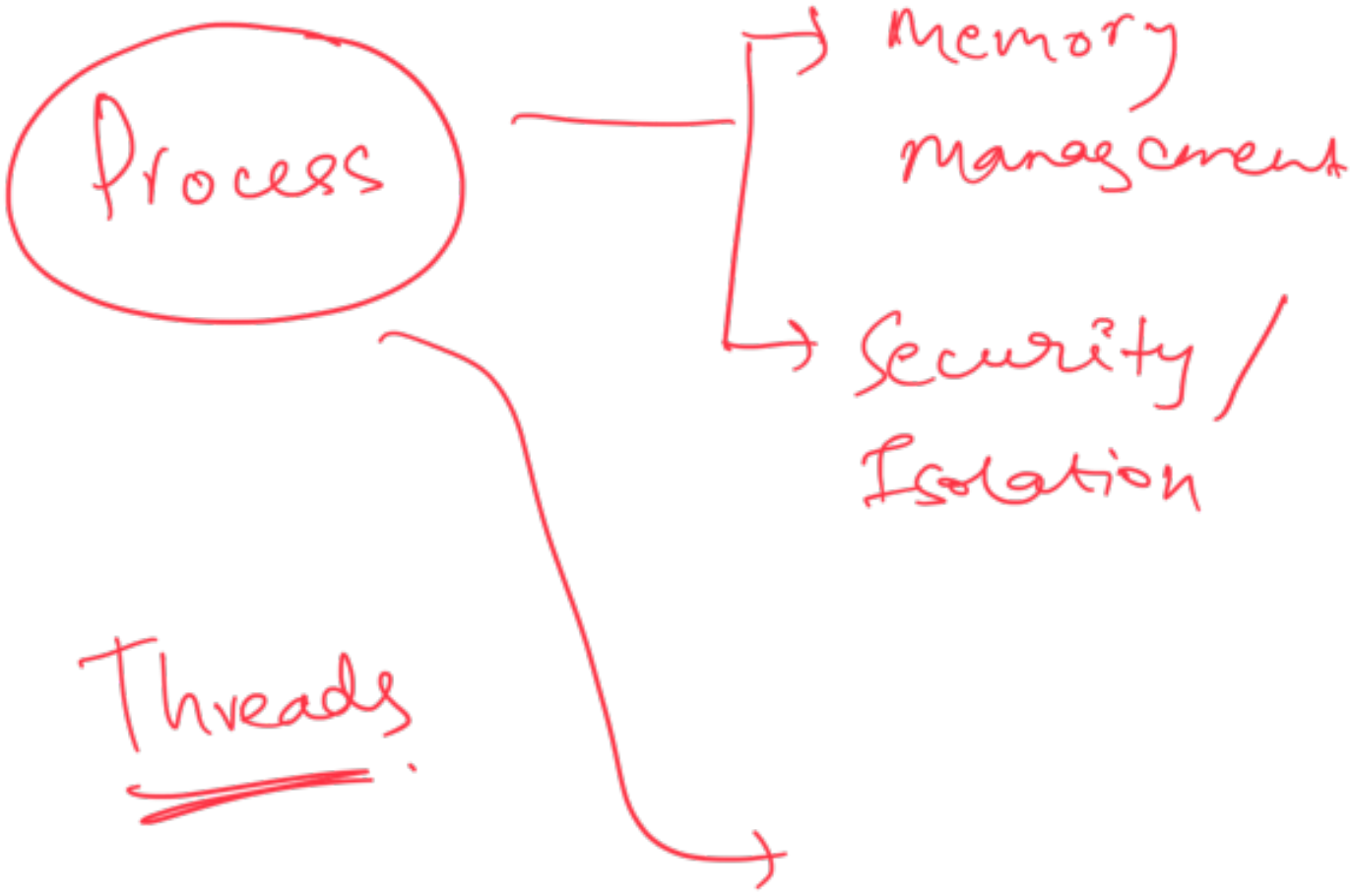
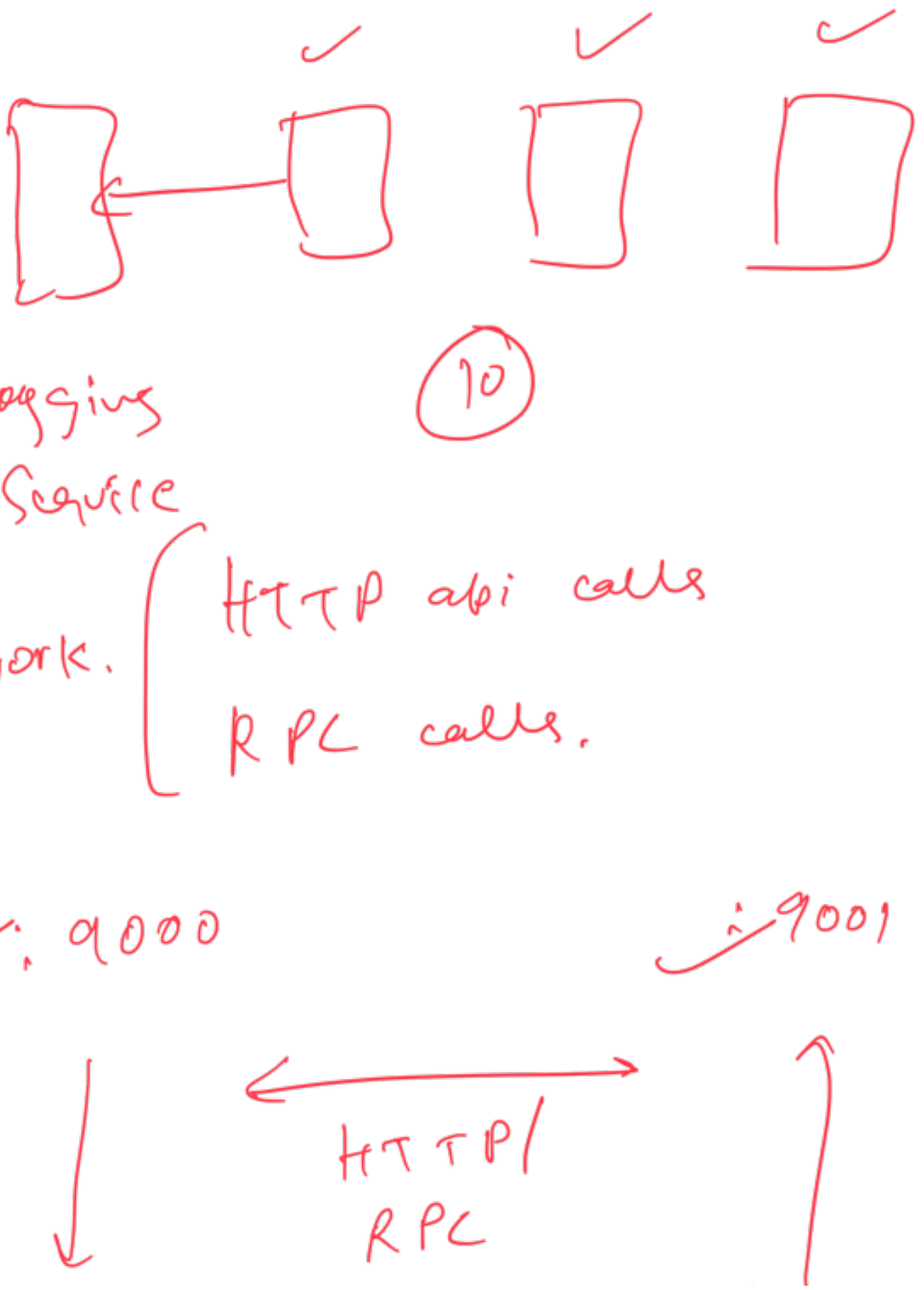
Operating Systems - 3

Threading & Synchronization

- Threads & why they're need.
- I/O & CPU bound.
- Producer / Consumer
- Race condition
  - mutual exclusion
  - progress
  - bounded waiting
- Peterson's alg.
- Mutex, Semaphores
- Deadlocks
  - What
  - How we can resolve
  - When.

Threads





logging Service

(10)

network. { HTTP api calls
RPC calls.

:9000

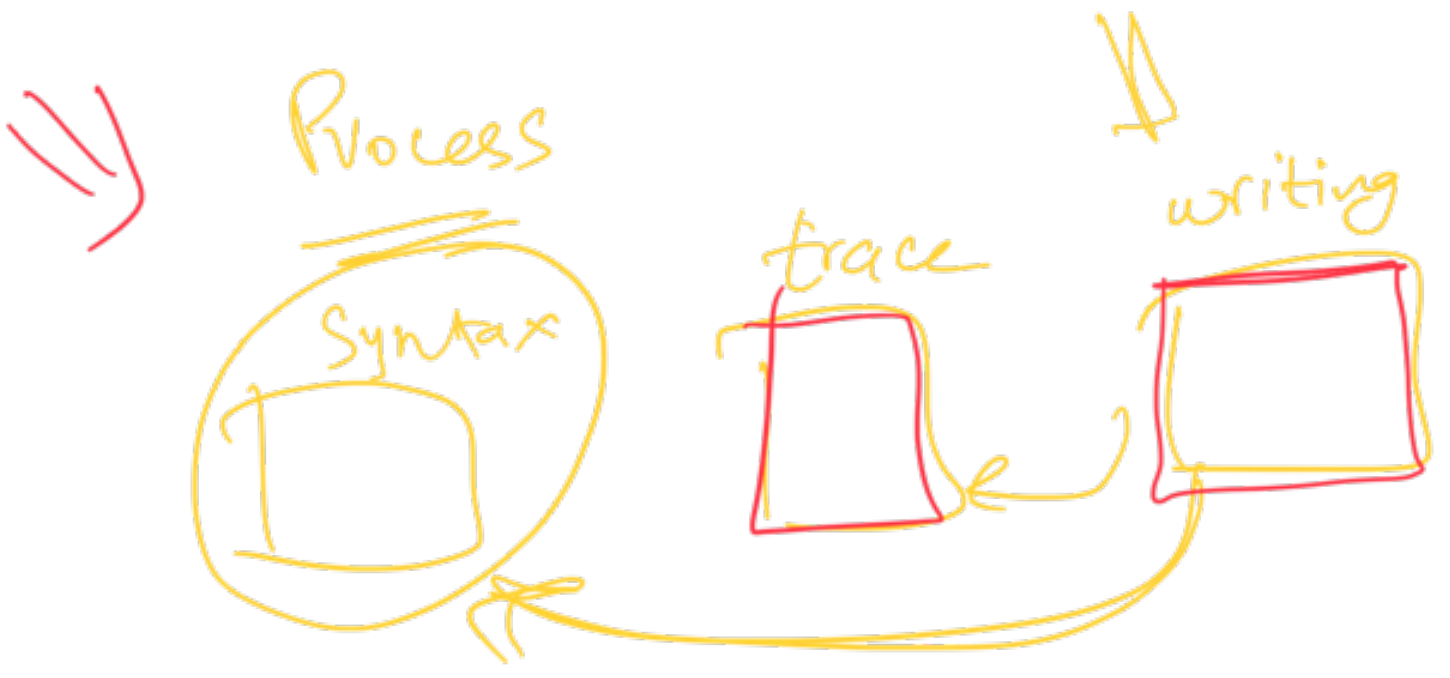
:9001

HTTP/ RPC

write Read

↳ When we want to share memory

Text Editor



X Design → expensive

→ Share memory.

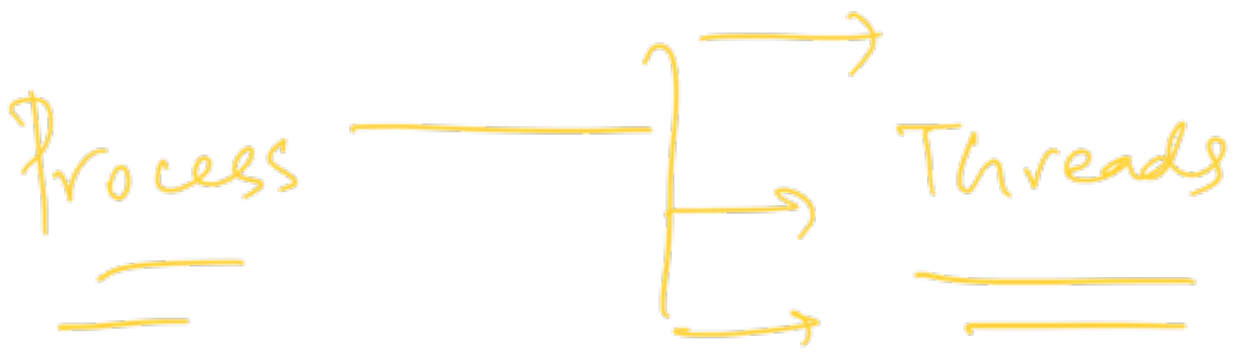
Threads

→ provided by the OS

Process → Threads

Text Editor

⇐ IntelliJ



X memory

X slow

OS

sharing memory ✓

fast ✓

PCB

[ ] threads.

TCB

	T_1	T_2	T_3
Data			
resources			
registers	—	—	
PC	—	—	
stack	—	—	

small

Process.

Text Editor → writing

→ syntax

→ grammar

Scheduling

OS

✓ Process → Scheduler.

✓ Thread. → OS threads.

→ P.L / library

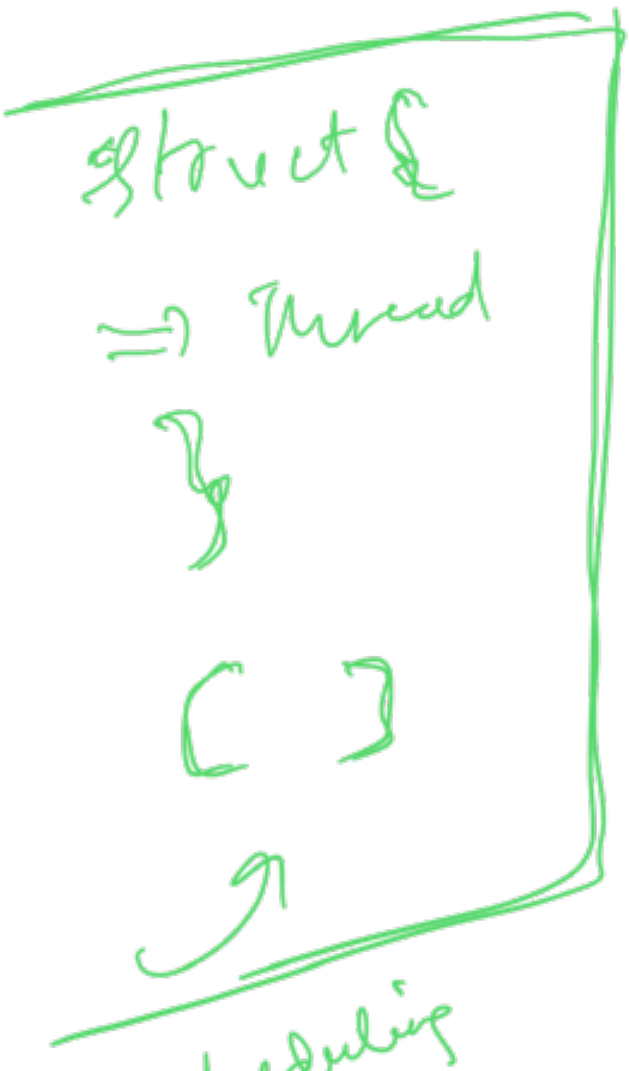
Java.

Python

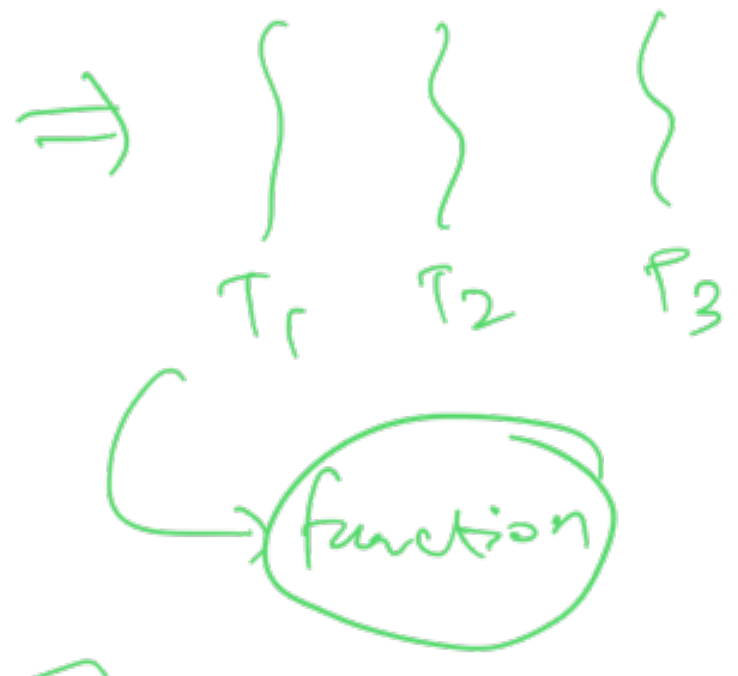
Greenlets.

Golang

Coroutines.



User space.



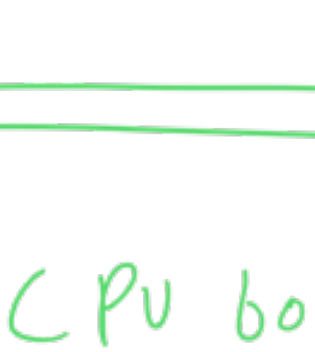
goroutines.

read
goto another
read

Cooperative

I/O bound	= CPU bound.
FS	FFT
Printer	Image filtering
Input	Crypto
Network.	Mining

CPU intensive



1 processor

threads

C1 C2 C3] ⇒ only one of

C_1, C_2, C_3 executed

Threads → I/O bound.

P_1

T_1 is waiting on I/O

schedule T_2

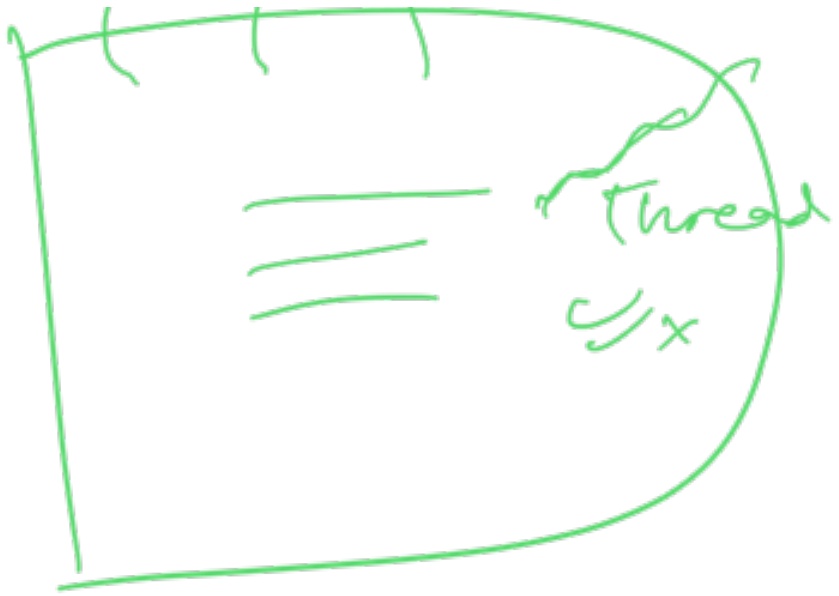
T_3

T_1 T_2 T_3

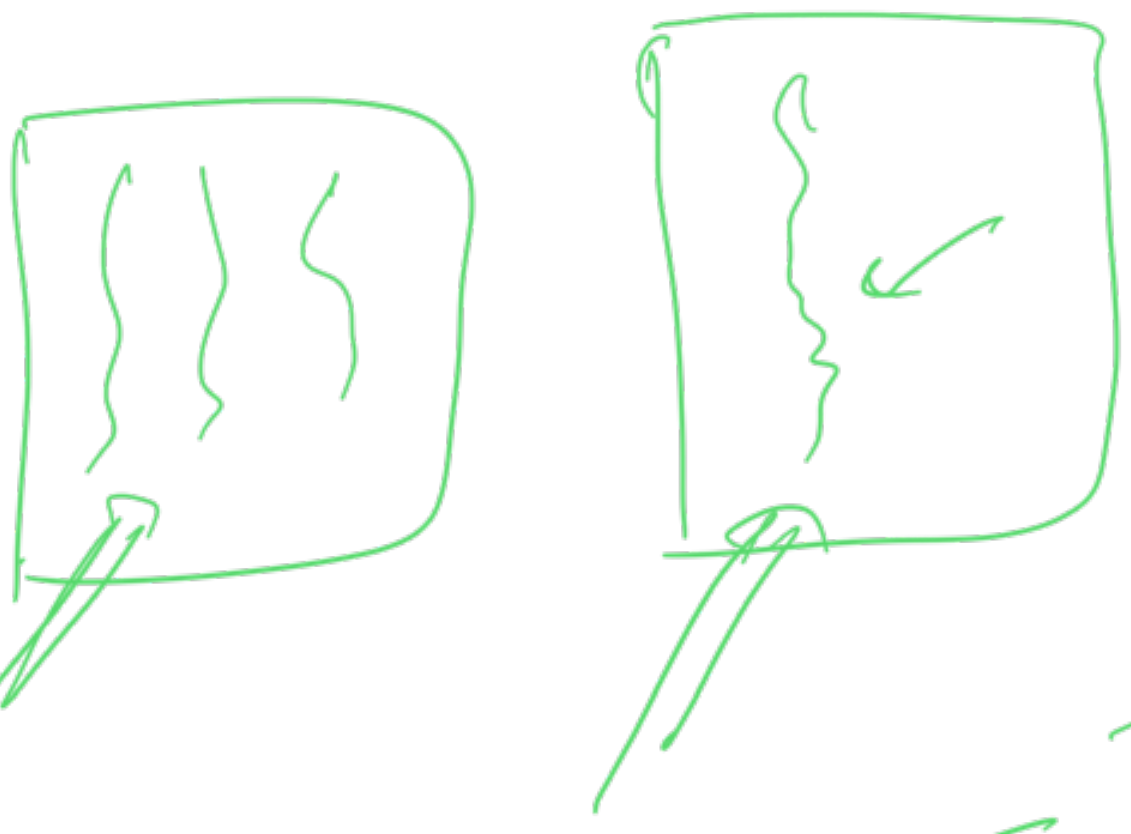
I/O

I/O

P_1 P_2

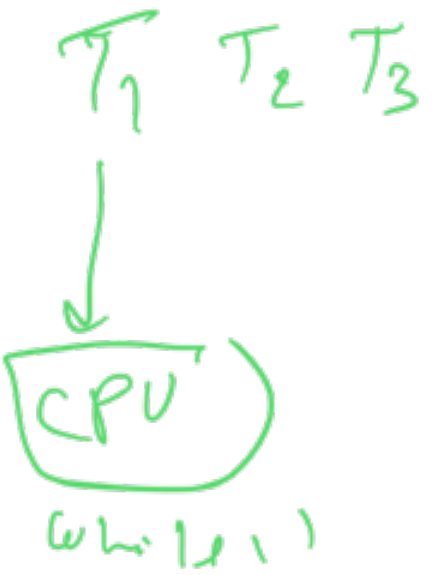


Text editor



Cooperative Scheduling

each thread
tells when it
wants to stop



⇒ Whi

Producer Consumer problem

* 1 producer which produces data & pushes to Q

fixed

1 Consumer which reads from the Q.

Examples
- Message Queues
- Streaming.
- Printing pool.

Constraints

1. Producer should not produce when buffer is full.
2. Consumer ~~in~~ ~~~~ ~~in~~ ~~~~ empty
3. Each item should be consumed exactly once

↳ overhead

↳ destroy logic.

buffer = [ ]N
cnt = 0

Producer | Consumer

write(){ | while(){

full

while(cnt==N); | while(cnt==0);

not empty

buffer[cnt] = item; | item = buffer[cnt];

not full

cnt++ | cnt--

}

T1 | T2

* Runs most times.

* Fail X

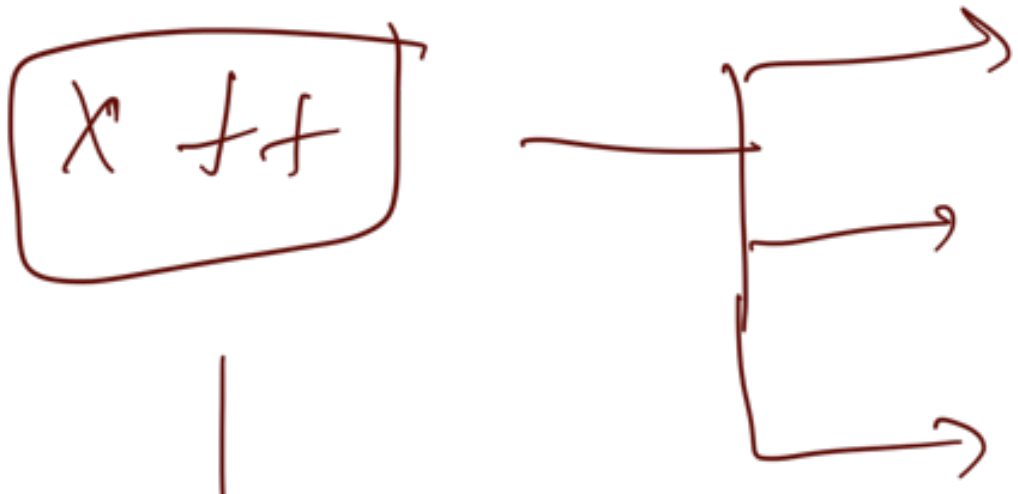
X++ ⇒ is not atomic. (depend on language)

Latin ≡ indivisible.

JMP
JNZ
:

ATOMIC

you can't execute these instructions halfway.



① Load value of x into a register

② Add 1 to that register

③ Write the value back to x.

T_1 T_2

cnt++

x+2

T_1 Load x into Reg_1

T_2 Load x into Reg_2

T_1 Inc Reg_1

T_2 Inc Reg_2] x+1

T_1 write Reg → x

T_2 write Reg → x

y

x

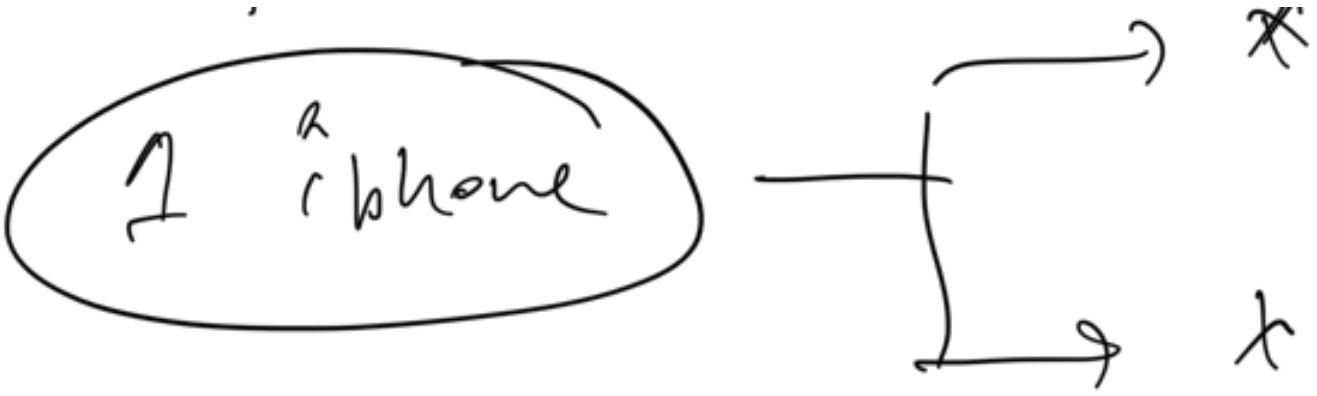
x+1.

Race Condition

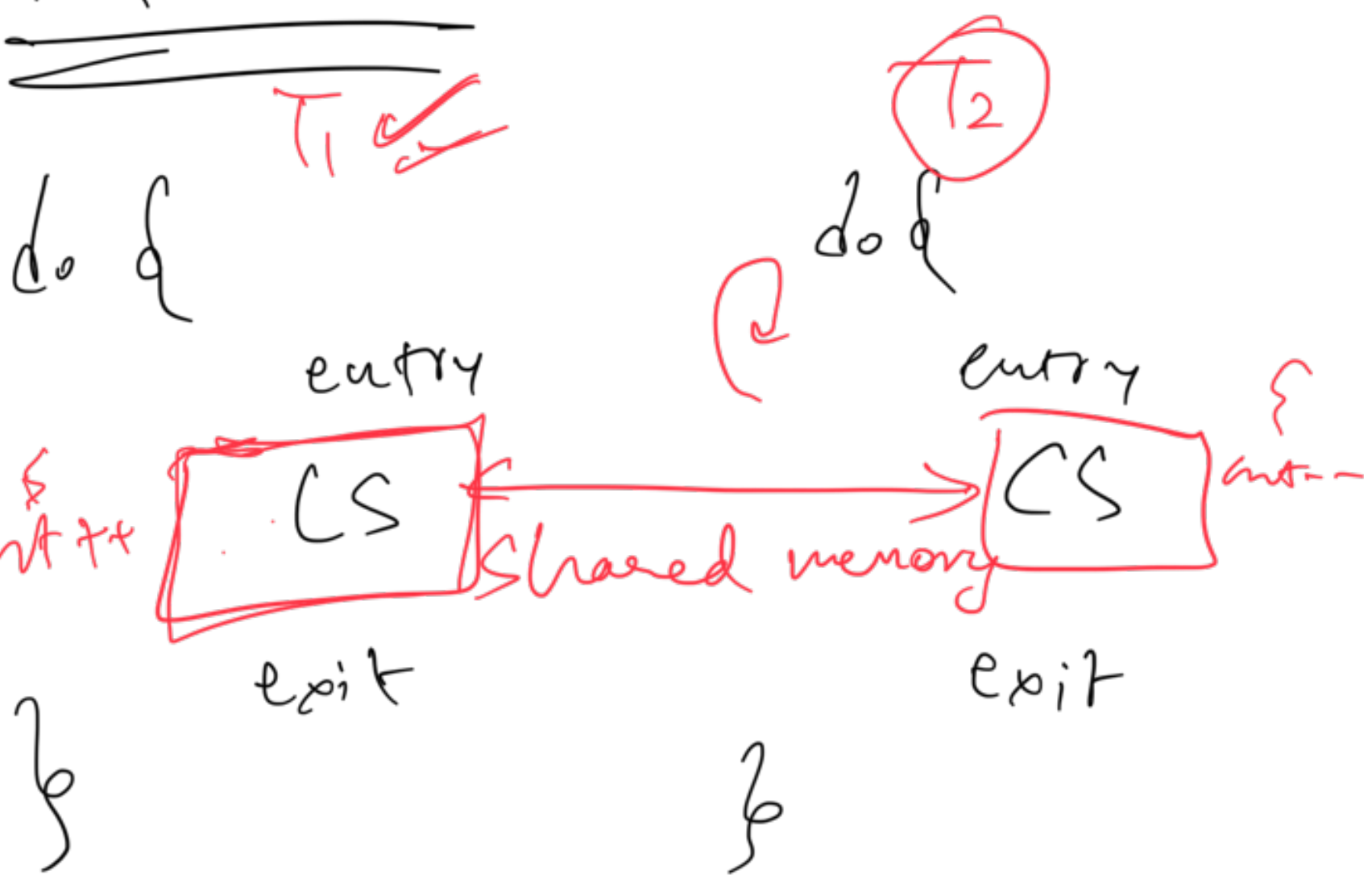
↳ Context Switch can lead to data inconsistency.

a) difficult to catch.

Amazon Flash Sale.



Reformulate



* Mutual Exclusion — only one process is in CS at a time.

* Bounded waiting —

T_2 entry

T_2 CS

CON

T_1 entry

Starved

CON T_2 CS

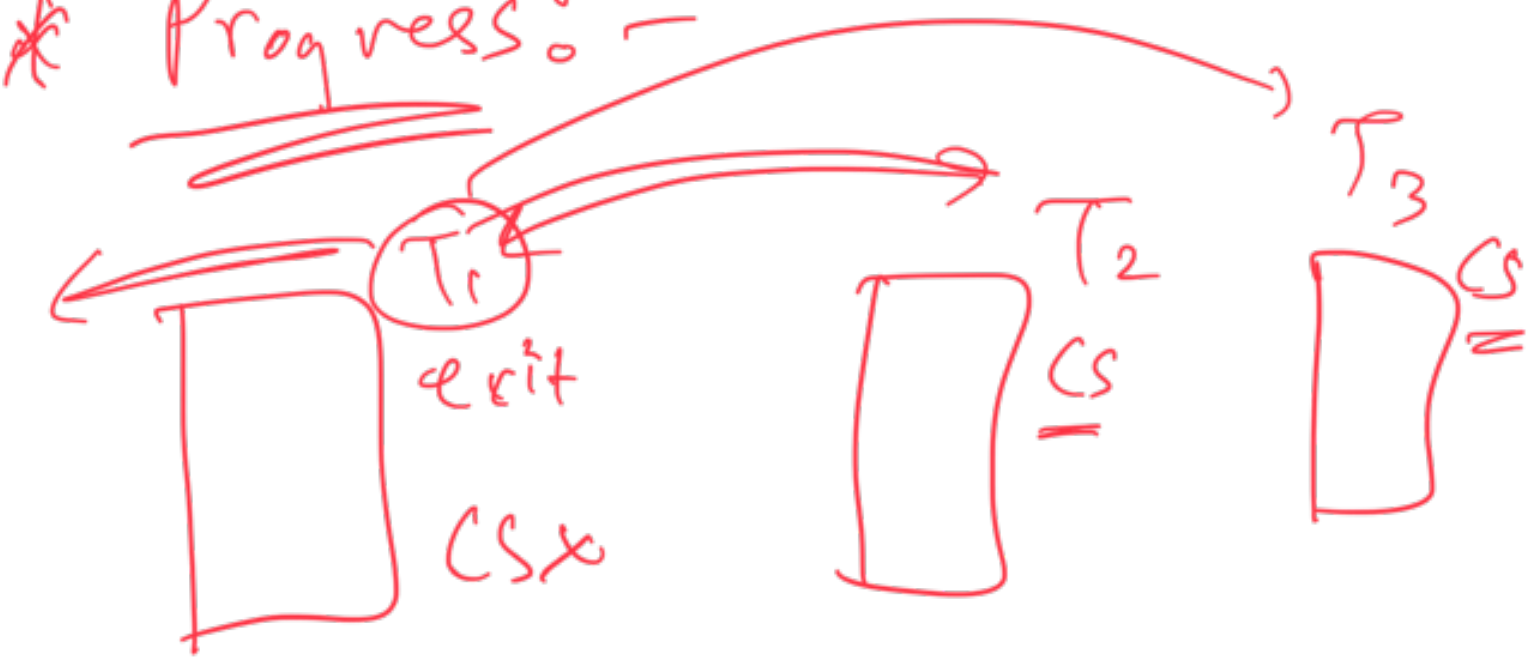
T_2 exit

T_2 entry

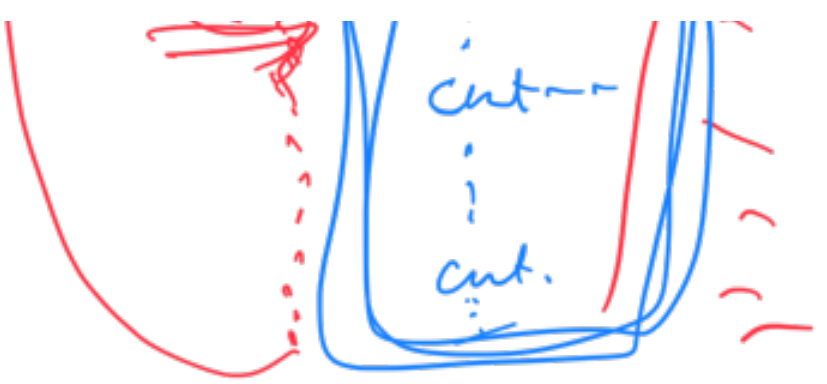
T_2 CS

CON T_1

* Progress:-



* Only processes in their CS can make decision of who to execute.



Peterson's Algorithm

I want to execute

T_0 ~

T_1 ✓

loop [$flag_0 = true$

turn → 1 → you can have the turn before me

while ($flag_1$ && turn == 1) ; → while it is your turn / a certain lock

// CS

$flag_0 = false$ } releasing lock

]

T_1	T_2
✓ $flag_0 = True$	✓ $flag_1 = true$
✓ turn = 1	✓ turn = 0
while ($flag_1$ && turn == 1) ;	while ($flag_0$ && turn == 0) ;
// CS	// CS
$flag_0 = false$	$flag_1 = false$

while(flag$_1$ &&
turn == 1);

CS

flag$_0$ = false

while(flag$_0$ &&
turn = = 0)

CS

flag$_1$ = false

flag$_1$ = false

flag$_0$ = True

turn = 1

CS for some time

CON

flag$_1$ = true

turn = 0

Complete CS

CON

flag$_0$ = false

Move to

T_0, T_1 = false

while(T_1 is in CS);	while(T_0 is in CS);
T_0 in CS = T	T_1 in CS = T
CS ✓	CS
T_0 in CS = F	T_1 in CS = F

-

Mutex

```
{ acquire_lock()

  CS

  release_lock()
```

remainder

Mutex

✓ LOAD
ADD

test_and_set(x)

HW atomic

Returns value of x & sets value of x as 1

lock=0

while(test_and_set(lock)==1

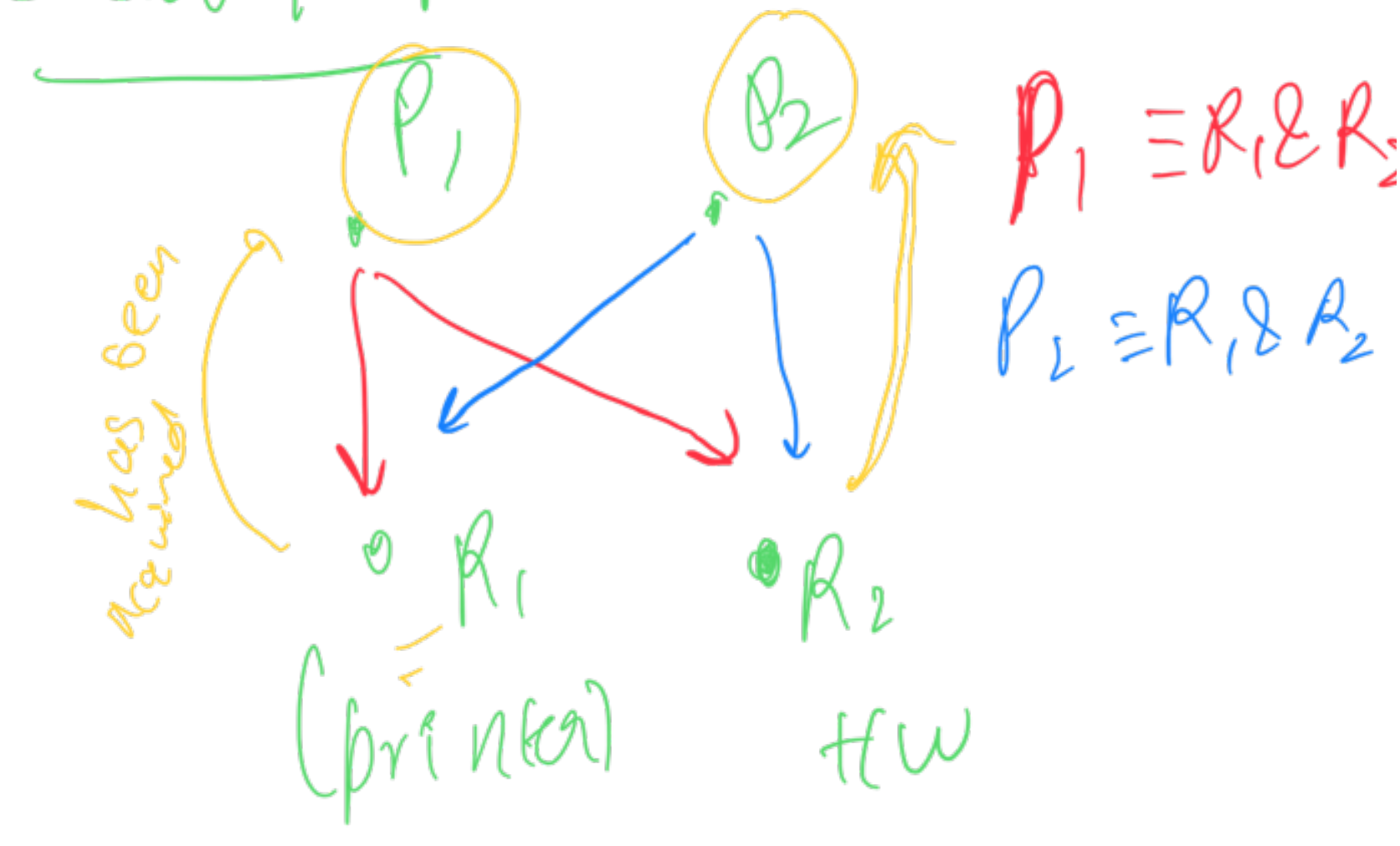
CS

lock = 0

HW

lock = 1] occupy

Deadlock



P2 is waiting for R1

to free up

P1 is waiting for R2
to free up

Deadlock

Conditions for deadlock

1) Mutual exclusion :- resource can only be used by one process @ a time.

2) Hold & wait

3) no preemption :-

Can't take resource away from process

4) Circular wait.

Handling ??

a) PREVENT

→ i) Mutex:

allow a resource to be used by multiple process

at the same time.

→ ii) H & W

Process ~~P2~~ sends

Bad X ← request to all the resources at the same time.

iii) No preemption

L ↳ Some state X

iv) CW

R_i — R_j

R_j R_i L_j

b) DETECT

Windows & Linux

→ ignore.

Semaphores

S = 0 ⇒ Resource not available

S = 1 ⇒ 1 of S is available

S = 2 ⇒ 2 of S are available.

wait(S): if S > 0, then grab it & decrement it.

signal(S): increment S by 1.

Process A	Process B
A_1	B_1
A_2	B_2

S = 0

signal(s)

A_3

B_3

wait(s)

A_4

B_4

A_5

B_5

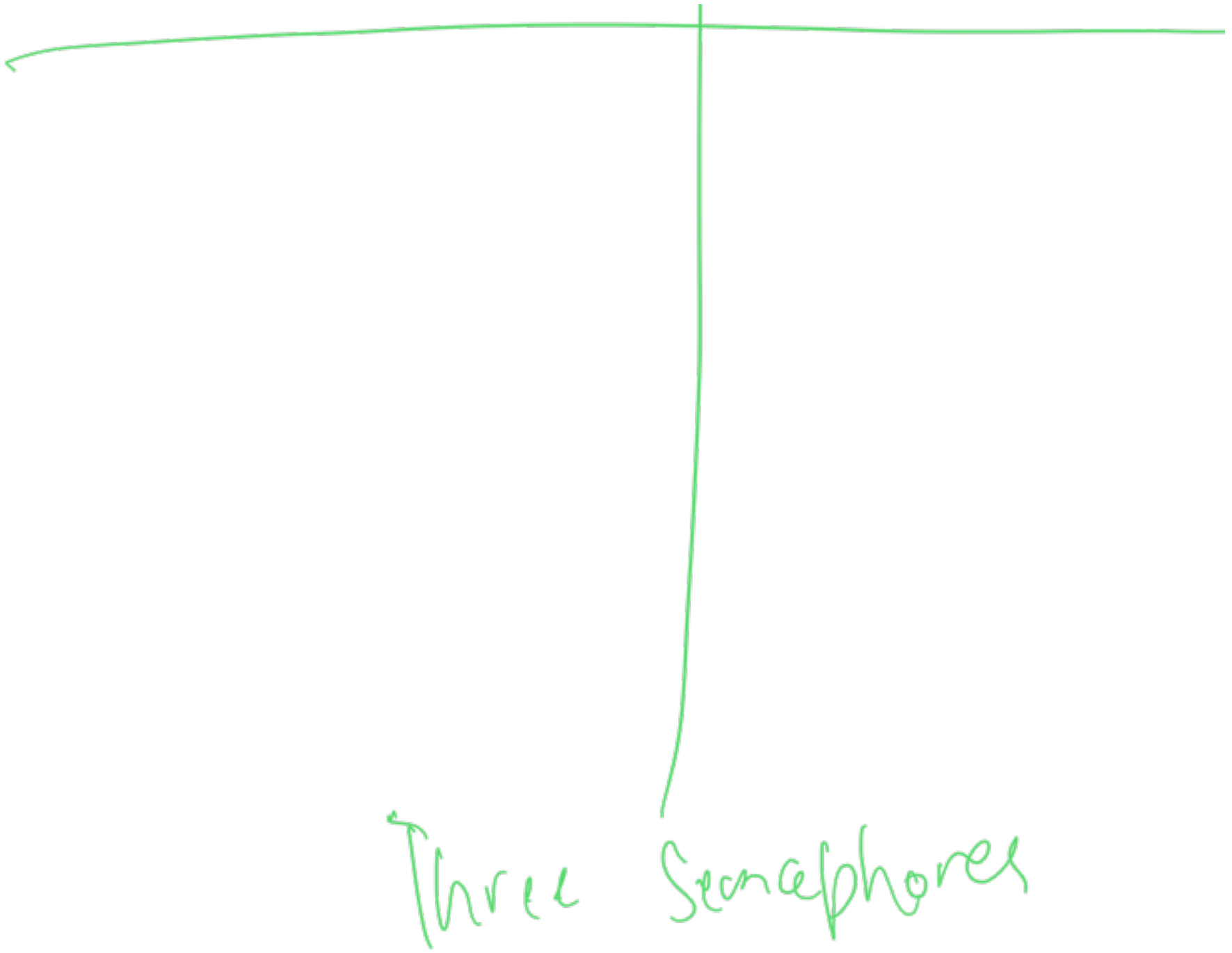
Precedence

A_2 ≤ B_4

wait(s) ⇒ allocation

signal(s) ⇒ Return to pool.

Producer	Consumer


Three Semaphores
full, empty & mutex