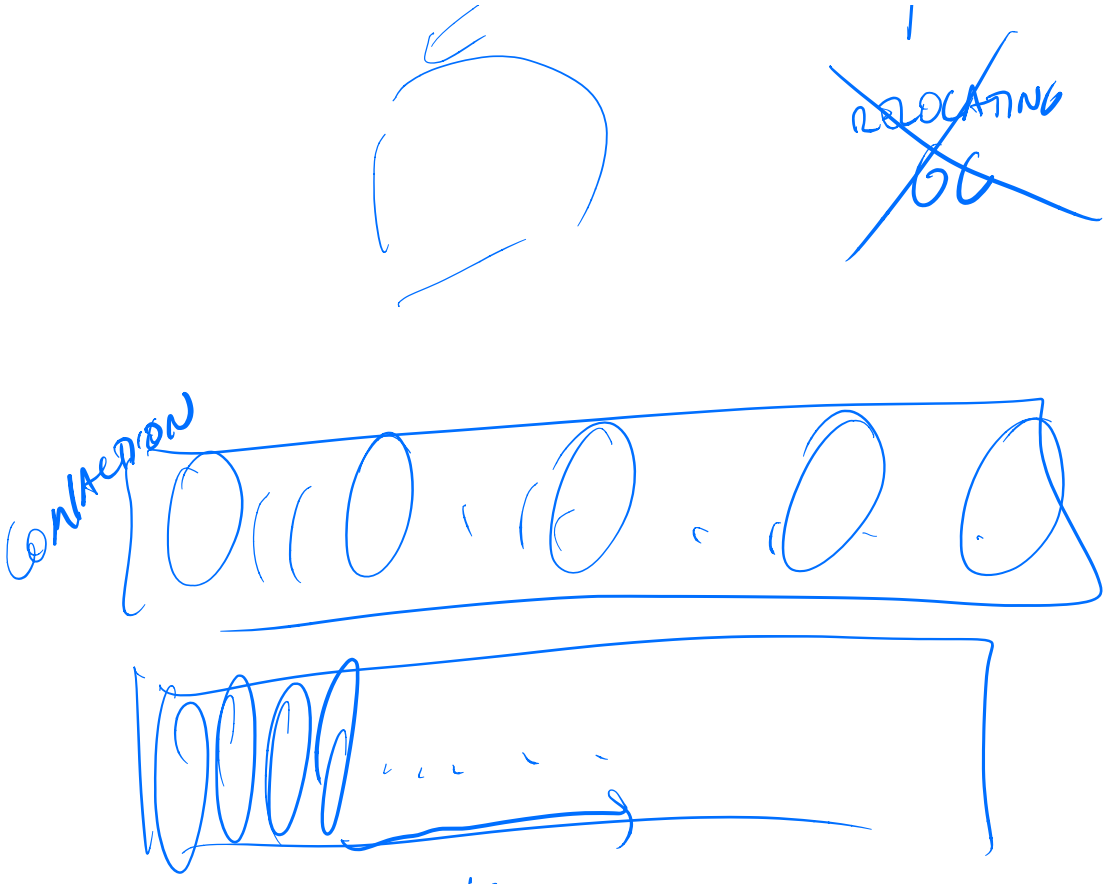
Hans Boehm

PRECISE GC

duck test

Conservative GC





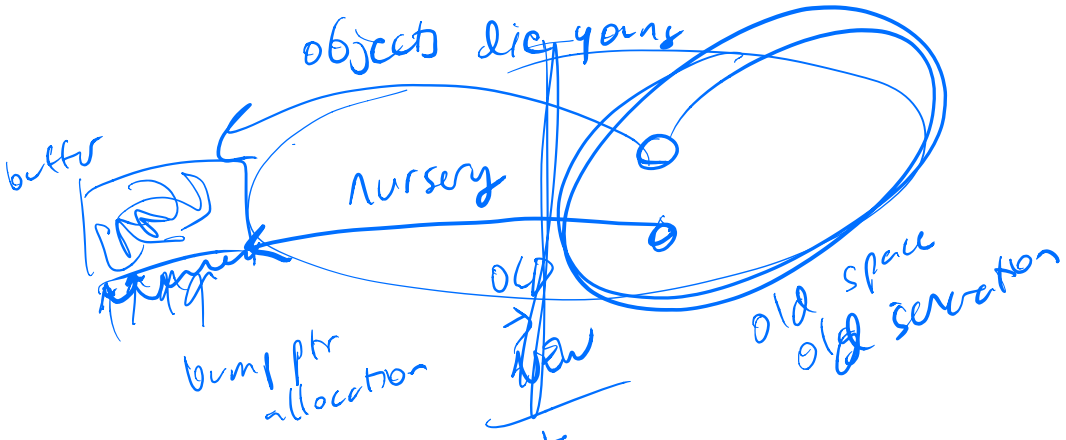
reachability

"scanning"

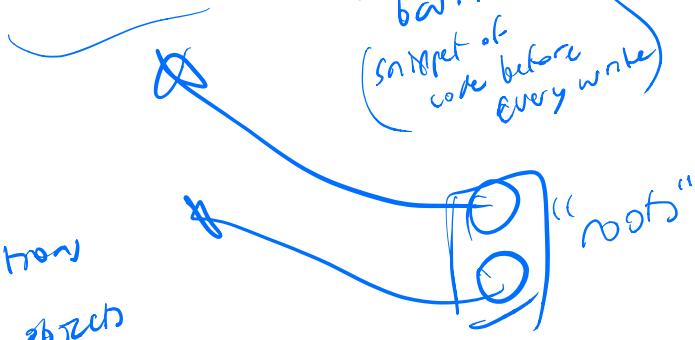
MS
MSC
generational GC

generational hypothesis

objects die young



write barrier (snippet of code before every write)



fast allocation

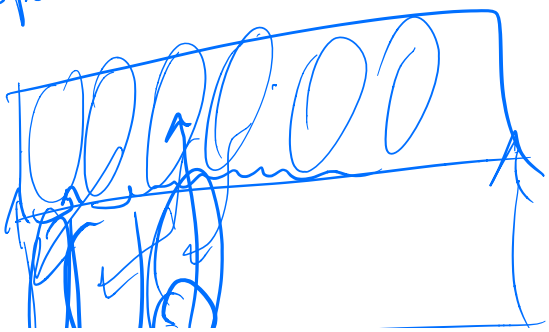
young objects are quickly reclaimed

⇒ delaying full heap GCs

pre-tenuring

immortal

semispace

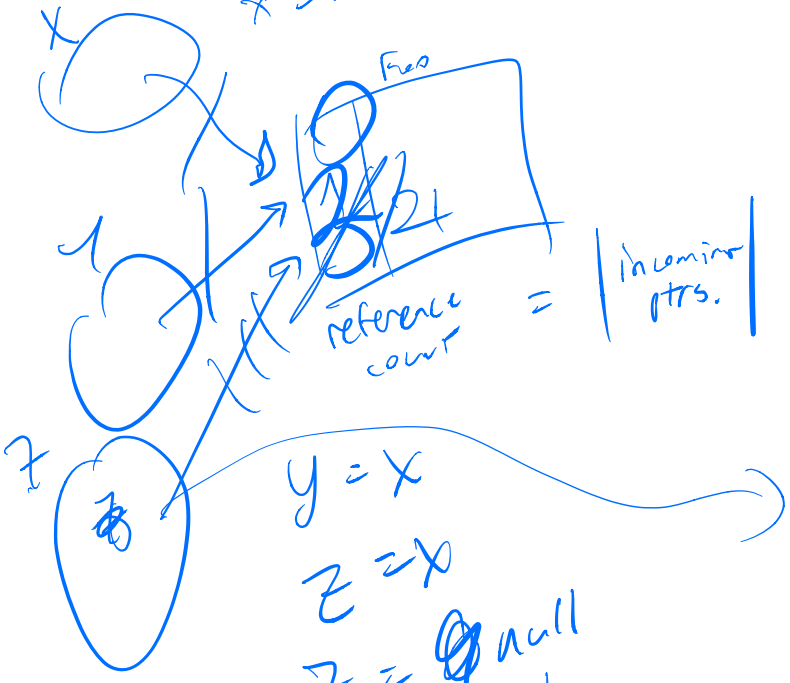


fixed pointers

new

reference counting

x = new Foo



y = x

z = x

z = null

y = null

x = null

ptrupdate (x, q)

if *x = q do nothing

else

ref(*x) --

ref(q) ++

if (ref(*x) == 0)

delete *x;

C++ smart pointers

cycles

X = null

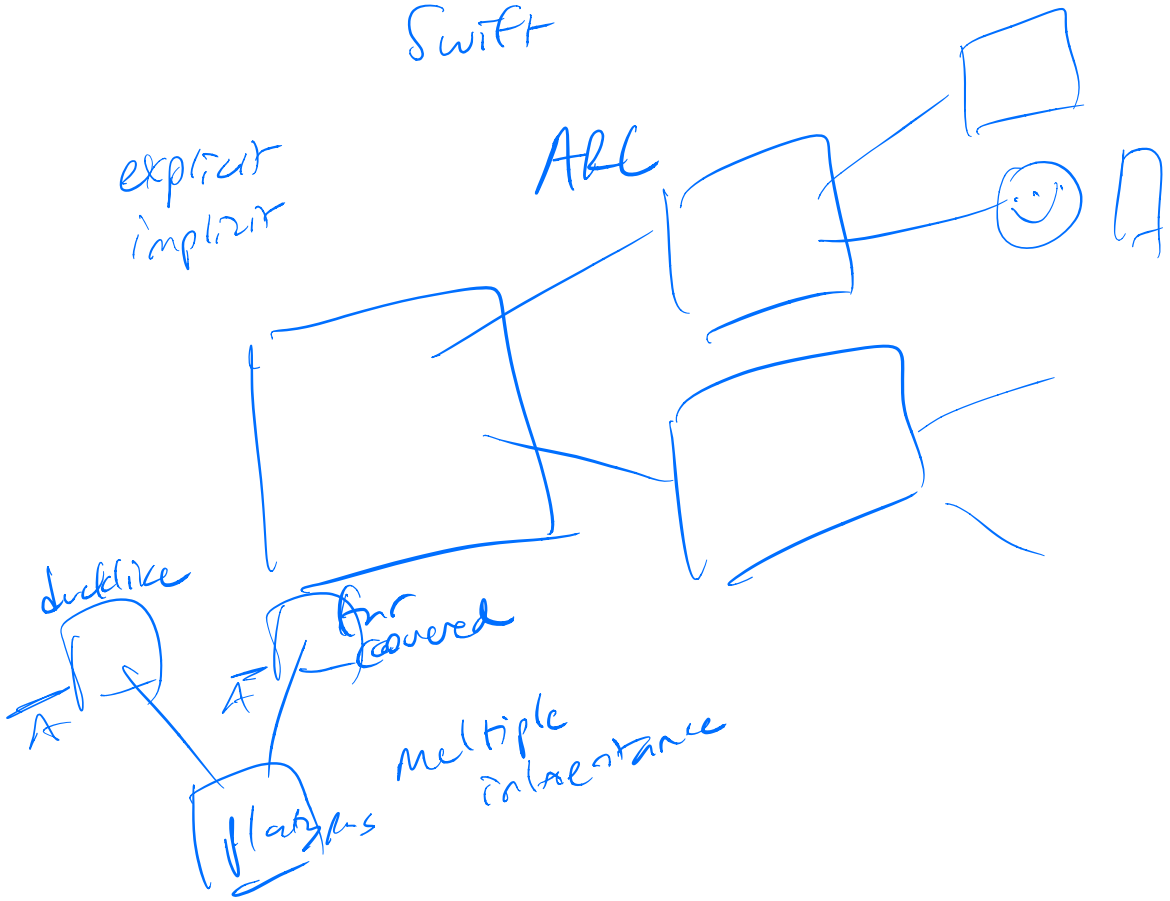
cycle collection
≈ GC

Finalizer
RAII

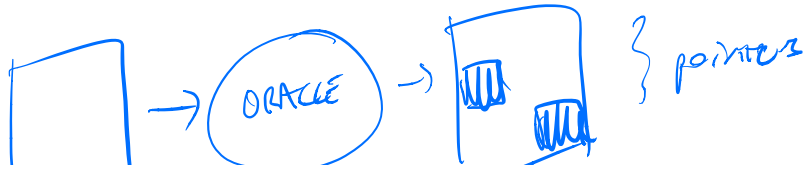
region
pool allocation
arenas

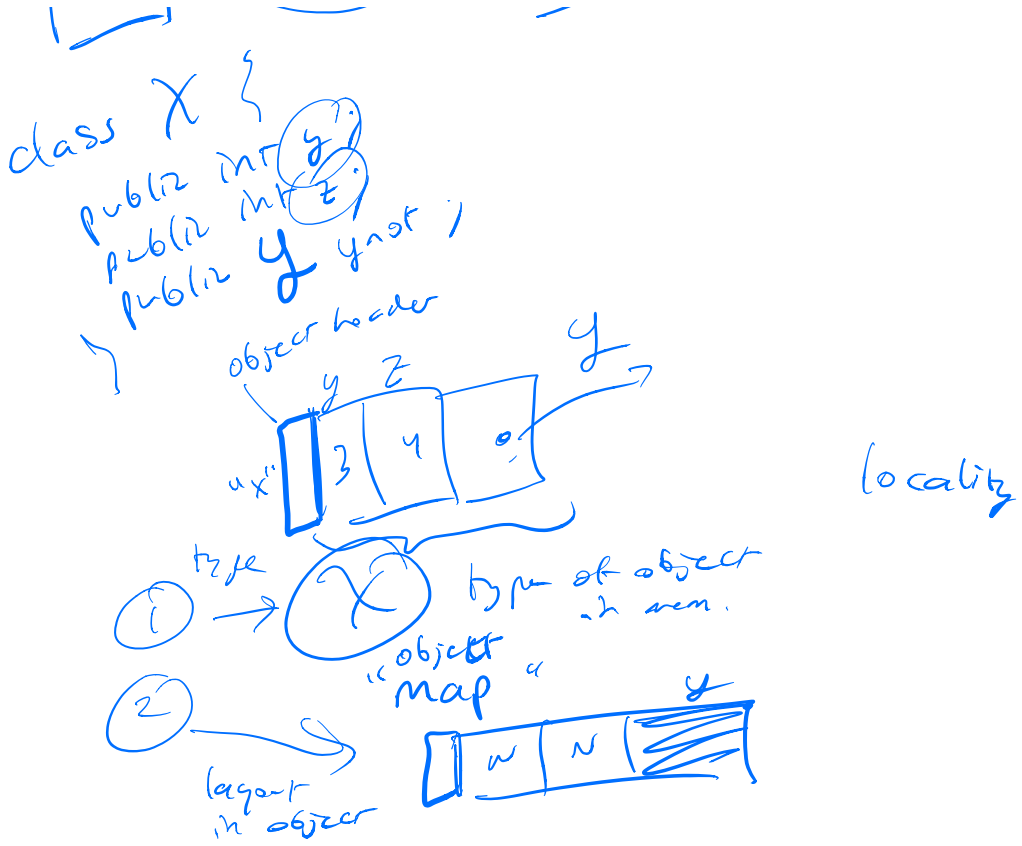
delete obj
class Obj {
~Obj()
}

RC → Objective-C
Swift



Boehm collector
conservative GC
— cannot distinguish ptrs. from values
precise GC
— can identify ptrs.





managed languages

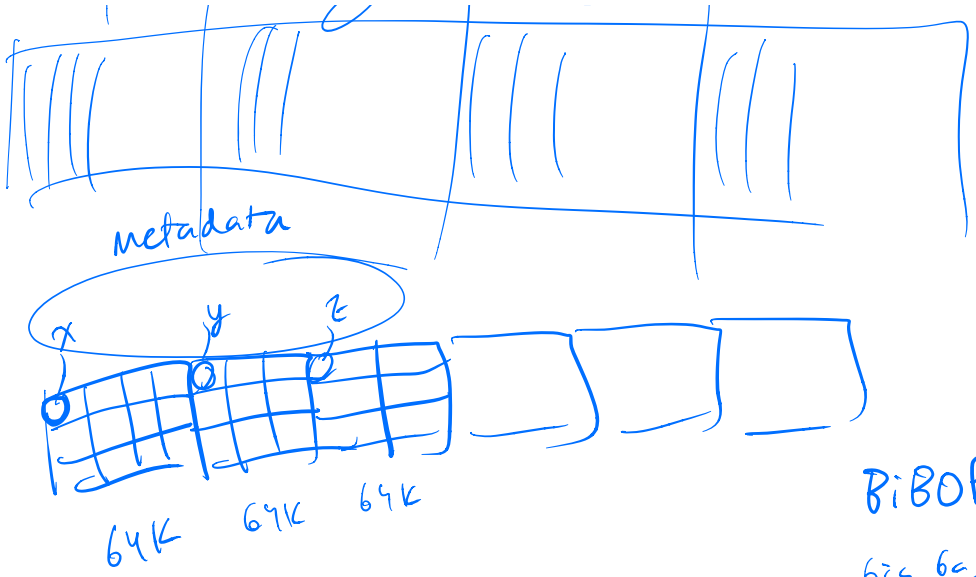
garbage collection

type safety

bounds checking

errors - uninitialized reads

X, Y, Z, A



BiBOP

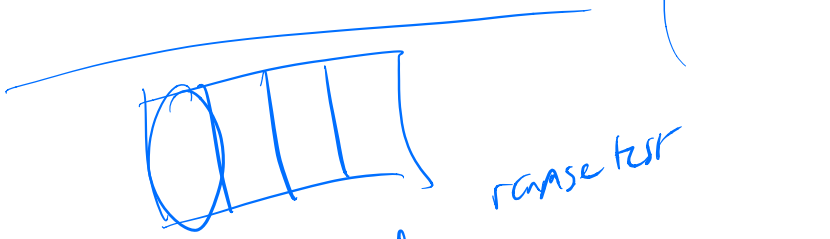
big bag of pages

thread free list

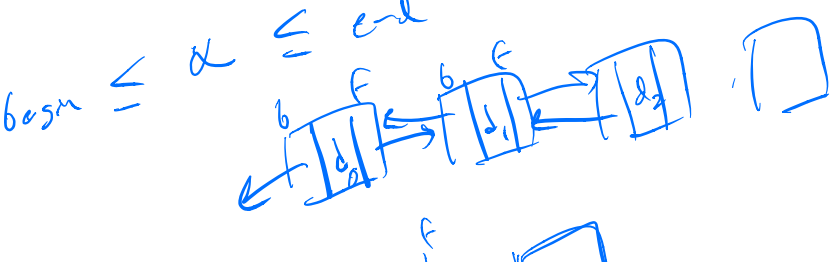
Morris worm

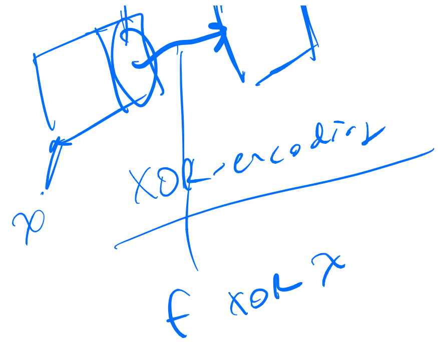
DDoS

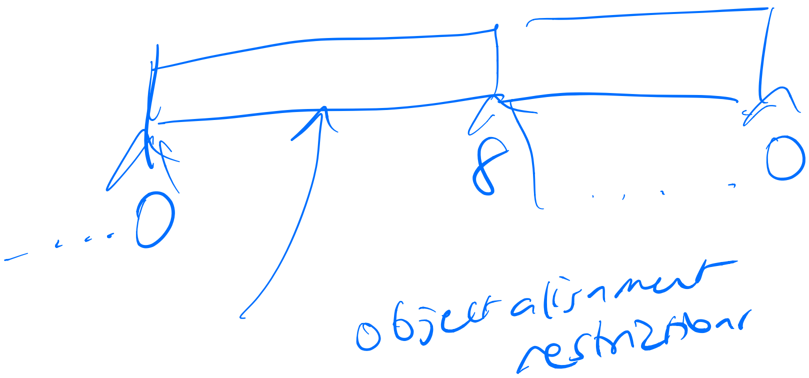
Malware

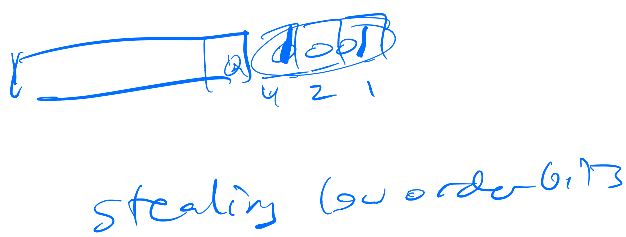


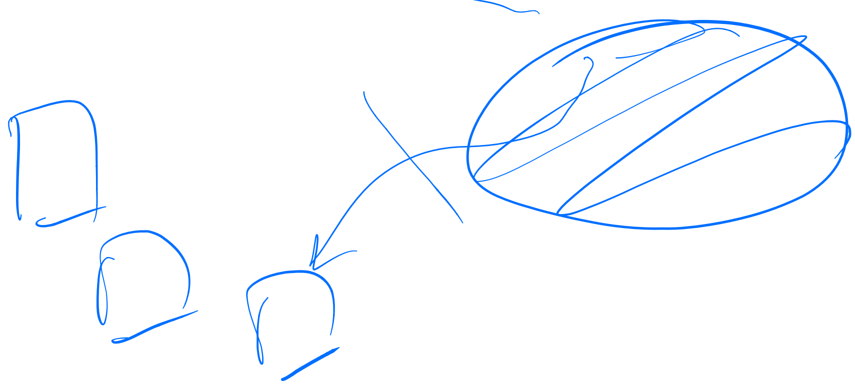
range test



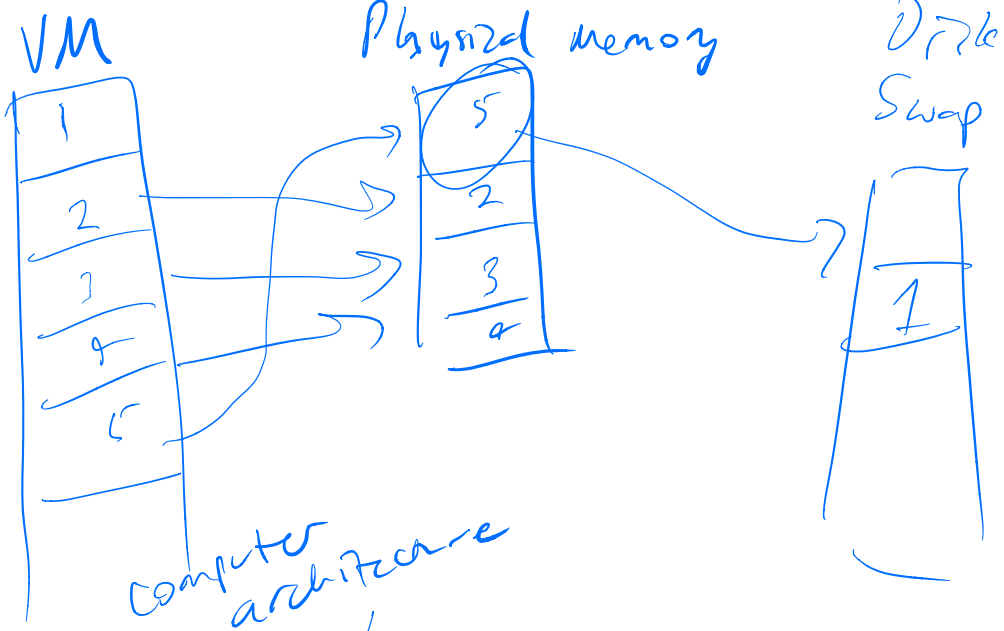








normal memory paging



computer architecture

ISA

instruction set arch

decoupling

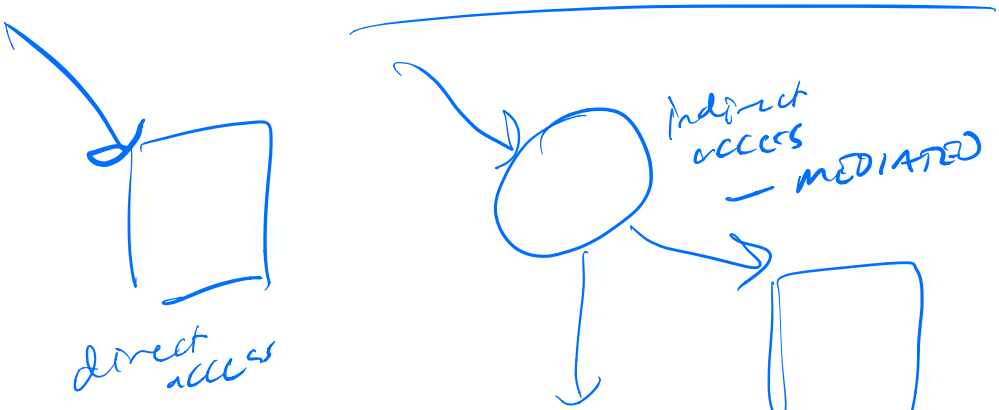
microarchitecture

virtualization

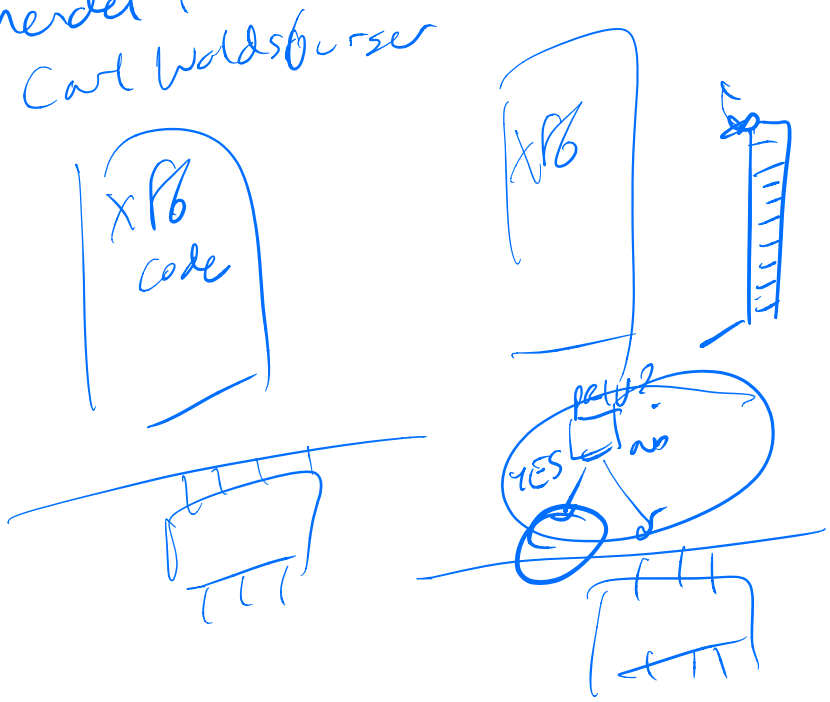
memory hierarchy

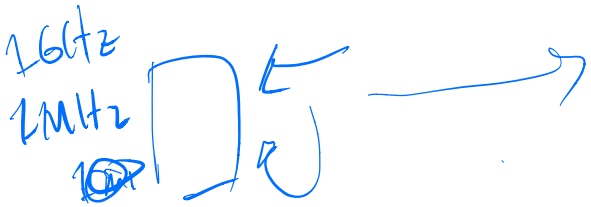
caches

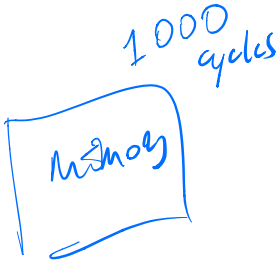
"all problems in CS
are solvable
by adding
one level of indirection"

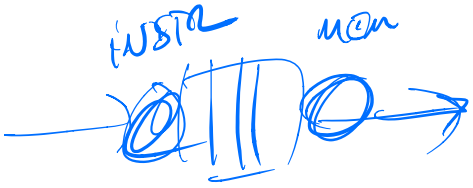


Mendel Rosenblum
Carl Waldspurger

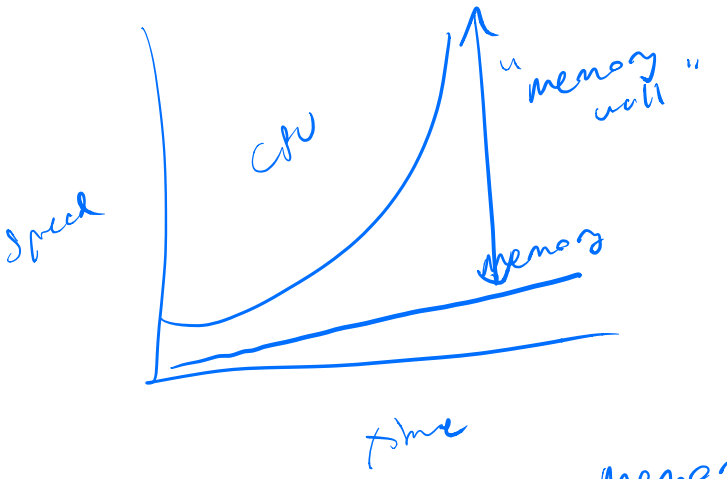






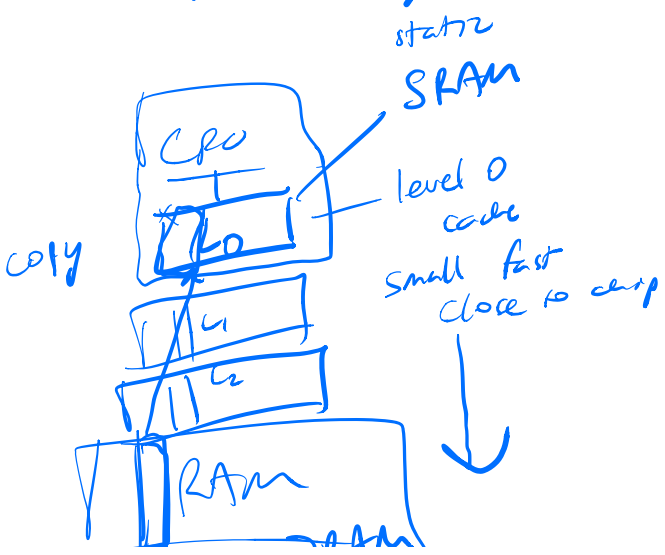


queuing model



CISC grows over solves item

memory hierarchy



ADD MOVE

cache line

326 → 1286 → 2566

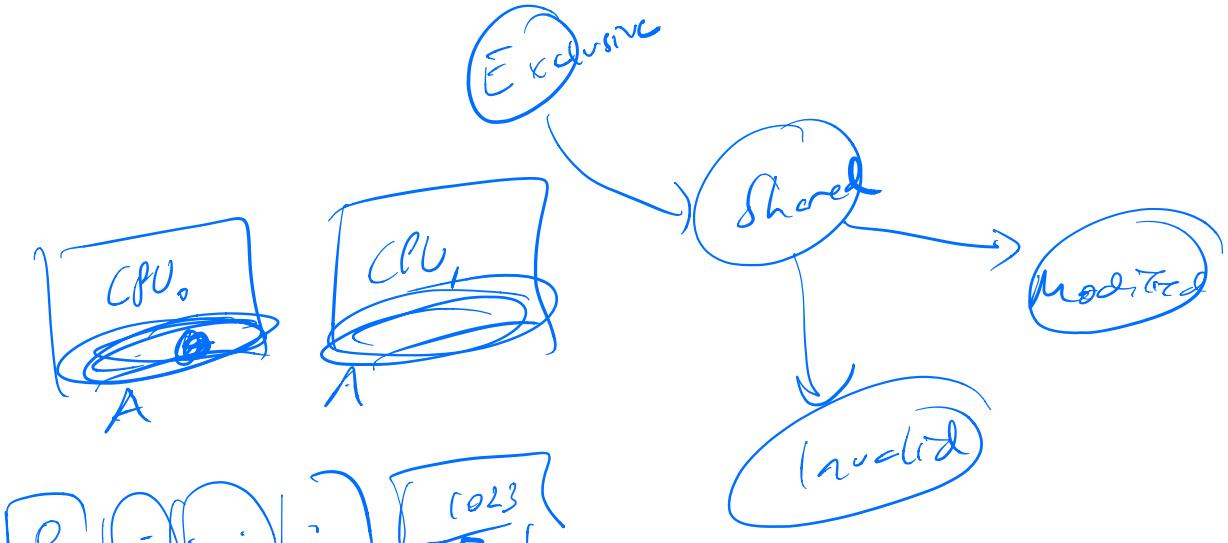
inclusive — copy in all levels
Intel

exclusive — copy in one level
AMD

"LRU" eviction

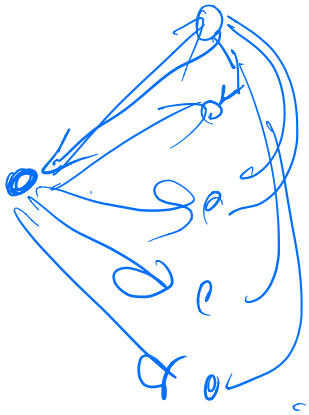
cache coherence protocol

MESI



scalability wrt N
⇒ execution time

~ polylog(N)

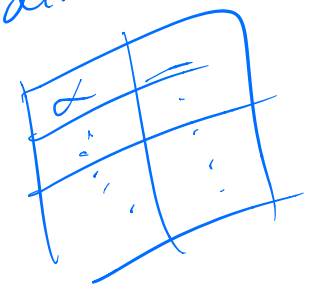


$N(N-1) \simeq N^2$

directory-based

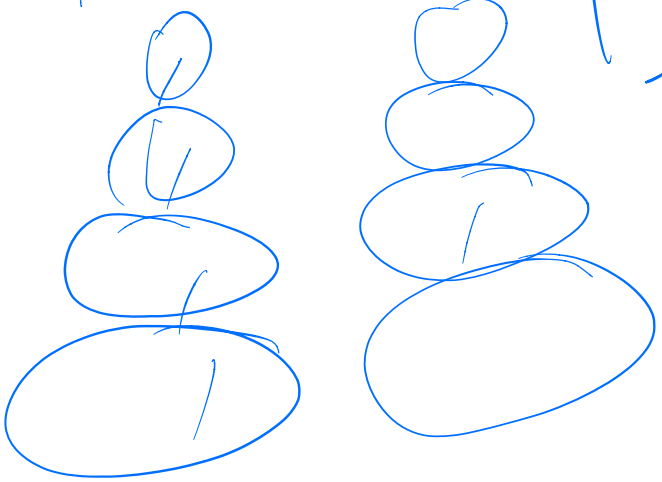
scalable

~ hash table



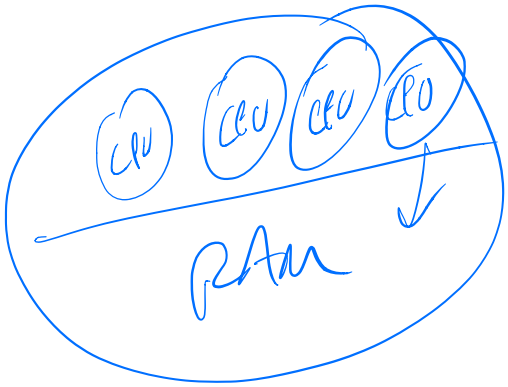
INCLUSIVE

EXCLUSIVE

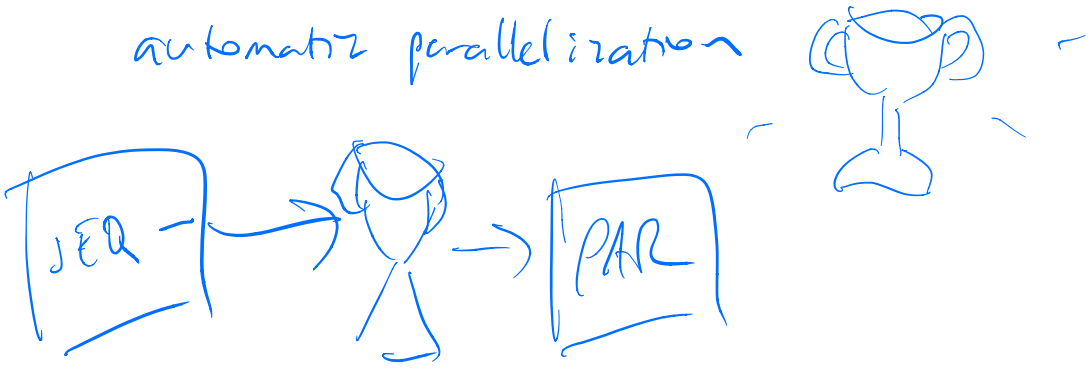


Cache details

Amdahl's Law

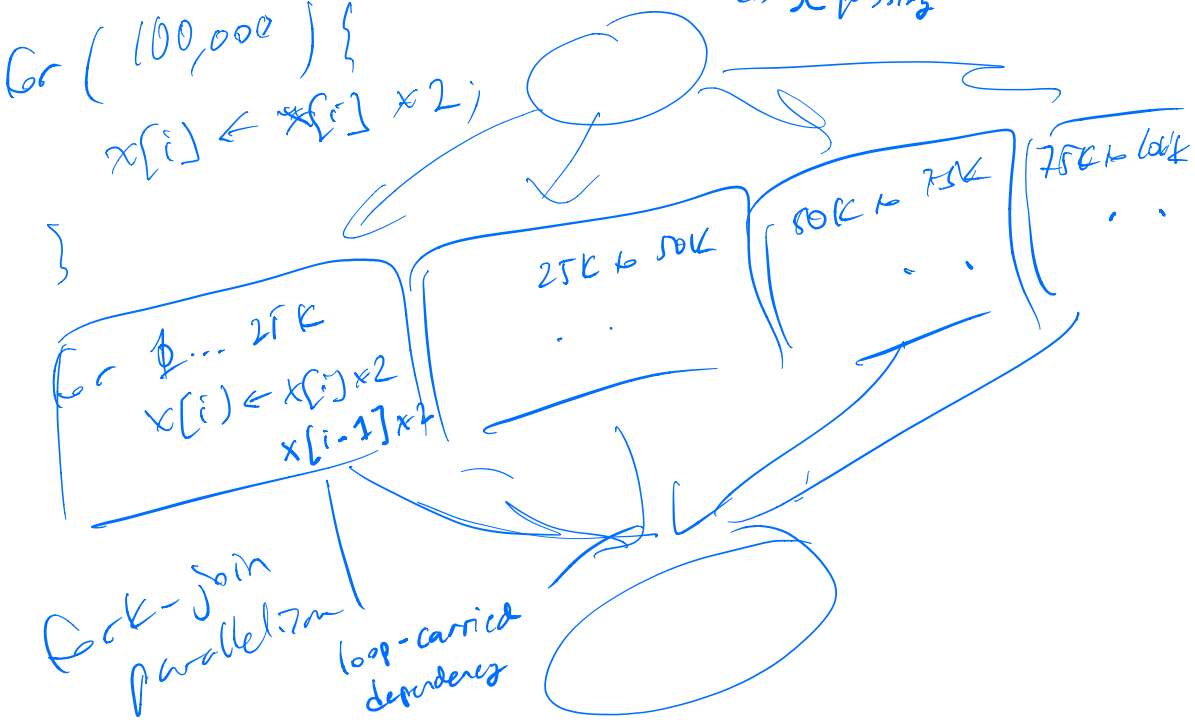


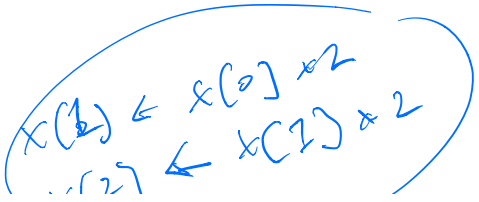
automatic parallelization



Multithreading

Multiprocessing





transform to eliminate loop carried dependency

x[1] ← x[0] * 4
x[2] ← x[0] * 4

x[i] ← x[0] * 2^i

polyhedral analysis

x[y[i]]

DSL

constrained computations
↳ easier to parallelize

manual parallelization

dynamically allocated objects
recursion
side effects
complex expr.

---

libraries

mmult(x, y)
↳

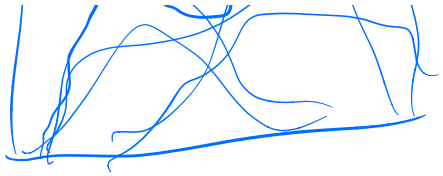
manually tuned

BLAS
Basic Lin Alg Subroutines

FFTW
↳ program synthesis

ATLAS

perf.



size

$A = 3 \times C + 2 \times E$

correctness

nondeterminism

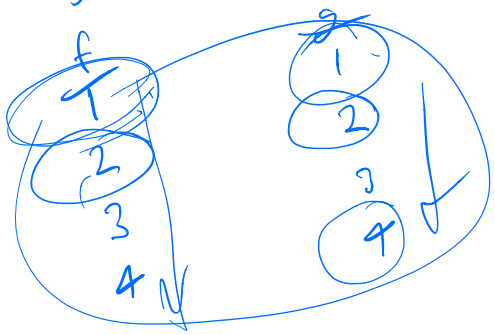
threads
processes

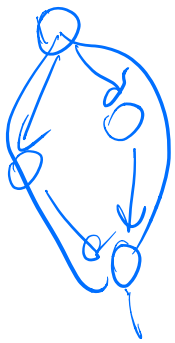
races

performance

```
f() {
  x = 1;
  printf("%d", x);
}

g() {
  x = 2
}
```





f1 g1 s2 s3 g4 f2 f3 f4

2^4

$\sim T^1$

"1"

Synchronization
contention (cache line level)
memory level
] — SEQUENTIAL part

Thread t1 = new Thread(t, arg1);
Thread t2 = " " (t, arg2);

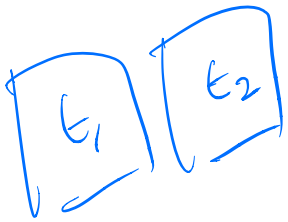
t1.wait()
t2.wait()

T << P — insufficient # of threads
T >> P — too many threads
T ≈ P ?

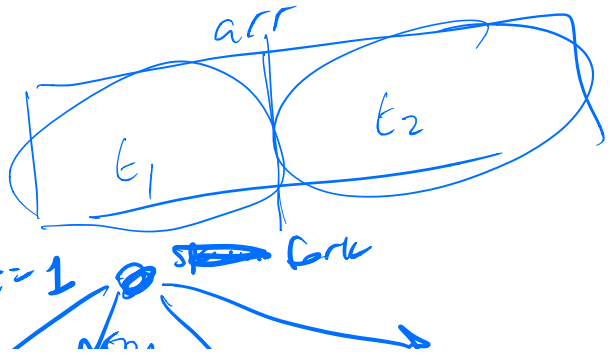
amortize cost

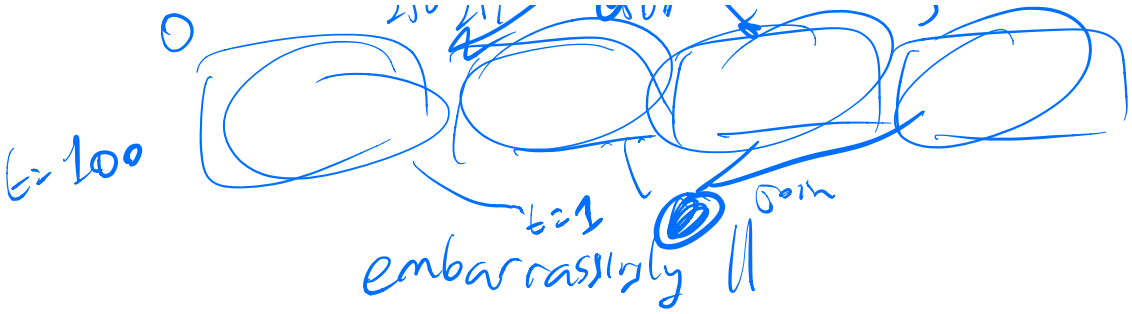
too fine-grained

granularity



Coarse-grained
T ≈ P





T_1 time with 1 processor (serial)

T_p time with p "

goal: $T_p \approx T_1/p$

T_∞ — "critical path"

$$T_p \approx \frac{T_1}{p} + O(T_\infty)$$

$\frac{T_p}{T_1}$ $\quad$ $\frac{p}{p-S}$ $\quad$ .9 $\quad$ .1 $\quad$ 1

Q2: $\frac{.9}{2} + .1$

.55 ~ x

1

$4: \frac{.9}{4} + .1 \sim 3$

$.325$

$9 \quad \frac{.9}{9} + .1 \sim 5$

$.2$

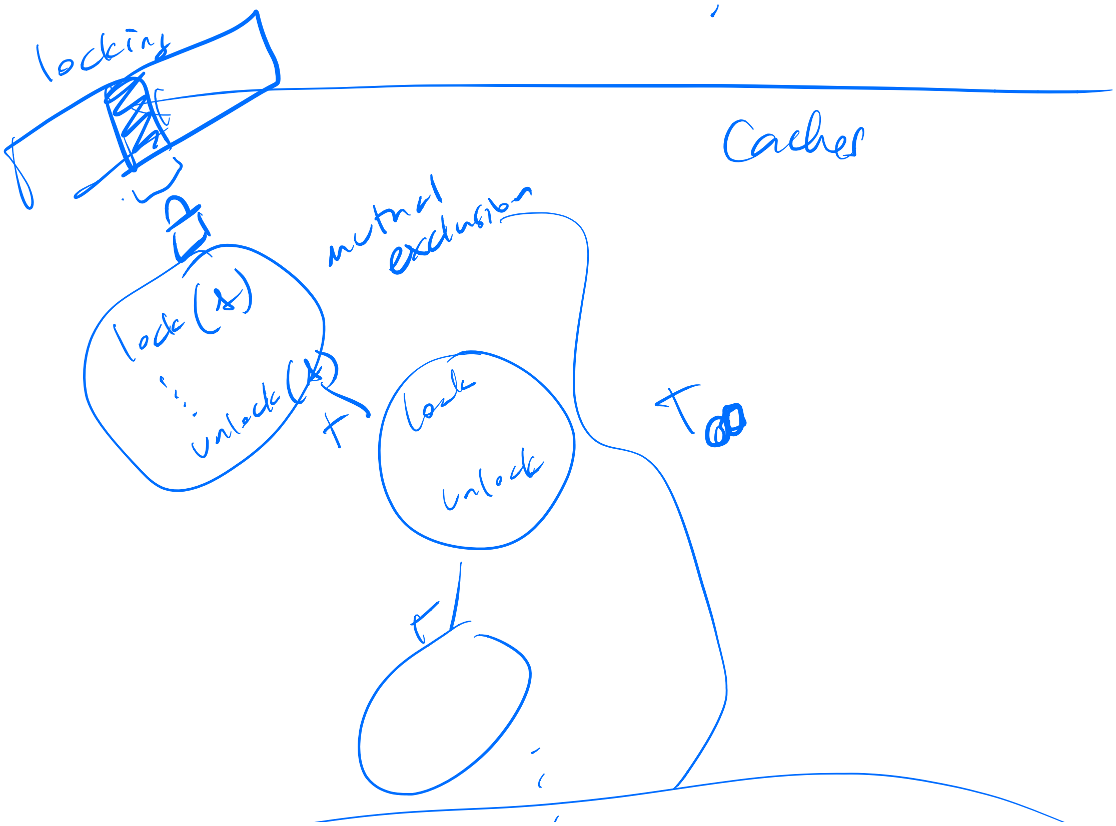
10

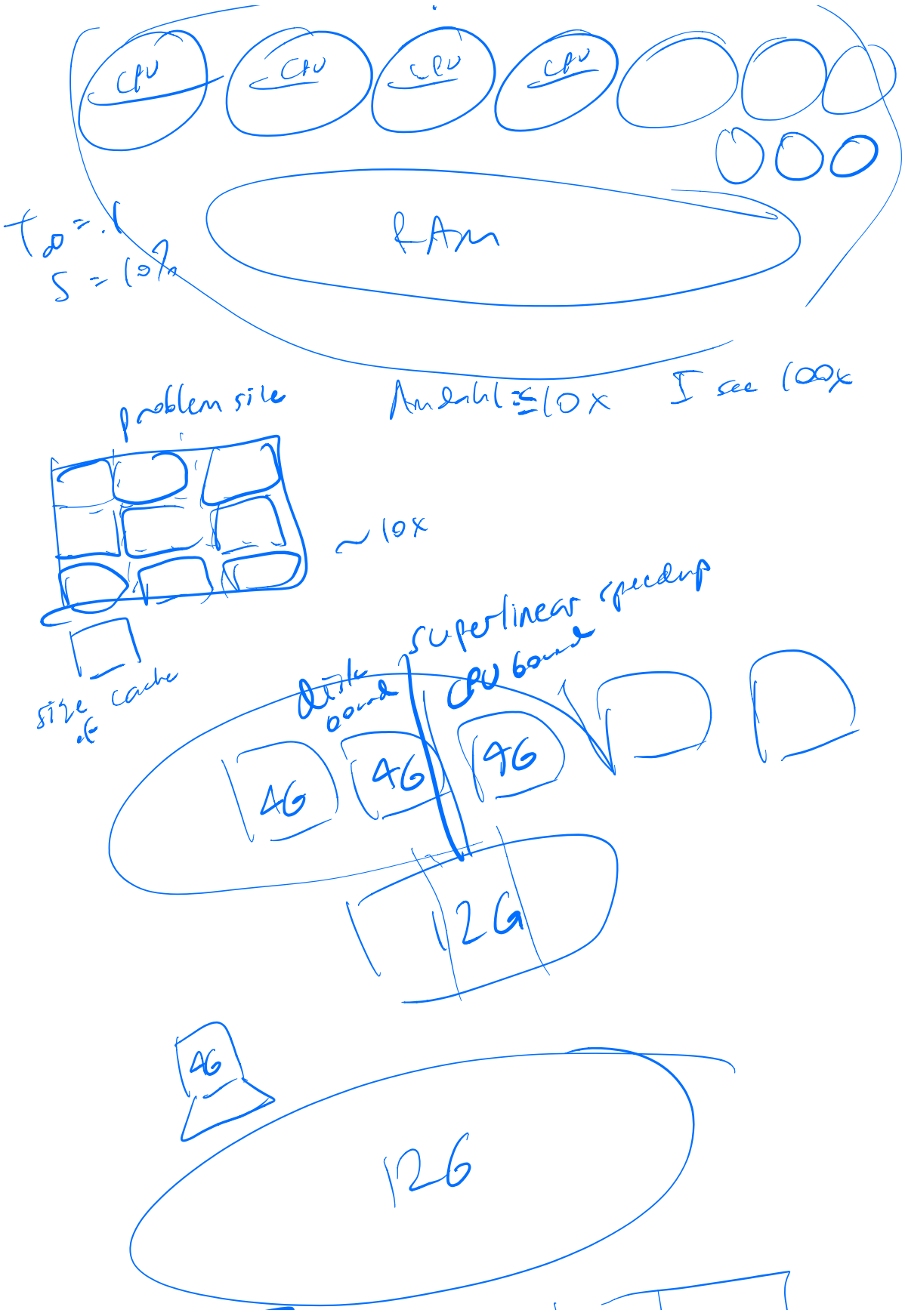
$\vdots$

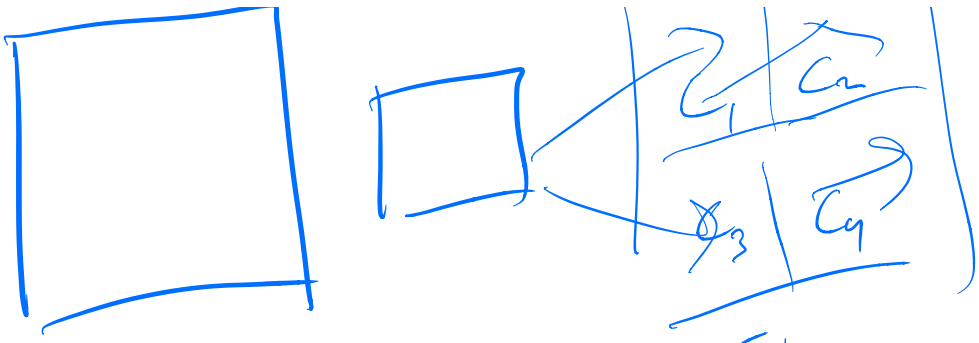
$T_\infty = .1 \approx 10x$ speedup

$T_\infty = .01 \approx 100x$

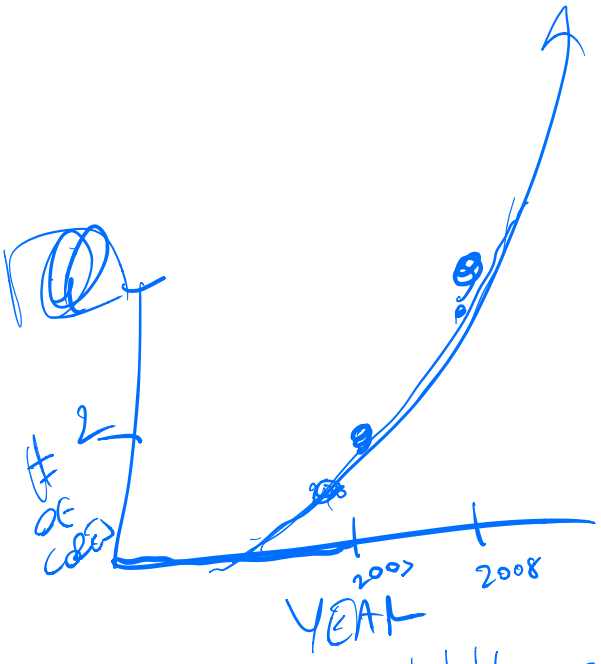
$\vdots$

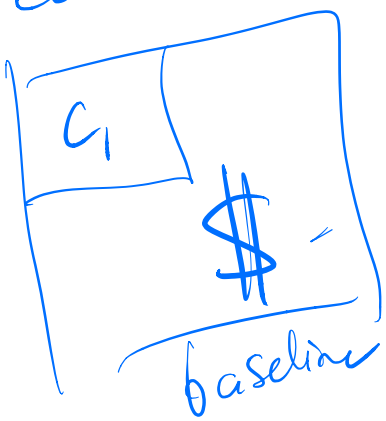


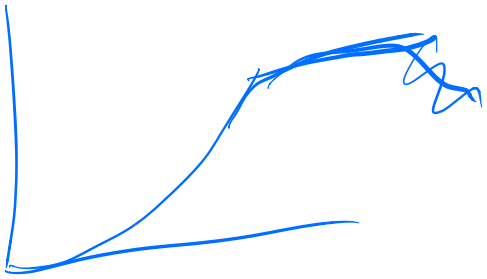

CPU
CPU
CPU
CPU
RAM
$t_0 = 1$
$S = 10\%$
Amdahl ≈ 10×
I see 100×
problem size
~10×
size of cache
disk bound
CPU bound
superlinear speedup
4G
4G
4G
12G
4G
12G

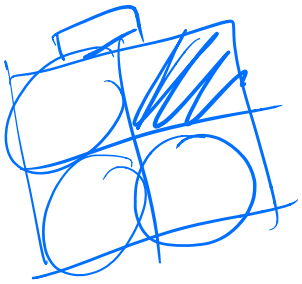


cores









7 core
5 core
7 core

caches

dataflow

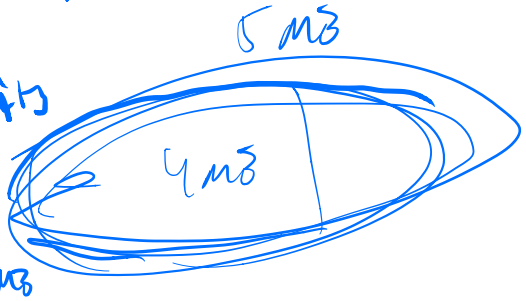
Cache misses

CAPACITY

COMPULSORY first access

CONFLICT miss

↳ not fully associativity (full)



COHERENCE miss

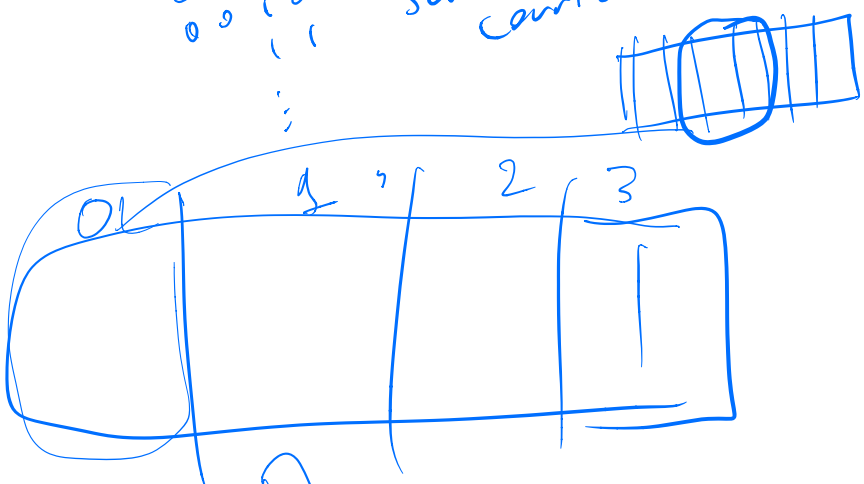
time L3 4MB

LRU

MTF
move to front

0000
0001
0010
11
:

saturating counter

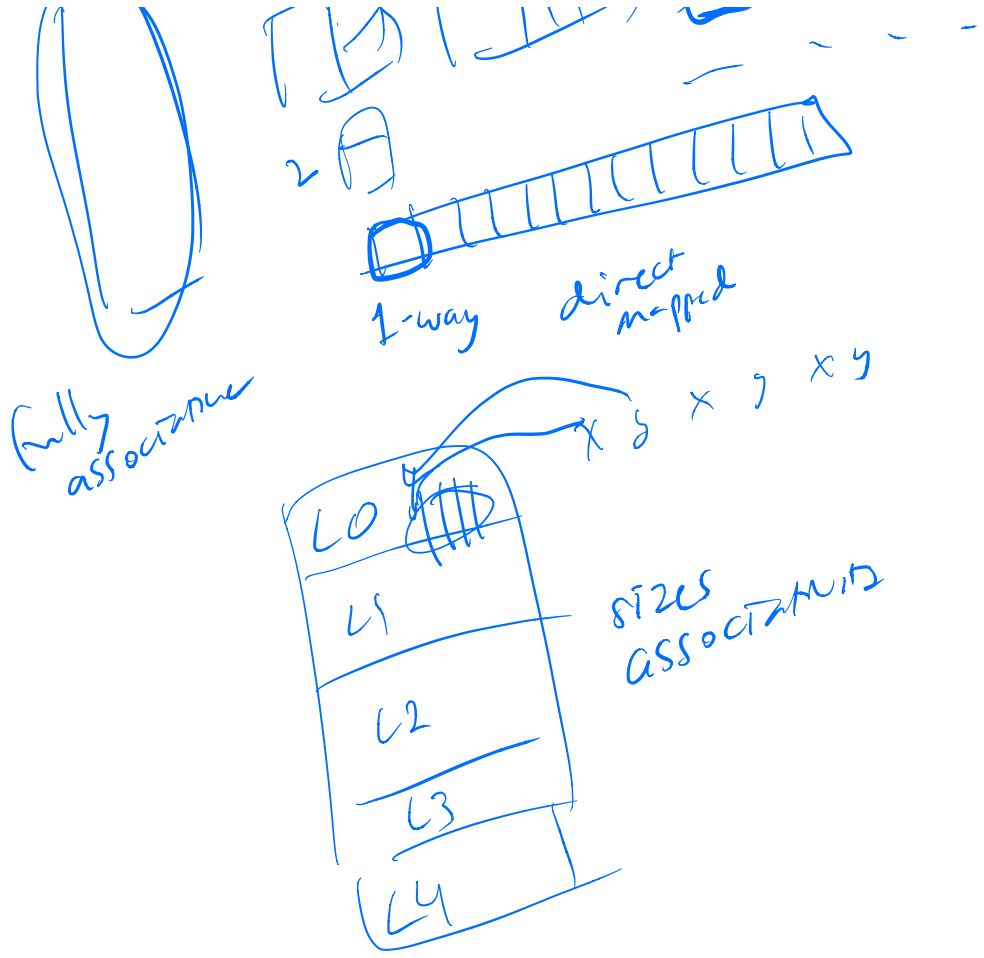


sets &

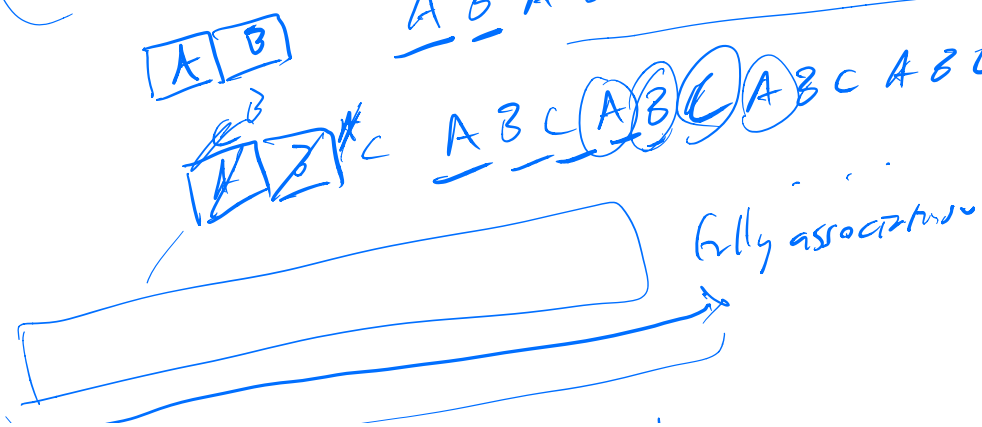
4

set - associative

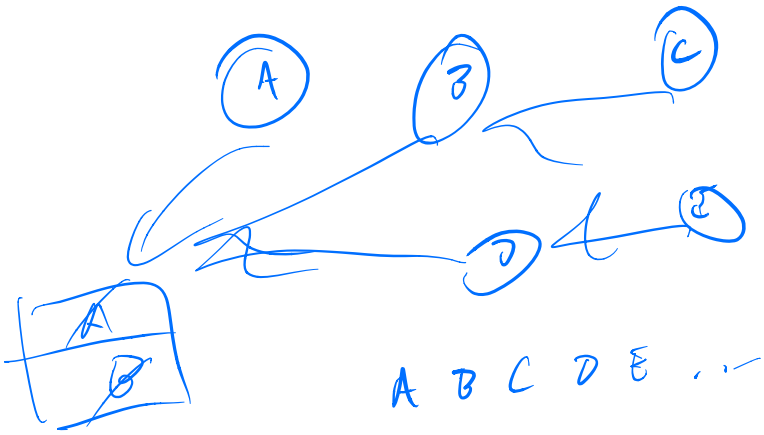
WAY




```
for(i=0; i<1024; i++){
    x[i] = ~ i;
}
```

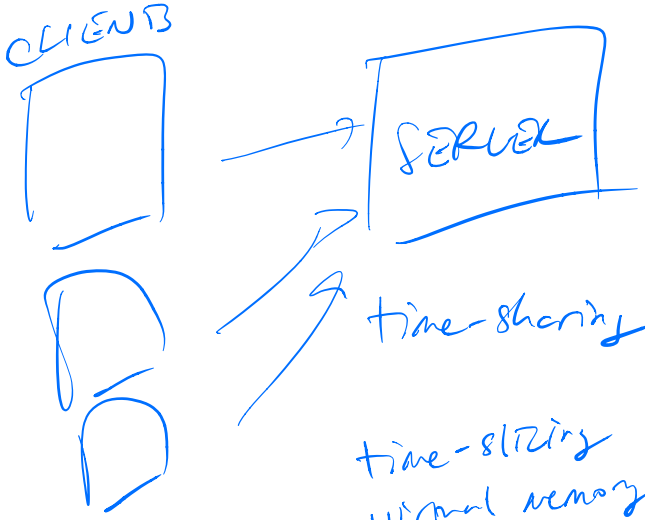


pathological
worst-case

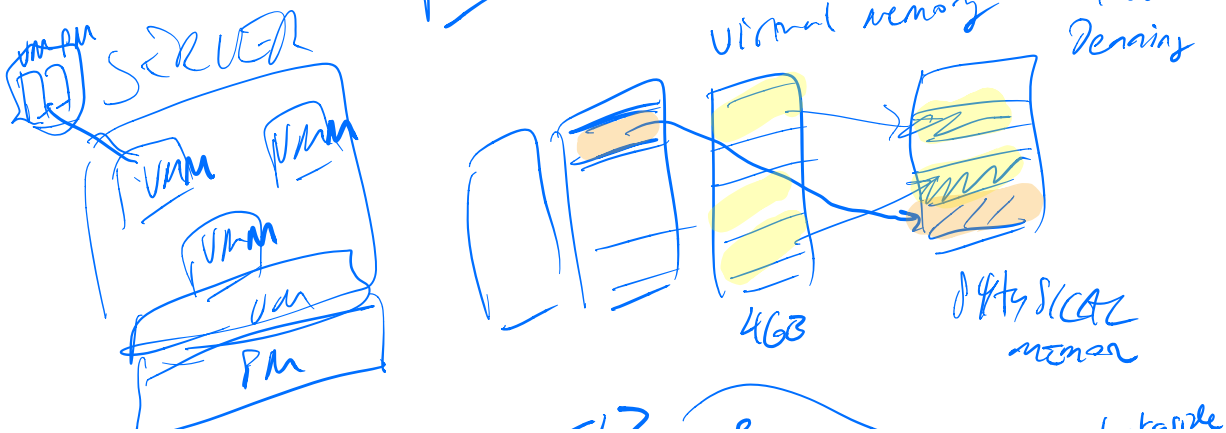


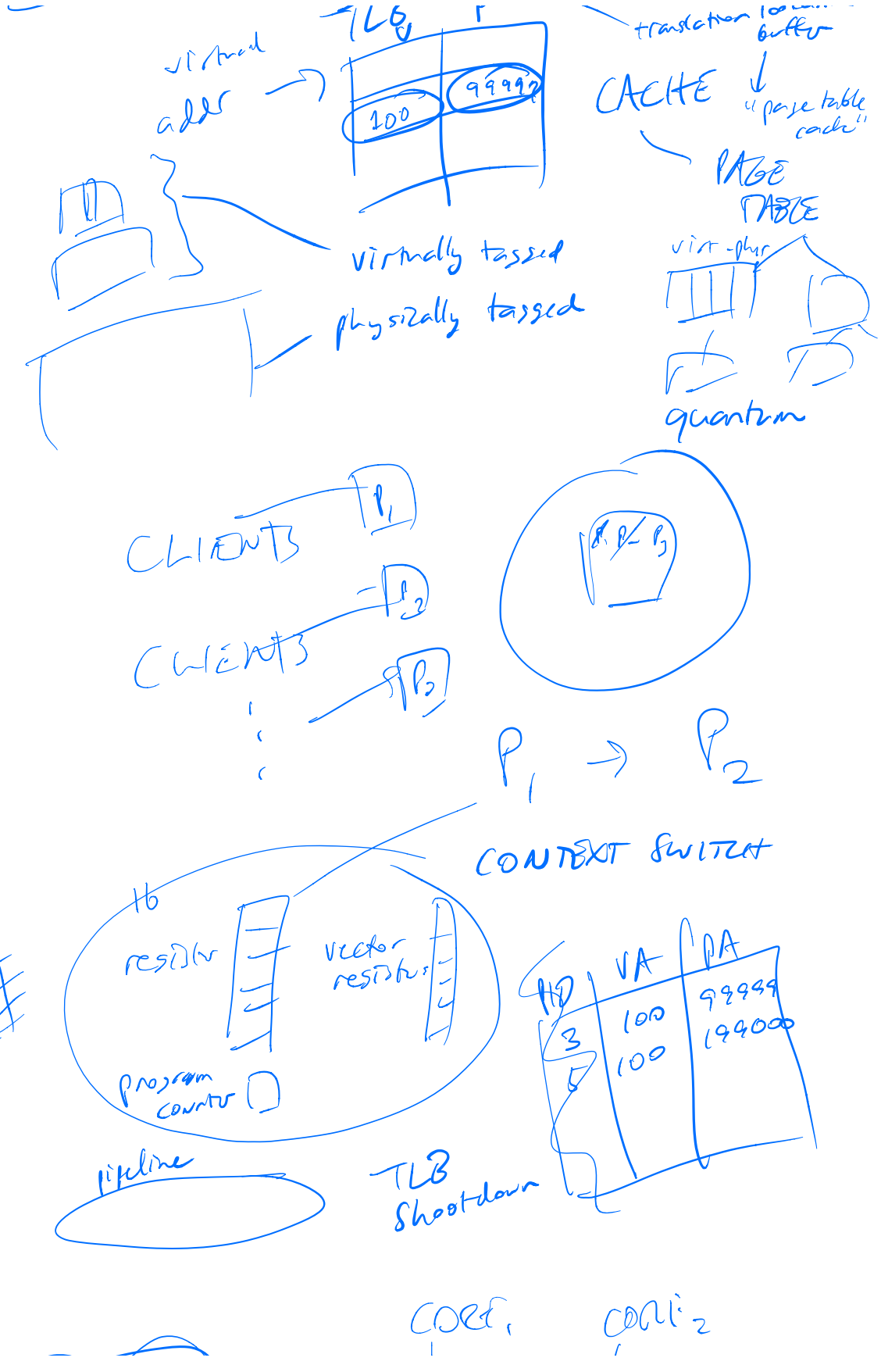
interlude

software architecture

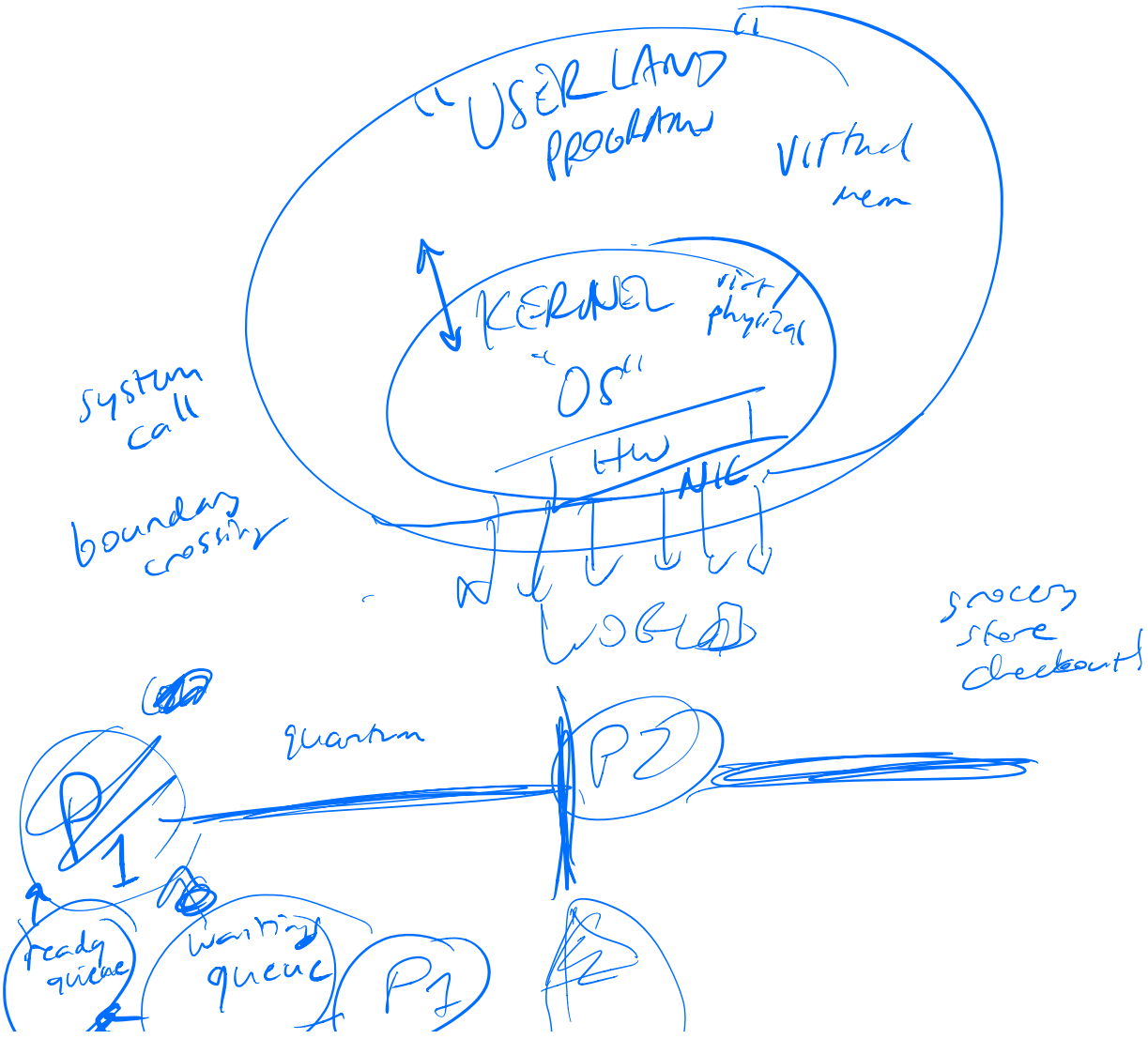


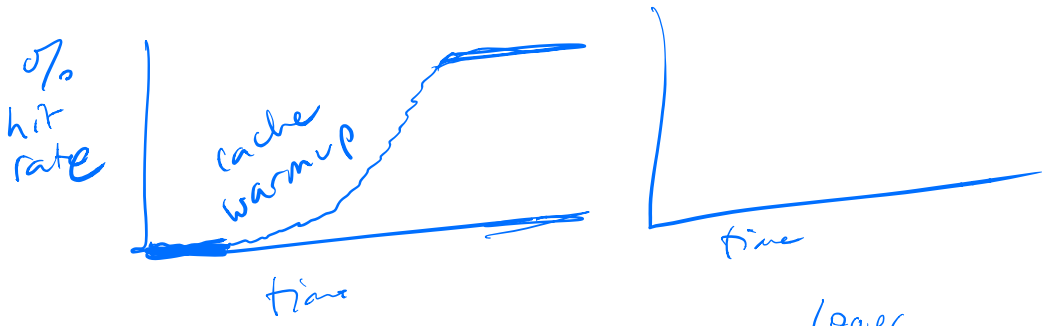
time-slicing
virtual memory ~ 1968
Denning




TLB
translation lookaside buffer
virtual addr
100
99999
CACHE
"page table cache"
PAGE TABLE
virt-phys
virtually tagged
physically tagged
quantum
CLIENTS
CLIENTS
P1
P2
P3
P1 → P2
CONTEXT SWITCH
FILES
IO
register
vector registers
program counter
pipeline
TLB Shootdown
HD
VA
PA
3
100
99999
100
199000
CORE1
CORE2


working set
working set


USER LAND PROGRAMS
virtual mem
KERNEL
"OS"
virt physical
system call
boundary crossing
HW
MMU
quantum
P1
P2
ready queue
waiting queue
P3
process store checkout

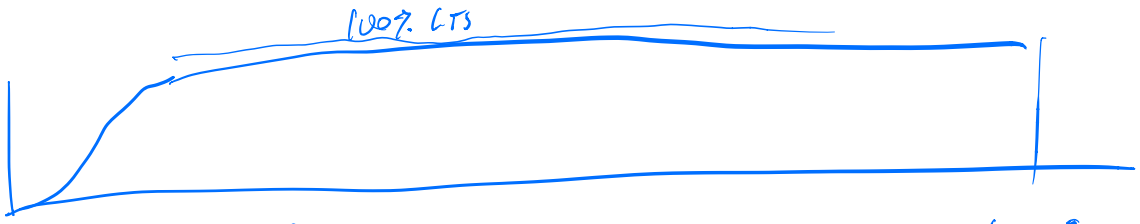


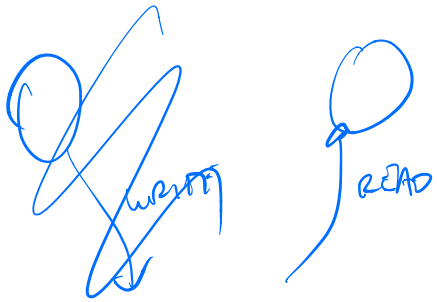
better responsiveness — lower latency

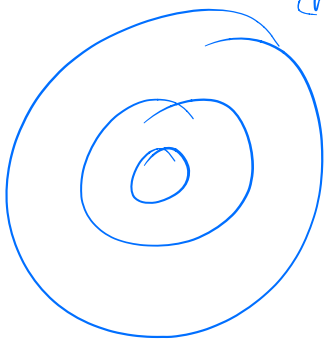
quantum — small

higher throughput "Faster"

quantum — large



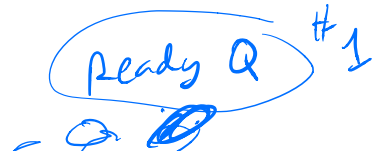




(PROTECTION) RINGS

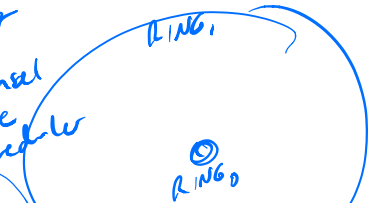
Multics

Bob Graham



priority based scheduling

proportional share scheduler



eventually max

real time scheduler

preemption → PROGRESS

deadlines

Carl Waldspurger

lottery scheduling

STARVATION

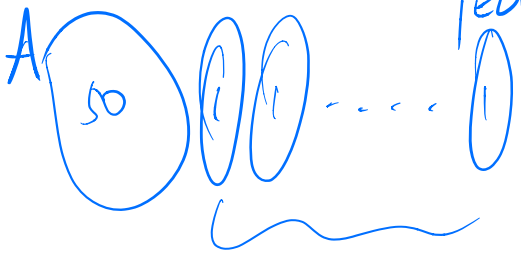
— absence of progress

P_1 10ms

P_2 30ms

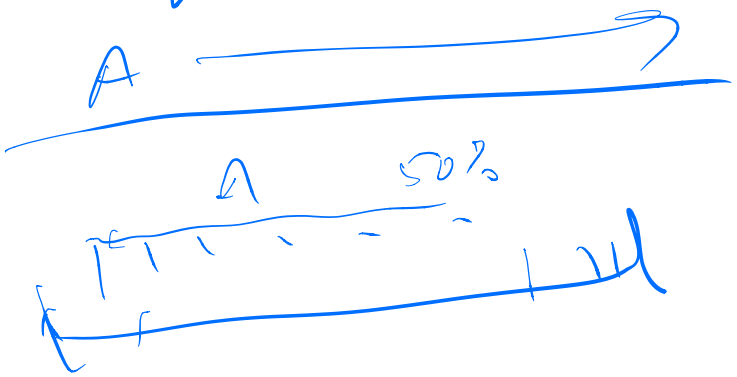
EDF earliest deadline first

greedy & optimal



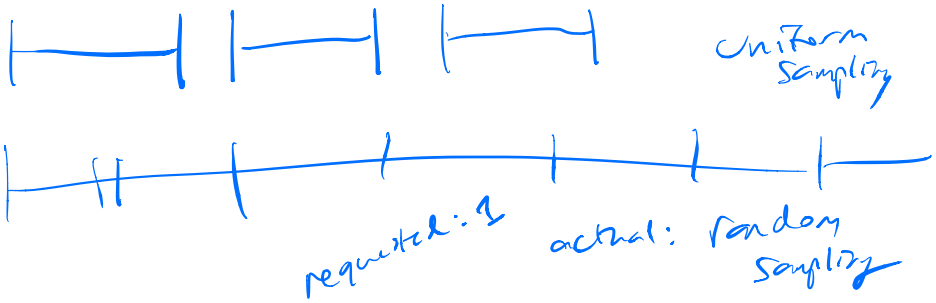
50

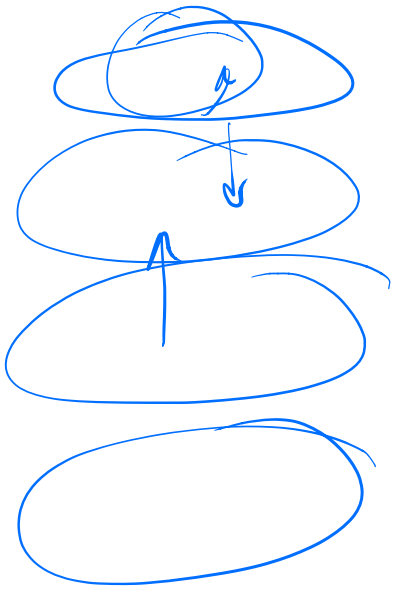
$Pr[A \text{ wins lottery}] = .5$



$A_i \in S$ $\quad corr(A_i, A_j) = 0$

$A_0 \ldots A_{n-1} \Rightarrow A_n$





1

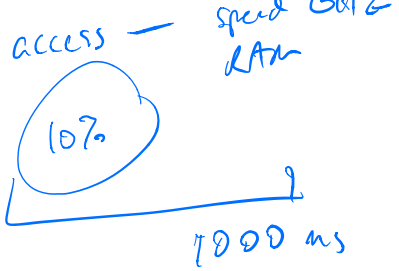
priority

100

CPU utilization : 100%

QoS guarantees

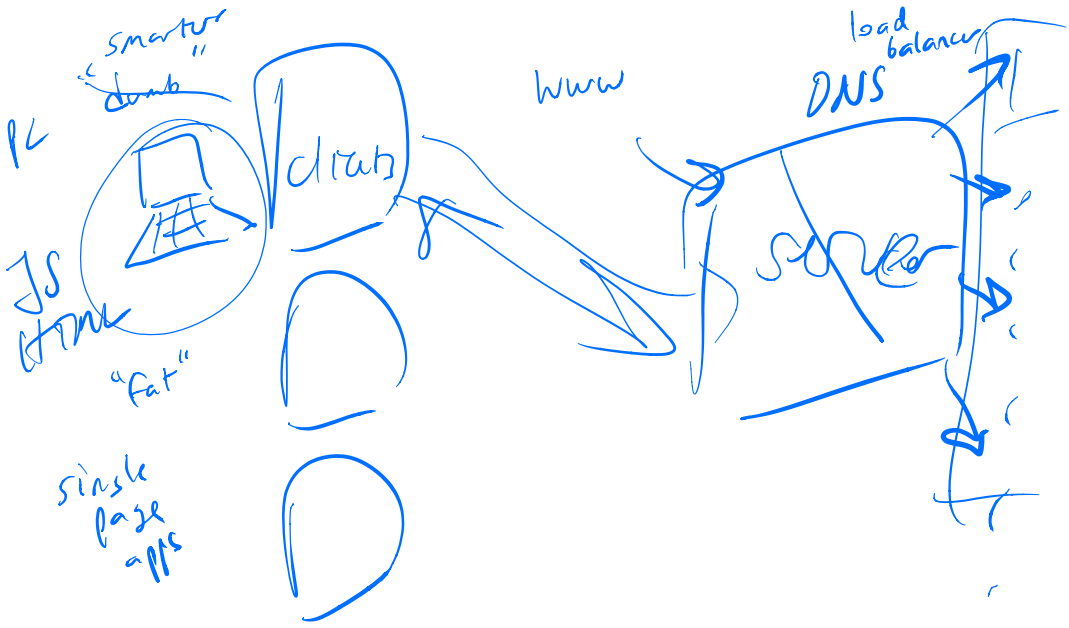
quality of service



availability

$\geq$ 99.9%

$Pr\left[\begin{array}{c}10\% \text{ service} \\ \text{every } 1000\text{ms}\end{array}\right] \geq$.



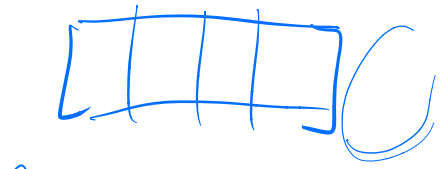
memory leak

Bleak

---

replacement mem allocator for Rust (in Rust)

"safe" language

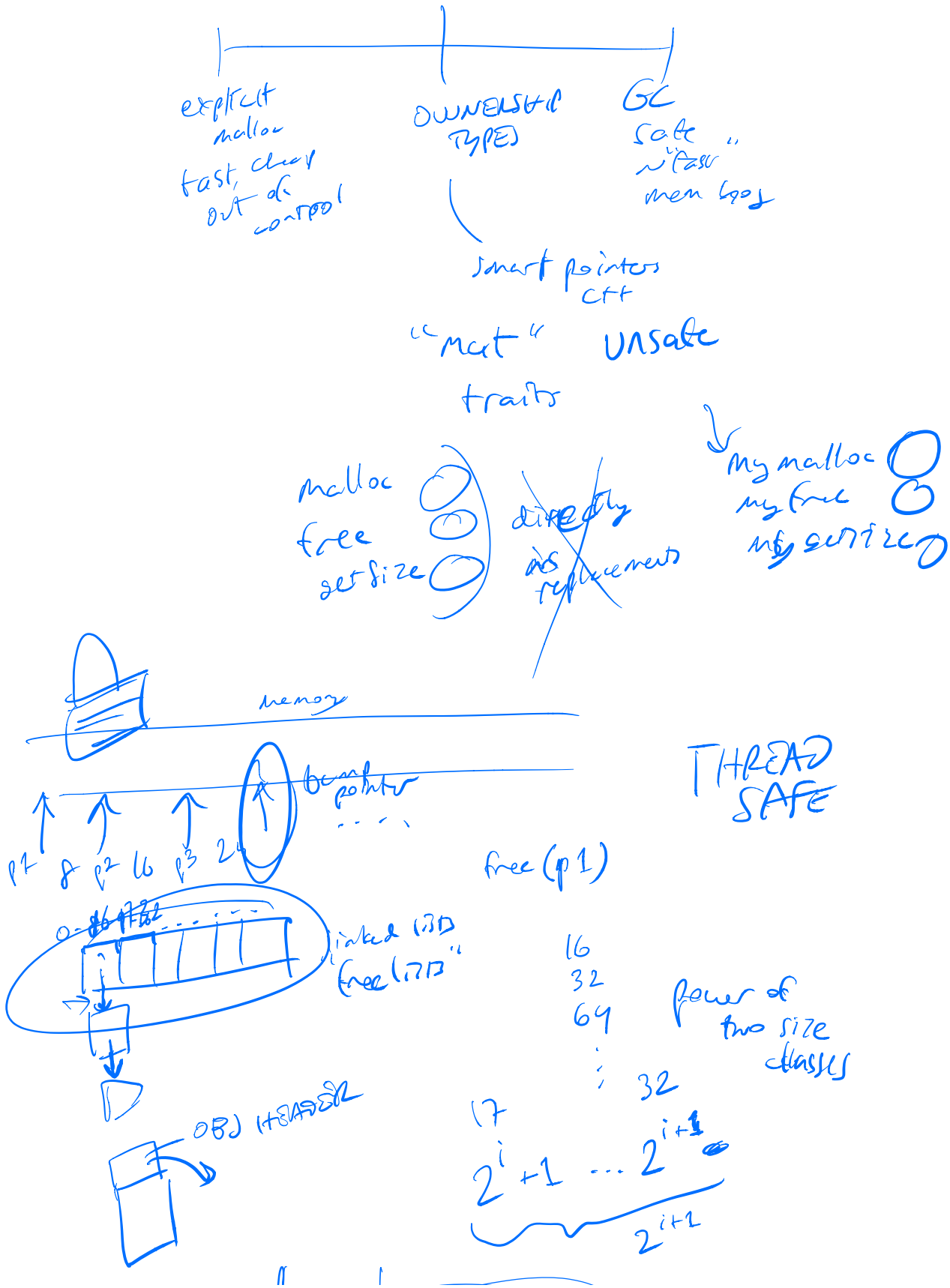


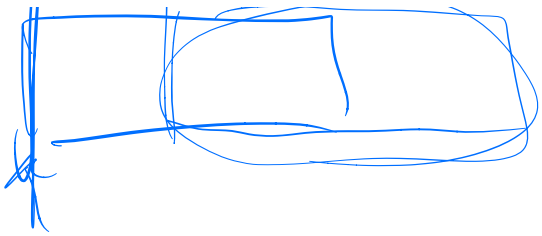
X bounds checking

mem mgmt "safe"

no dangling pointer errors

Rust → no GC

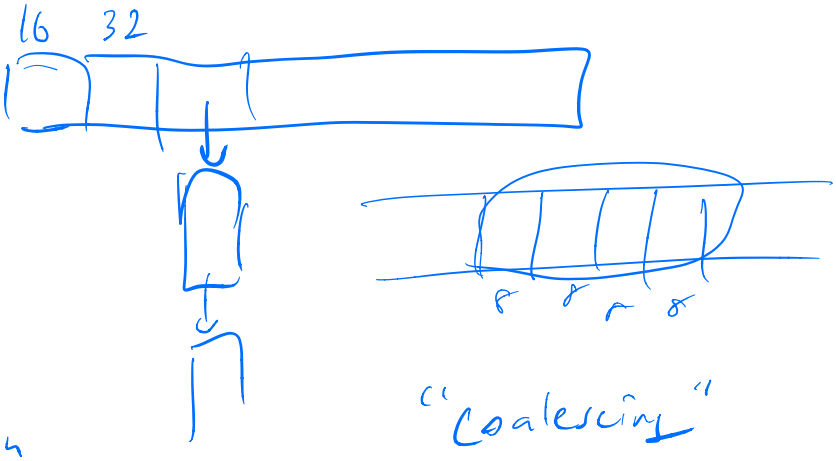

explicit
malloc
fast, clear
out of control
OWNERSHIP
TYPES
GC
safe
"fast"
mem leak
smart pointers
C++
"not"
traits
unsafe
malloc
free
setSize
directly
as replacements
my malloc
my free
my setSize
memory
pointer
p1 8 p2 16 p3 24
free(p1)
THREAD
SAFE
linked list
free list
16
32
64
:
32
power of
two size
classes
17
$2^i+1 \dots 2^{i+1}$
2^{i+1}
OBJ HEADER



all objects MUST be "aligned"

SEGREGATED FITS (BEST-FIT) ALLOCATOR

4-byte 0 8
16-byte 0



Paul Wilson 92/94

"coalescing"

adjacent small freed objects → one big freed object

surveys of uniprocessor GC mem allocator &

Robson 1972

$\left(\log \frac{M}{m}\right)$ biggest / smallest

M × m

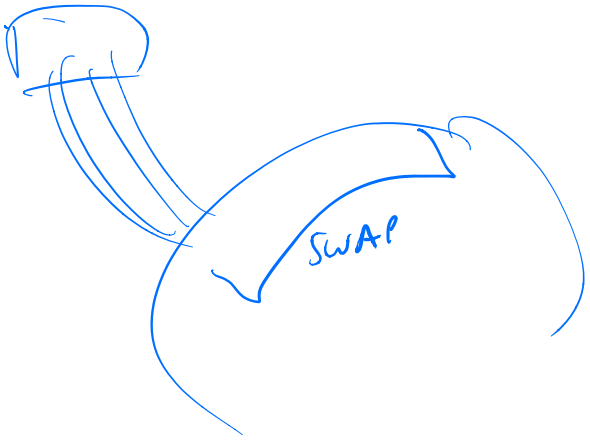
memory

power-of-two size classes

~~Intel x86~~

arrays (stacks)

64 bit systems



OS —

hey I exist

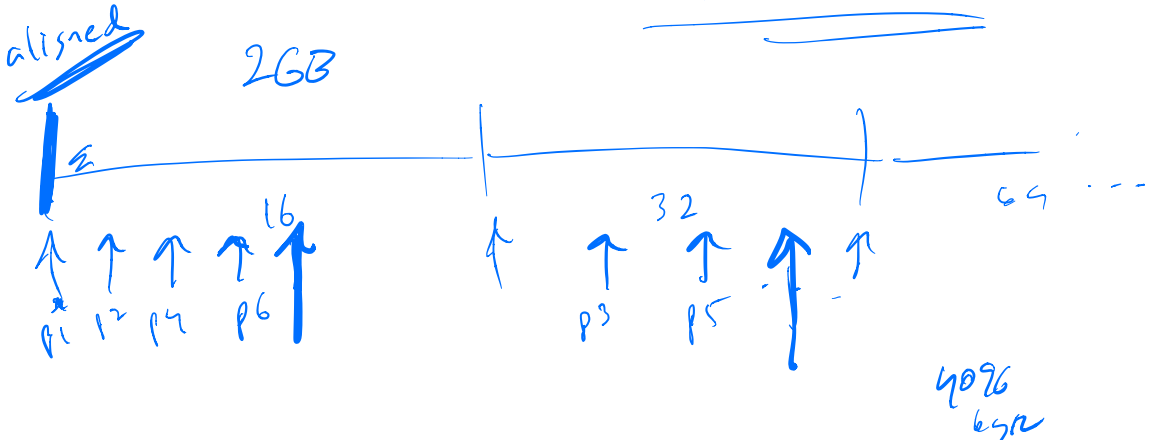
Memory mapped

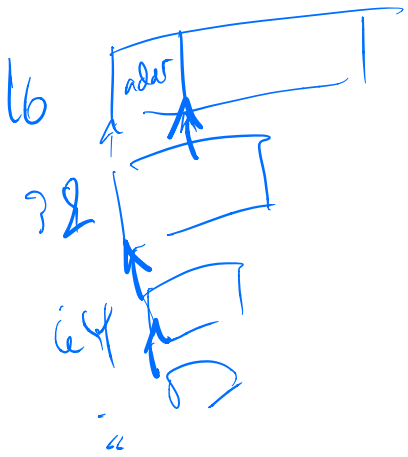
"mmap"

Readable writeable anonymous private

Madvise (DONTNEED)

MAP_NORESERVE





malloc: if stk ! empty
pop stk & ret ptr
else
bump memory ptr
& return

free: push ptr onto stack

inUse
maxAllocated

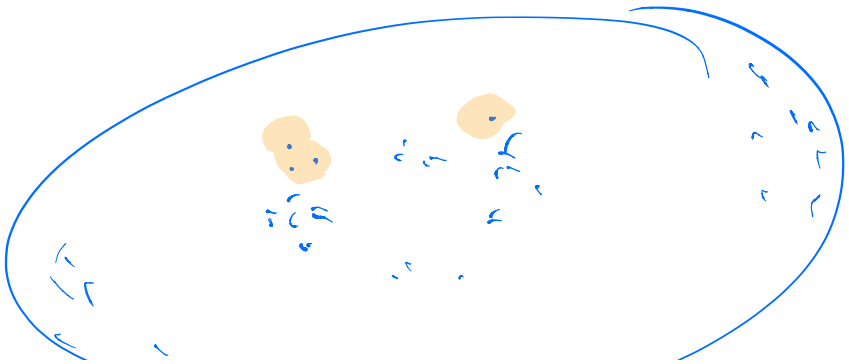
"CHeap"

zero on-demand page

Dave Patterson
Hennessy

RISC
RAID

reduced-instruction set computing



every new instruction for Ite

CISC

Conventional instruction set computing

"BAD for compiler writer"

RISC-V

Krste Asanovic

RISC

2000s

ARM

power

energy consumption

very few addressing modes

same size

MOV

ADD

Intel Xeon

8088

8086

80x86 ...

SSE

SSE2

SSE3

crypto

MMX

TSX

I$

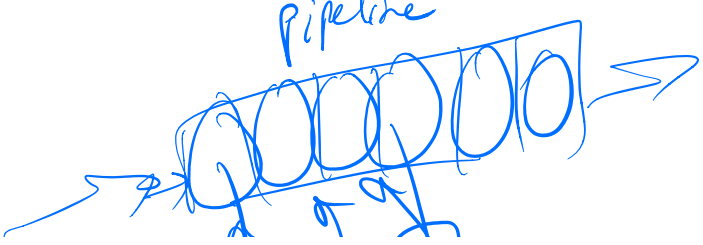
fetch

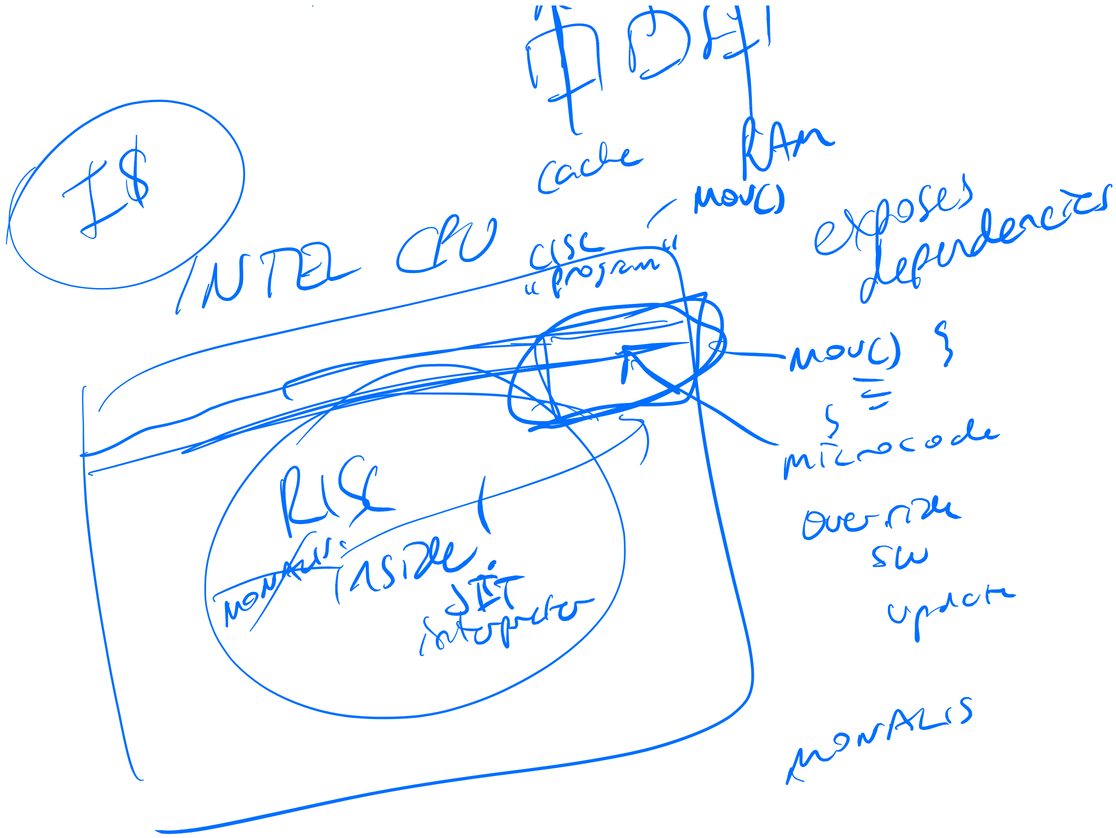
decode

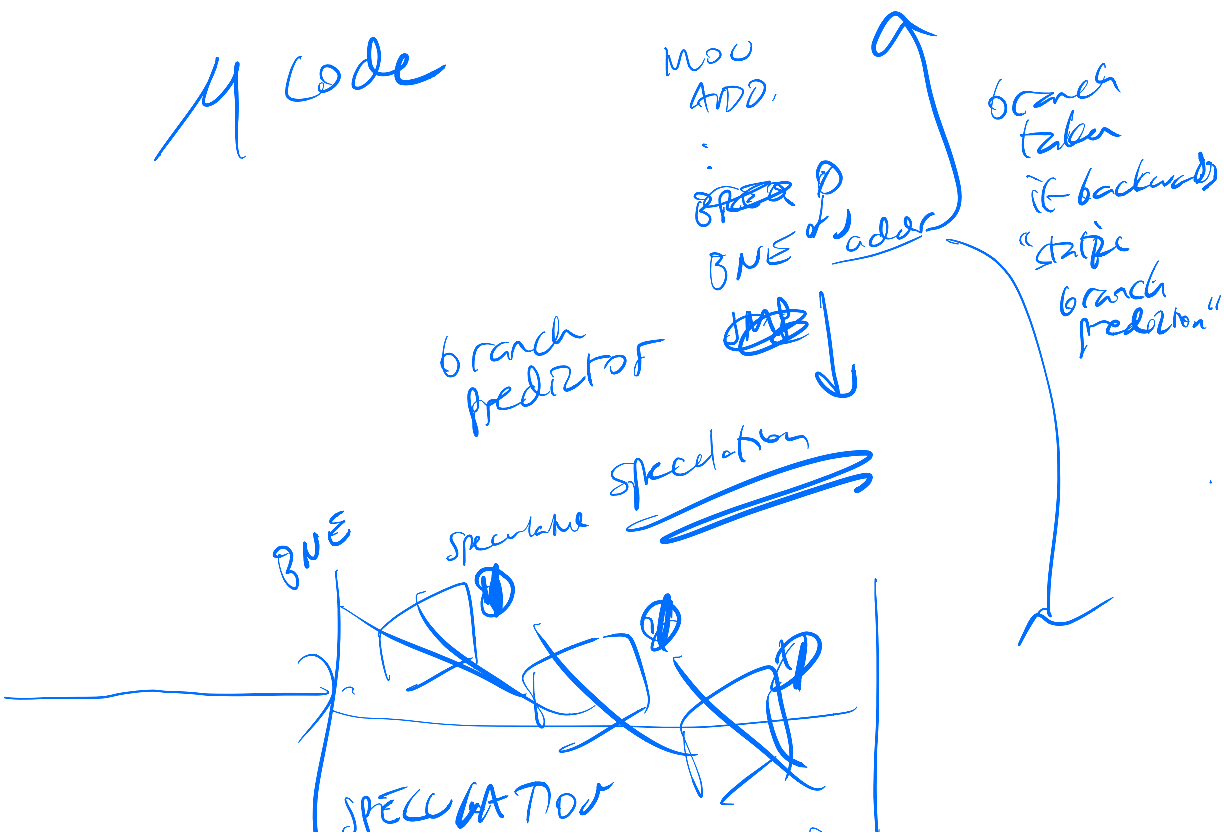
execute

ROP

pipeline




I$
cache
RAM
INTEL CPU
CISC
"program"
MOV()
exposes dependencies
MOV() {
}
microcode
override
SW
update
RISC
INSIDE
JIT
interpreter
MONALIS


μ code
MOV
ADD
BNE
addr
branch taken if backwards
"static branch prediction"
branch predictor
speculation
BNE
speculate
SPECULATION

speculative state

YES! NO

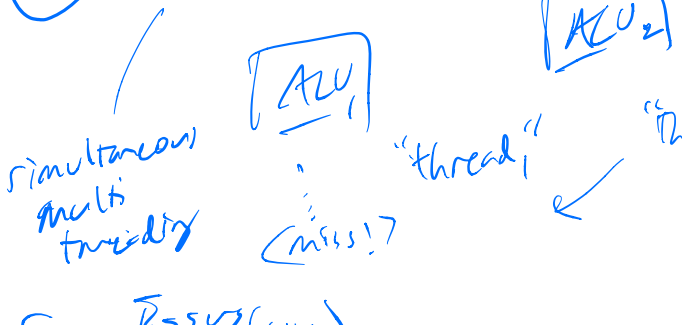
fetch
decode
executes — JMP PC = ...
BNE cond flag, PC = ...

LATENCY of mem ($$)

(1) Speculation
prediction = failed prediction
discard speculative state
"roll back"

(2) "SMT" / hyperthreading

Use concurrency to hide latency

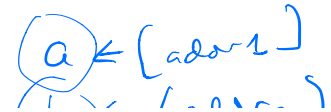


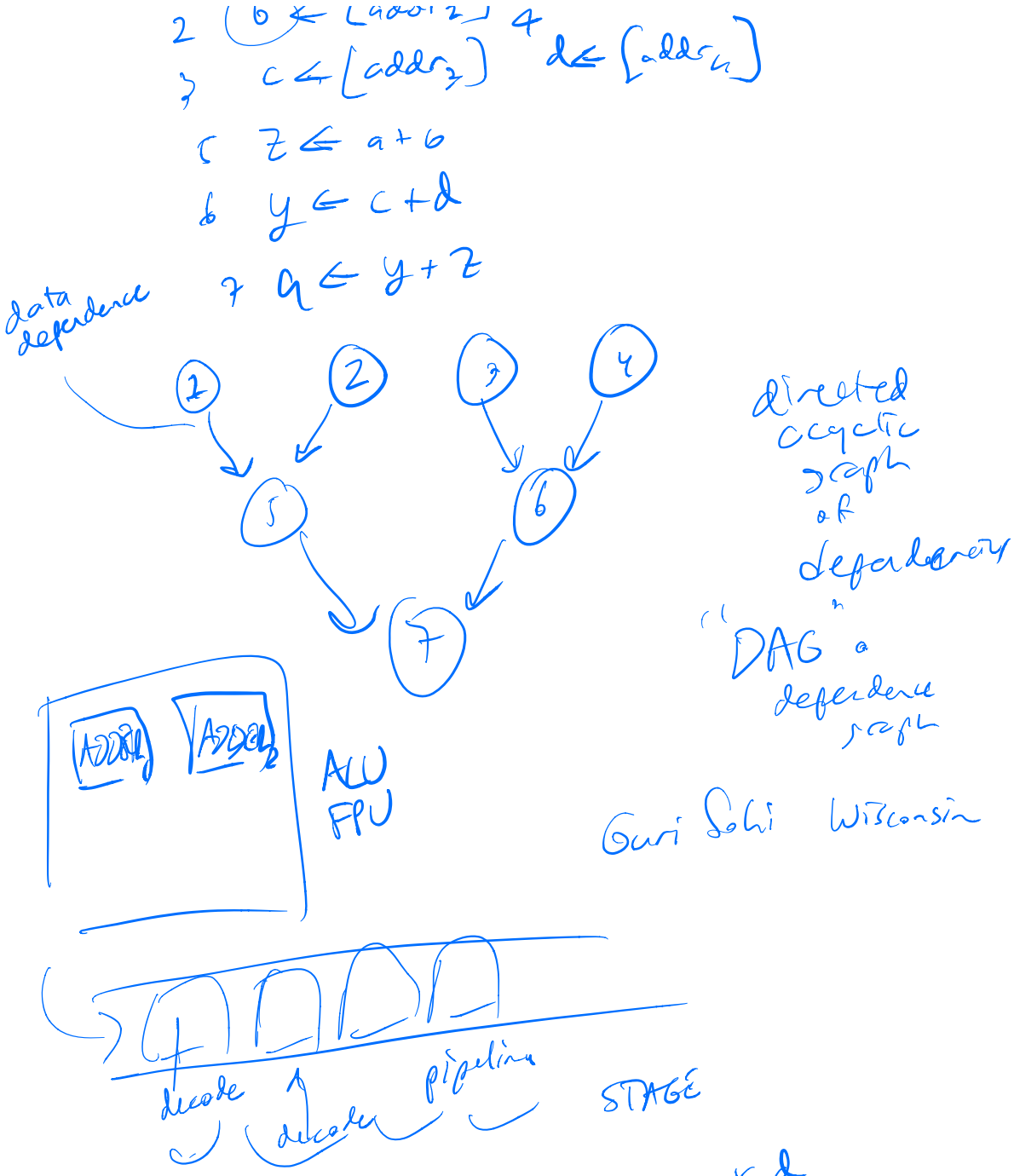
simultaneous multi threading

Susan Eggers (UW)
Dean Tullsen

across thread ll ism

(3) "intra-thread parallelism" ILP
instruction-level



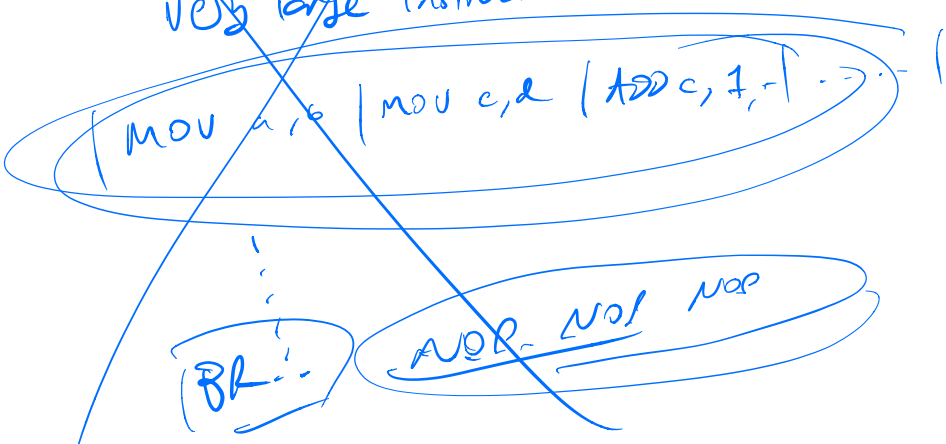

2 b ← [addr2] 4 d ← [addr4]
3 c ← [addr3]
5 z ← a + b
6 y ← c + d
7 q ← y + z
data dependence
1
2
3
4
5
6
7
directed cyclic graph of dependences
"DAG" = dependence graph
ALU
FPU
Guri Sohi Wisconsin
decode
decode pipeline
STAGE
e ← a x b + c x d
available ILP
"IA-32" 80286 3 4 5th version

IA-64

Itanium
Itanic

VLIW

very large instruction word



PA-RISC
↳ Itanium

EPIC

explicitly parallel instruction coding

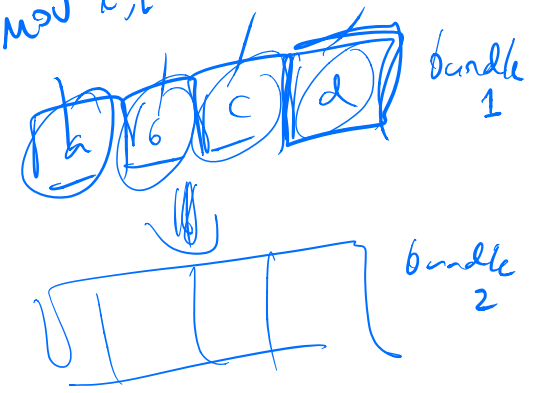
- conflict detection

"An EPIC Fail"

VLIW

(branches)

MOV x, [addr]



in-order

OOO
out of order

alias analysis

foo(int *a, int *b)

*a = 12 + x

*b = 12 + y

Pure LISP

(cons

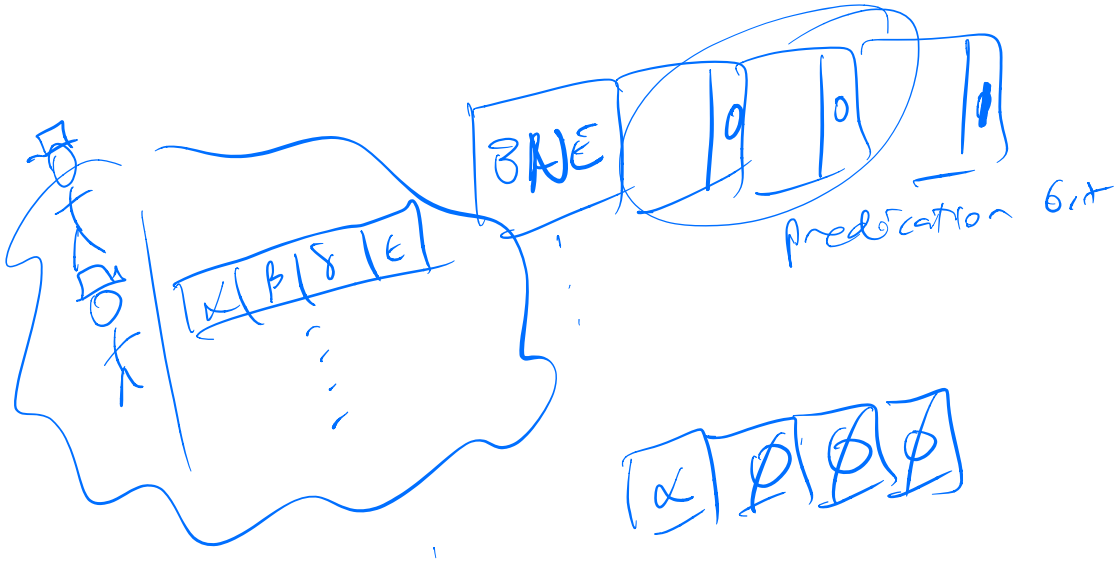
(car head

(cdr tail

(let ((a (12))

(b (+ 13 9)))

(+ a b))



ITANIUM

STATIC
vs
DYNAMIC

prefetcher

AMD-64

— Q

Java object → monitor

synchronized void - - (

recursive
locks
condition variables/
monitors

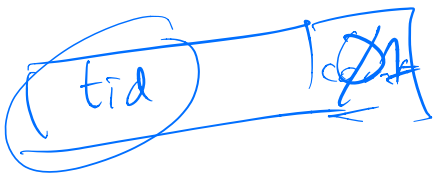
)

synchronize (this) {

}

obj.wait()

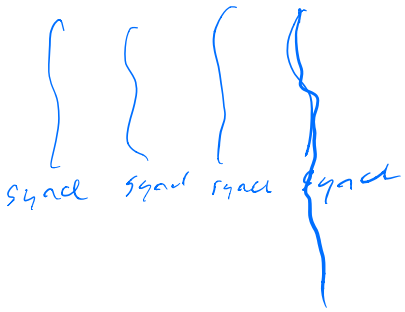
obj.notify()
notifyAll



WAIT WAIT WAIT WAIT

NOTIFY
NOTIFYALL

thundering
herd



STARVATION

progress property

something good eventually happens

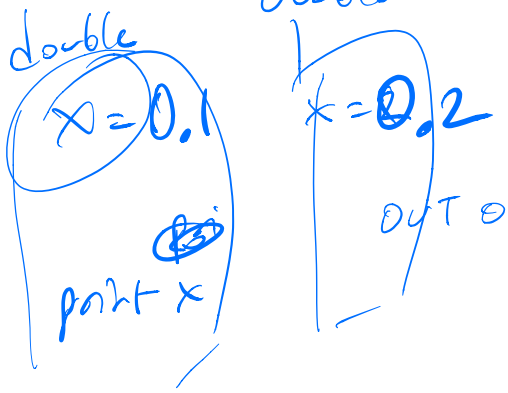
safety property

nothing bad ever happens

nondeterminism not necessarily the enemy

mutexes — mutual exclusion

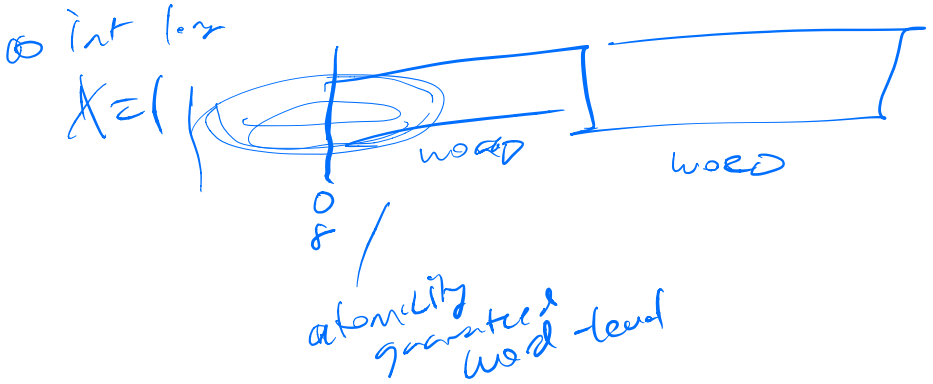
no races

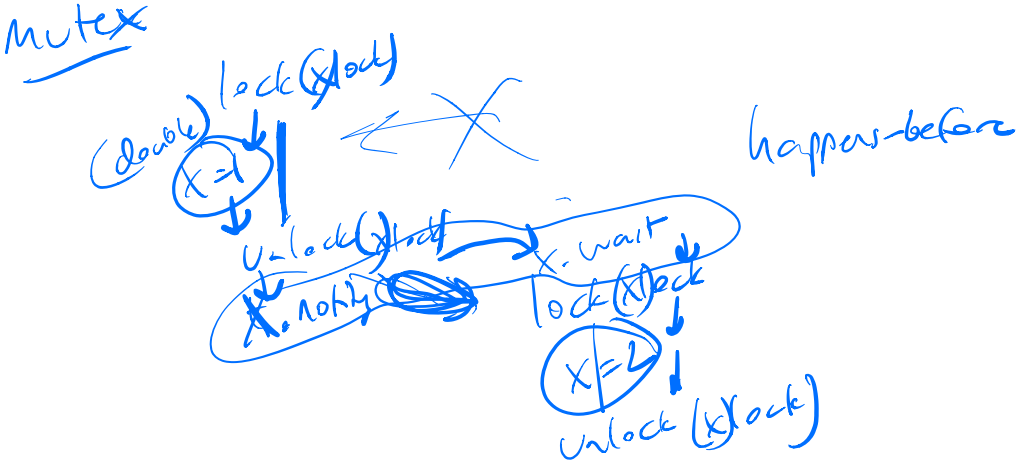


nondet result based on time.

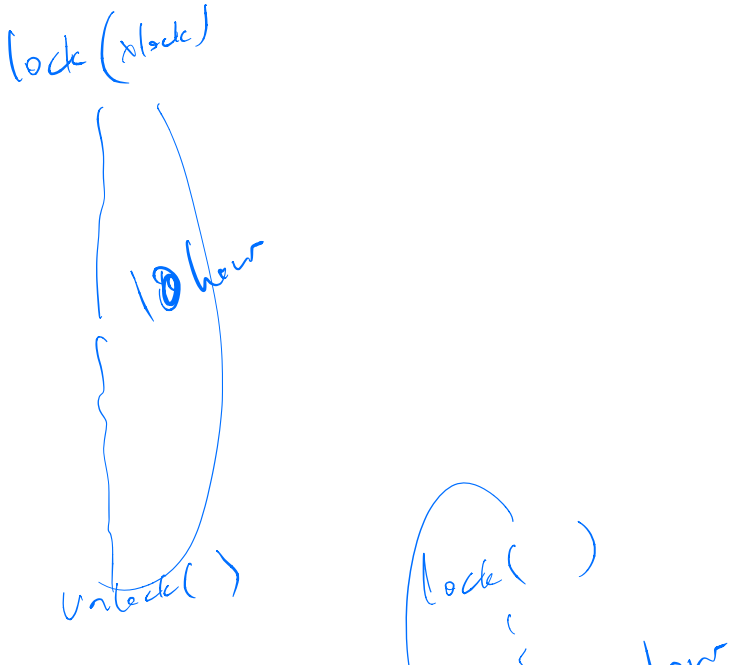
atomic all or nothing

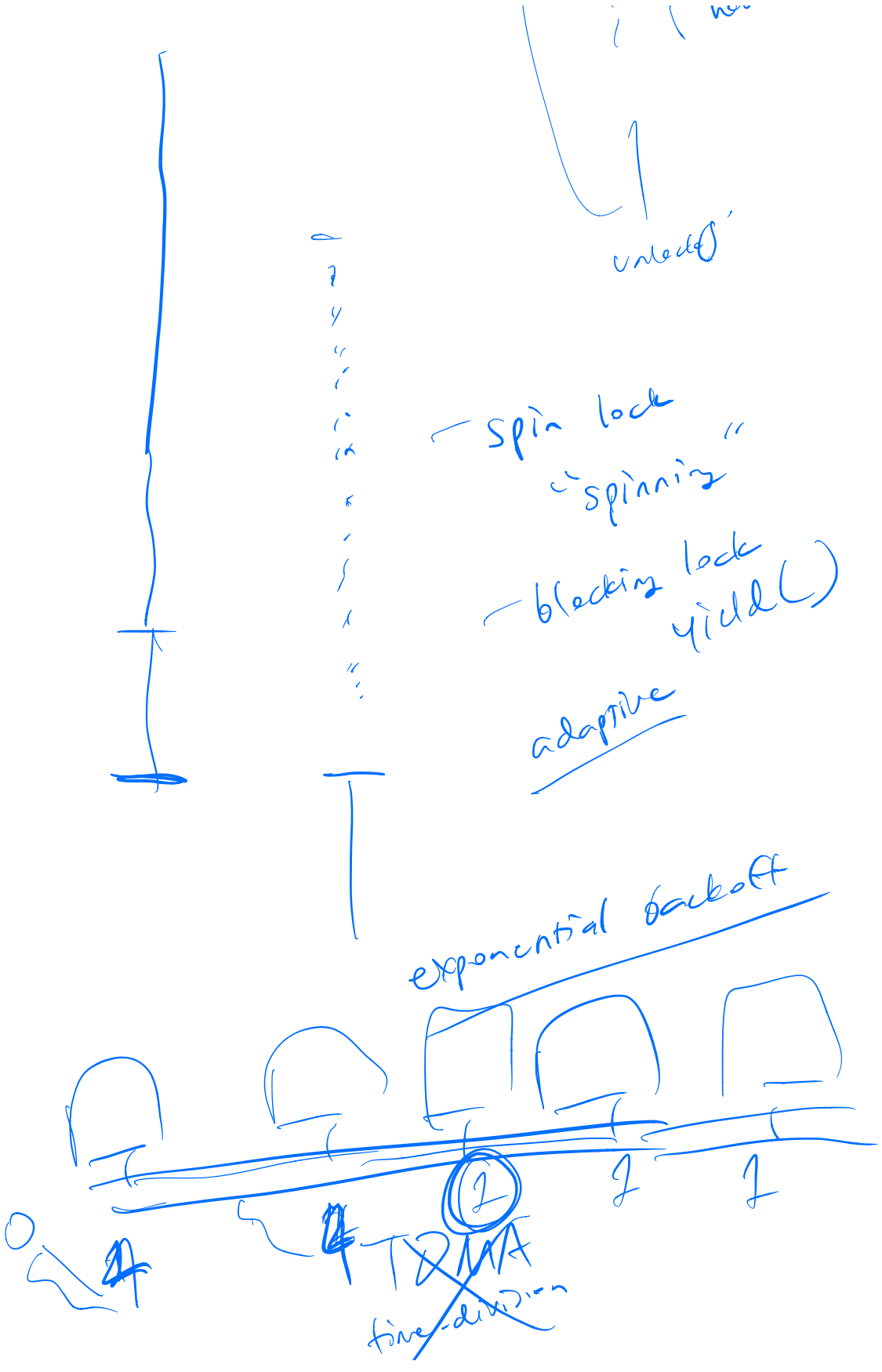
atomos





no happens before relationship = race




unlock()
spin lock
"spinning"
blocking lock
yield()
adaptive
exponential backoff
2
2
2
TRMA
time-division

randomized / exponential backoff

TRUE
interval *= 2

MCS lock

security aside

dead-store elimination

x = ack(300, 199) {
}
live ranges
print x;
liveness analysis
for performance

Xi Wang
UWash
(MIT)

scrubbing

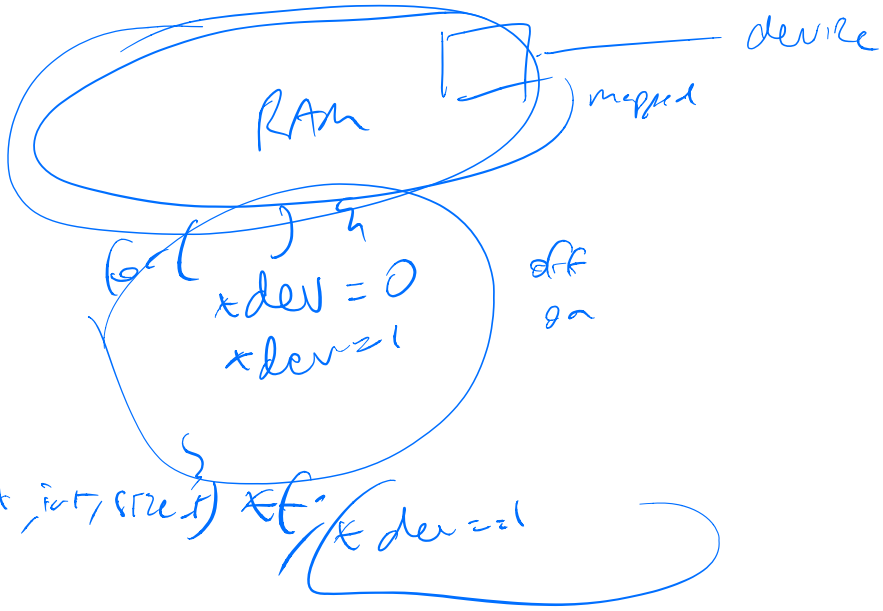
memset(buf, 0, sz);

buf[0] w
buf[1] w
⋮

asm ( ; read(buf, —)]

buf[0] w
⋮ w

volatile



volatile
int * dev;

volatile
void (void *, int, size_t) *f;

f = memcpy

---

atomicity of words

double x;

x = 1

x = 1000

atomic double x;

x = 1;

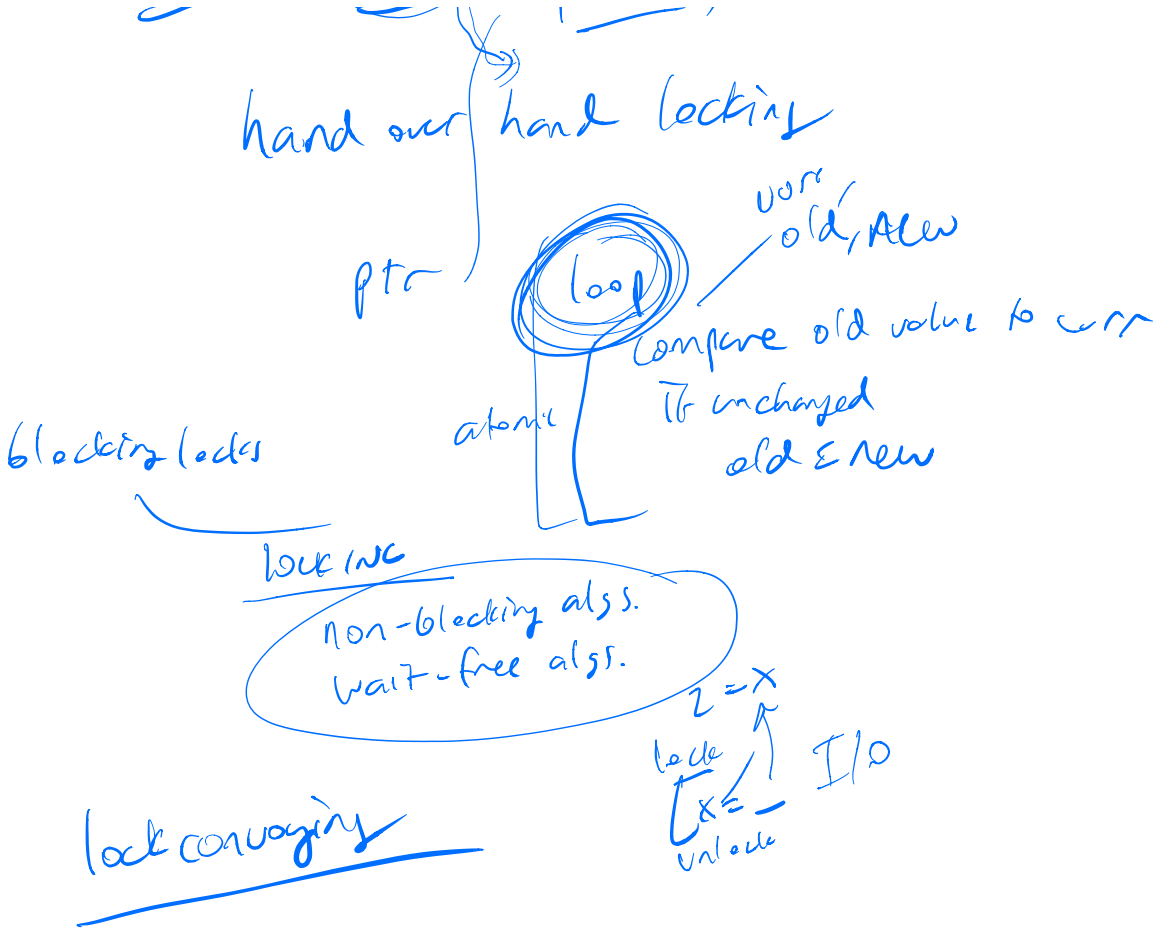
LOCK MOV ...

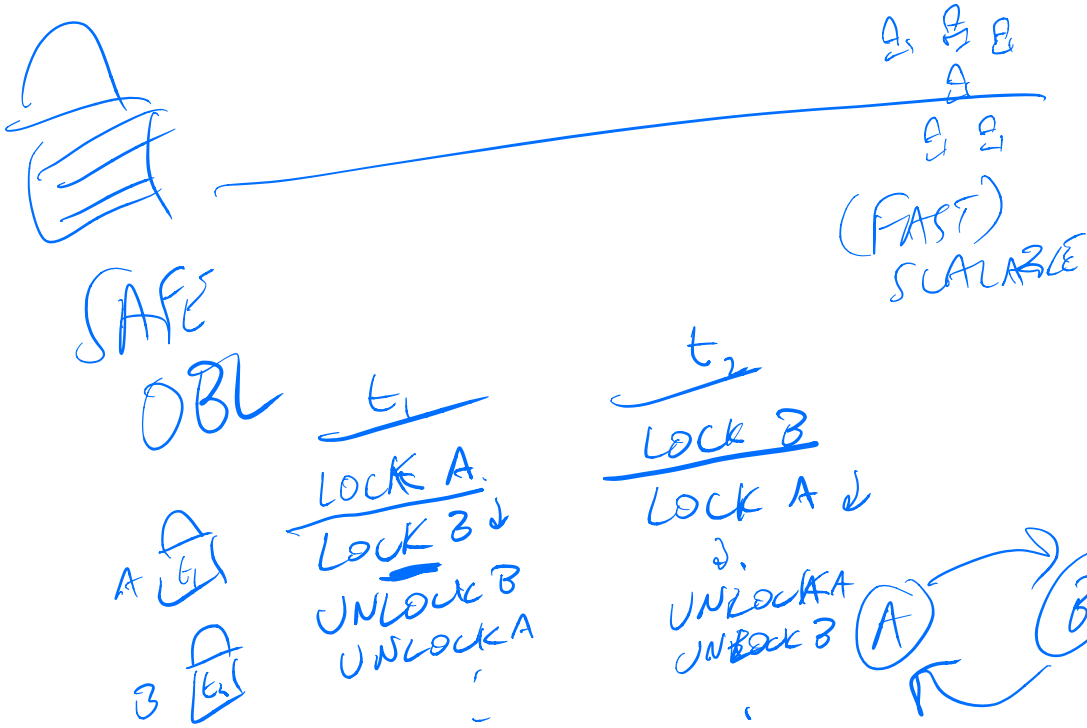
LOCK INC

LOCK TST

LOCK CMPXCHG8B

"atomic instructions"


hand over hand locking
ptr
loop
compare old value to curr
If unchanged
old & new
atomic
blocking locks
lock inc
non-blocking algs.
wait-free algs.
z = x
lock
x = _
unlock
I/O
lock coupling


(FAST)
SCALABLE
SAFE
OBL
t1
LOCK A.
LOCK B ↓
UNLOCK B
UNLOCK A
t2
LOCK B
LOCK A ↓
UNLOCK A
UNLOCK B
A
B

Edsger Dijkstra

mutexes

semaphores

lock test-and-set atomic

"deadly embrace"

deadlock

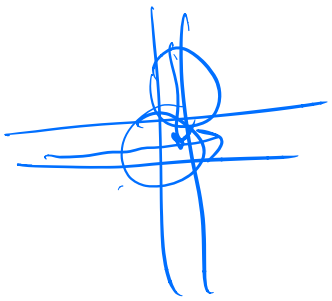
if unlocked (locked == 0)

locked = 1

binary semaphore

proberen te verlagen

Verlagen



deadlock

prevention

recovery

detection

Bakery algorithm

cycle detection

Hoare triple

weakest precondition

semantics

{ } precond

{ } postcond

database

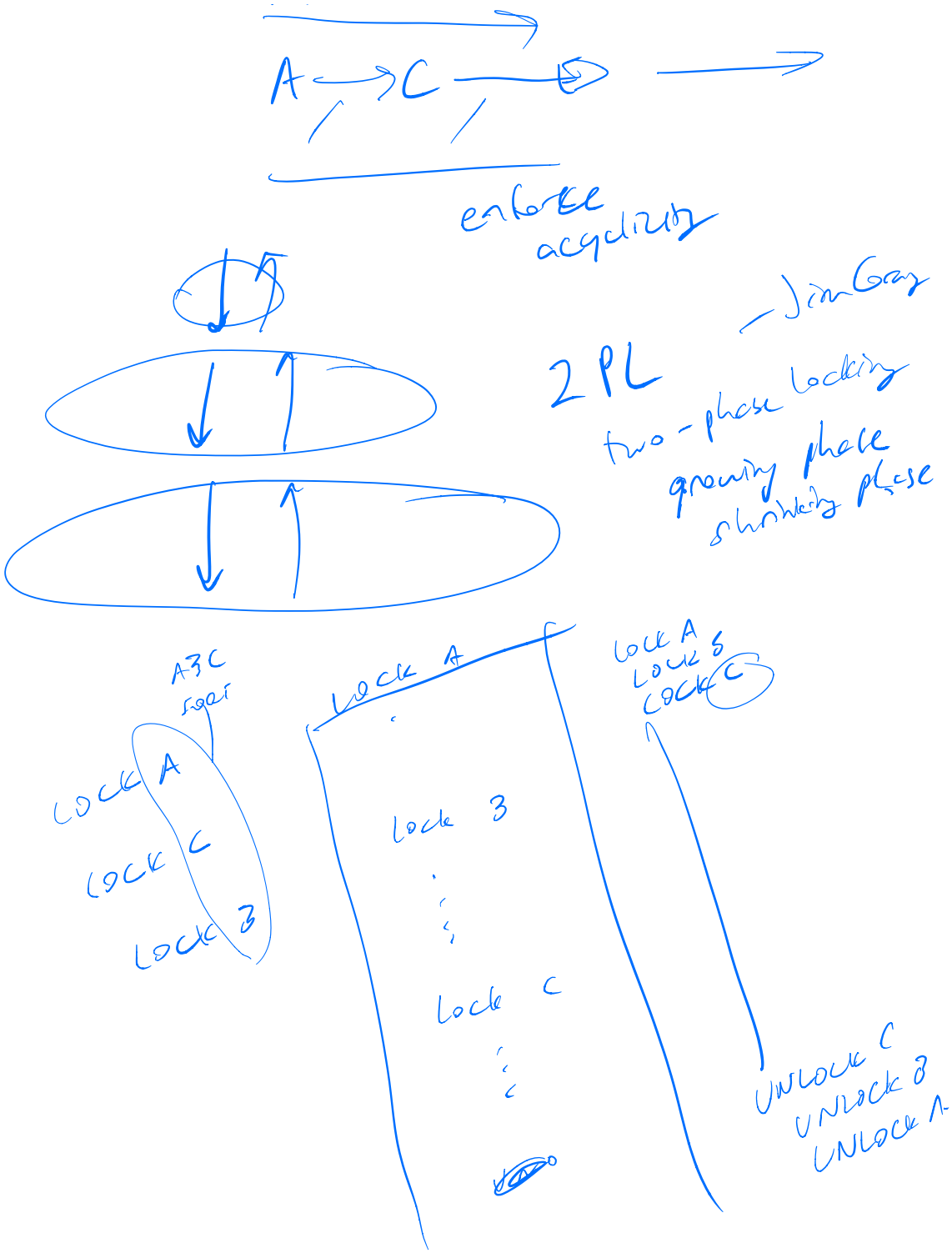
transaction

atomic

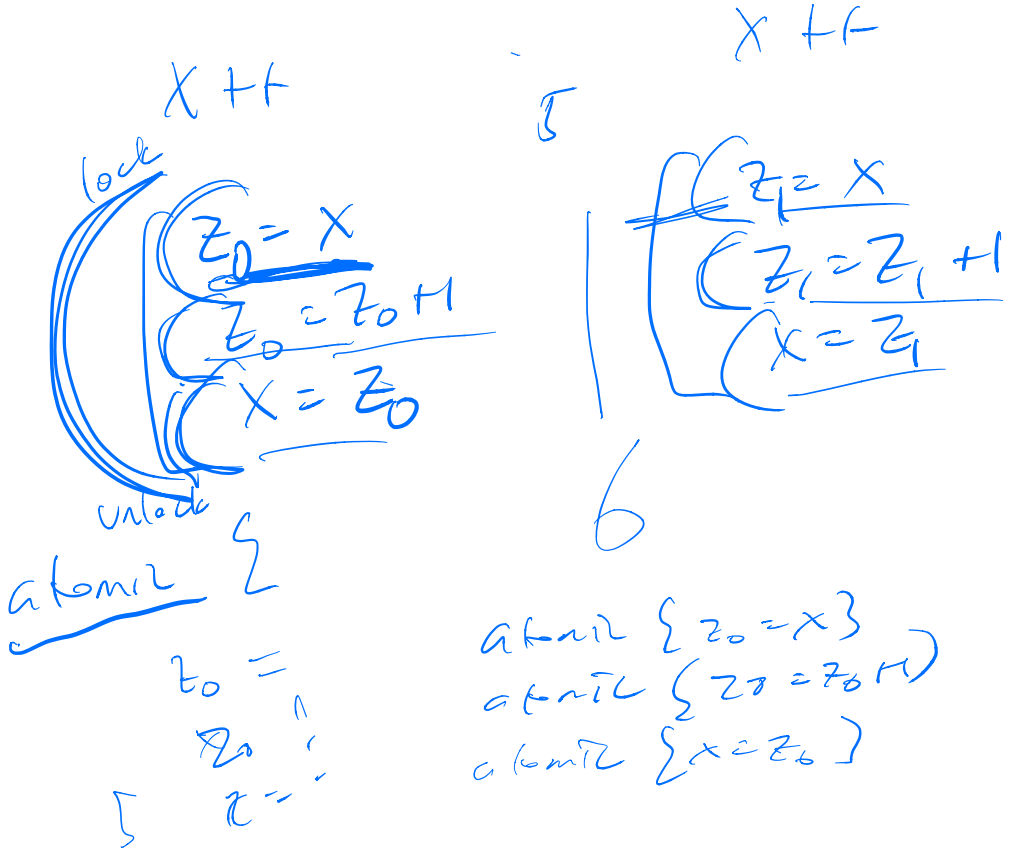
establish

canonical order

A B C D E


A → C
enforce acyclicity
2PL
- Jim Gray
two-phase locking
growing phase
shrinking phase
A → C
legal
LOCK A
LOCK C
LOCK B
LOCK A
LOCK B
LOCK C
Lock B
Lock C
UNLOCK C
UNLOCK B
UNLOCK A

race free



atomicity violation

"The Specification problem"

spec ? where from? spec what are they good for?

spec ⊂ impl.

total correctness

partial correctness

eg. impl. doesn't deadlock

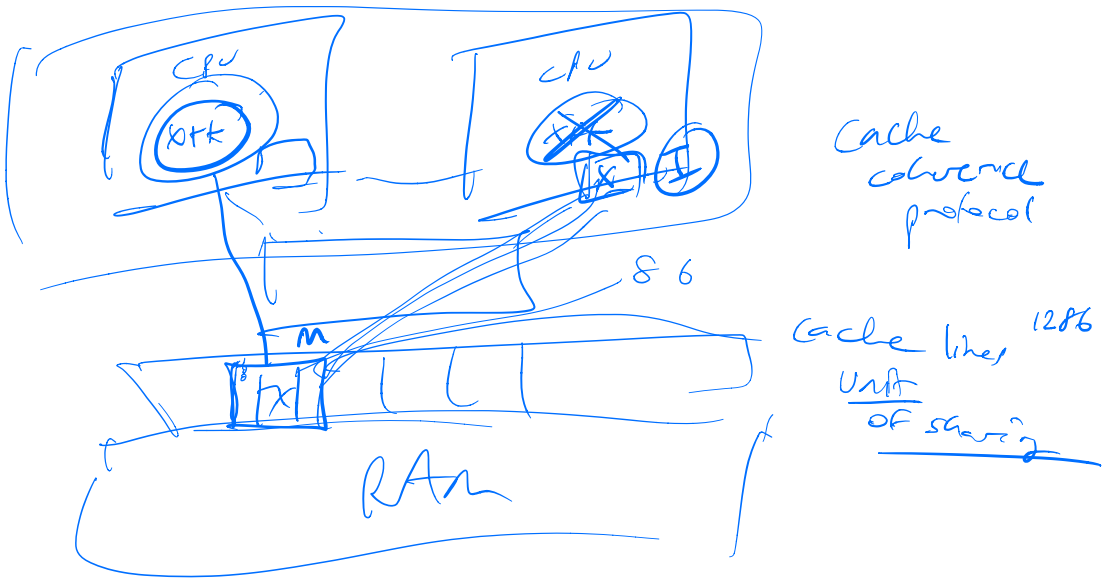
assertions pre and post cond.

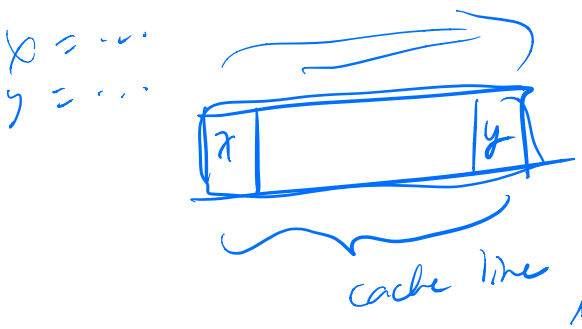
Ed Clarke
Allen Emerson
} model checking

Concurrency issues

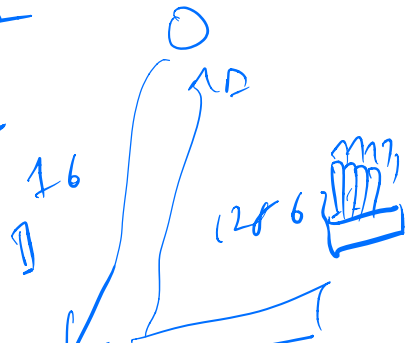
deadlock
overly coarse locks
⇒ serial execution

atomicity errors
races data races
unordered synch (locks)





mov r_1 → addr
mov addr$_1$ → r_2

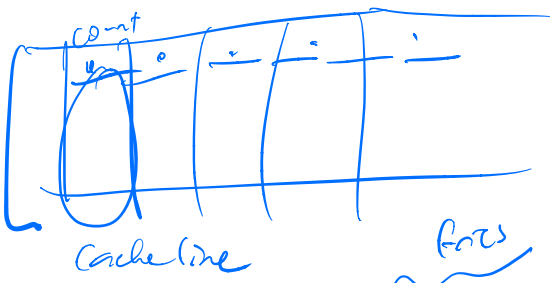


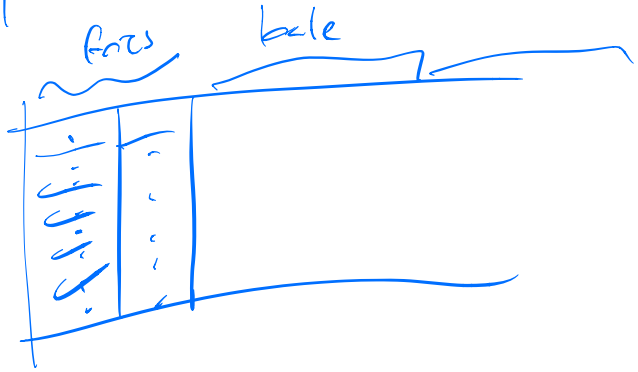
VE

RAM

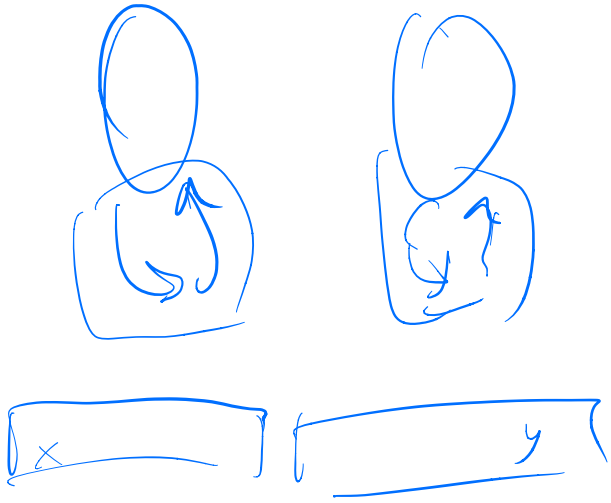
arrays
structs
classes

spatial locality
(speculation, banking)

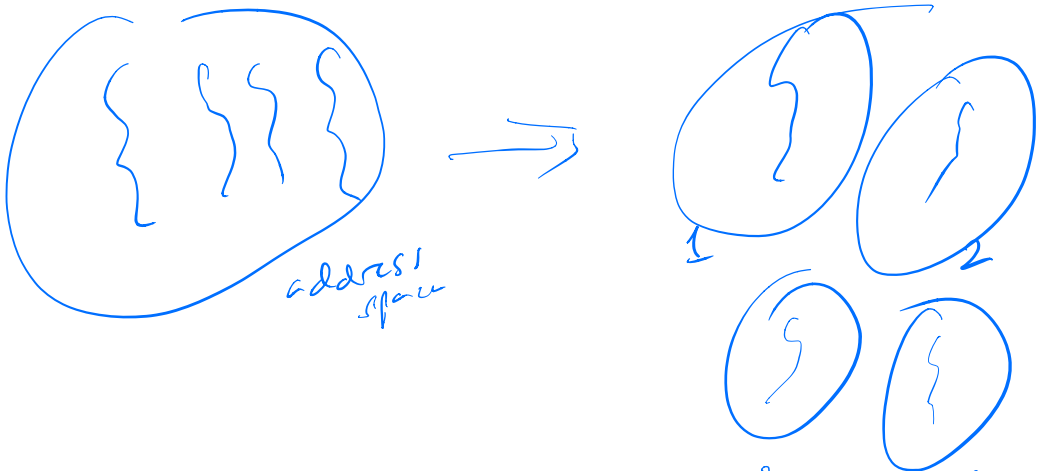


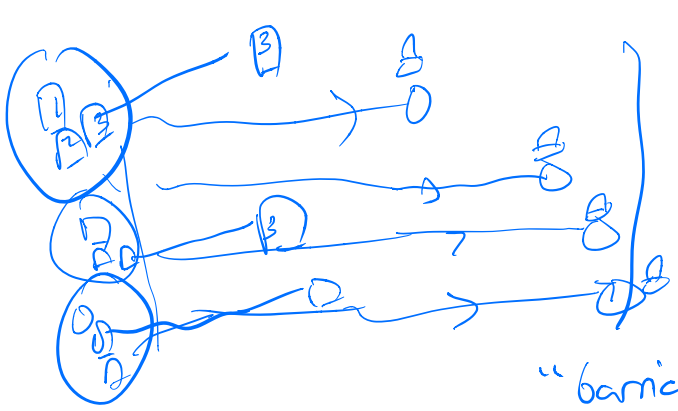


amortizes
- execution time overhead (fetching)
- space consumption



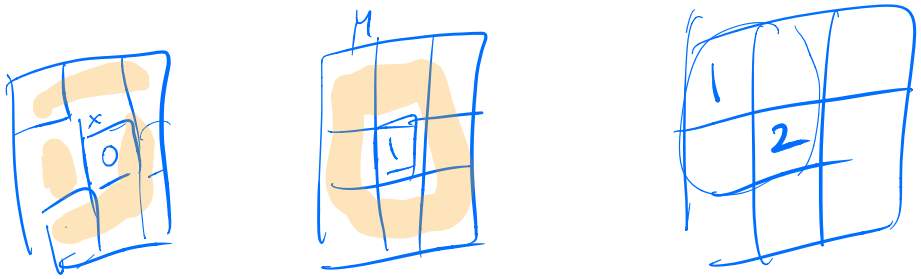
"false" sharing

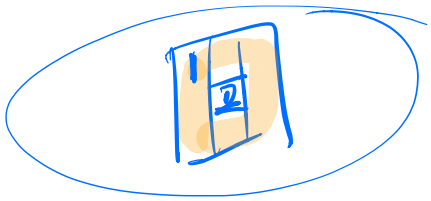




t_1 $X=1$

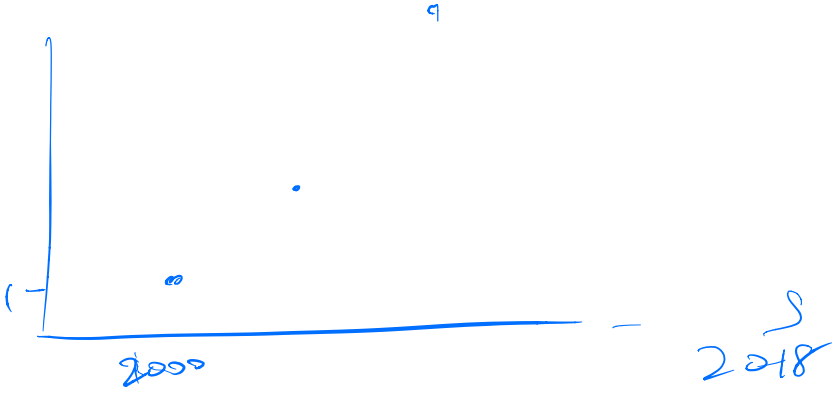
t_2 $X=2$ $Y=1$

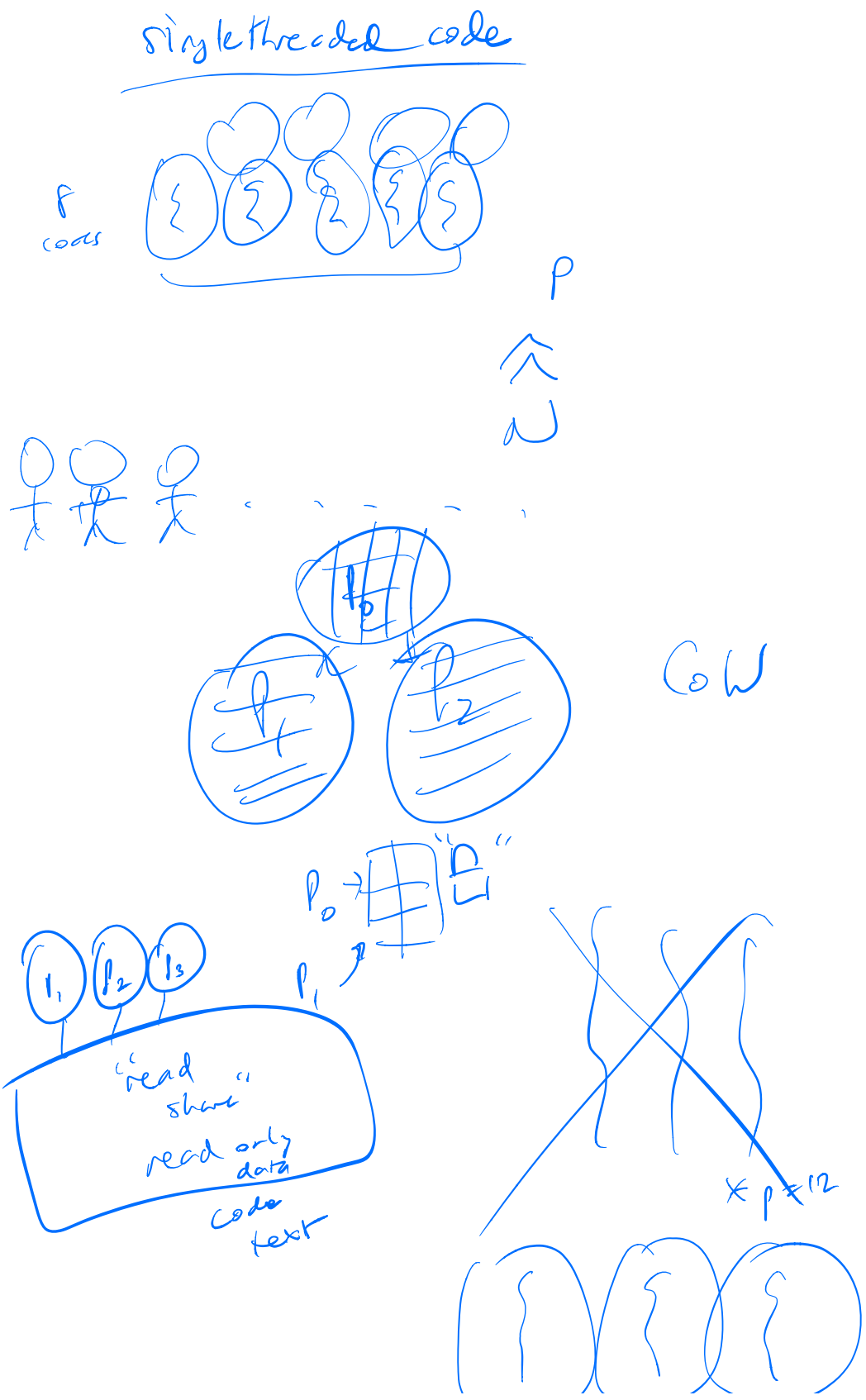


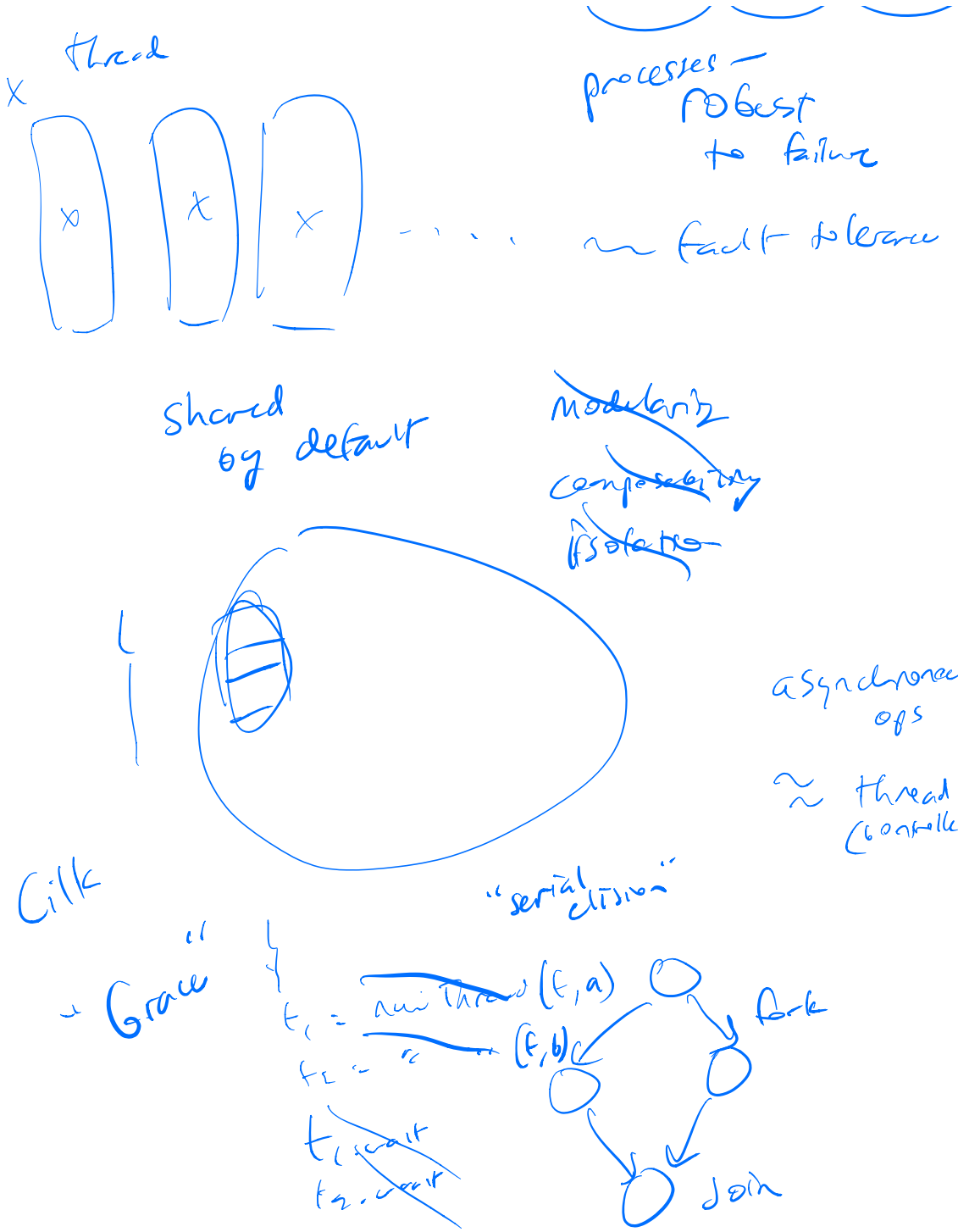


2 threads — isolation, determinism & commit order

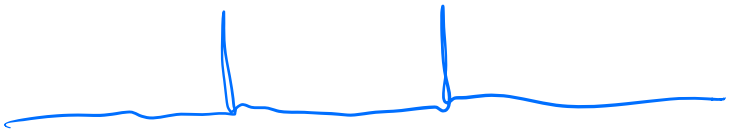
"determinism"

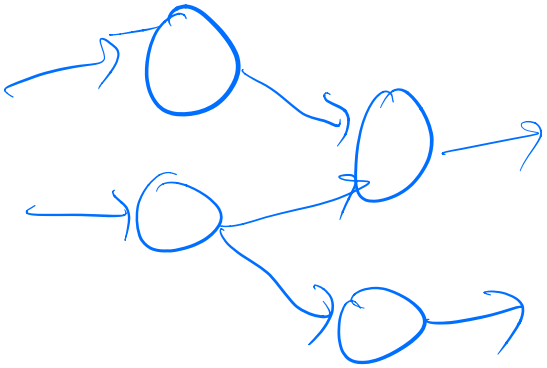


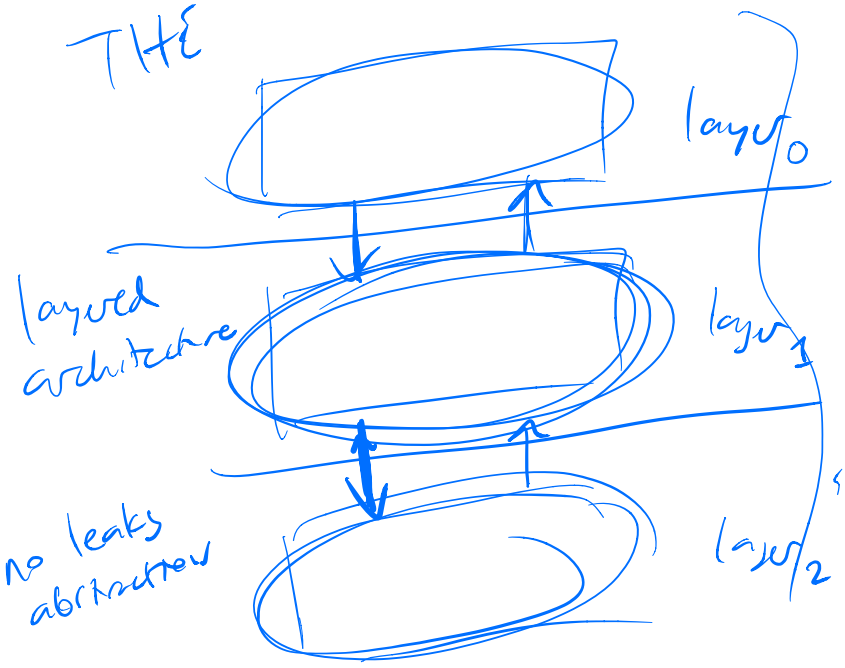

single threaded code
8 cores
p
CoW
"read shared"
read only data
code text
*p=12

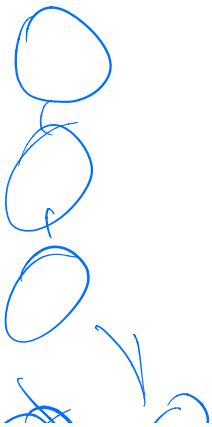

thread
processes —
robust
to failure
fault tolerance
shared
by default
modularity
composability
isolation
asynchronous
ops
≈ thread
(controlled)
Cilk
"serial elision"
fork
join
Map Reduce
SQL
limited expressiveness
"easy"
more effective

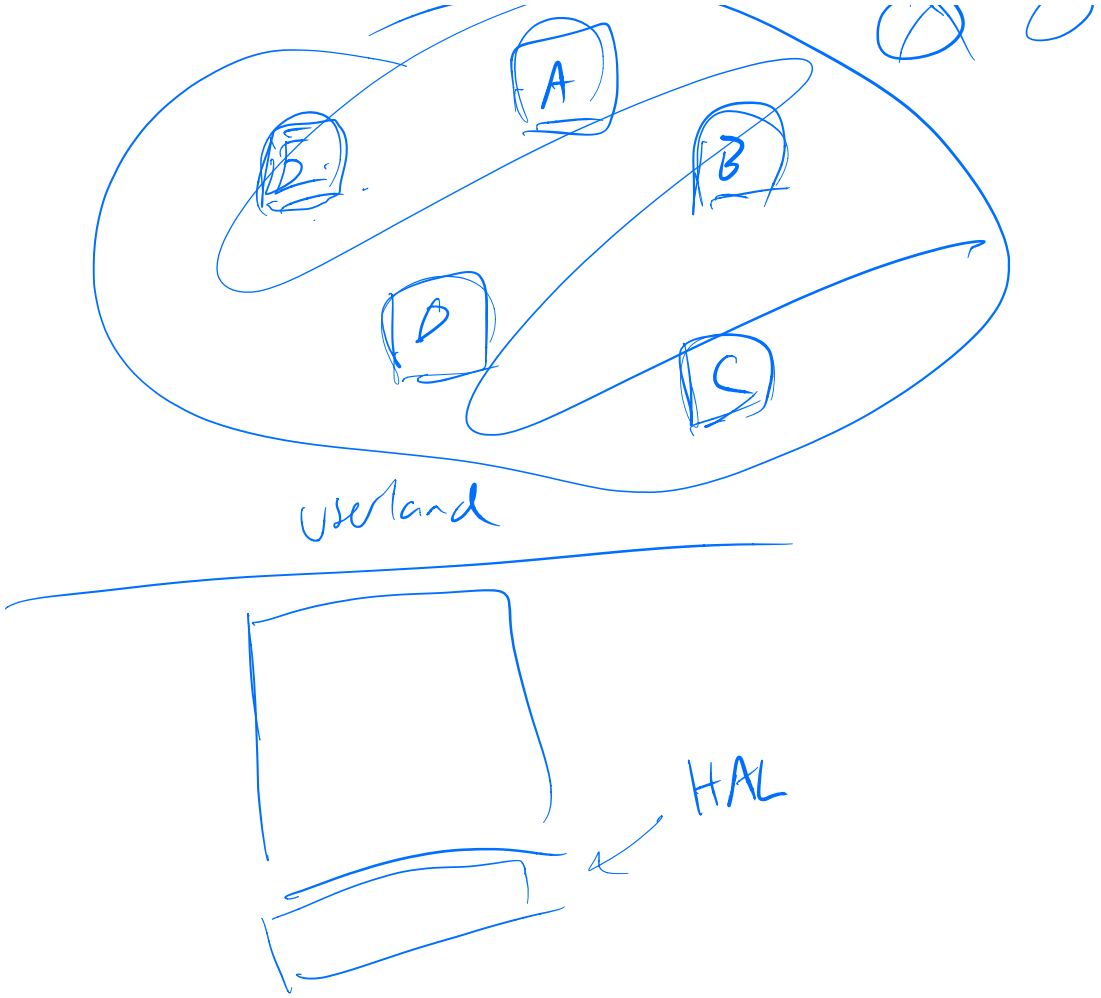
Fault tolerance

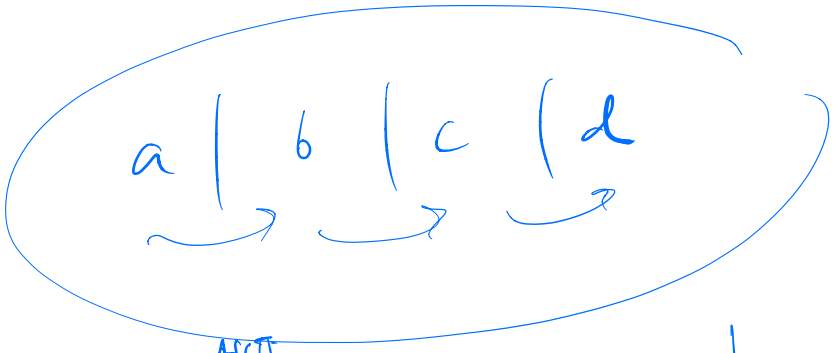


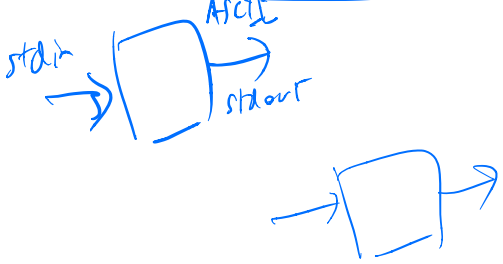




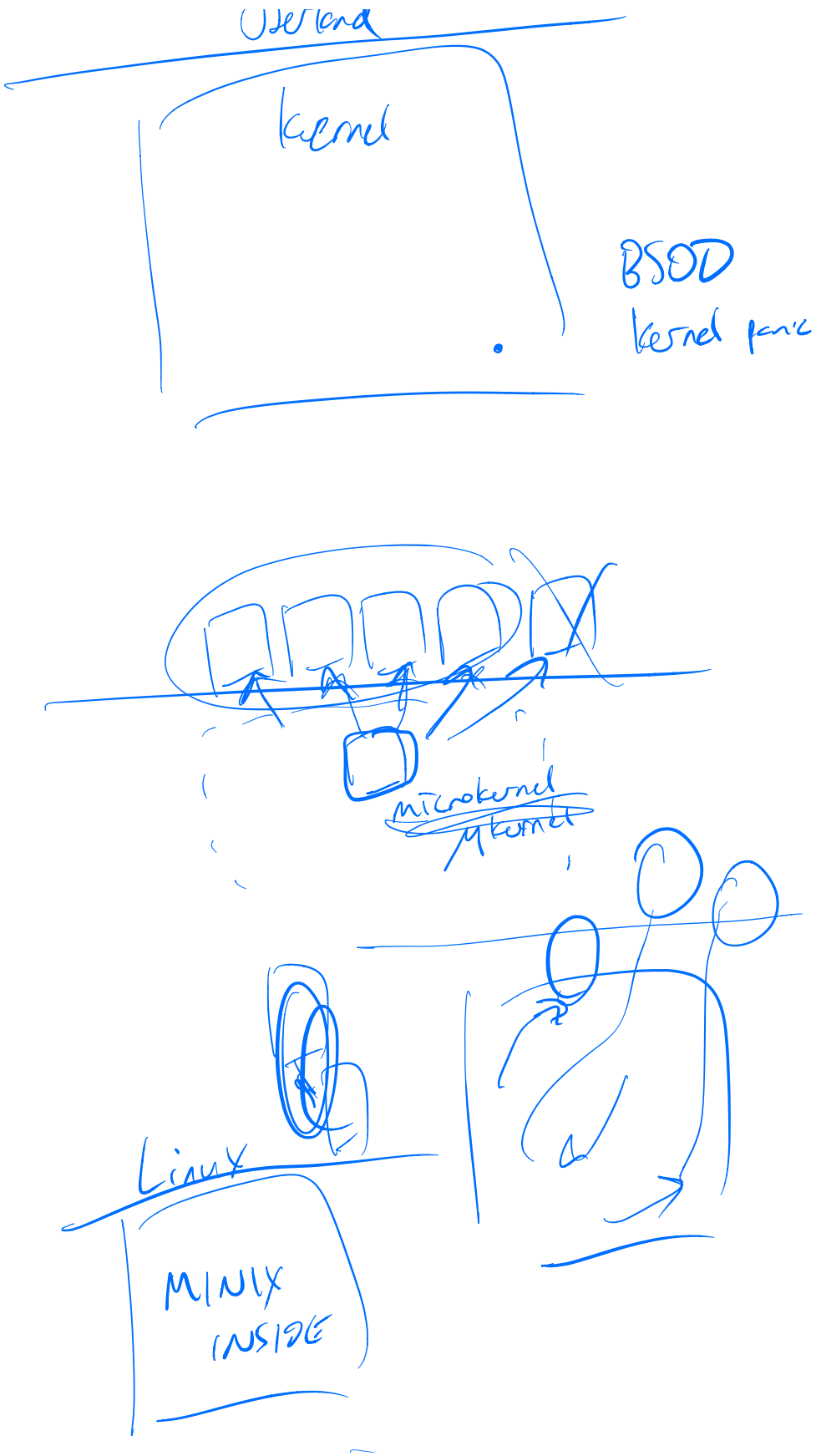


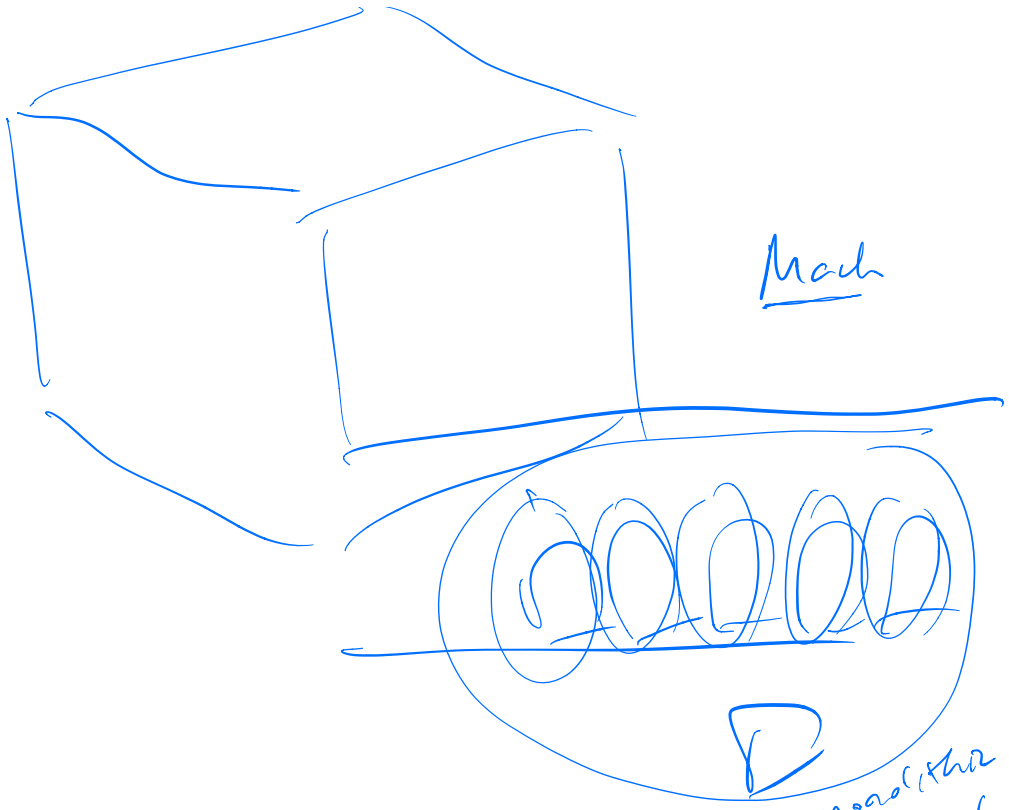






cat f | grep Grades | sort | uniq | wc -l


Userland
kernel
BSOD
kernel panic
microkernel
Linux
MINIX
INSIDE



"Worse is Better"

"PL"

MIT ~~Stanford~~ style Scheme MULTICS

"OS" NJ style Unix & C Common Lisp

"worse"

better for adoption → long-term survival

simplicity "cleanliness" clean elegant "purity"
interface
correctness
consistency
completeness
APIs — e.g. functional vs. state modifying

simplicity interface & implementation
correctness
consistency
completeness — meh

The Cathedral & the Bazaar

NOT a scientific paper ~ essay

"R" - diamond-like jewels

big complex system

LISP 1960

GenGC 1985

Self 1990s

JVM V8 2000s

dynamic compilation → JITs

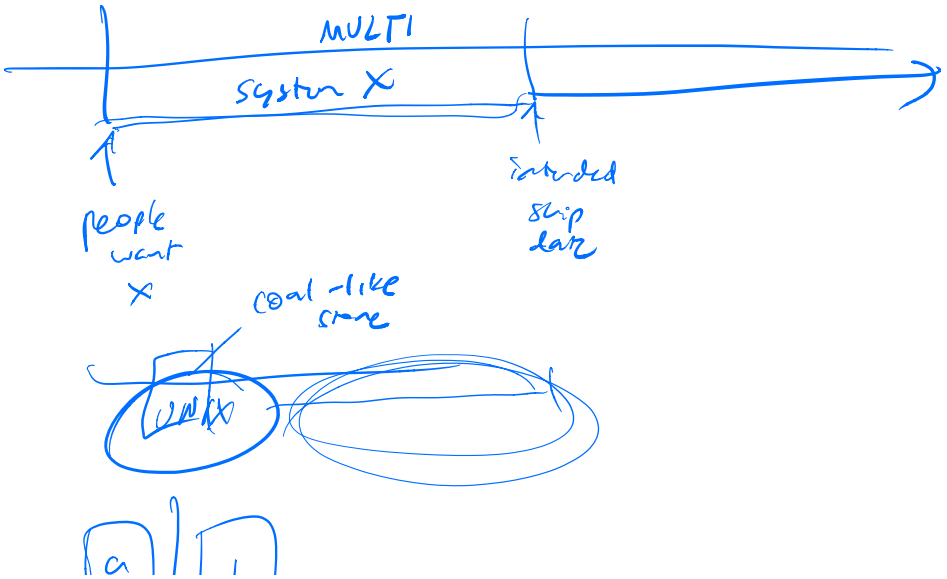
concurrent & // GCs

C

x++
++x
⇒ translated to PDP/11 instruction set

~1972 "portable assembly"

FORTRAN ~1957 — 1992

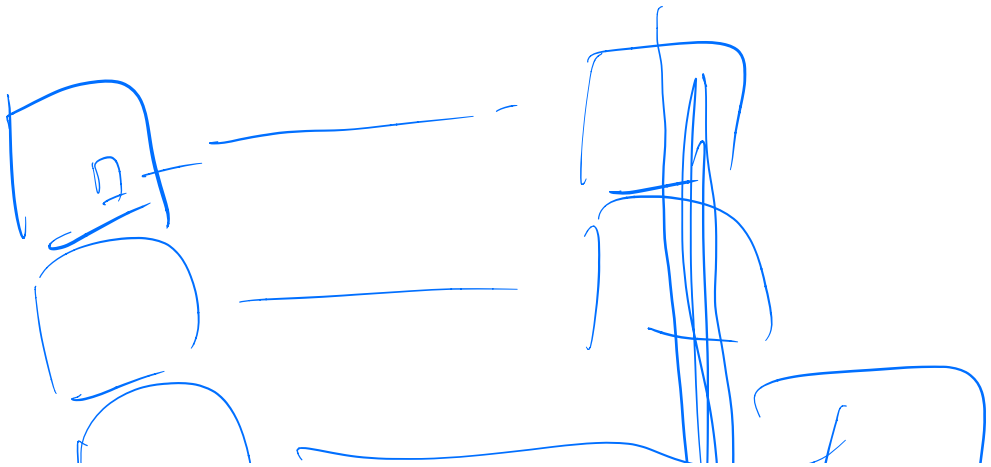


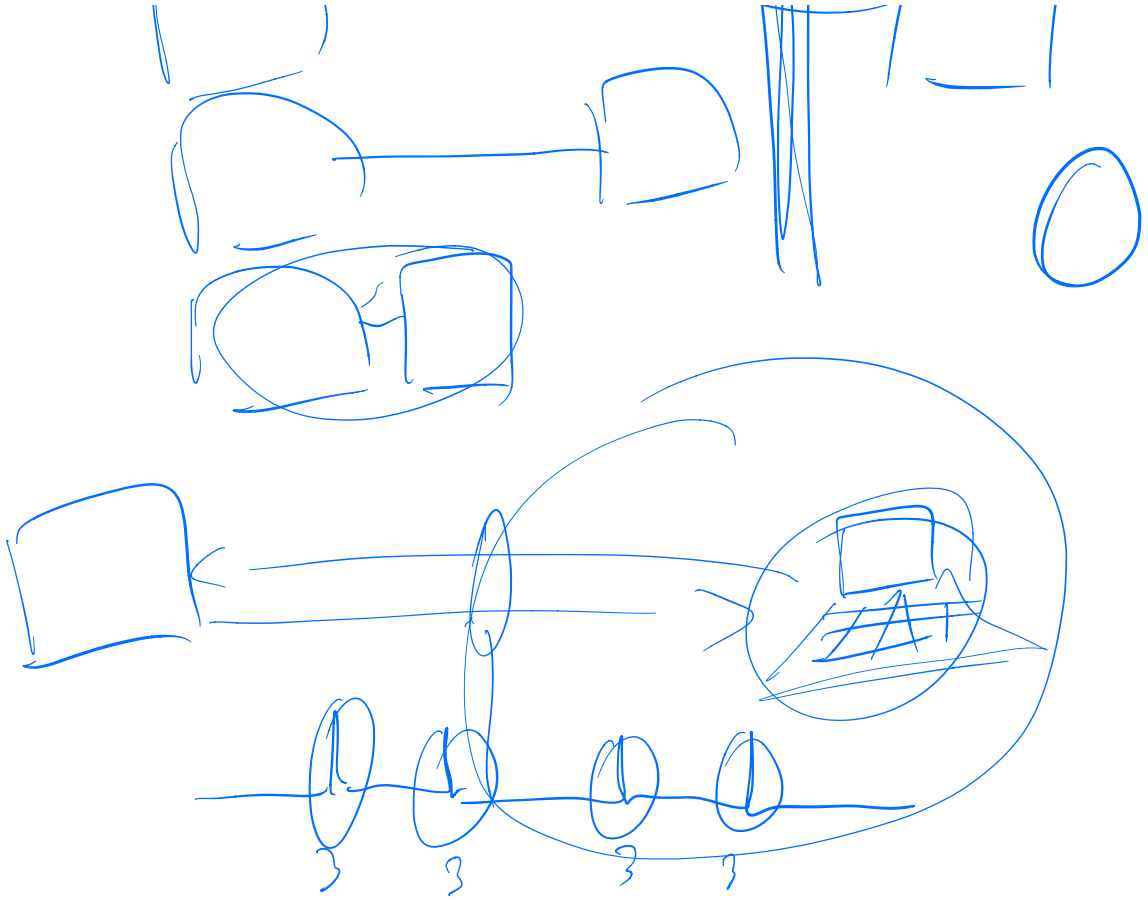
Communication channels → print, send network, display etc.

~~Covert~~ overt

Covert channels — implicit communication

timing channel

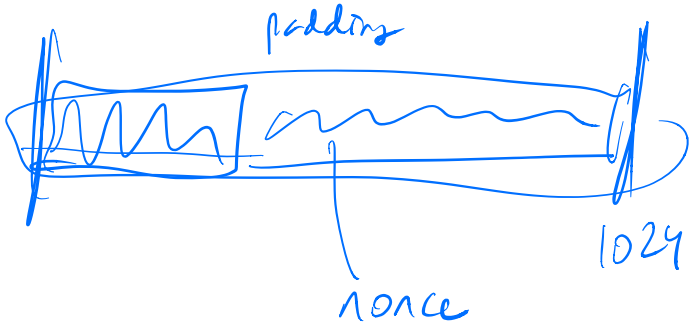




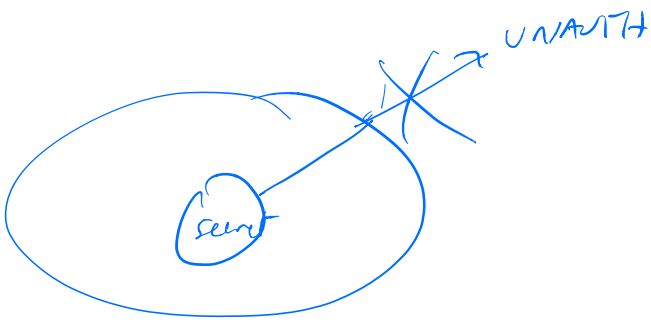
exposing raw time

⇒ introducing random noise

batching / make take same amount of time



Information flow



int secret x = password;

"tainted"

Perl.

int s = x;
int q = s + 12;
print q;

never print if type is secret

overt
covert

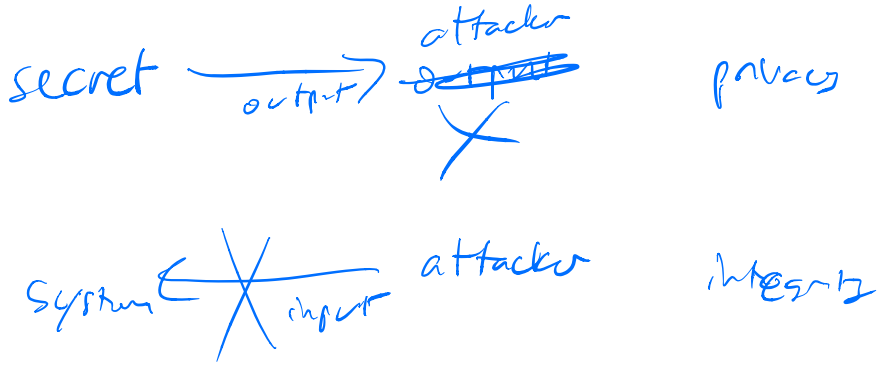
taint mode

s = readsocket(...); taint input
system(s);

— taint propagated by
copying
& assignment

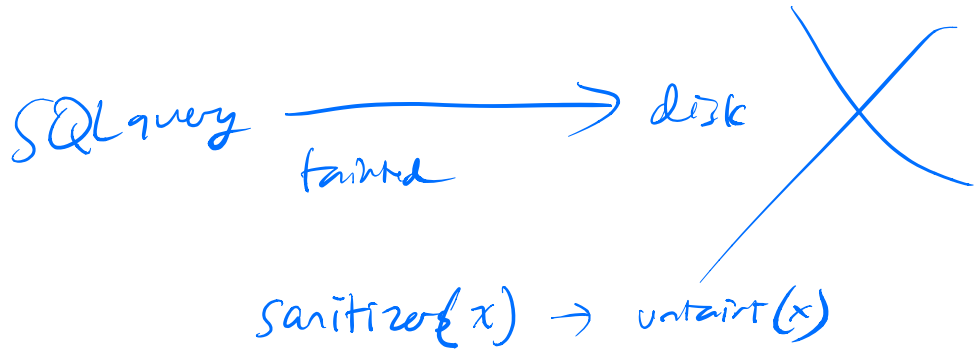
1 ← intent

conservative
guarantees either privacy or integrity

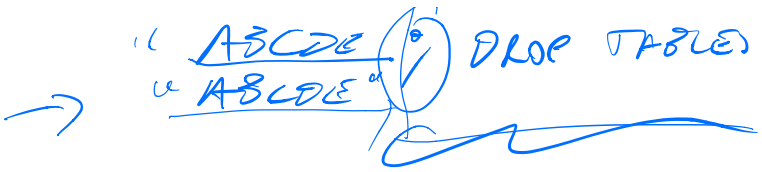


x = a secret
y = ... x
z = ... y
⋮

taint explosion



sanitize(x) → untaint(x)



dual of sanitizers – declassification
declassifier

declassify(tainted input path) ⇒ (redacted)

domain-specific

JIF Andrew Myers
JavaInfo. flow

taint mode in Perl
– dynamic IF analysis

JIF – static IF analysis

QIF quantitative info. flow
McCamant & Ernst ~2000s
– 30s (info flow)

Shannon bit

```
int a[N];
if (x[i] == secret[i]) {
    a[i] = 1
} else {
    a[i] = 0
}
```

control dependency ← taint

implicit information flow

print a

copying & assignment — data dependency

timing channels

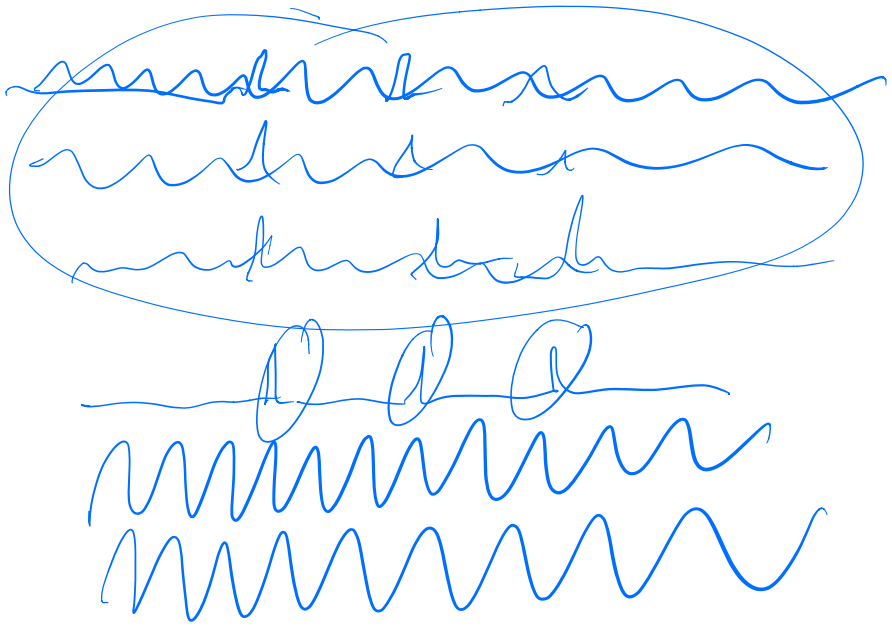
if ( ) {

} else {

}

timing attack on RSA

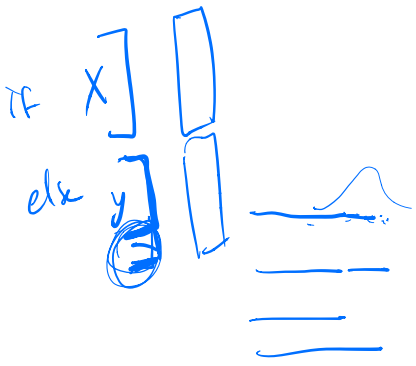
exfiltrate private key by observing timing of branches



Fort Knox vs. Window locks approach
perfect good enough

"raise the bar"

buffer overflows

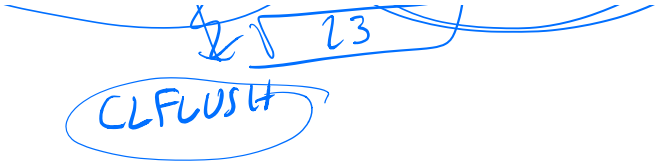


threat model
Who the attacker is
What power they have

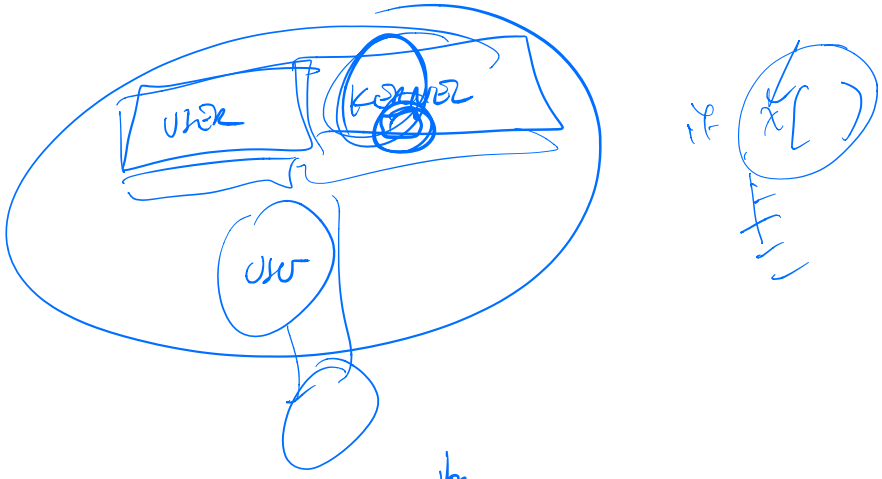
~~rubberhose crypto~~
physical access
supply chain
rootkits
hypervisors
port knocking
~~social engineering~~
~~phish~~
insider attack
disgruntled employee

Meltdown
Spectre

FLUSH & RELOAD
EVICT & RELOAD

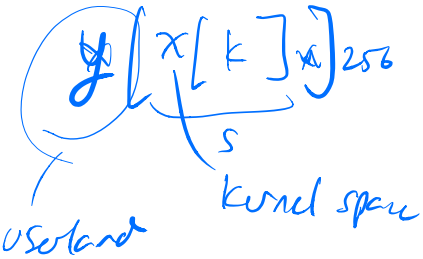


Speculative execution — caching



KSLR
kernel space layout randomization

KAISR



Spectre

Google — ($ ads, ... cloud services) index the web (& more) — Android "hw vendor"
DoubleClick, YouTube

Amazon — stock everything & ship it (physical) + search
$ cloud services, + video ...

Microsoft — desktop apps → cloud "hw vendor"

Facebook — social net ($ ads)
Instagram, WhatsApp

Apple — hw & sw ("cloud services" iTunes / app store, Apple Music, video, iPhone)

Twitter — social net ($ ads)

Samsung

Netflix — subscription model

IBM services & cloud — "walled garden"

impressions vs. clicks

multiarmed bandit

single armed bandit

$E[\text{payoff}] = \sum p(\text{payoff}_i) \times \text{payoff amt}$

exploration vs. exploitation

cloud services

("outsourcing") pay as you go model
avoids capital expenditure

eliminates config/admin overhead
eliminates hosting A/C physical space electricity

elastic

redundancy / fault tolerance) replication

security

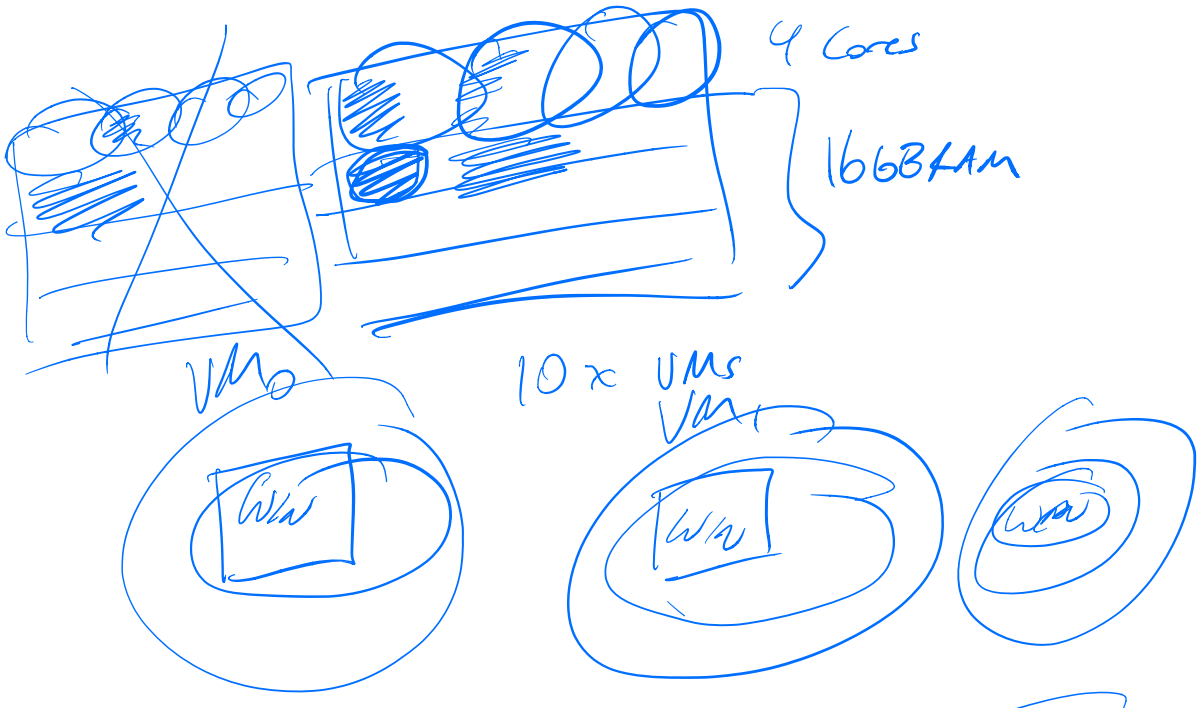
economy of scale

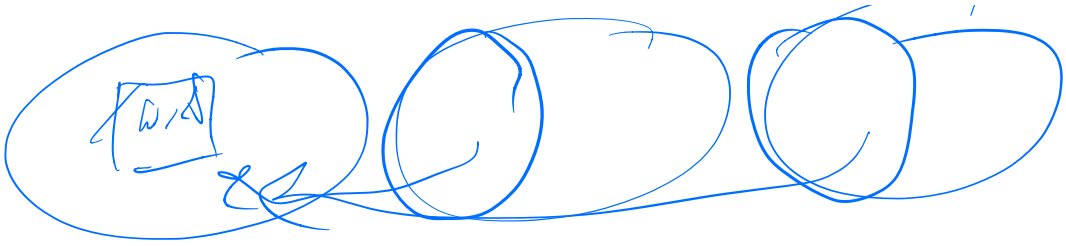
cheap electricity
hw
real estate

Collocation

commoditization

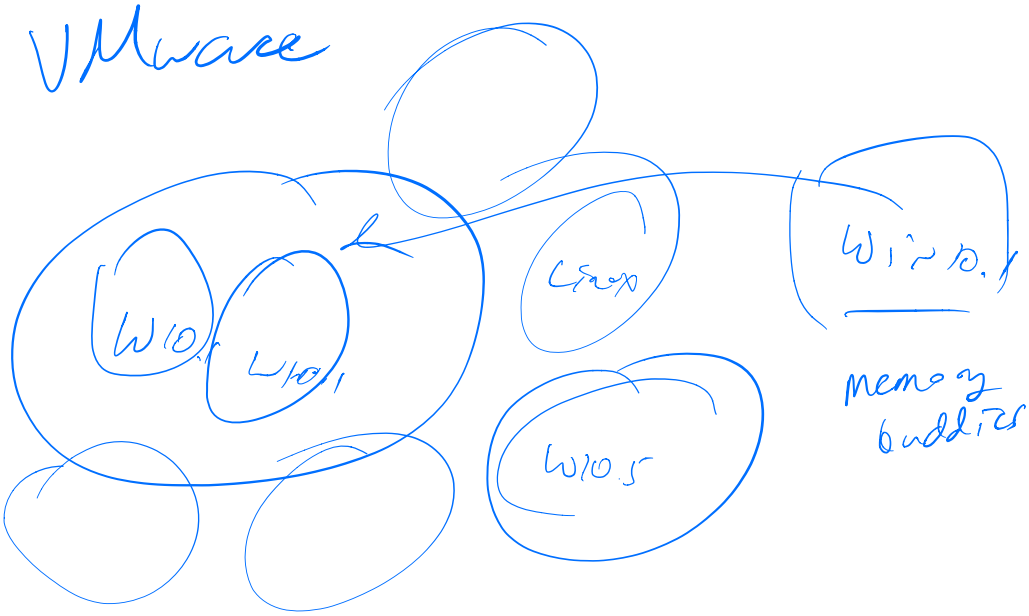
Google TPU TensorFlow
MS Project Catapult FPGA

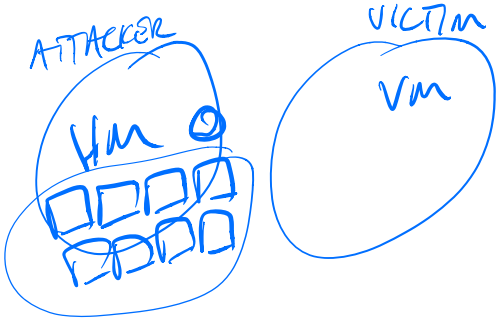




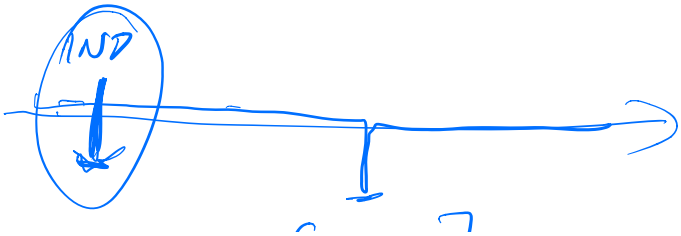
Xen

VMware





FLUSH & RELOAD



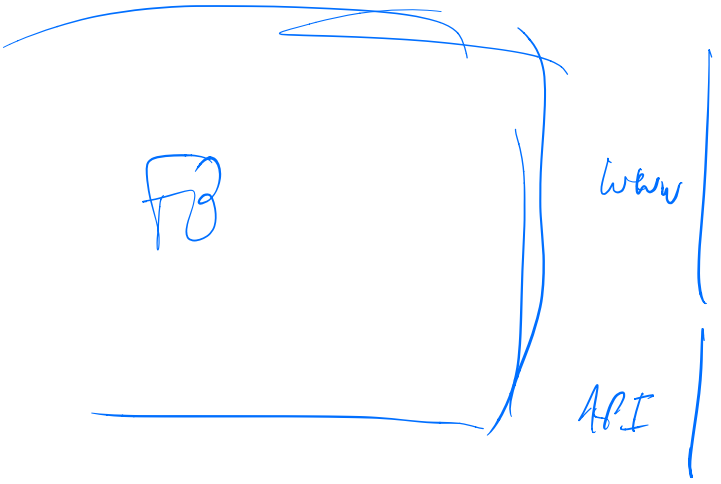
x [ ]

[access kernel memory KAISR

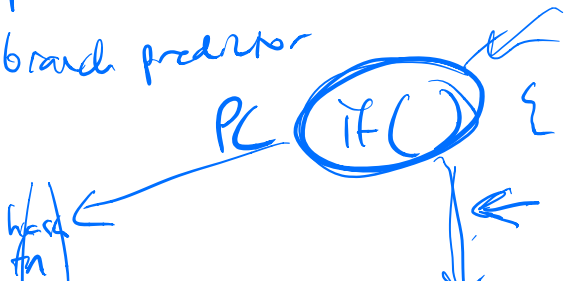
; de shift) speculated

MySQL ~> ~15%

Meltdown
Spectre) fundamental threats to business model

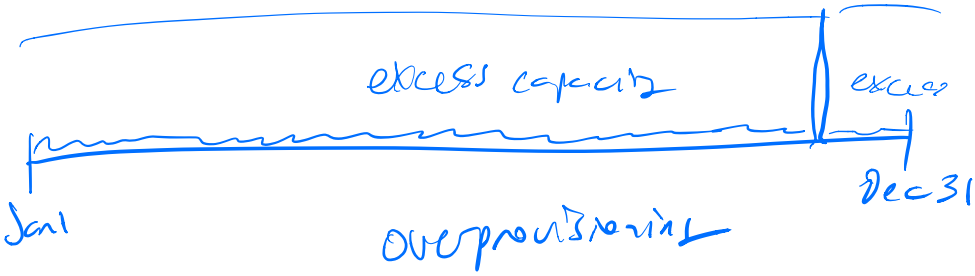


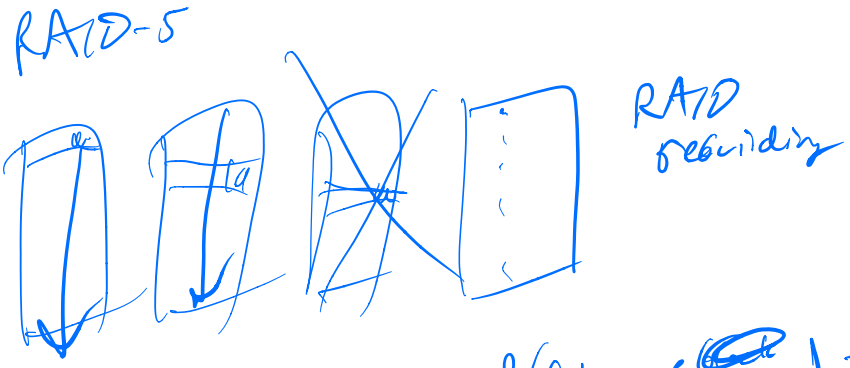
Spectre
speculative execution
branch predictor



IBRS ?

OceanStore



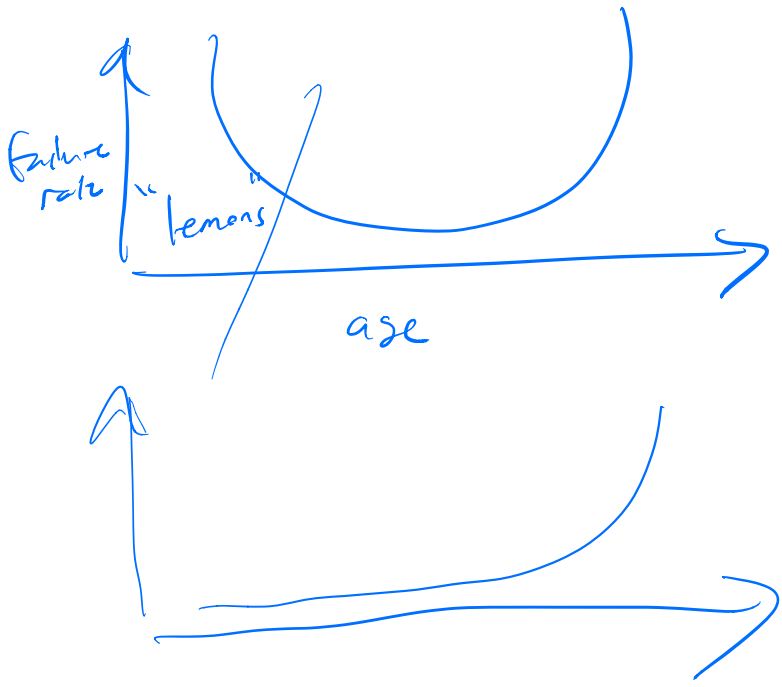


P(failure and data wasted) = $\frac{1}{100}$

trinodular redundancy

(3 copies)

P no failures $1 - \left(1 - \frac{1}{10^{20}}\right)^3$



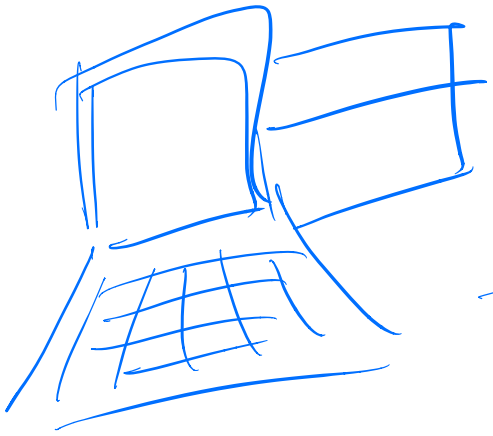
bathtub curve

Fault Tolerance – Karthik

Security (applied crypto) – Adam

Distr. Sys – Heather Miller

Fault tolerance



- power outage (UPS)
- hardware failure
  - hard drive
    - mechanical
    - electrical
  - (disk controller – electric sensors)
- software errors
  - kernel panic / BSOD
  - server – SEGFAULT

fail-stop

- hardware defects

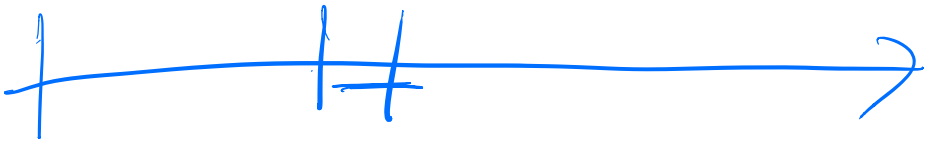
MS Word / Chrome

Software fault tolerance

Checkpoint-based recovery

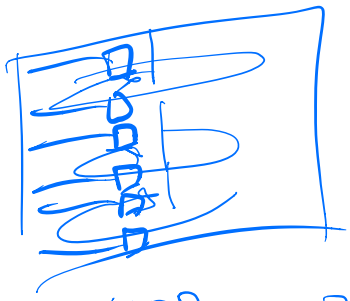
~ 10 min

"Snapshot"
= checkpoint



5 min

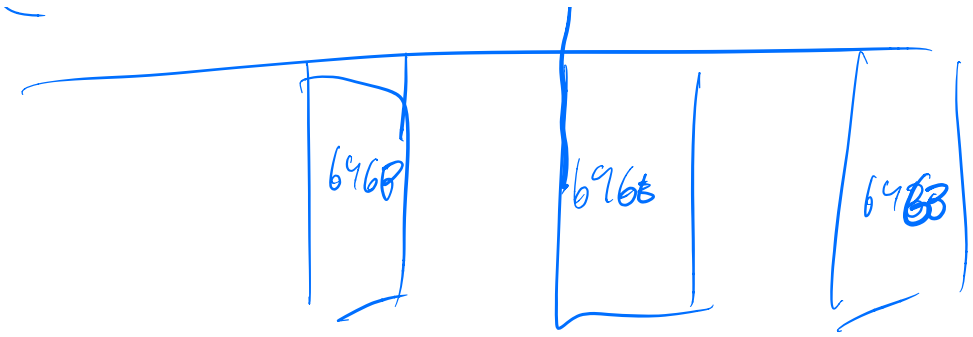
Coasistat snapshot

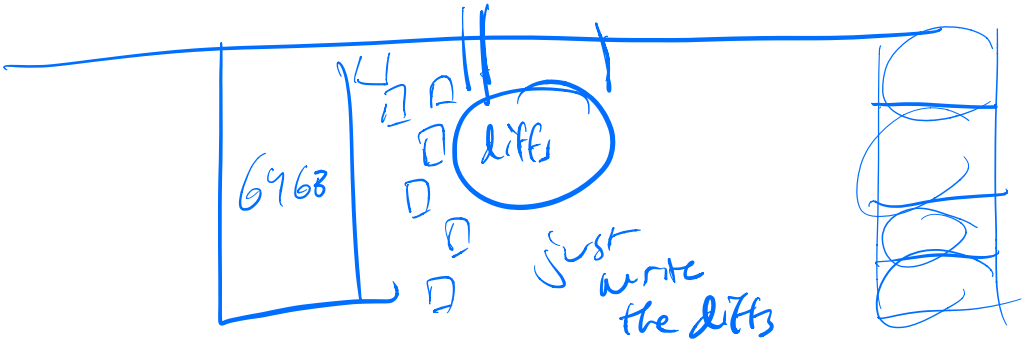


parallel writes

HDDs → high r/w

(replication)





can be utilized

SPoF

single point of failure

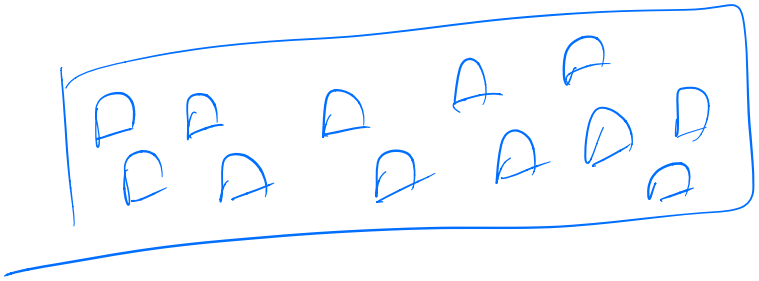
availability %

90% 99% ...

"5 9's"

99.999%

5 min downtime / year



$\frac{1}{10000}$

1

10 odds all are up

$\left(1 - \frac{1}{10000}\right)$ $\left(1 - \frac{1}{p}\right)$ chance one machine is up

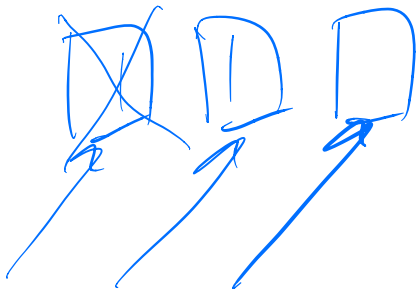
$\left(1 - \frac{1}{p}\right)^n$ all n are up

10 : $\frac{1}{1000}$

100 : $\frac{1}{100}$

1000 : $\frac{1}{10}$

replication

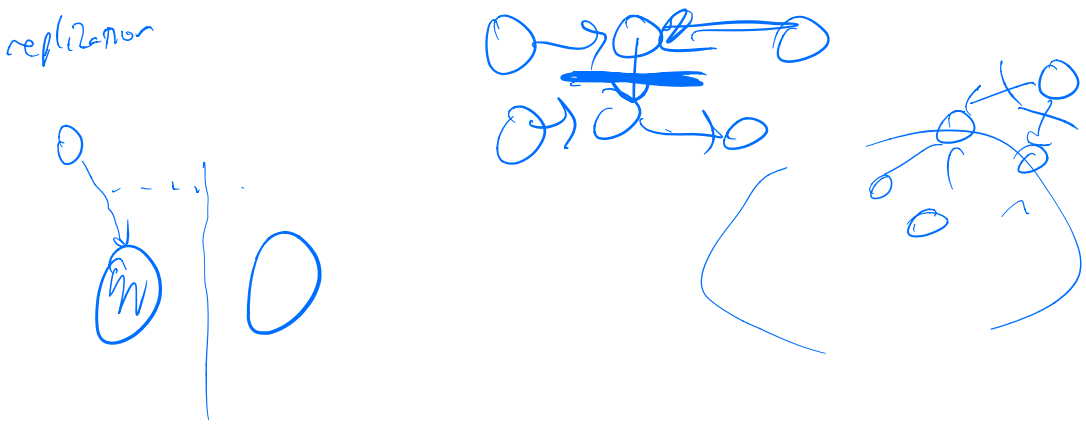


CAP theorem

Eric Brewer

Consistency — replicas in sync
Availability
Partition-tolerance

replication



eventual consistency

immutability — guarantees/facilitates Consistency

RAID

redundant array ~~inexpensive~~ independent disks

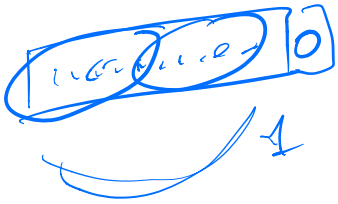
parallelism & fault tolerance

striping

mirroring

coding

"checksums"



silent corruption

ECC

error-correcting codes

correct errors

ROC

recovery oriented computing

"boot faster"

RAID

RISC

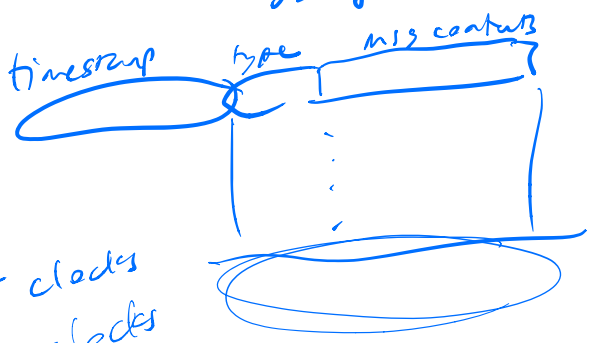
NOW network of workstations

SPUR

RAMCloud

Snapshots (checkpointing)

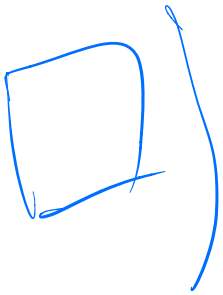
message logging



ordered

Lamport clocks

Vector clocks



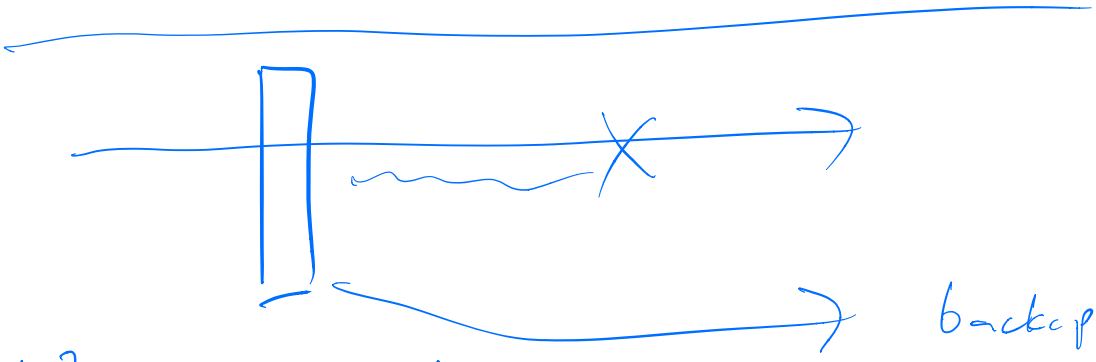
NTP

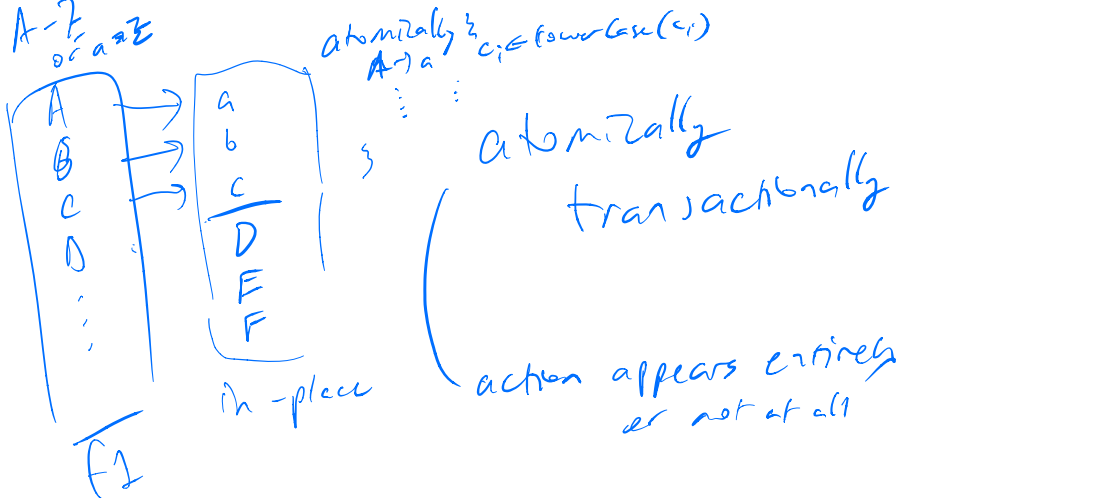
Causal order

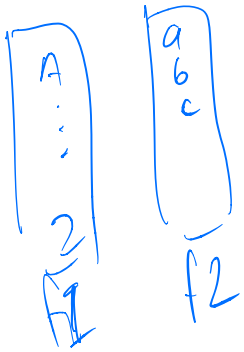
A happens-before B $\Rightarrow L(A) \leq L(B)$

Synchrony

asynchrony





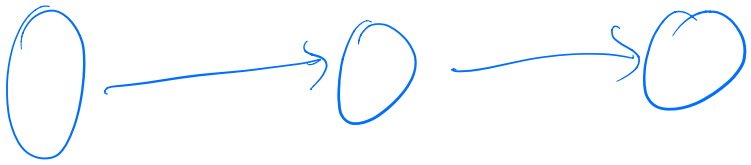


write updates "somewhere else"

ATOMICALLY update

↳ rename f2 to f1

transactions ensure system always consistent



CONSISTENT

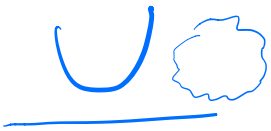
OPEN TRANSACTION
TR. BEGIN
⋮
TR. END

Fault tolerance

RAM ECC
error correcting codes
cosmic rays!

- disk HDDs
manufacturing momentum

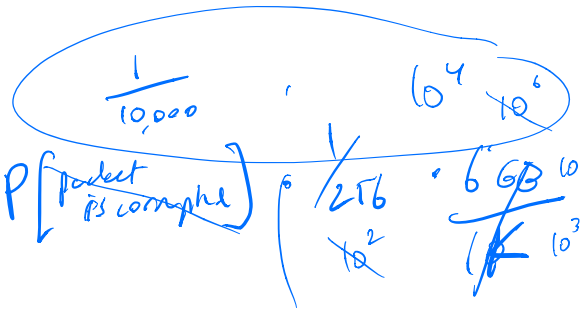
RAID



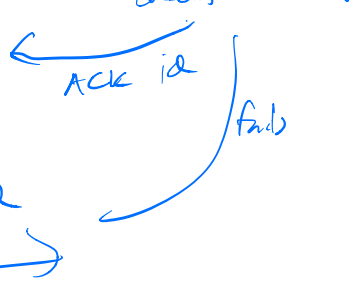
end-to-end argument

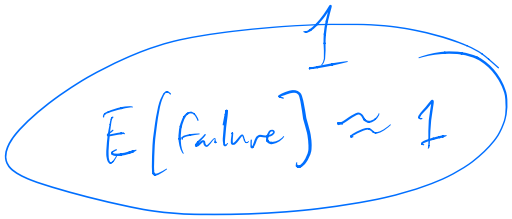
UDP — full buffers → packet lost

TCP/IP
"reliable" delivers in order



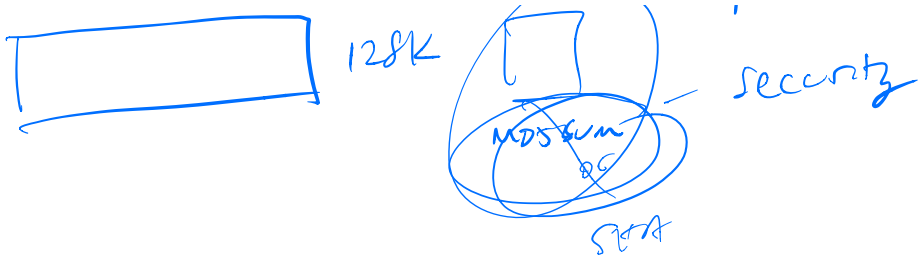
checksum





random faults

— — — reliability



probablyistic

Byzantine Fault Tolerance ≈ 100% accurate (modulo crypto)

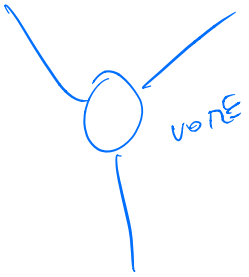
malicious faults

3n + 1

2n + 1

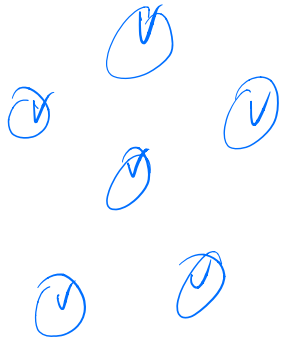
n = 8 generals

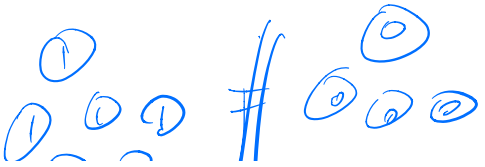
rootkit



Re FLP result — impossibility of (fair) asynchronous consensus

Fisher Lynch Paterson





Synchronous

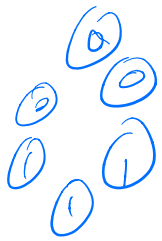
in round 1:

count 0s & 1s

if $0^s > 1^s \Rightarrow 0$

~~$1^s > 0^s \Rightarrow 1$~~

else $\Rightarrow 1$



breaking symmetry

- timeouts
- probabilistic protocols

(1990)

Paxos

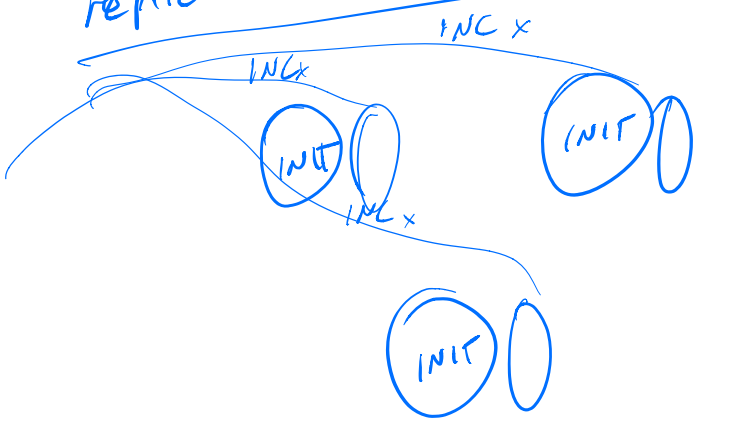
(1998)

Part-Time Parliament

ledger (log)

consistent view (identical) of log

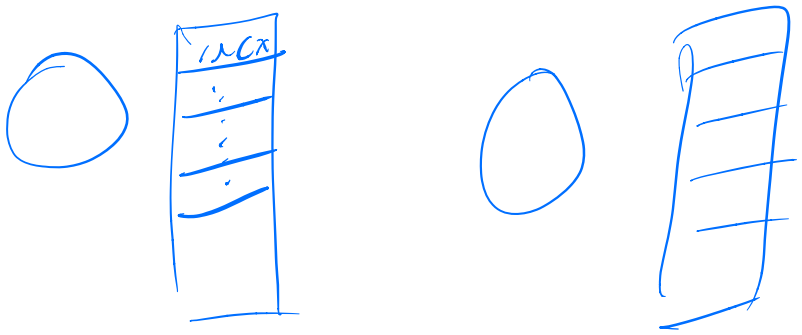
replicated state machines



execute same inputs in order

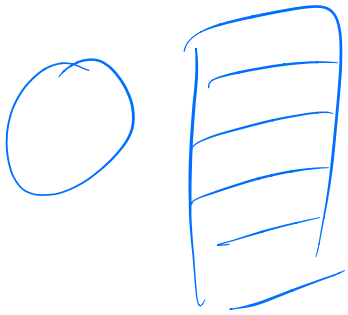
⇩

same state



Identical log

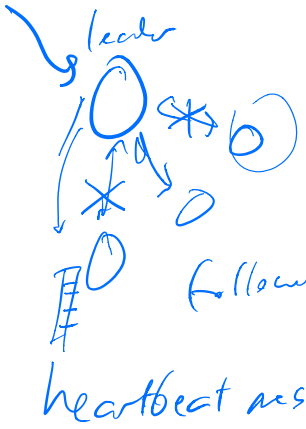
≡ identical state



Paxos Made Simple (2001)

RAMCloud → RAFT

- leader election
- log replication
- safety



RPCs
& copying
log
consistent

heartbeat messages

2PC
two-phase
commit

Amazon TLA+

Coq

gonna let breakfast talk breakfast

RAID ECC
SISC RAID

FILE
SYSTEMS

journaling
= logging

---

information flow

secret → don't let secrets out

← malicious

don't let malicious input in

x = secret
print x ← explicit info flow leak (data flow)

declassify

if (secret == "foo")
print A
else
print B

implicit (control flow)
control dep. on classified data

---

x = read from internet
System(x)

"rm -rf /"
--no-preserve-root

Sanitize

$s = secret

if ($s[i] == 0) { 10 ns

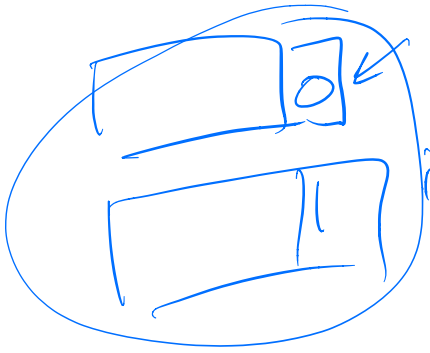
} else { 1 ns

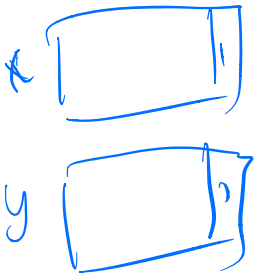
}

covert channel

X implicit flow

Perl dynamic taint analysis





$x = read_from_internet()

$y = $x

system($z)

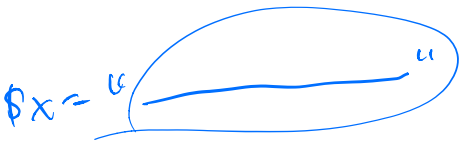
if tainted($z) then fail

SELECT * ... → DROP TABLES

/([^;]*)/ OR

$x ~ = / / -> 0
Untrusted

dyn. info flow
+ reasonably efficient
- no implicit flow
- "taint explosion"



len($x)

quantitative information flow } Stephen Chong, McCamant, Mike Ernst, Mike Hicks

1 bit vs. 256 chr. | UMd

static info flow

+ efficient
+ implicit flow
- taint explosion
- new language (Jif) — Andrew Myers
- error messages ...

---

replicated log ⇝ replicated ledger

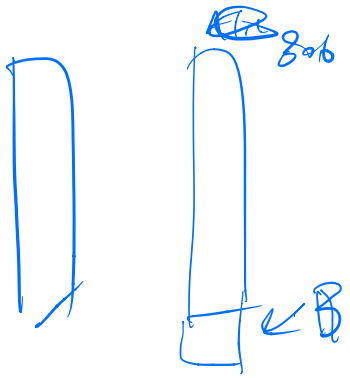
Bitcoin = distributed consensus

zero-day vuln

"wallets"

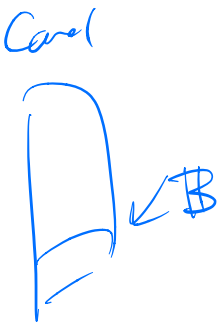
liquidity

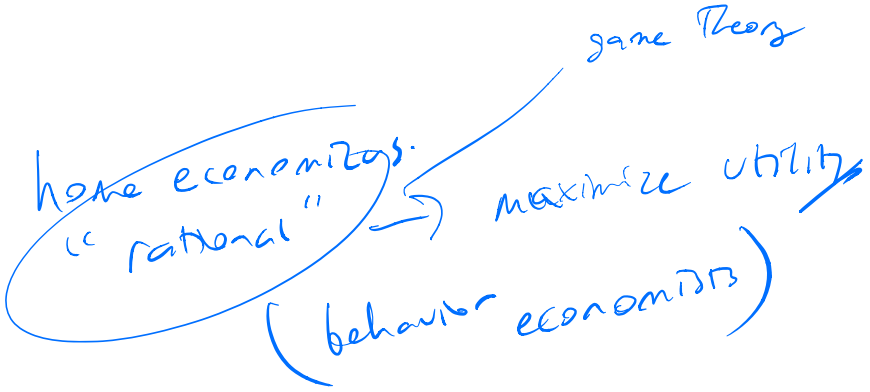
internet + money = crime

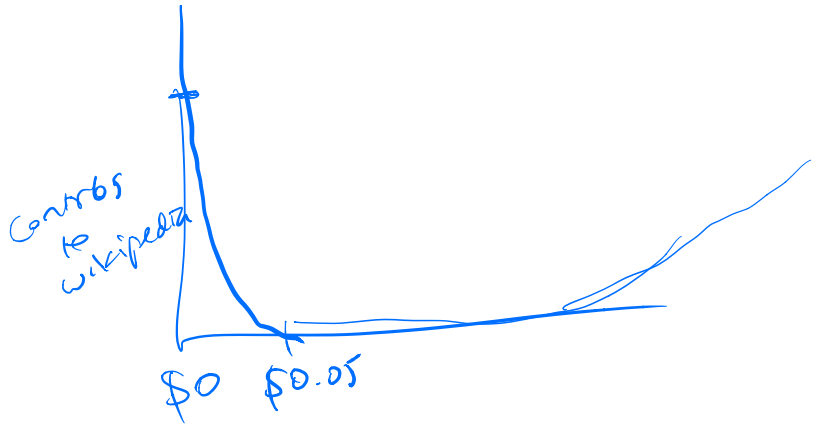


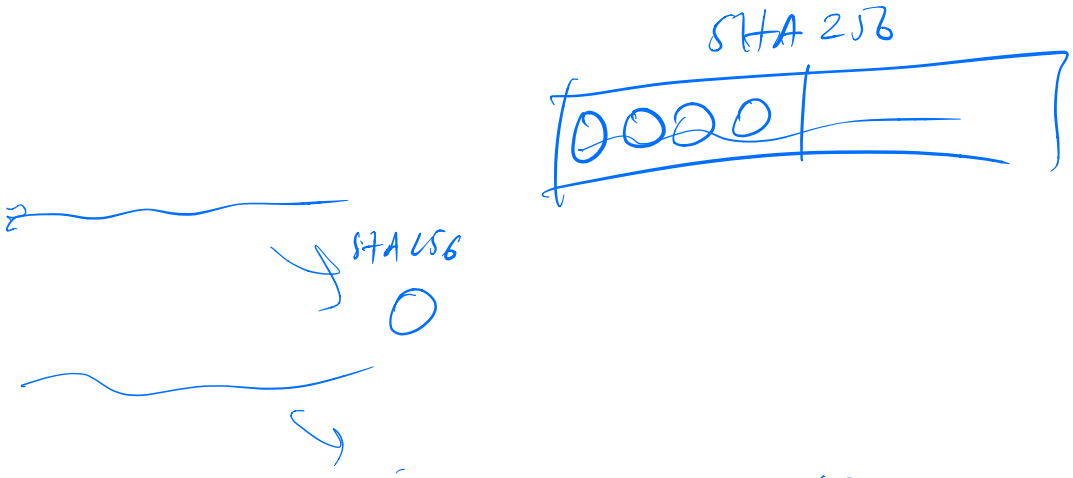
double spending

Alice → $0b → Carol





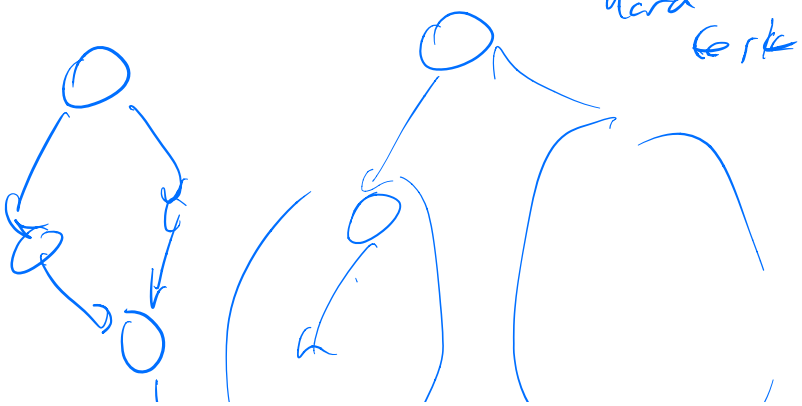




CPU → GPU → ASICs → China

less power
more efficient

mining pools / selfish mining attack



TOLTTOU

file system race

time of change to time of use

incentive-participation ₿ (mining)

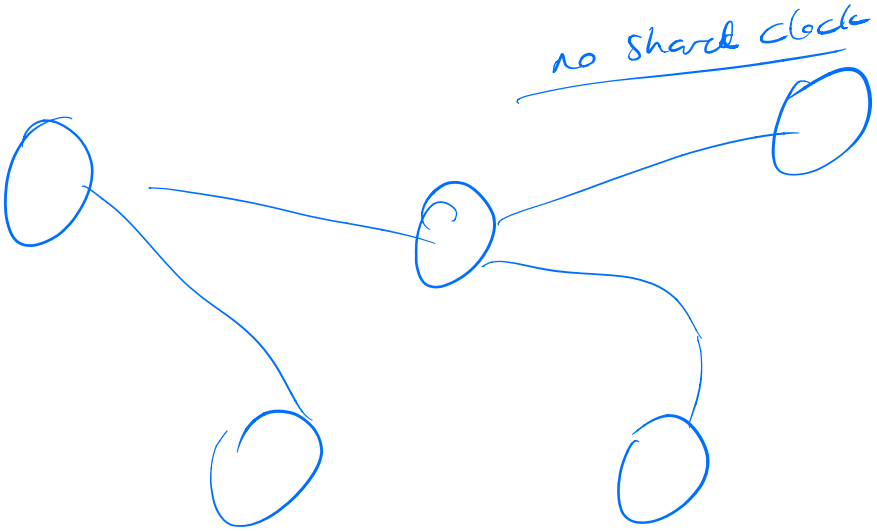
Currency — universal exchangeability

Yap stone of value

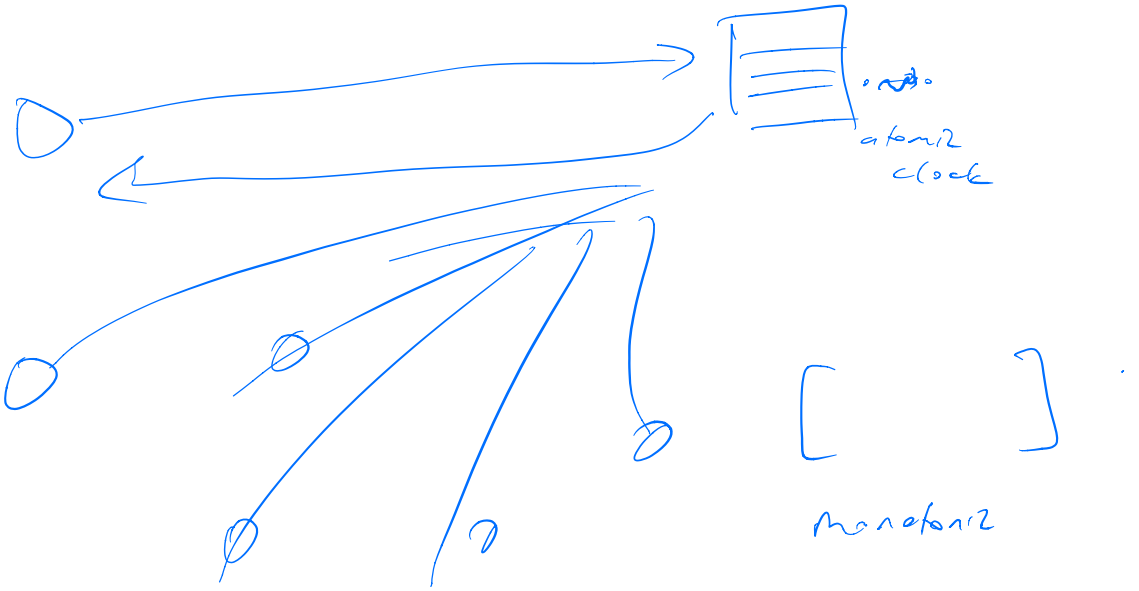
Ponzi scheme pyramid "

Lamport

clocks
notion of time

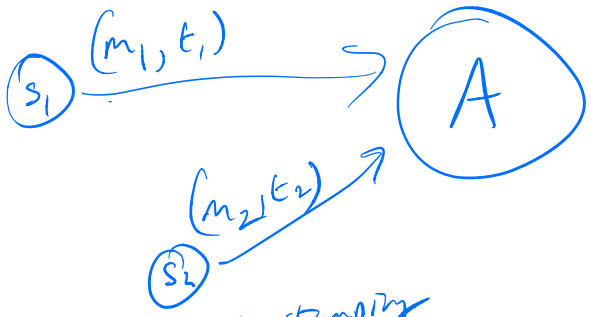


$3 \cdot 10^8$ m/s



monotonic

loose synchrony



possible!

$t_1 < t_2$

BUT

time(m_1) > time(m_2)

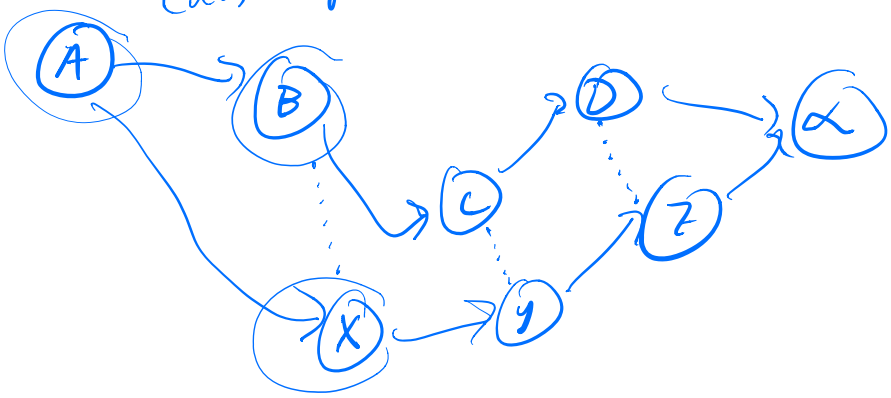
happens-before

"time"

sender Action

Relativity

Causality



$A < X$

A happens before X

$A < B$

$B < C$

$C < D$

$X < Y$

$Y < Z$

$D < E$

$Z < E$

no happens-before for D, Z

partial order

numbers have total order

$\alpha < \beta$

$>$

$=$

$\not\to\beta$: $\alpha \not\to \beta$

$\alpha \not< \beta$

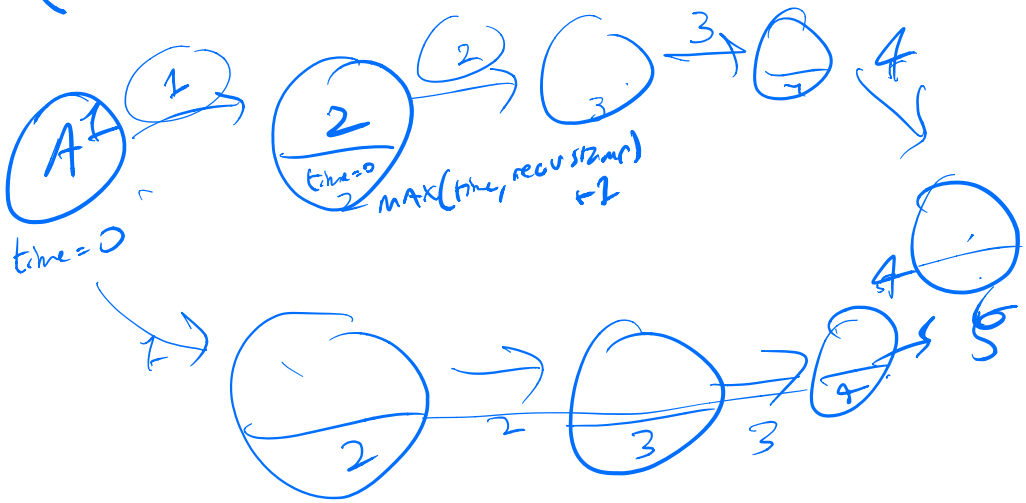
$\alpha \not> \beta$

$\alpha \stackrel{?}{=} \beta$

"Lamport clock"

one int added to messages ("piggy-backing")

LC = 0 ← message received



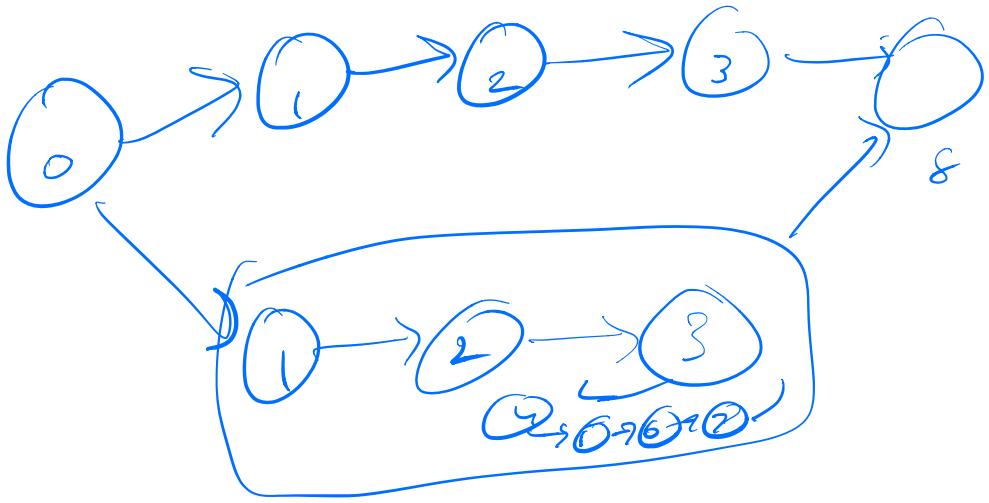
if $\alpha \to \beta$ then $LC(\alpha) < LC(\beta)$

if $LC(\alpha) \not\subseteq LC(\beta)$ then $\alpha \not\to \beta$

$LC(\alpha) \supseteq LC(\beta)$ $\alpha \to \beta$

total order
(arbitrary (for me))

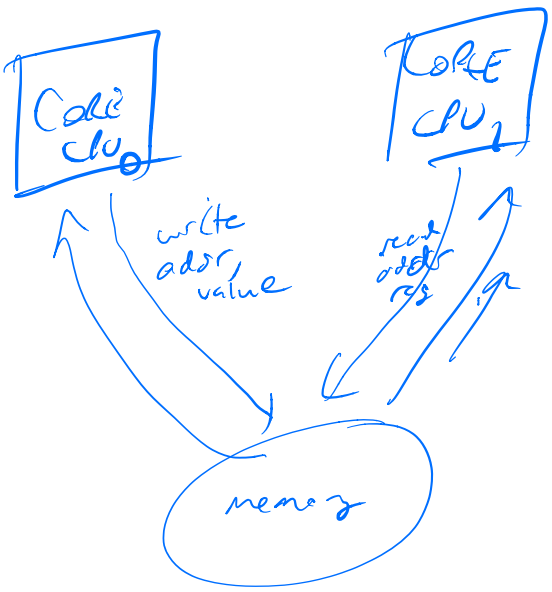
uint 32 64

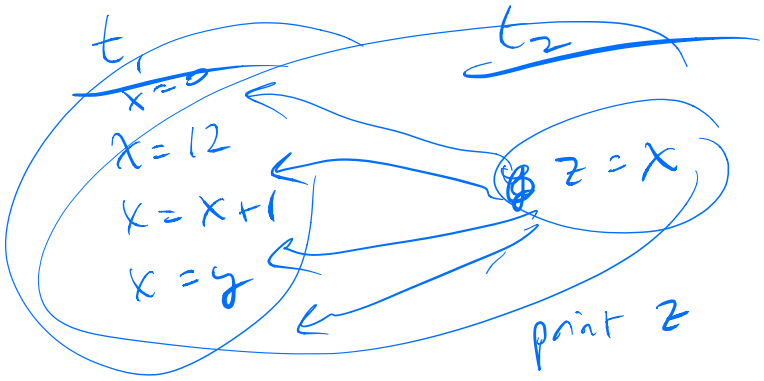


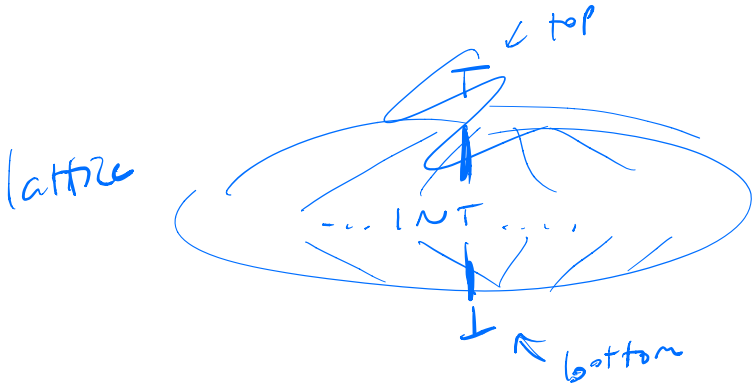
Threads (races)

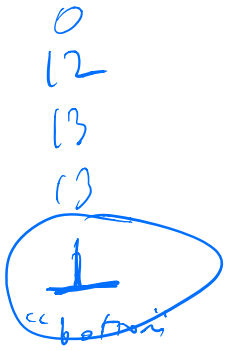
absence of a happens-before edge

≥ two threads accessing a shared value AT THE SAME TIME
one of them is writing


Core CPU 0
Core CPU 1
write addr, value
read addr
memory


t1
t2
x = 0
x = 12
x = x + 1
x = y
z = x
print z


lattice
top
... INT ...
bottom


0
12
13
13
bottom

cout << 1/0

"benign races"

Systems

PL

NO BENIGN RACE!

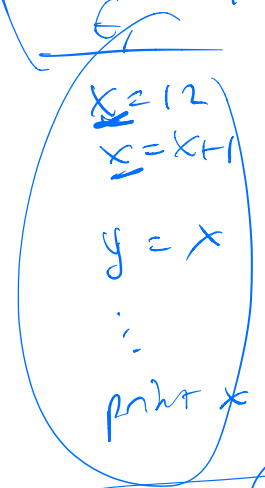
```
if (x == 0) {
  print —
}
```

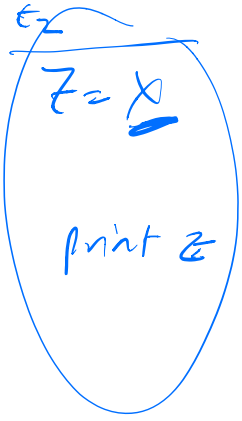
dead code analysis

backwards analysis

~~STOP~~

```
int x;
int y;
int z;
```





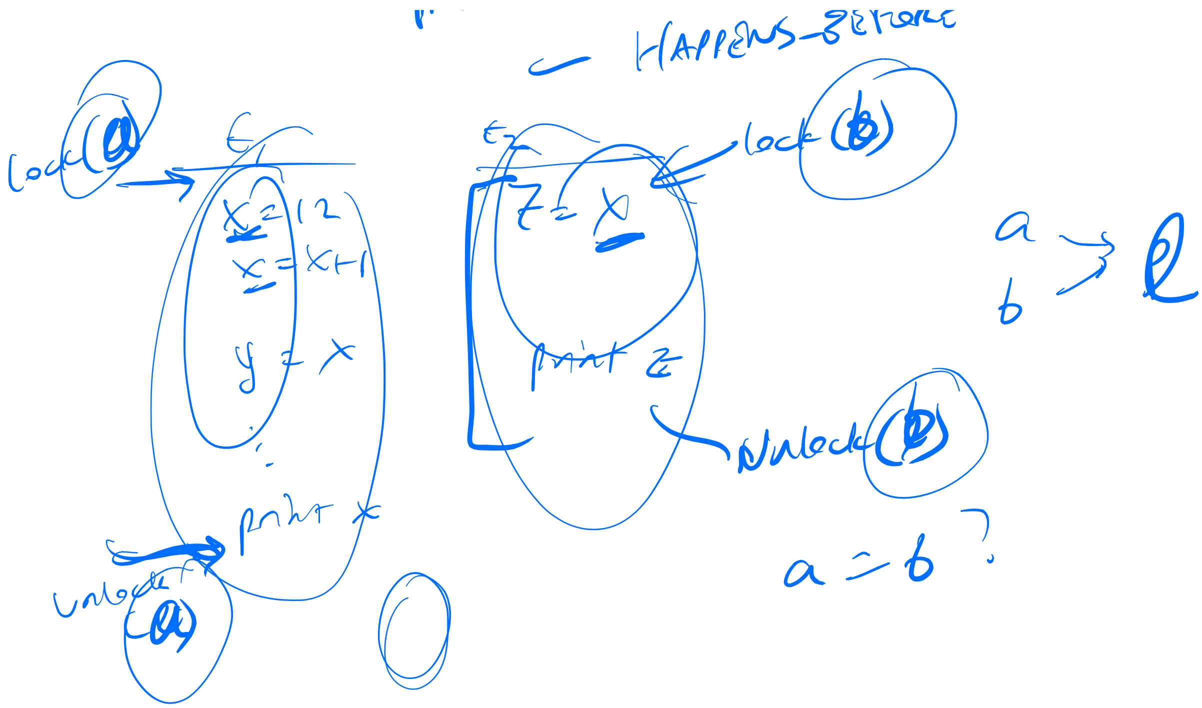
static race detection

WRITE: {x, y}
READ: {x}

WRITE: {z}
READ: {x}

— STATICALLY (at compile time) identify read & write sets

static analysis — halting problem



Halting problem:

given program p

does p terminate on all inputs?

p : while (true) { } ✗

p : straight line code, no loops, no fn calls ✓

p : if (ack(a,b) < _) {

}

if (halt(p))
  while true { }
else
  exit

t_1 — int x

t_2 — int x

$R = \emptyset$
$W = \emptyset$

$R = \emptyset$
$W = \emptyset$

Sound

reports no races ⇒ definitely no races

no false negatives

$R \cap W \stackrel{?}{\neq} \emptyset$

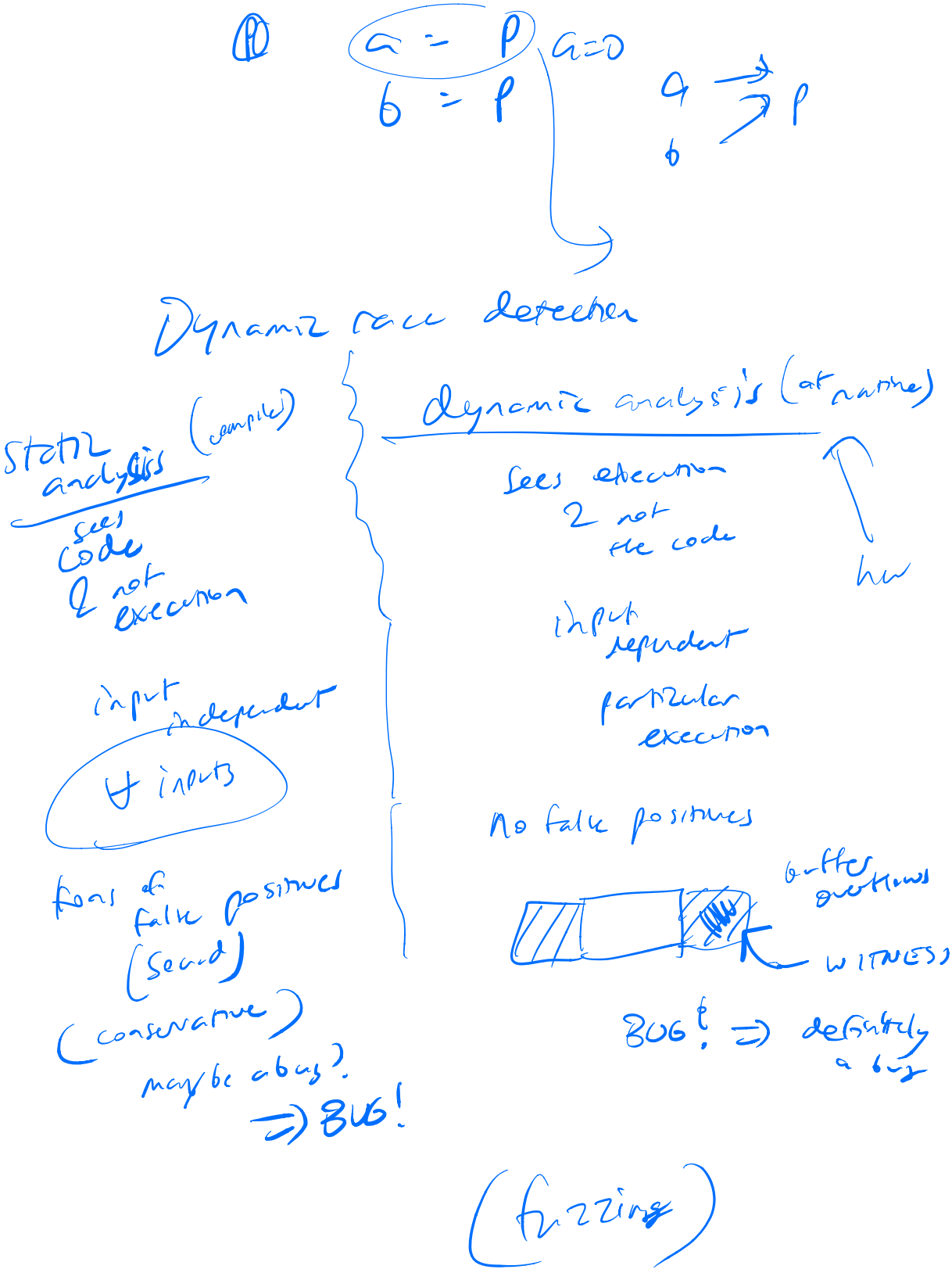
Complete

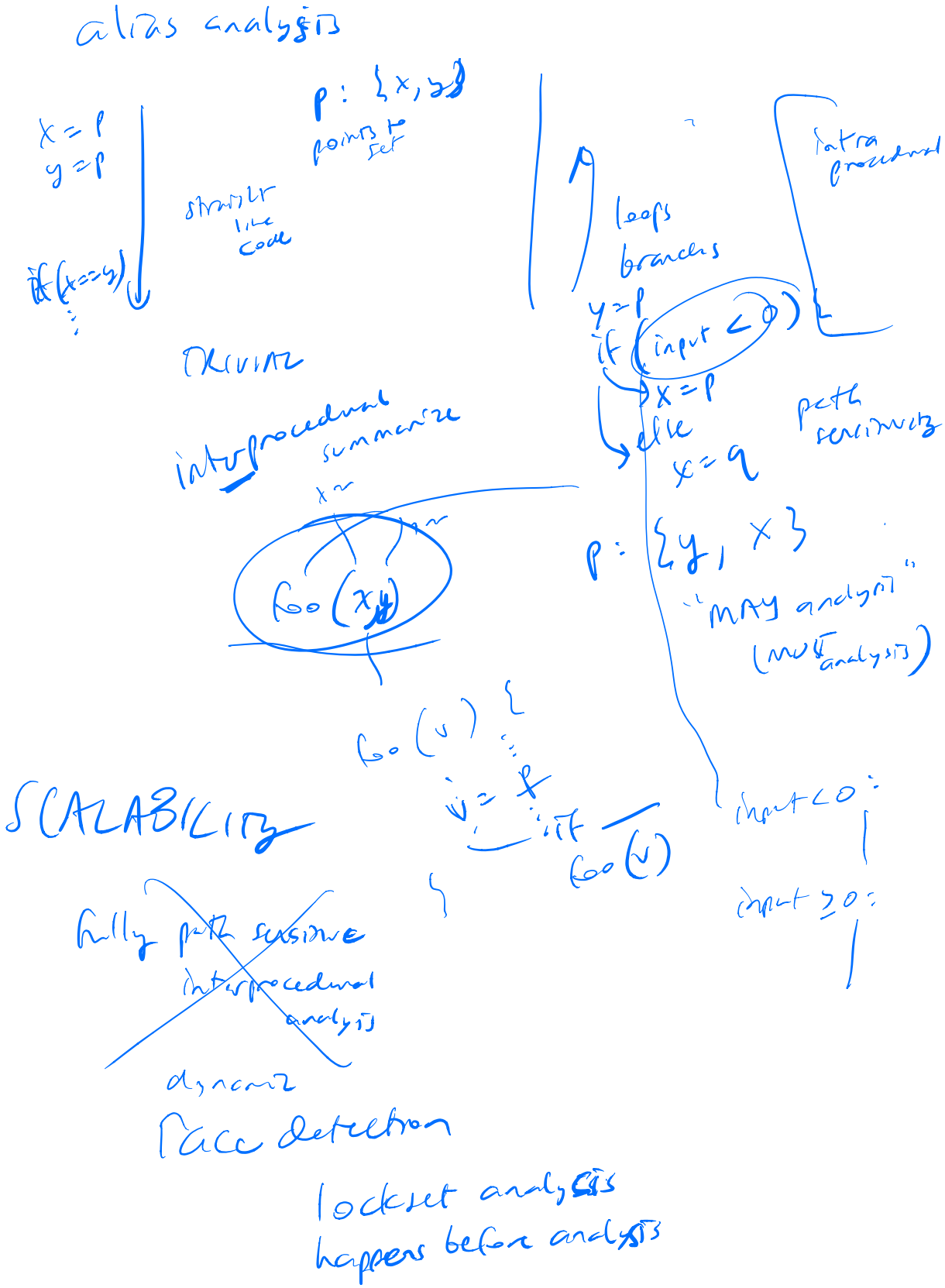
if reports race ⇒ race is real

no false positives

$a \stackrel{?}{=} b$ (not necessarily (lock held) ⇒ (complete) RACE!

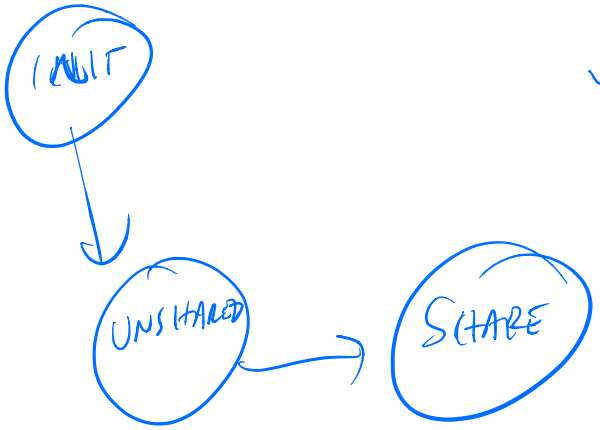
STATIC RACE DETECTION
FALSE POSITIVES


a = p
b = p
a=0
a
b
p
Dynamic race detection
static analysis (compiled)
sees code & not execution
input independent
∀ inputs
lots of false positives (sound)
(conservative)
maybe a bug?
⇒ BUG!
dynamic analysis (at runtime)
sees execution & not the code
hw
input dependent
particular execution
no false positives
buffer overflows
WITNESS
BUG! ⇒ definitely a bug
(fuzzing)


alias analysis
x = p
y = p
p: {x, y}
points to set
straight line code
if (x==y)
flow insensitive
interprocedural summarize
foo(x)
loops
branches
intra procedural
y = p
if (input < 0)
x = p
else
x = q
path sensitivity
p: {y, x}
"MAY analysis"
(must analysis)
foo(v) {
v = x;
if
foo(v)
}
input < 0:
input ≥ 0:
SCALABILITY
fully path sensitive
interprocedural
analysis
dynamic
race detection
lockset analysis
happens before analysis

x	{U}
y	{U}
z	{U}

	CURRENTLOCKS
lock(l)	{l}
lock(m)	{l,m}
x = 12 ←	$x_{LOCKSET}$ =
unlock(m)	$x_{LOCKSET} \cap$
unlock(l)	currentlocks
⋮	= {l,m}
lock(m)	
⋮	
x = x+1	
unlock(m)	{m}
x = x+1	∅



ERASER — lockset analysis

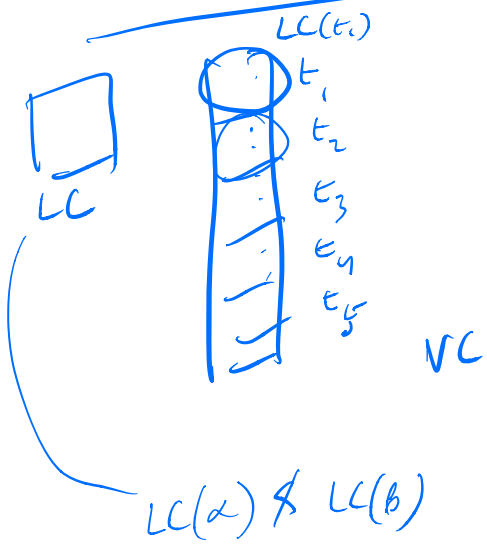
exhaustive

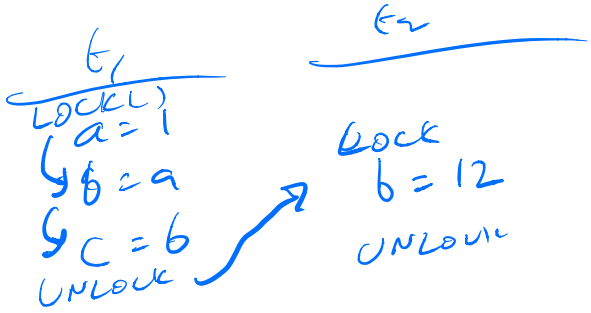
- false positives

+ "predictive"

HB

Vector clocks

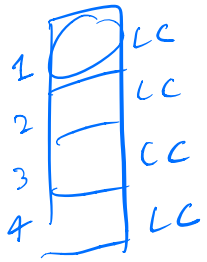




FastTrack

laziness

ThreadSanitizer "TSan"

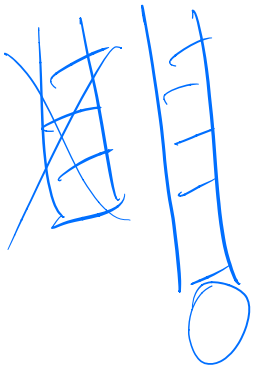


Ethereum

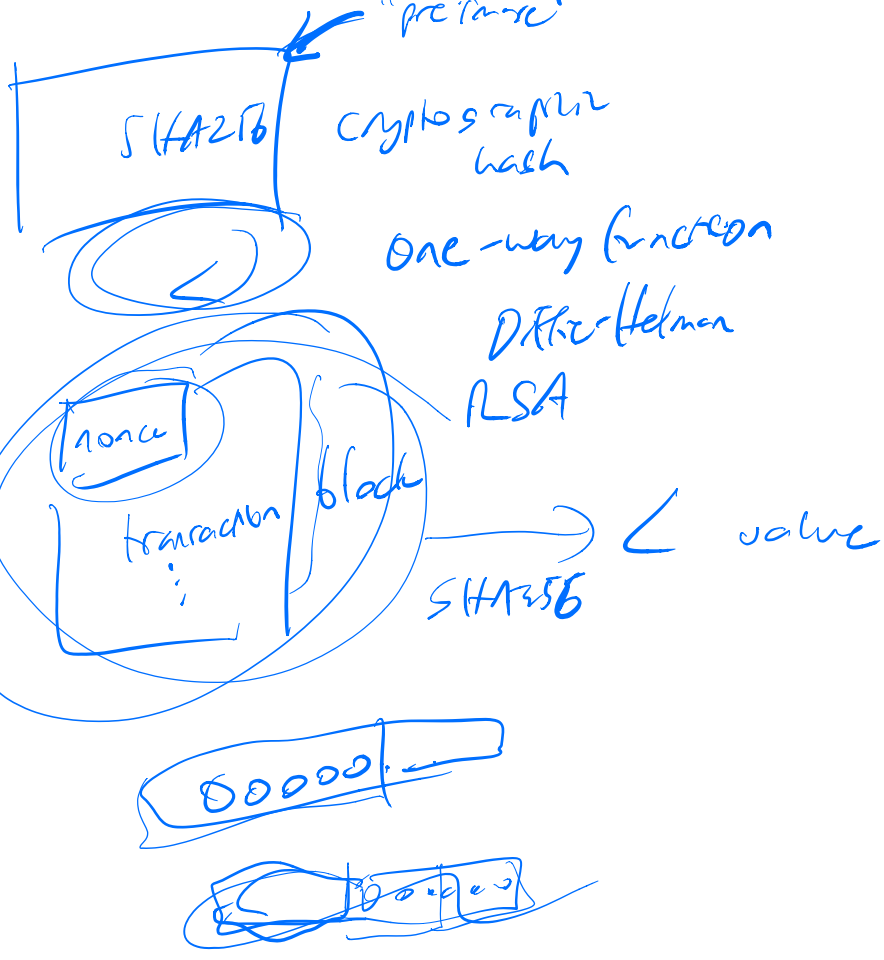
proof of work

>50% mining capacity → own the system

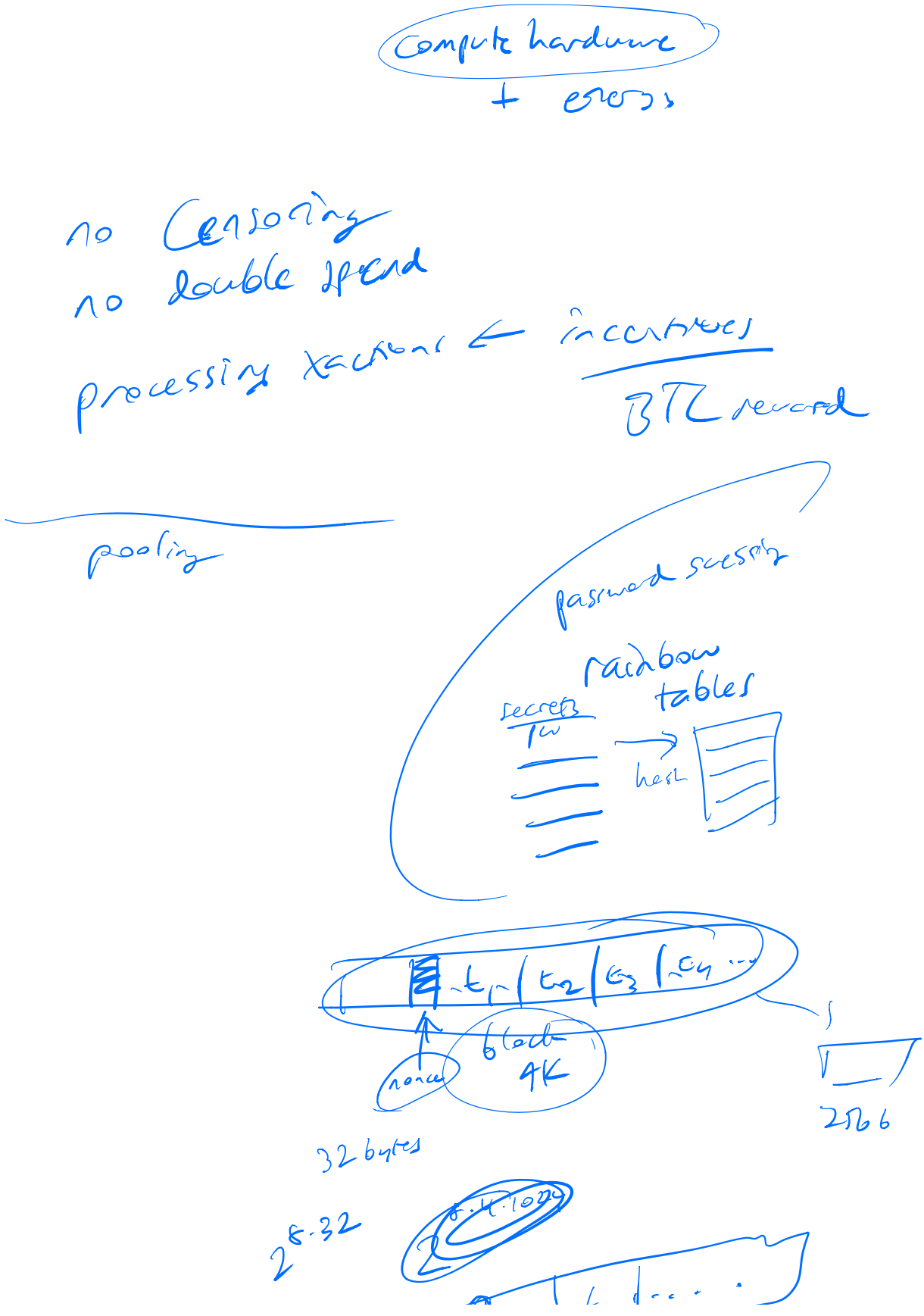
"house always wins"

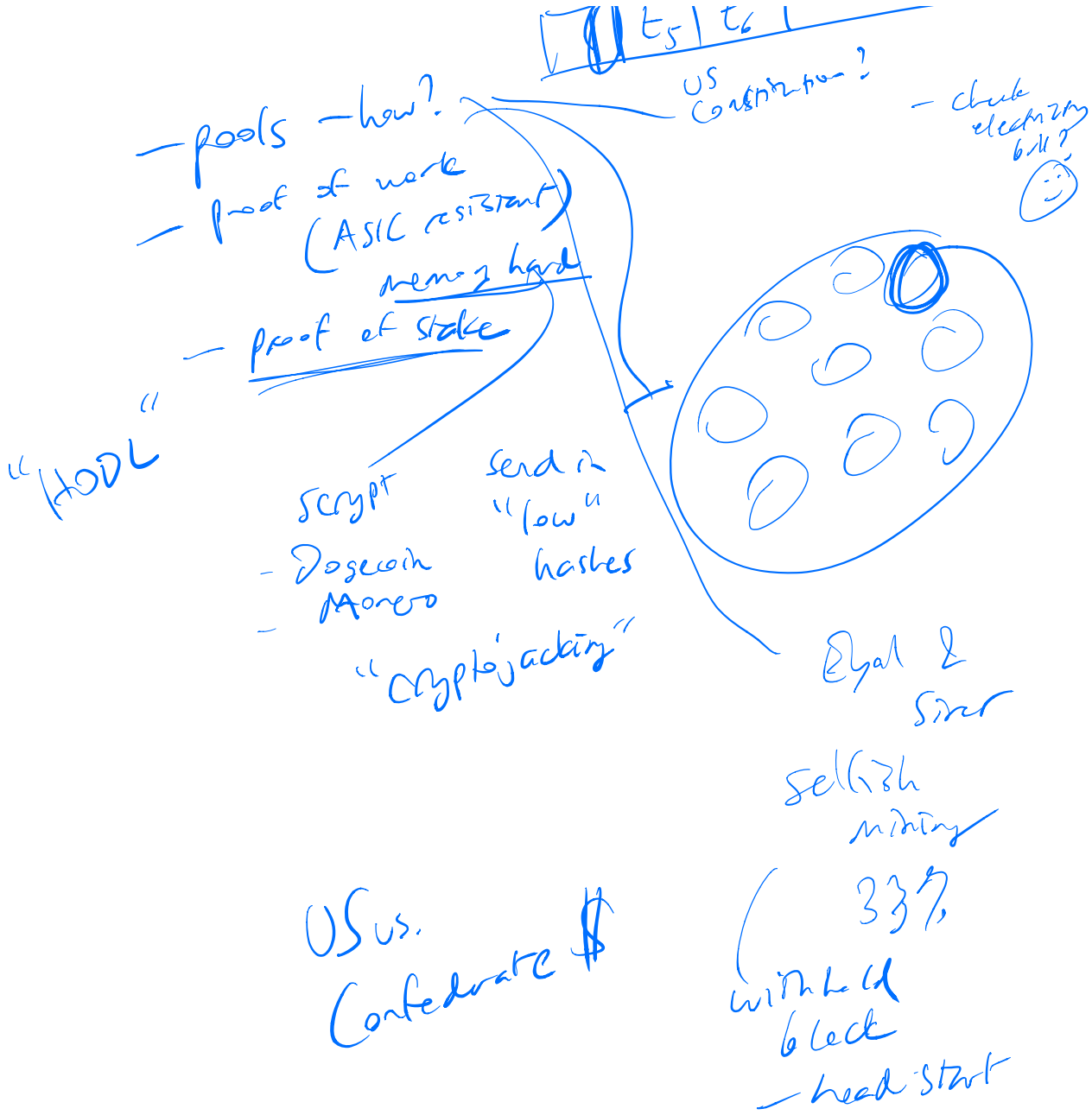


selfish mining attack



CPU → GPU → ASICs


Compute hardware
+ energy
no censoring
no double spend
processing xactions ← incentives
BTC reward
pooling
password guessing
rainbow tables
secrets
pw
hash
block
4K
nonce
32 bytes
256
2^{256}



legit uses of blockchain

censorship-resistance

unforgeable

Bitcoin - not TC durable (Nick Camuto)

- Ethereum

"Smart contracts"

dynamic & static analysis ("sound")

Valgrind

10x - 100x

witness

Coverity

~~sound~~

~~complete~~

low false positive rate

lock(l)

no HB →

"potential race"

unlock(l)

lock(l)

unlock

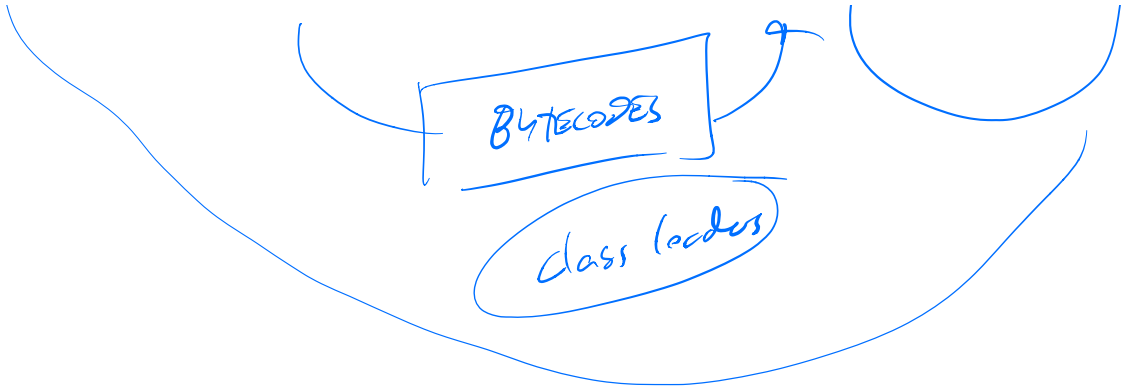
rare (usually security to other executions

memcheck

buffer overflows
invalid reads/writes
dang ptrs.
etc.

cout << "nope" << endl.

precision-recall tradeoff

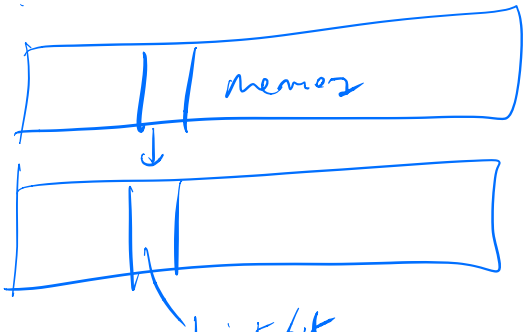


closed-world assumption → sound analyses (assume closed world)

(helgrind race detector

The Secondest Monkeys)

<memcheck>



but live in open world

"fast"

Pin

DBI dynamic binary instr.

pintool

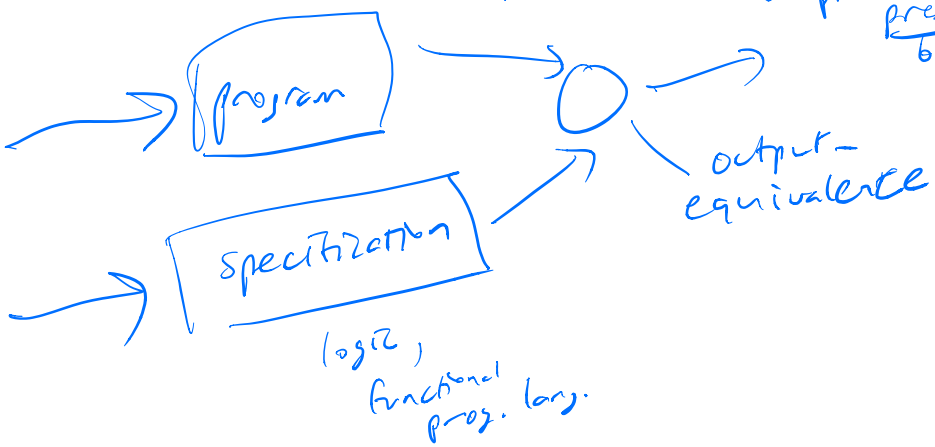
Testing

"total" / complete

Verification:

"proves the absence of bugs"

testing — Dijkstra "proves the presence of bugs but not their absence"



$\forall i \in \text{inputs}: \text{output}(\text{spec}) = \text{output}(\text{program})$

$\text{sorted}(f) \equiv \forall i \in \{0, \text{len}(f)-1\} : f[i] \leq f[i+1]$

precondition — sort(f) — postcondition sorted(f)

$\{\text{pre}\}\ \text{sort}(f);\ \{\text{post}\}$ — Hoare triples

weakest predicate transformers — Dijkstra

partial verification:

absence of some bugs

- no buffer overflows

Theorems for Free — type systems

dependent type theory

```
int x;
struct y;
x = x + y;
```

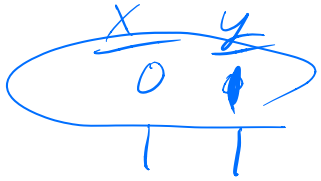
logical constraint

```
int x : x < 10;
```

program → constraints

SMT } solver
SAT }

model checking approaches




```
x = input
bool x
bool y   if (x)
           y = 1
  :
{x ∧ y}  else
           y = 0
{x = y}
```

x : inp = 0 ⇒ f

FAILURE

- program wrong
- spec wrong
- both are wrong

Eiffel — "contracts"
precond
postcond
} assertions on steroids
"runtime verification"

— execution time overhead
— input dependent
tells you about current/past executions
not the future

GOALS: working software
secure
Fast? — asymptotic complexity
correctness wrt SPEC
bounded execution time
availability
correctness
safety privacy
integrity
ABS < 0.1s

bool x = inp 2
bool y = inp 2
4

int32 x 2^{32}
int32 y 2^{32}
2^{64}

hard realtime constraint

SMT solvers
Satisfiability Modulo Theory

{x=1}
{x>0}

SAT NP-complete

$(x \wedge y) \wedge ((z \vee y) \vee (x))$

x y z
T T x

T, T, T

DPLL

x	y	z
F	F	F
F	F	T
⋮		

k variables
N possible values

worst-case: N^k

"common case": $O(N)$

SAT solvers / SMT solvers — practical

Z3 Microsoft

"secret sauce of DART"

The Specification problem

write a spec (hard!)

longer than the code

3X-5X

V

is it correct?

"quis custodiet ipsos custodes"

verify
spec
code ✓

P ≫ ≡
S

$P_1 \in P$ $P_1 \equiv S_1$

$S_1 \in S$

temporal logic
safety properties
progress properties

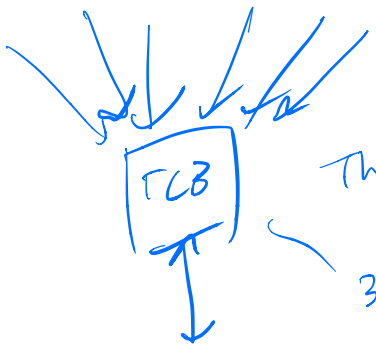
absence of deadlock
bounded response

model checking

component
testing
important
"easy to specify"

low-level
buffer overflows
privilege escalation

TCB
trusted computing base



Thompson's Turing

300 LOC

TESTING

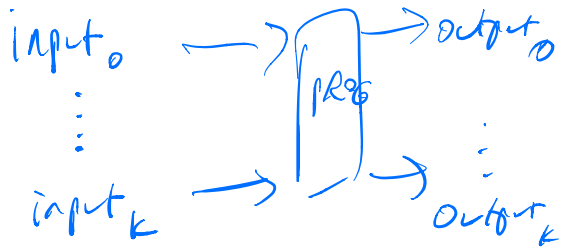
Knuth

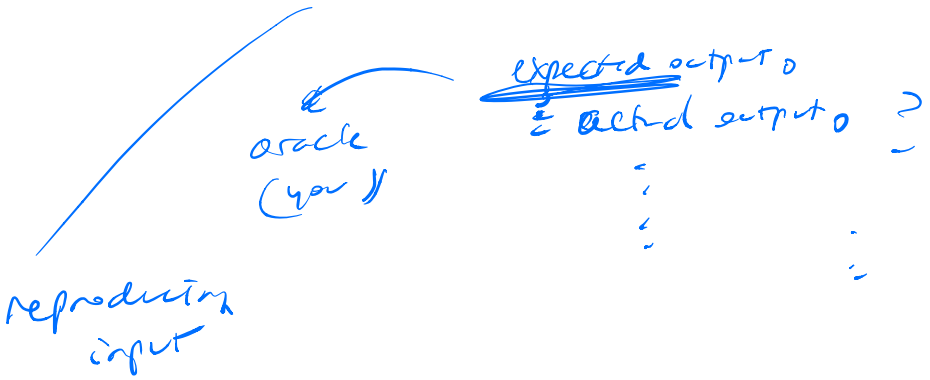
"I have only proved it correct, I have not tested it."

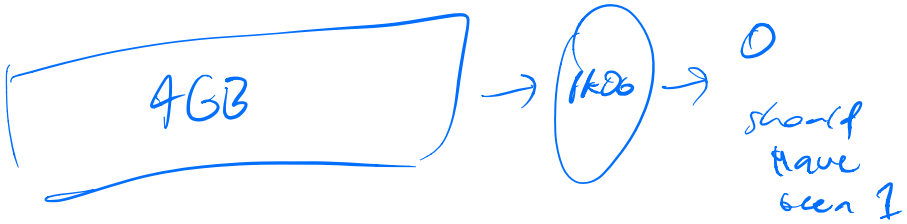
TESTING

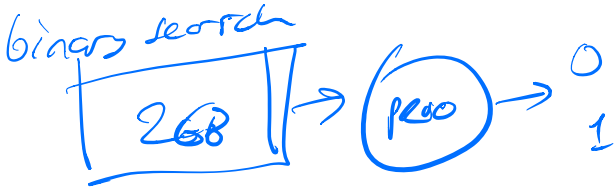
system testing

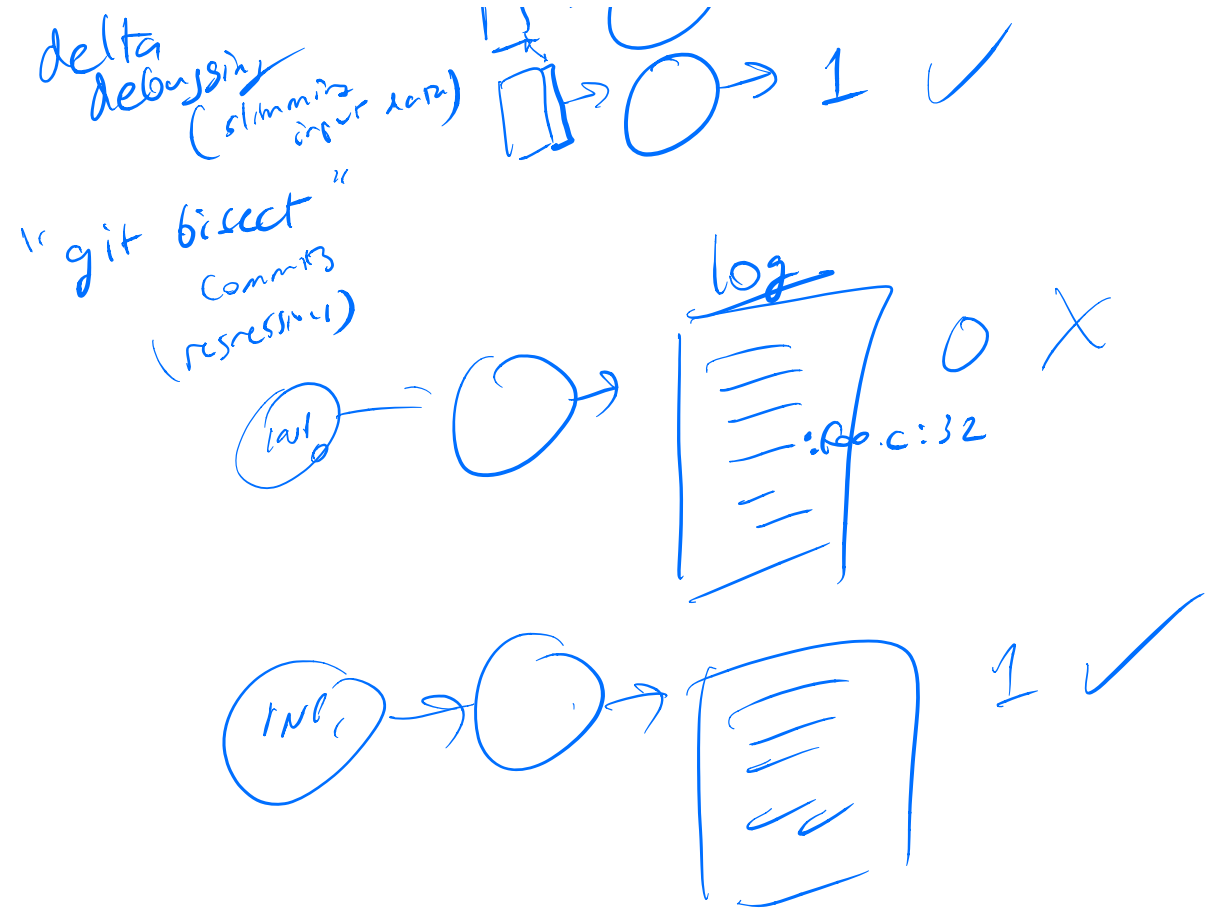
end-to-end testing



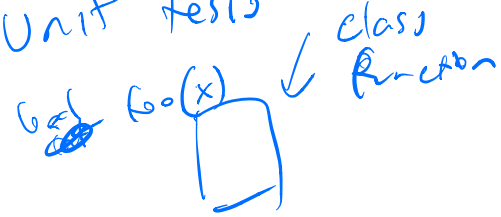






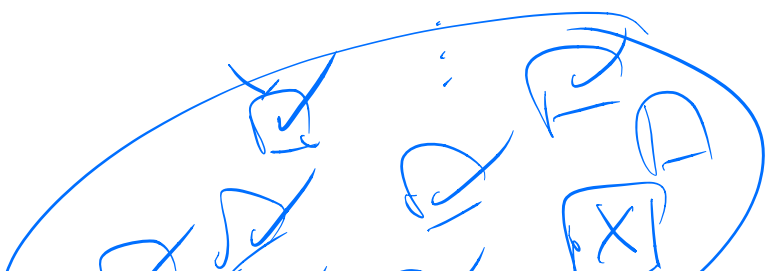


Unit tests



"input/output examples"

foo("—") : 1
foo("—") : 0



human-created test suites

− (1) correlated with the code

+ (2) test created post hoc
→ could be good insurance
(prevent future regression)

+ (3) documentation

− (4) low coverage

→ % of LOC executed (line)
% of branches taken (branch)
% of paths (path)
exponential in # branches

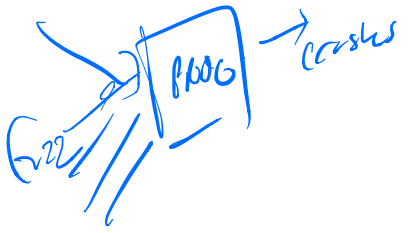
Fuzz testing

Randomly-generated "tests"

oracle: crash is BAD

availability
⇒ security implications

"line noise"

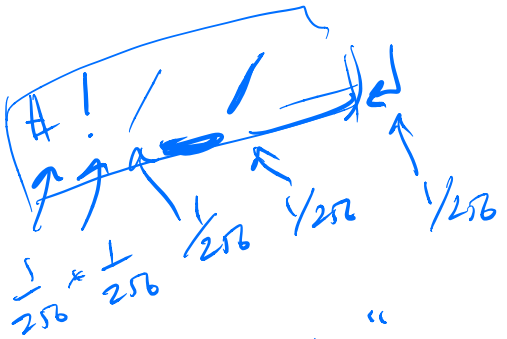


AFL

American Fuzzy Lop

DART

```
if (X[0] != "#")
    abort();
```



"for (i=0;;);"

$1/2^{64}$

```
if (x == 0.357) {
}
```

Parsers

URL regex

CSS
HTML
Command-line

| x % 256 == 0

Z3

~~x = 0.35f~~

~~x = 0.35f~~

if ( ) {

x = 0

}

"concolic testing"

Concrete + Symbolic

3rd party device drivers

BSD

DDK

"SAGE"

Project Everest

OpenSSL

Heartbleed

F* → C