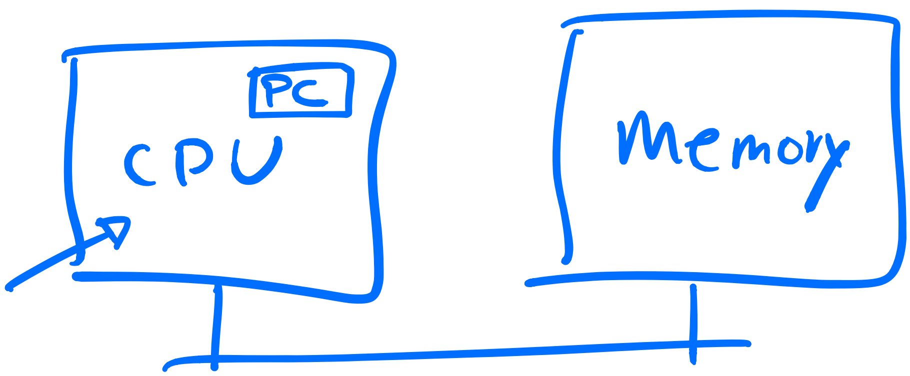
Today

→ Review (last time)
CPU virtualization
[→ mechanisms]
→ policies (scheduler)

Background: CPU




```
while (1) {
    Fetch (PC)
    Decode : Figure out which
             inst is
    (inc. PC)
```

Execute: (could change PC)

3 Process Interrupt(s)

before execute: modes: user, kernel

check permission:

is this inst OK to exec?

if not OK: raise exception

=> OS gets involved (exception handler)

Handle pending interrupts

=> OS gets involved

CPU virt. mechanisms:

OS: implement Limited Direct Execution

@ Boot time: OS runs first
(privileged mode)
↑ "kernel"

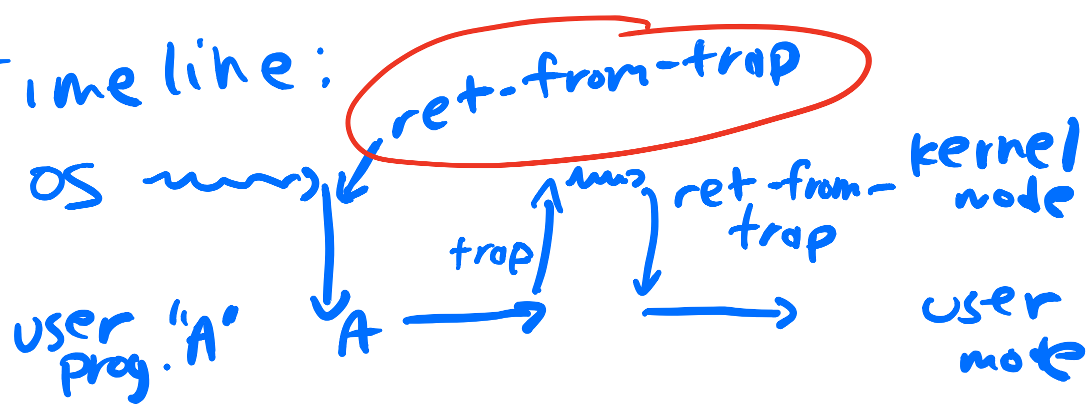
① => install "handlers" (code)
(tell H/W what code to run on exception/interrupt/traps)
[done by privileged inst.]
x86: lidt

unpriv/restricted
⇓
"user"

② => init. timer interrupt
[privileged]

Ready to run user programs

Timeline:



1) A wants OS service
(system call)
issue special instructions:
=) trap instruction
(x86 called "int")

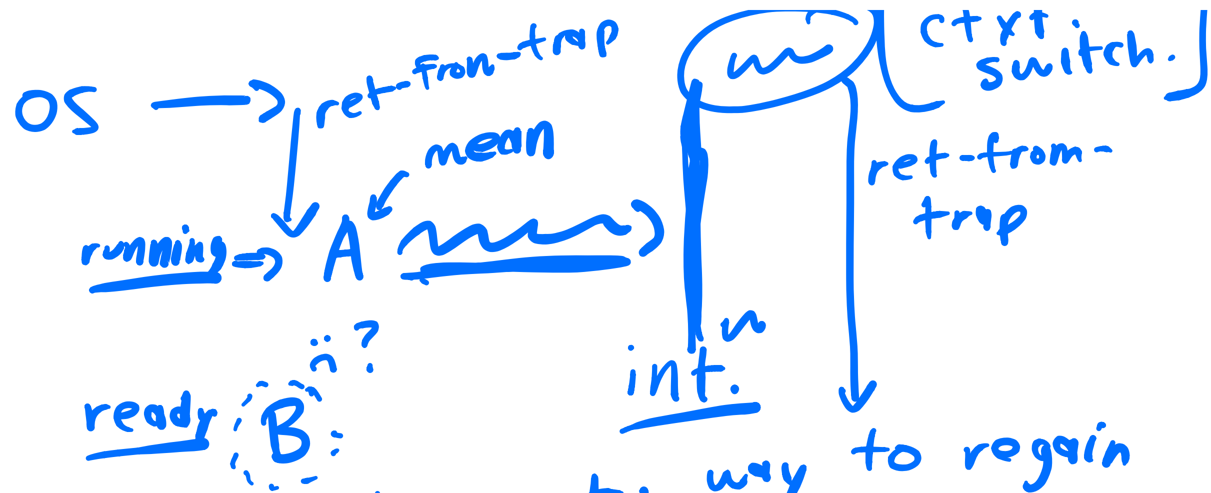
=) transition
user mode =) kernel mode

=) jump into OS:
target: trap handlers

=) save register state
(so as to enable resume exec. later)

OS sys call handler: runs

=) ret-from-trap
(opposite of above)



timer interrupt: way to regain control of CPU (by OS)

=> OS handler

=> can switch to diff process

"**context switch**"

Part 2: CPU virt. (mechanism)

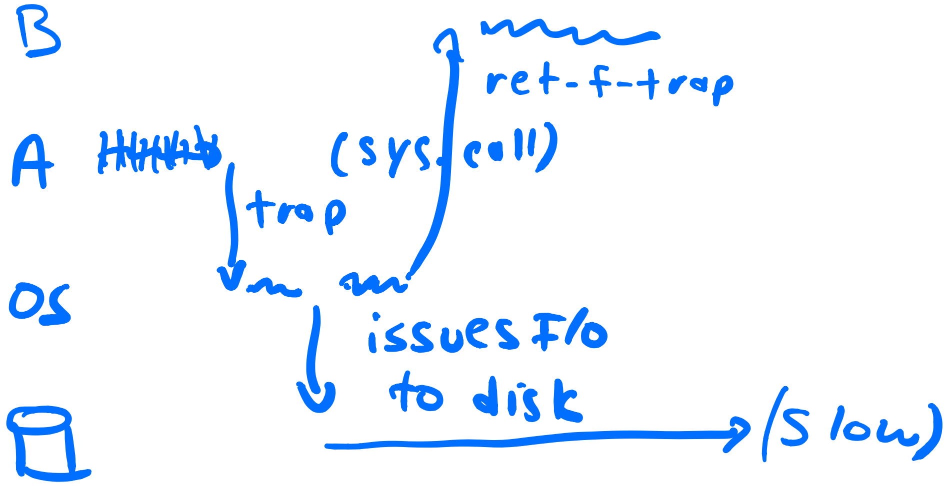
OS: must track diff. user processes

["**Process List**"] => in xv6 (future)

└ per-process info:

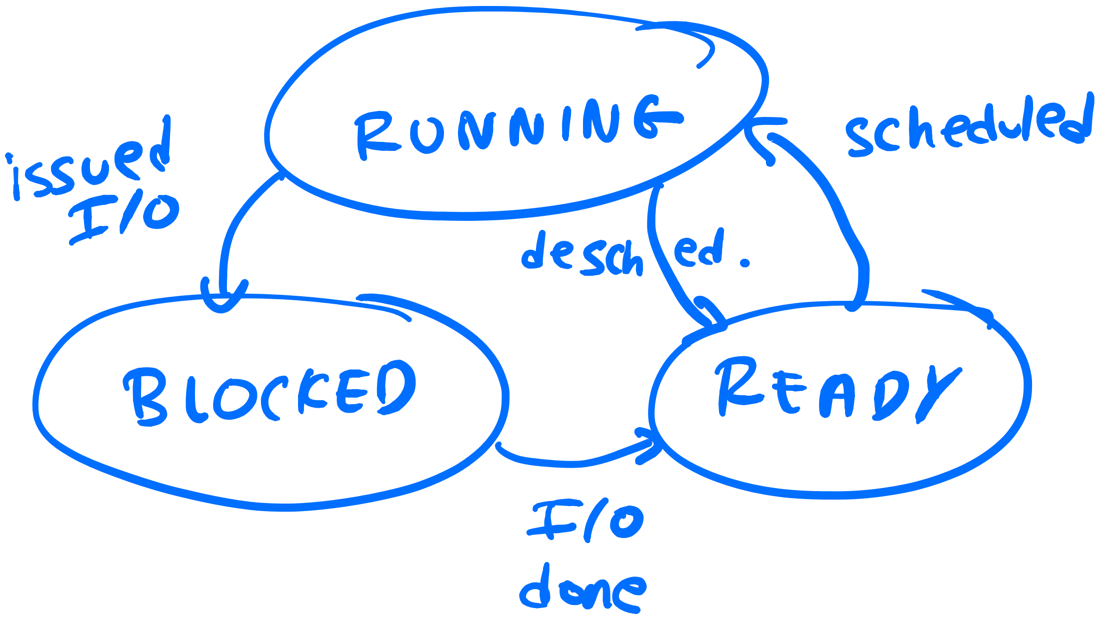
=> "state": READY, RUNNING

Problem #3: slow operations (I/O)



allows: better util. of CPU (overlap)

states: READY, RUNNING, BLOCKED (on I/O)



OS : tracks

=> goal : efficiency

Summary :

Limited → security

Direct Execution → efficiency

Mechanisms => Policies

OS: Policy

youtube in
100 years?
(IS ANYONE
LISTENING?)

2118 : wow!

=> flying cars

=> watching on
your brain

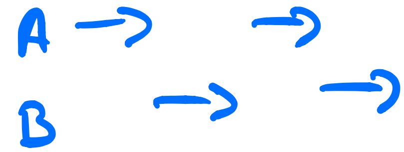
=> on mars!

=> maybe?
self-flying
"cars"

=> instant
language
translation

=> (remzi is
everyone's
teacher)

OS : Policy



How? ✓

what??

Assumptions : { each process : "job" }

⇒ (set of jobs, all arrive @ same time)

⇒ each job only uses CPU (no I/O)

⇒ each job runs some amt of time T_i

T_i is same for all processes (known ahead of time)

⇒ (Metric : turnaround time
time end − time arrived)

(complete)

Ex:

A, B, C: all arrive at $T=0$

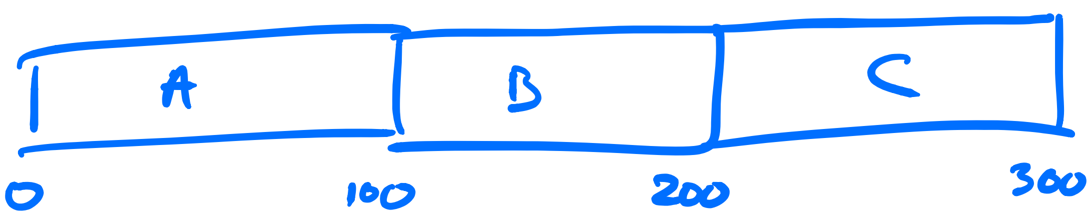
run time: 100 time units

Algorithms: FIFO (first in first out)

(aka FCFS) first come, first served

(A, then B, then C)

@ first: run to completion



$T_A = 100 - 0 = 100$

$T_B = 200 - 0 = 200$

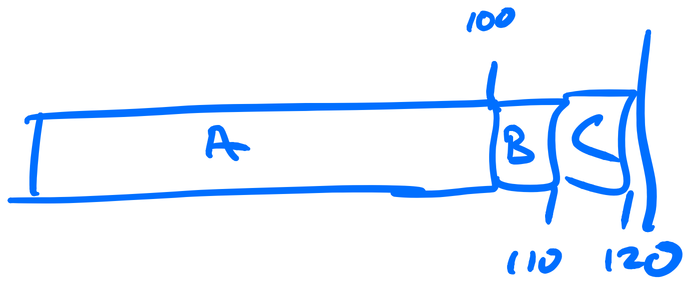
$T_C = 300 - 0 = 300$

T_{AVG}: (200) math

Relax: ~~[All jobs have same runtime]~~

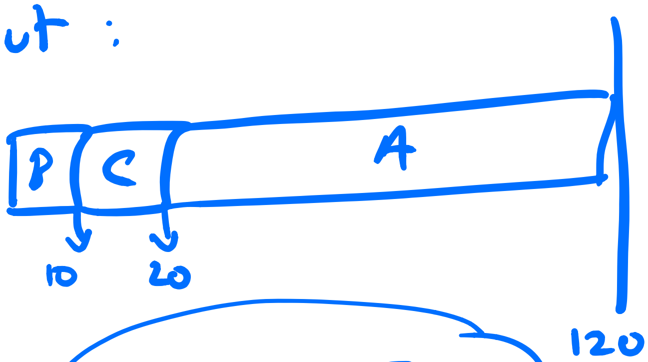
- → A : 100
- → B : 10
- → C : 10

pick runtimes so that FIFO looks **bad**



$T_{AVG} = 110$

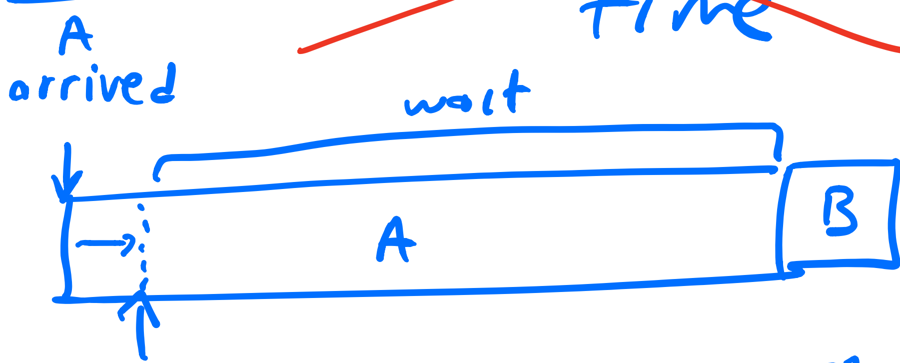
but :



$T_{AVG} = 50$

⇒ goal: schedule shortest job first (SJF)

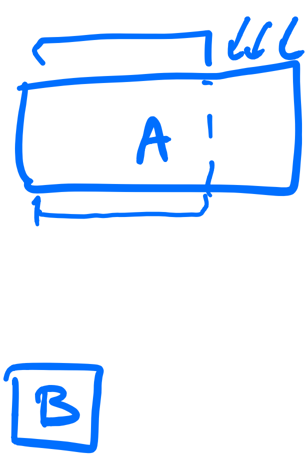
Relax: ~~all arrive @ same time~~



B arrives: B run-time short (10)

generalization:

shortest time to completion first (STCF)



new:

preempt job

(in some cases, stopping existing, starting another)

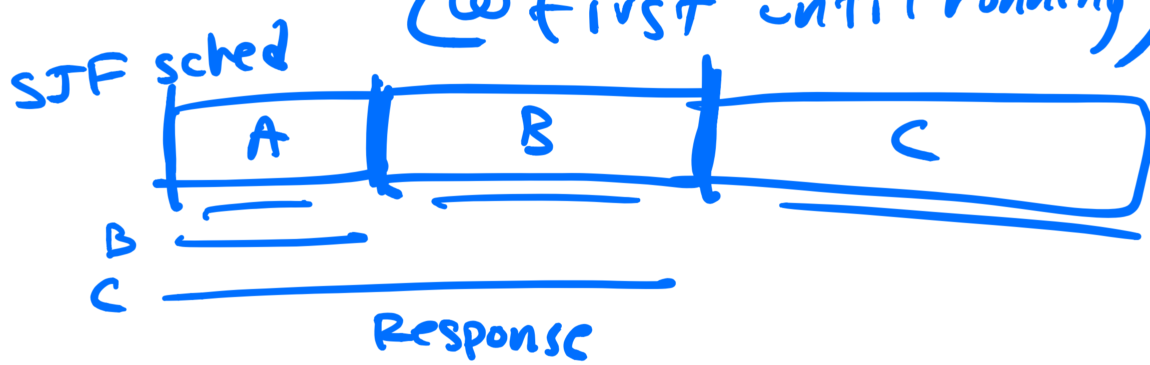
New metric: [Response time]

some define: time until process generates a "response"

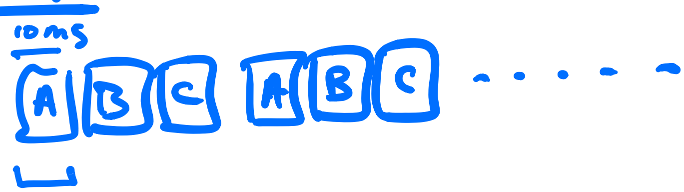
our definition: (response time)

$time_{first\ runs} - t_{arrives}$

(how long it waits @ first until running)



Policy: Round Robin



quantum, time slice: 10ms

(multiple of
timer interrupt period)

trade off:

short time slices:
better response time,
but high ctxt
switch overheads

longer:
worse response,
more efficient

idea: ~ SJF (STCF)
but, we don't know job
lengths

and
responce time (not just
turnaround)

=) how to build real
scheduler?