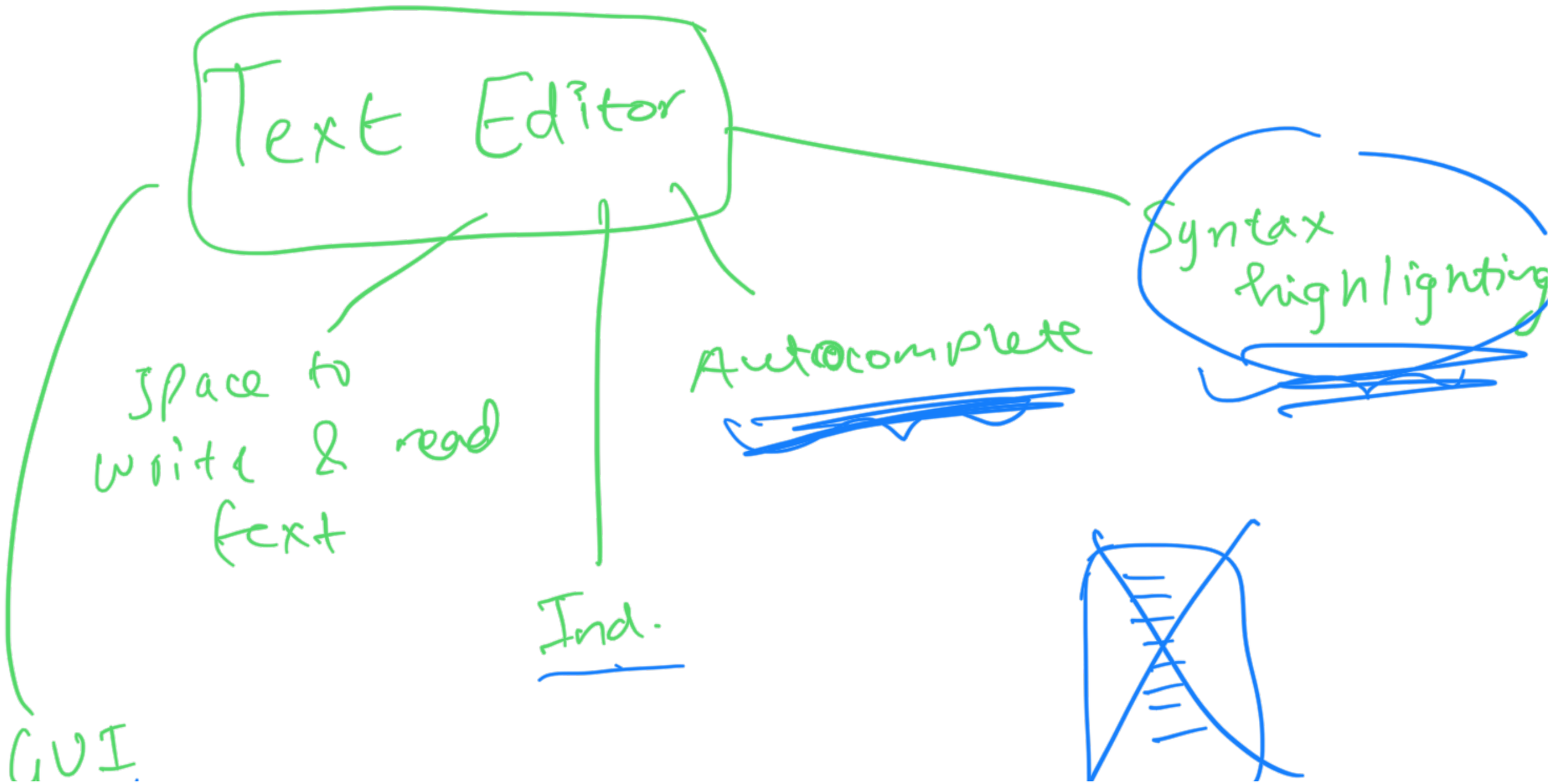
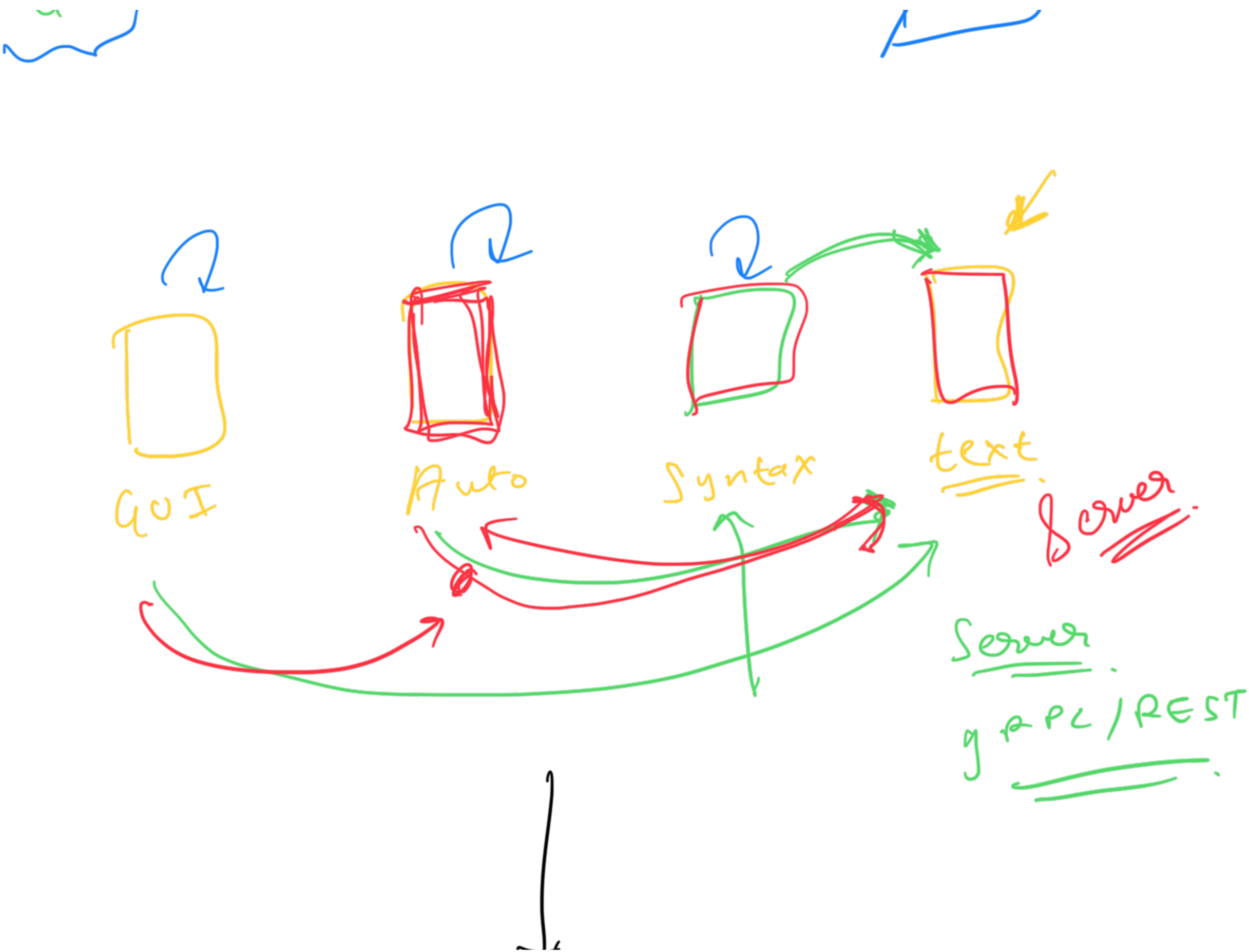
Operating Systems - 3

* Threads.




GUI
Auto
Syntax
text
Server
Server
gRPC/REST

1 process , different tasks

Threads

1 process.

Threads → Shared data

Processes → comp. diff kinds of tasks.

1 process

I/O vs CPU

100 calls to google.com

100 threads

100 threads make
a req to google.com.

100 x++

100

4 threads

25 25 25 25

100

wait for them to finish.

Response.

wait for all of them to finish.

$T_{0,1}$
$T_{1,1}$
$T_{0,2}$
$T_{2,1}$
$T_{3,1}$

2 2 2
2 3 3 3
3
T_0

Max(RTT)

for (i=0 to 100)

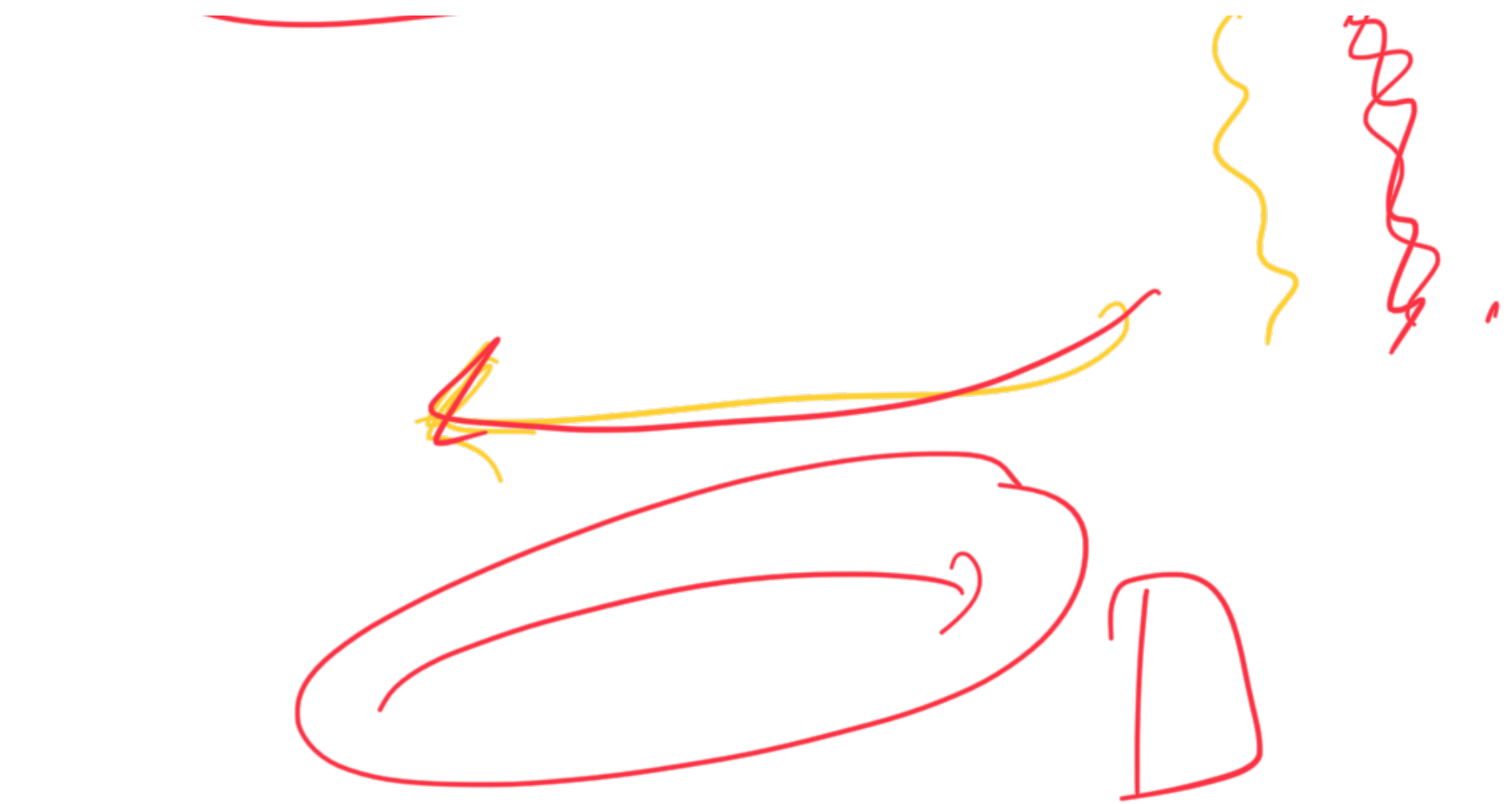
make call to google.com &

get response.

} 100 * (Time for each RT call)

waiting

→ process

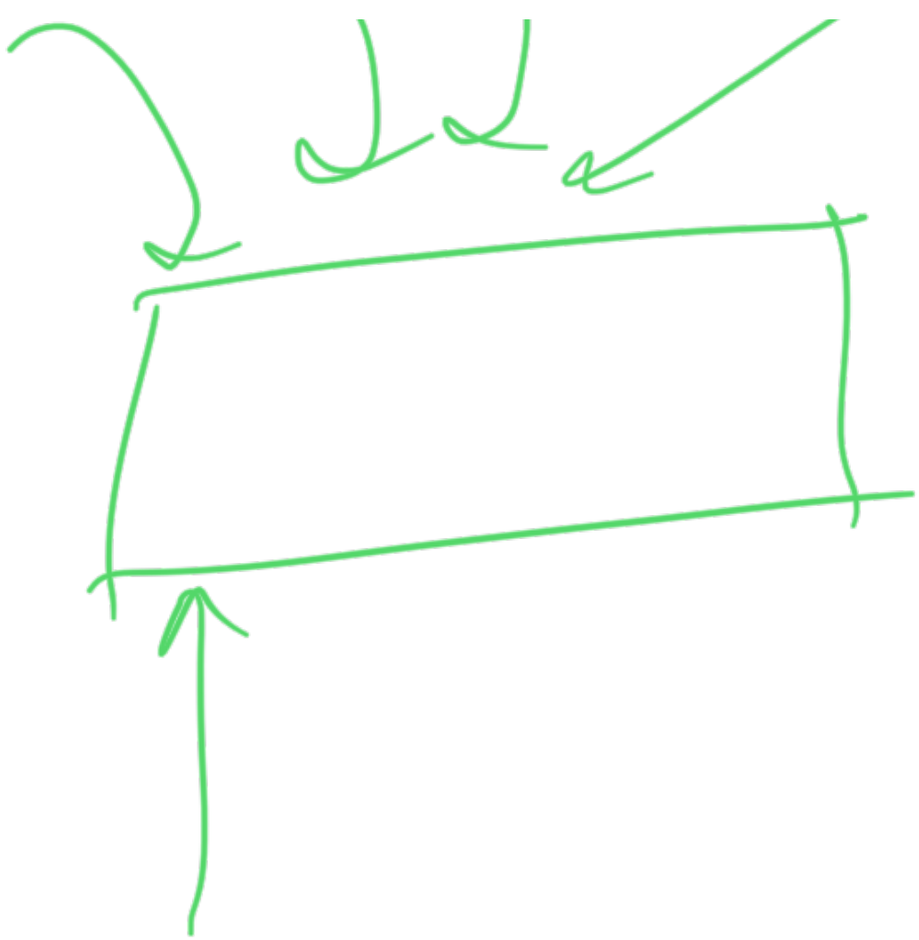


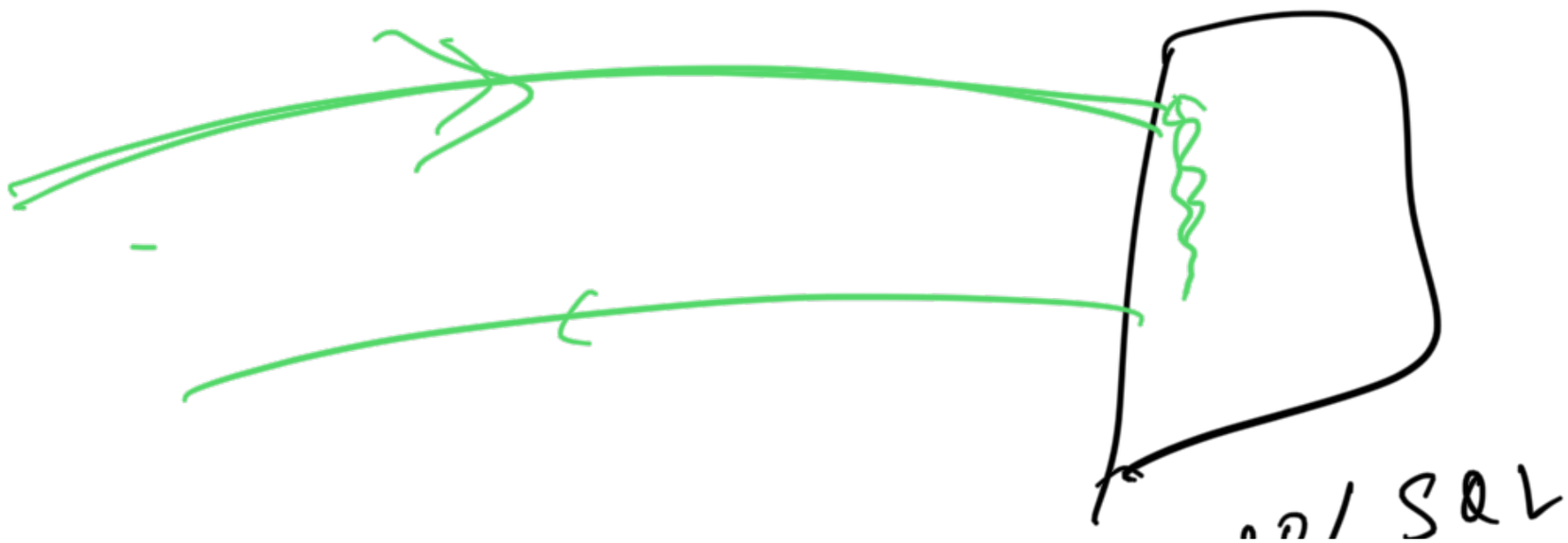
100

Thread pool

T_0 T_1 T_2 . . .

T_{99}





mongo?

25

* Producer - Consumer

Producer

Consumer

while(true){

while(cnt==0);

while(cnt==N);

item=buf[cnt]

buf[in]=item()

cnt++

in++

cnt--

in--

T_0

T_1

* Works most of the time

Sometimes, it doesn't work.

why ??

What addition does BTS.

To

T1

x++

x++

{ 1) load x into a registy

2)

3) write x back from register.

To load x Reg1

T0 load x Reg1

T1 load x Reg2

T0 Inc Reg1

T0 Inc Reg2

T0 Inc Reg1

T0 write Reg1 x+1

T1 ...

T1 ...

To write Reg1 , X

T1 write Reg2) X

T1

X+2

X+1

X

X++ is not an atomic

entry1

To

2

T1

entry2

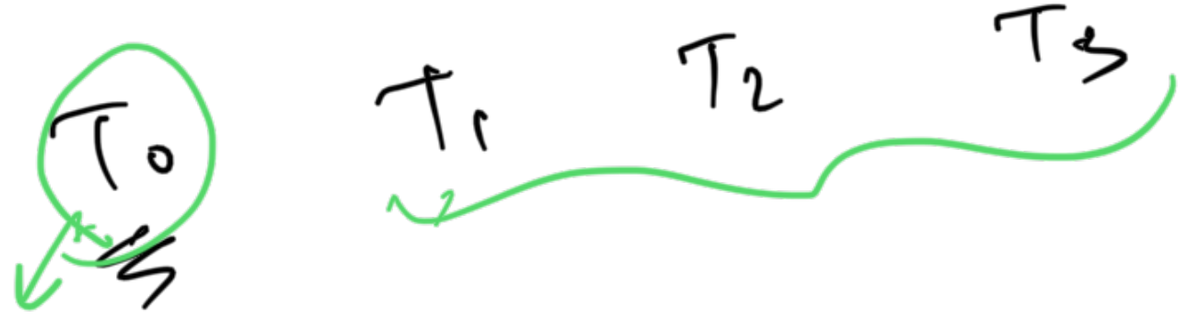
Race condition

* Context switches can lead to data inconsistency..

Mutual exclusion

* At any time, only one thread is in its CS.

Bounded waiting

* 

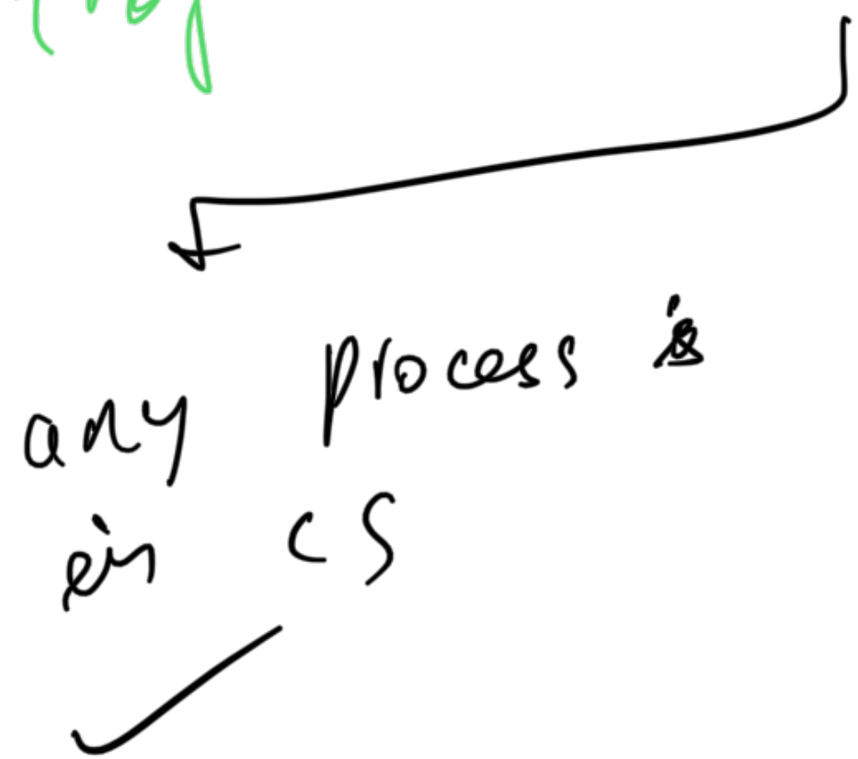
A bound must exist on the # times processes can enter CS

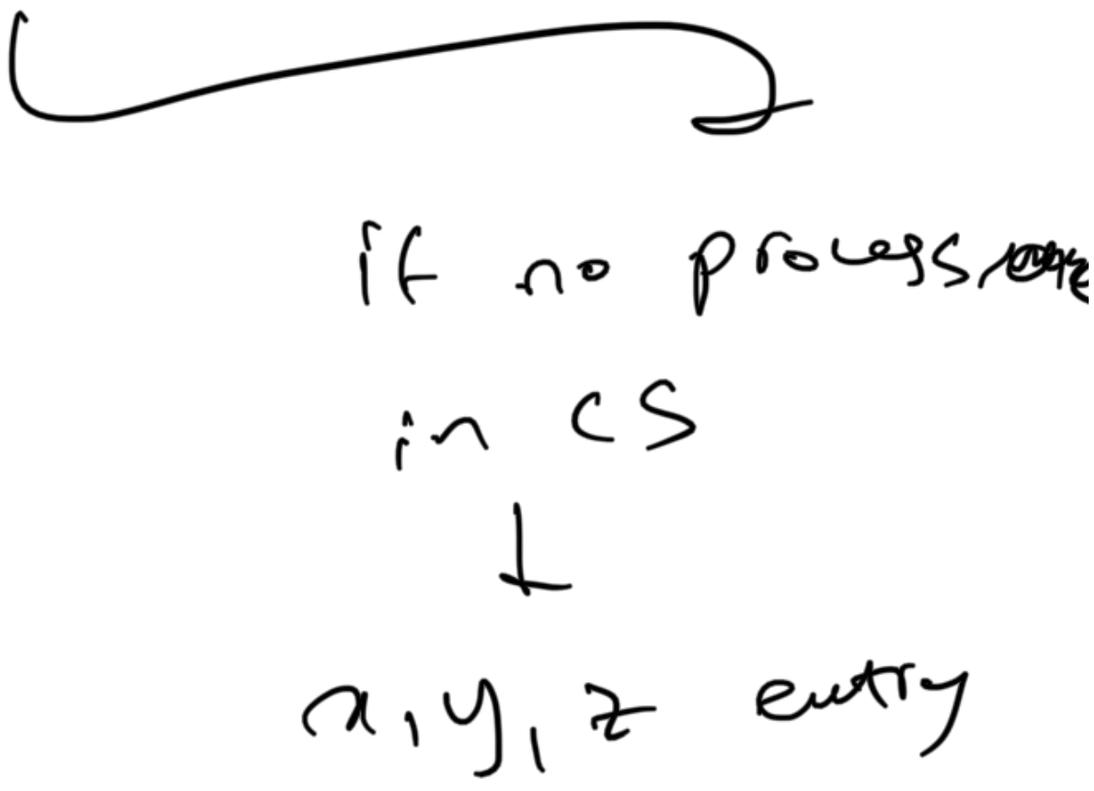
after 1 process has made a

req. to enter & before the request

is granted.

Progress





↳ CS

FINITE TIME

App #1

Loop &

→ turn = 1

→ while (turn == 1);

(CS

Loop &

→ turn = 0

while (turn == 0);

22 ↳ CS

T_0

T_1

T_0 in crit = T_1 in crit = False.

loop d

1 while(T_1 is in crit) ;

loop d

2 while(T_0 in crit) ;

T_1 in crit = T

T0 in crit = T

{ CS }

T0 in crit = F

{ CS }

T1 in crit = F

}

{

Mutex

Peterson's

T_0

$flag_0 = flag_1 = False$

Intent

T_1

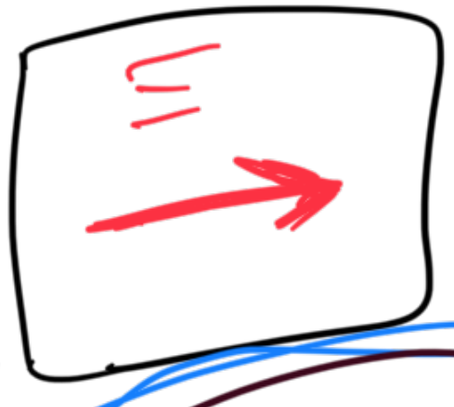
loop {

1. $flag_0 = True$
2. turn → 1

you can go first

while ($flag_1$ && turn == 1) ;

checking if T_1 is running its cs.



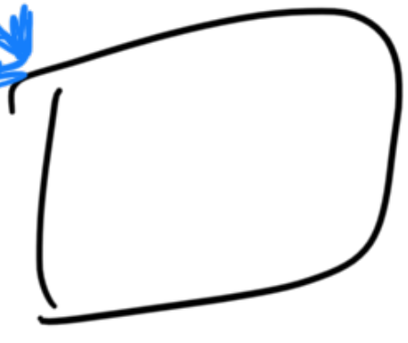
cs_1

$acq.$

loop {

3. $flag_1 = True$
4. turn → 0

while ($flag_0$ && turn == 0);



cs_2

$flag_1 = False$

$flag_0 = False$

No ~~interest~~ intent to run CS

get.

Lock.

acquire()

CS

release()

CAS :- Compare & Swap

H/W

Simultaneously get the value & set it to 1.

lock = 0 ⇒ free.

```
Critical() {
    while( test_and_set(lock) == 1);
    CS
    return 0
}
```

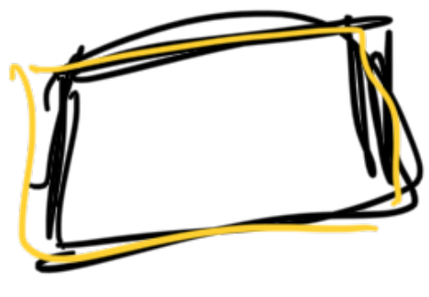
lock = 0

set lock = 1

}

Semaphores

T_0 T_1 T_2 T_3 T_4 T_5



→ 4 threads

}

Wait(S) : if S > 0, then grab it
& decrement it by 1.

Signal(S) : increment S by 1.

S >= 2 ⇒ 2 of S are available.

P_A

- A_1
- A_2
- A_3
- A_4
- A_5

signal(s)

P_B

B_1

B_2

B_3

wait(s)

B_4

B_5

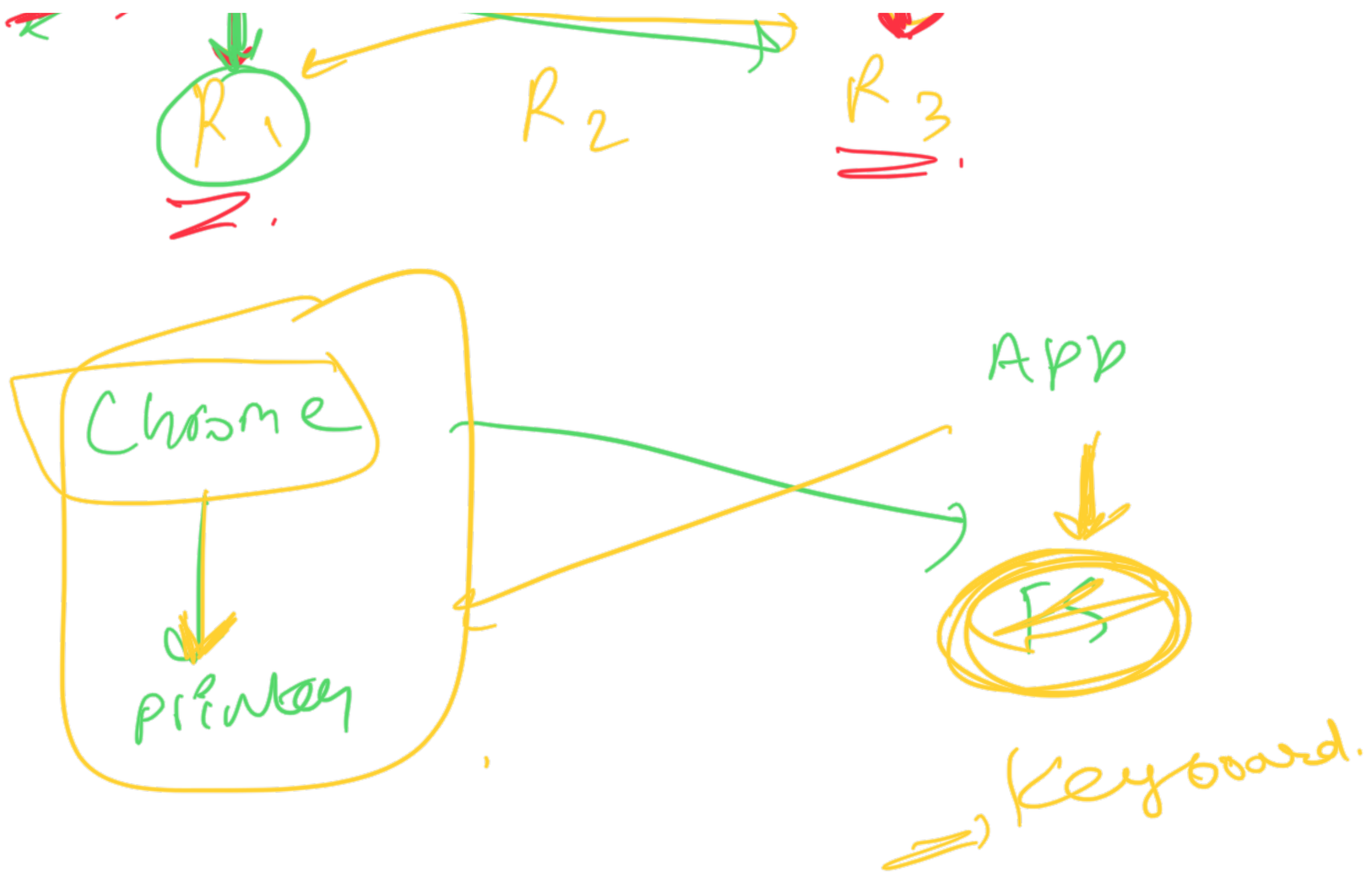
$A_2 \perp B_4$

S = 1

wait(S)

{

Deadlocks


R1
R2
R3
Chrome
print
App
Keyboard.


Ignore