03 Feb 2025 - Operating Systems - II - Week 05

→ Real time CPU scheduling

- → deadlines
- → hard real-time deadline
- → soft real-time deadline

→ event latency

→ dispatch latency : take current process off CPU and switch to another

- → identifying conflicts (other processes complete or quit
- → dispatch

Priority based

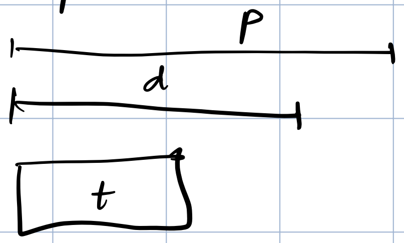
→ Real time scheduling : needs scheduler to support preemptive scheduling

→ but only guarantees soft real time.

assumed here

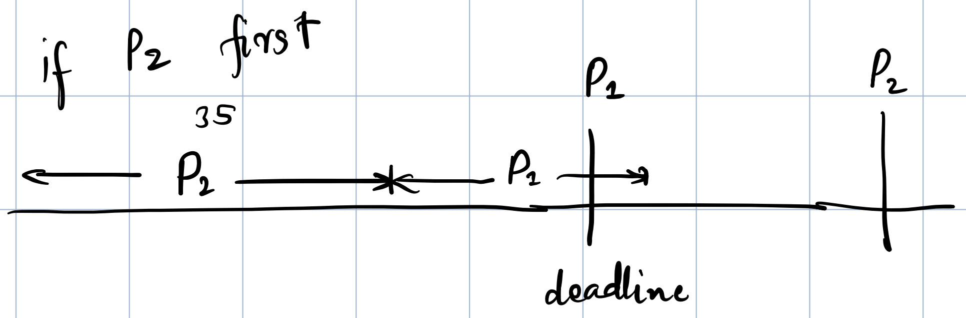
→ processing time t, deadline d, period p

$0 \leq t \leq d \leq p$



	$p = d$	T
P_1	50	20
P_2	100	35

come at same time



→ shorter period first

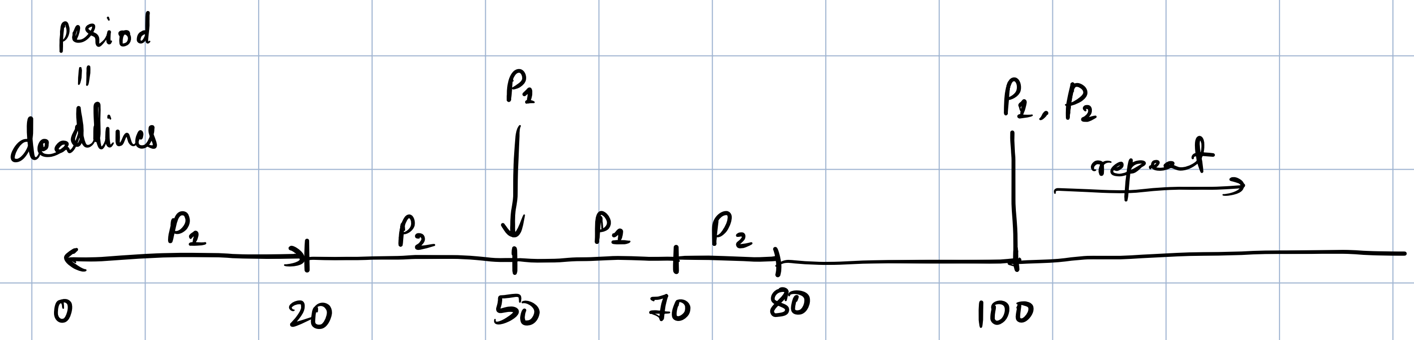
$$\text{utilization} = \frac{t_i}{p_i}$$

$$P_1 : \frac{20}{50} = 0.4$$

$$P_2 : \frac{35}{100} = 0.35$$

$$\left.\begin{matrix} 0.4 \\ 0.35 \end{matrix}\right\} 0.75$$

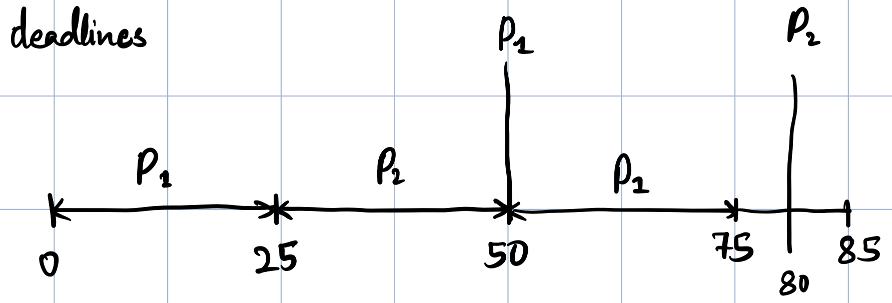
→ smaller period process has higher priority } rate-monotonic scheduling



	$p = d$	T
P_1	50	25
P_2	80	35

} cannot be accomodated

P_1 higher priority



utilization : $P_1 = \frac{25}{50} = 0.5$

$P_2 = 0.4375$

total ≈ 0.94

→ despite being optimal, → CPU utilization is bounded,

$$\max = N\left(2^{1/N} - 1\right)$$

- try to derive

$x = \lim N(2^{1/N} - 1)$

$N \to \infty$ max utilization = 69%

if utilization $\geq$ max → deadlines will not be met.

Disadvantages

→ optimal, but static priority

→ periodic

non-periodic $\rightsquigarrow$ can occur any time

Earliest deadline first

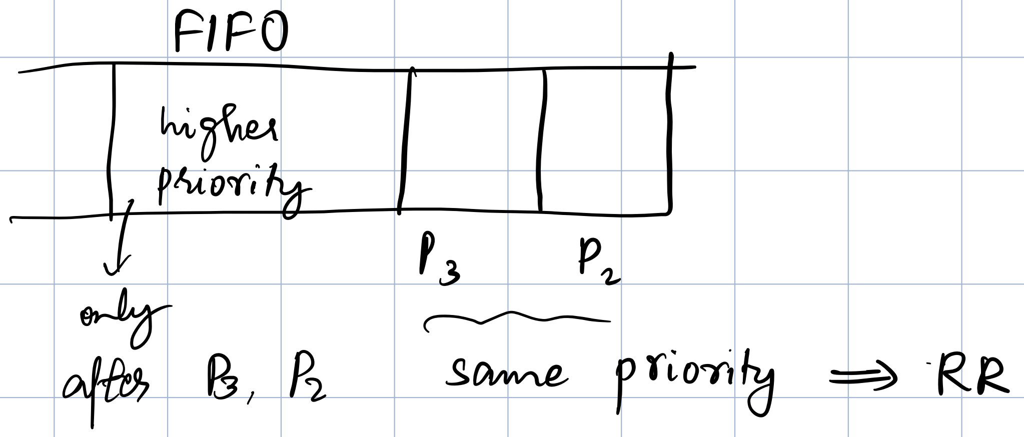
earlier the deadline, higher the priority
later the deadline, lower the priority.

- diagram

→ disadvantage: keep computing priorities.

Proportional share scheduling

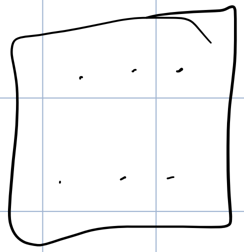
Posix real time scheduling



Scheduling in Linux

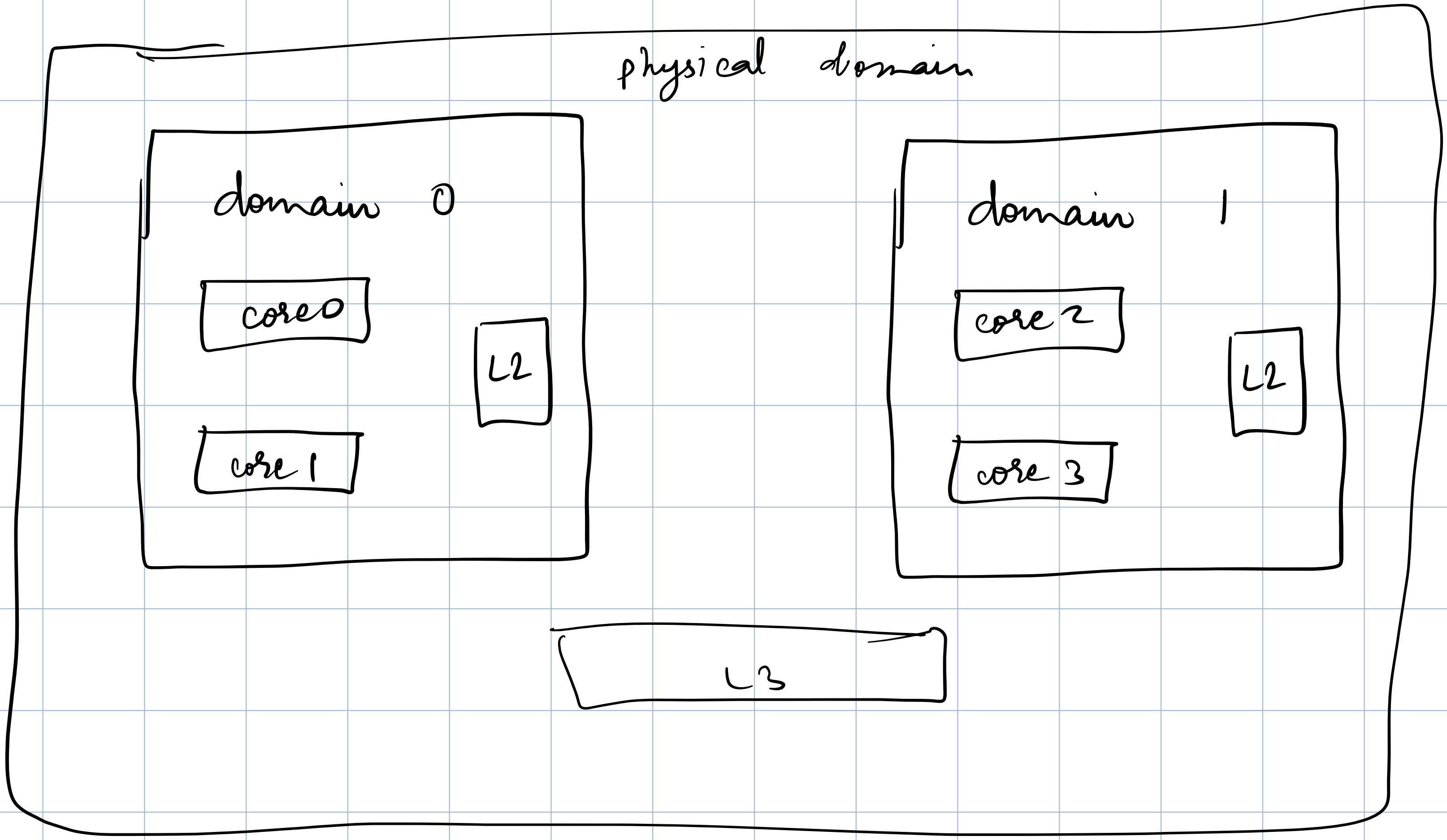
→ CFS (completely fair scheduler)

→ Scheduling classes



NUMA aware

→ far away core context - switch should be avoided.



Scheduling
Algorithm Evaluation

→ Deterministic evaluation : calculate minimum average waiting time.

→ theoretical

→ may not capture reality → may not work for all inputs

→ Queuing Models

→ develop models : describe arrival of processes and I/O burst probability

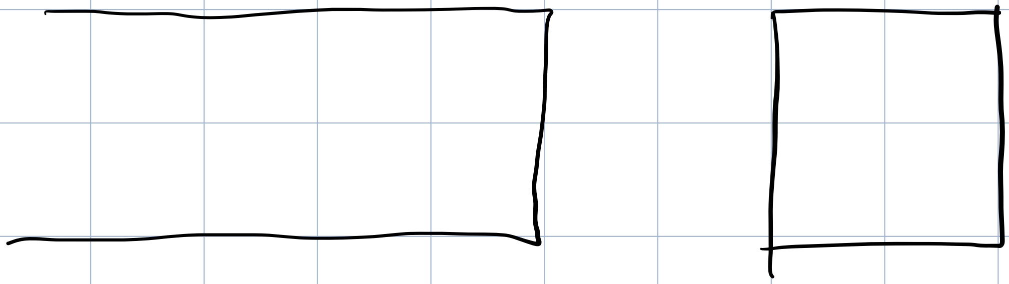
→ some limitations :

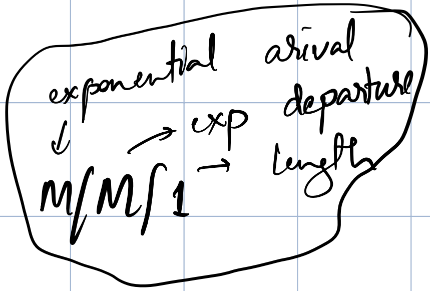
– still an approximation

Little's formula

n = average queue length $\qquad \lambda$ = arrival rate

W = average waiting time per queue $\qquad n = \lambda \cdot W$





04 Feb 2025

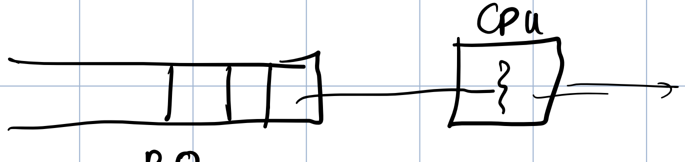
→ Assigment due 11 Feb

→ Quiz on 14 Feb 6:30 PM Chapters 4, 5

① Examples

② Models

③ Simulation → programmed model of a computer



Scheduling when

① new job

② RQ → CPU

③ CPU → RQ

④ CPU → Devs

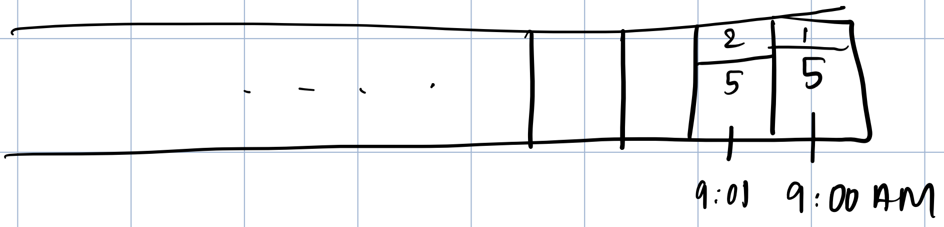
⑤ CPU → Term

→ clock is a variable

Discrete event simulation

struct ~~task~~ event {

id, time }

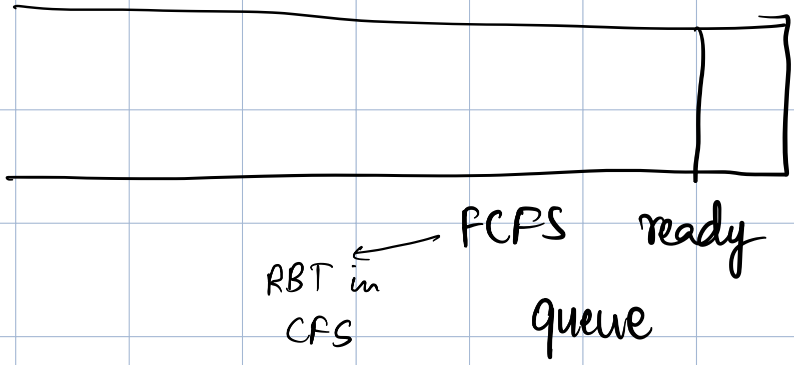


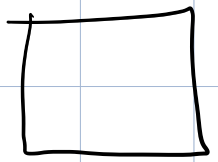
every timer interrupt is an event

Event queue → RBT, priority queu

→ some input events

→ each event triggers more events





clock is a variable

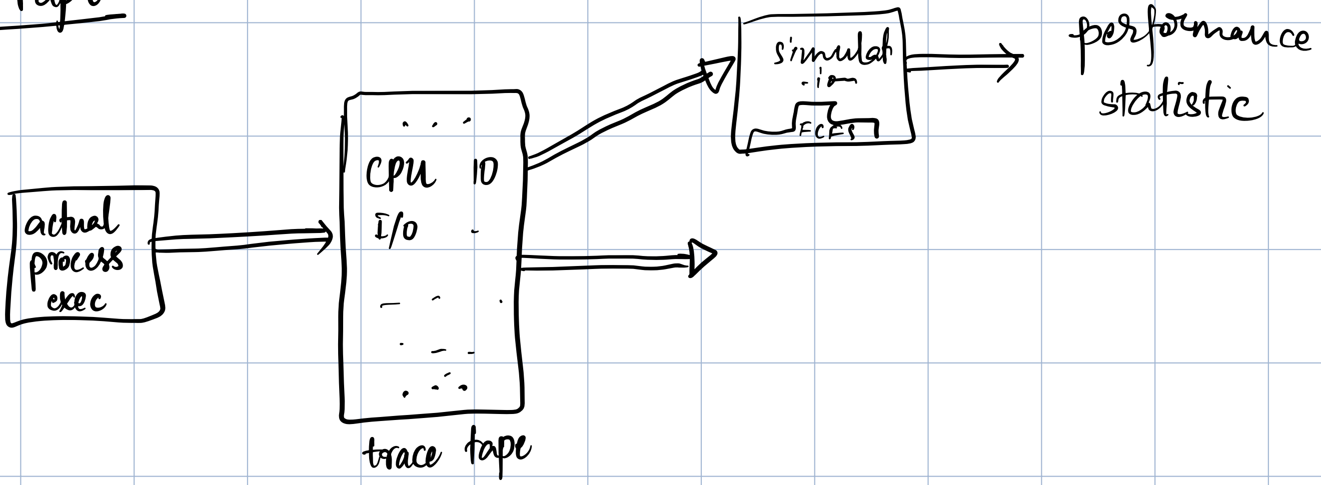
clock ticks on every event

Event queue

↳ subclasses: process, interrupt etc.
in

PCBs ≠ events

trace tape



drawback : logging time , space

service time affects arrival time.

→ still doesn't capture reality

④ implementation

→ high risk

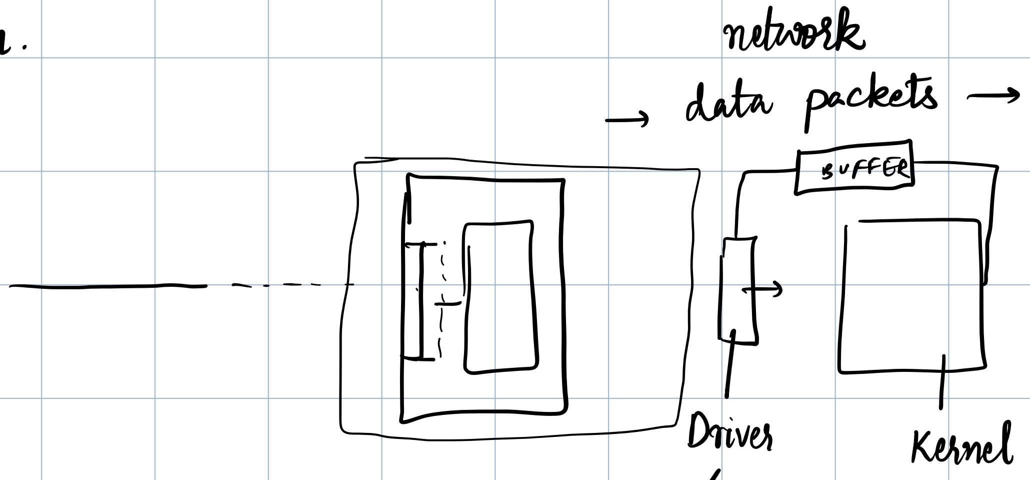
→ ideal schedules → tunable

→ more for real time systems
banks , etc.

05 Feb 2025

→ Producer consumer problem.

- → network
- → spellcheck

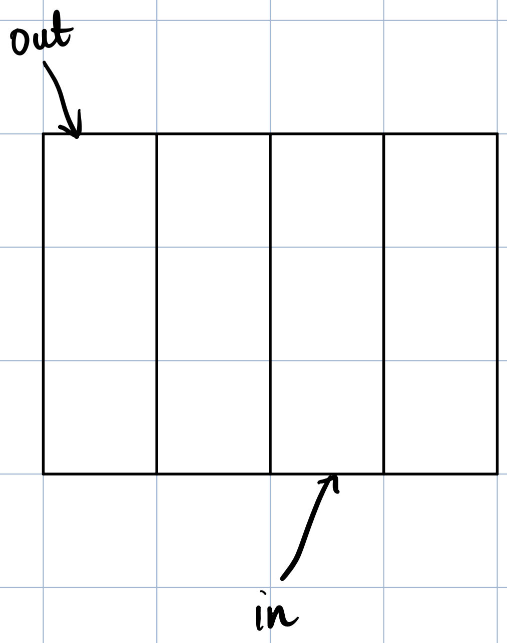


Asynchronous
- → Driver keeps pushing data into the buffer
- O.S. keeps reading the buffer

Driver ↙ specialized software for interfacing a device
→ hides hardware intricacies

Producer → must overwrite only those entries read by consumer

Consumer → must not read empty buffer



5
9 → critical section

6 − 8 → critical section if multiple producers

Producer

```
3 while (true) {          // assume packets keep coming
4         /* produce an item in next_produced */
5         while (counter == BUFFER_SIZE)      } keep spinning
6                 ; /* do nothing */          }
7         buffer [in] = next_produced;
8         in = (in + 1) % BUFFER_SIZE; 9 counter++; }
```

Consumer

- code

→ Both producer and consumer cannot be stuck in the while loop

o Deadlock

Race condition

~> common variables : counter, buffer

counter++	counter --
register1 = counter	register 2 = counter
registers 1 = register 1 + 1	register 2 = reg2 - 1
counter = reg 1	counter = reg 2

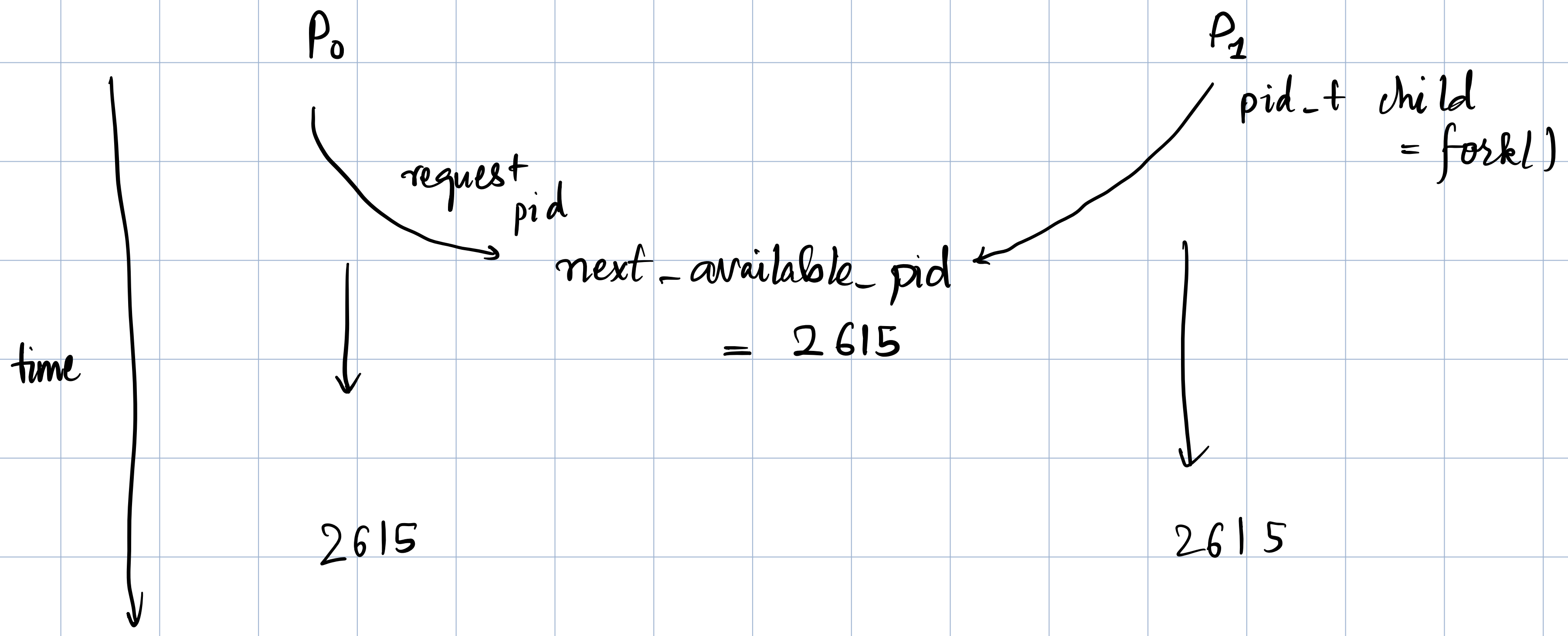
Consider counter = 5

example slides

We think sequentially

More than 1 producer threads ⟶ race conditions : in , buffer, counter

More than one consumer thread ⟶ race condition : out , buffer, counter



Critical Section Problem

- how to design a protocal to avoid this?

① while loop

② sleep

→ push out of CPU , PCB to some other quene

→ ready quene

general structuse :

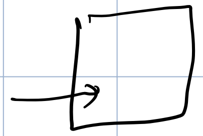
```
do {
    entry section

        critical section

    exit section  → allow only one anothes

        remainder section

} while (true);
```



Kaushal → sperm

if one thread is in critical section, no other thread should be in the respective critical section.

Solution

① Mutual exclusion

② Progress

③ Bounded Waiting

Without ②, point ① is meaningless
(lock the room for all)

without ③, process can starve

→ Assume that each process executes at a non-zero speed } → assume no core crashes
benign

malfunction
malignant

→ No assumption concerning relative speed of n processes

→ round robin

~> thread needs to fetch

4th) Fairness

- → preemptive
- → non-preemtive → easy in single core

Peterson sol^n

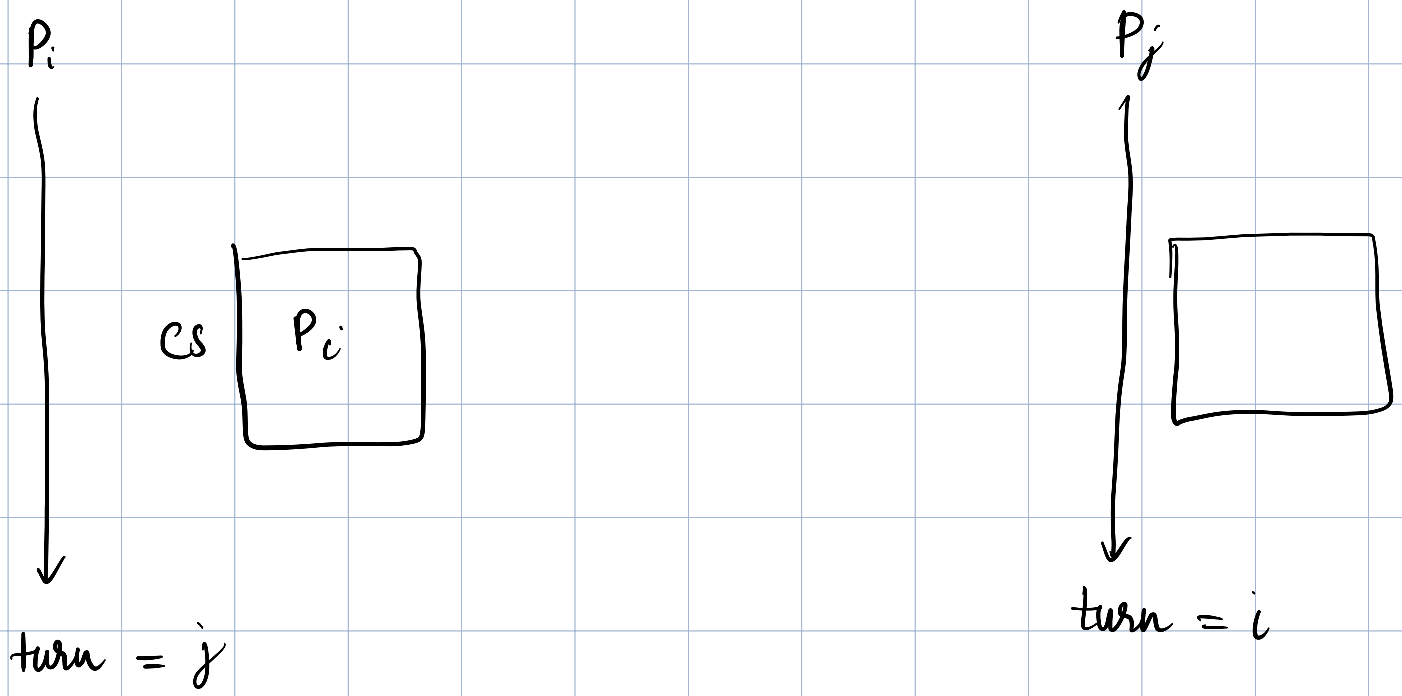
- → not for modern systems
- → two processes
- → assumes load and store operations are atomic

  somehow magically

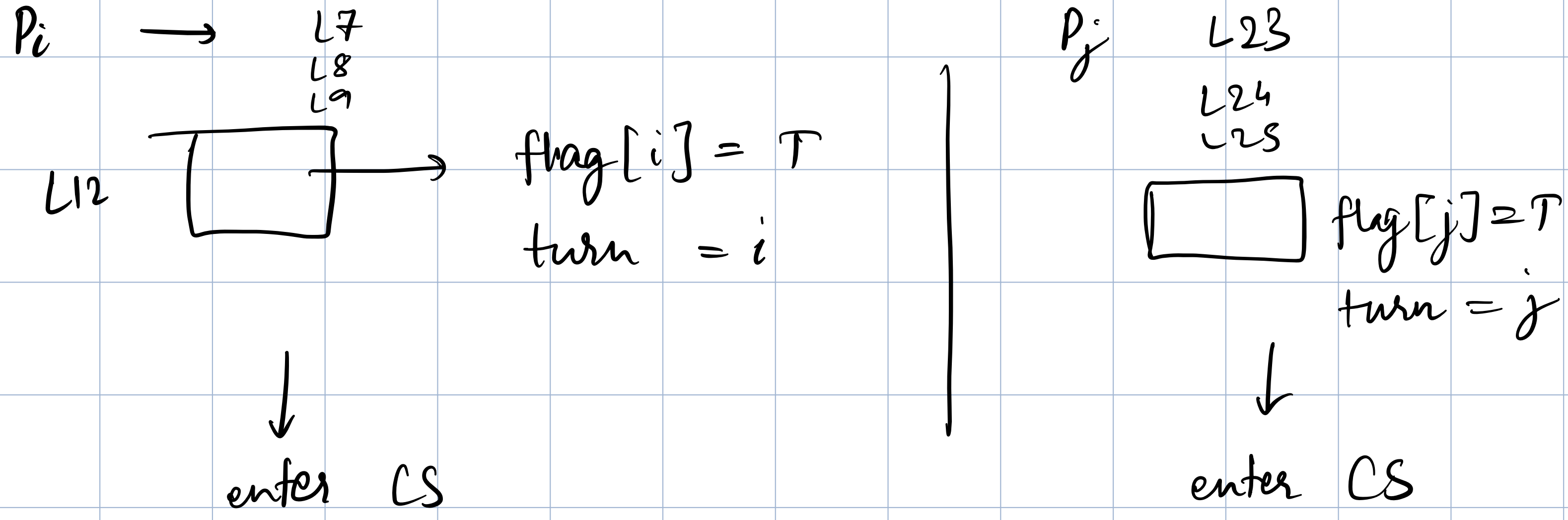
  cannot be interrupted.
- → share 2 var :

```
int turn ;
boolean flag [2];
```

peterson cpp → see book or whatever



If P_i sets turn to i
P_j sets turn to j } → Both can reach critical section



* provable that the three CS requirements are met.

* Although useful in understanding, it won't work
  - Compiler optimization will break things

How to solve problems due to compiler optimizations?
cache

- Take hardware support

→ Memory barrier

Next class : Hardware instructions