CSE 531: Distributed and Multiprocessor Operating Systems

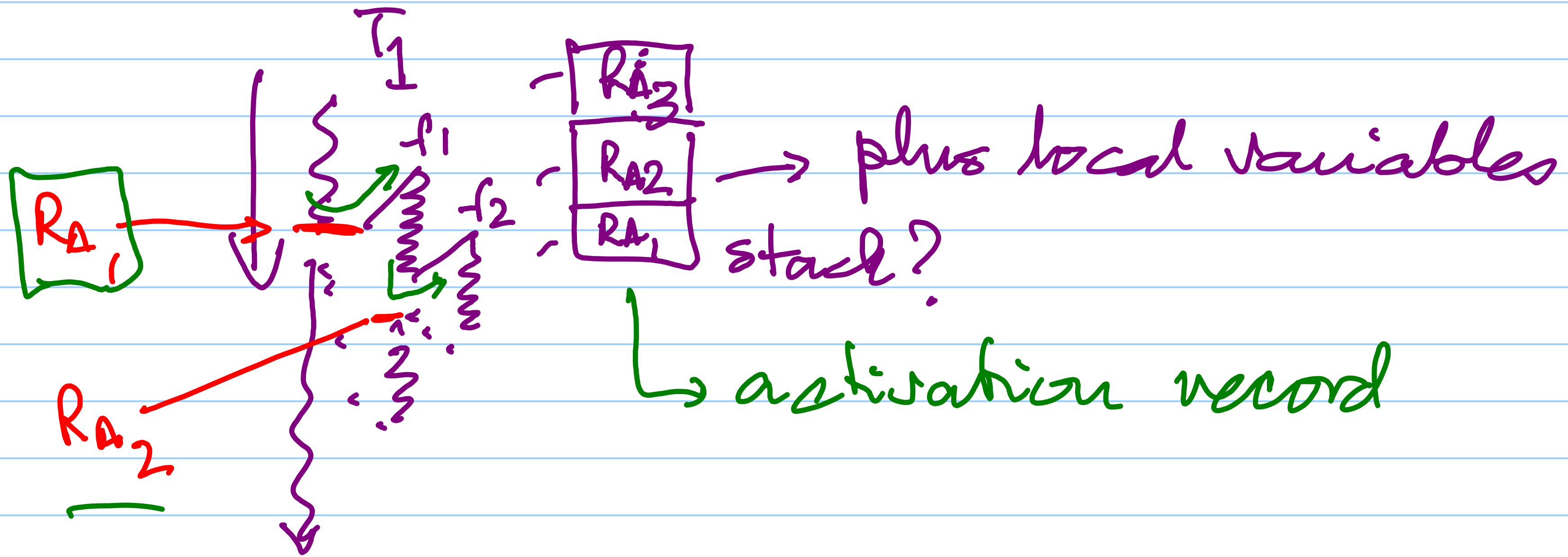
Fall 2020

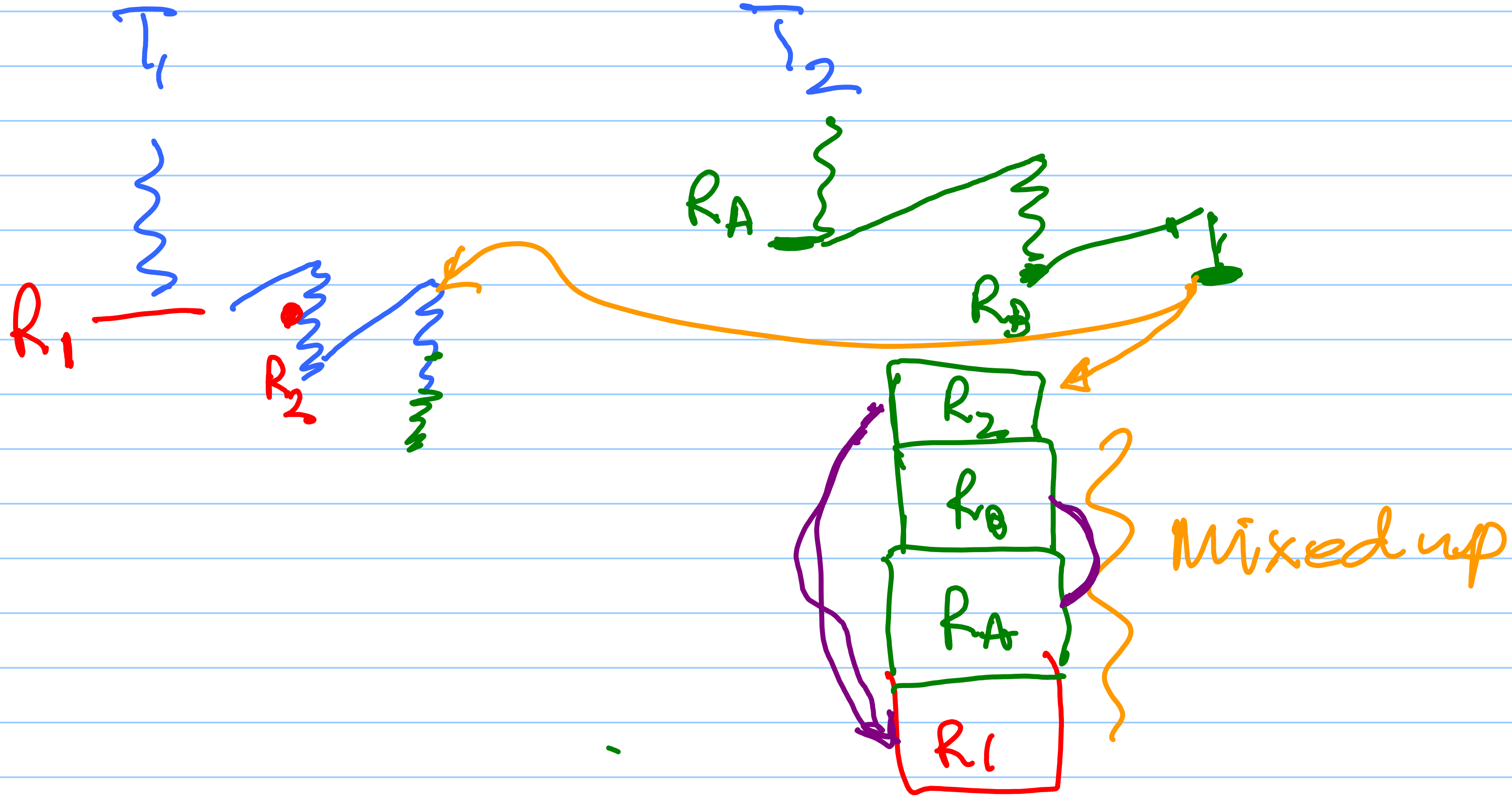
Class: 12 Date: 10/5

Note Title

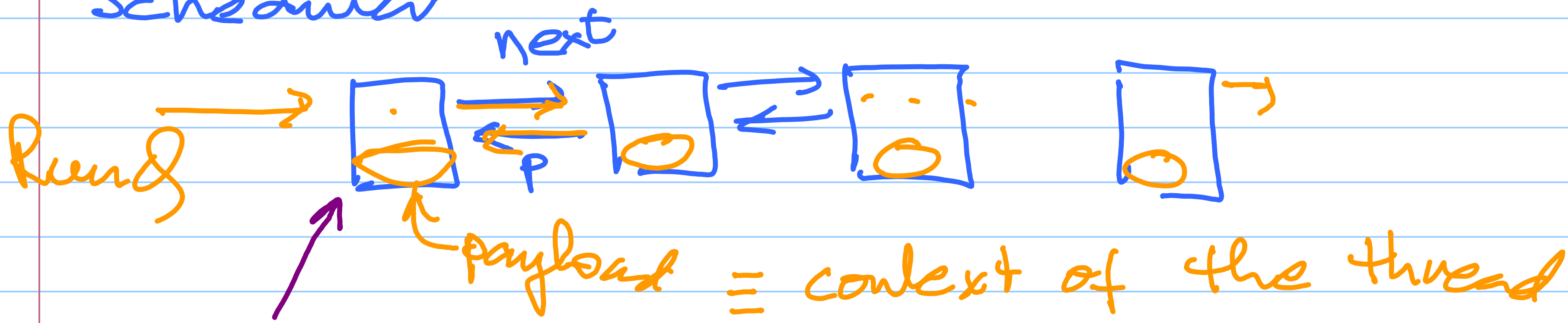
Coroutines / threads

- need separate stacks




T1
T2
R1
R2
RA
RB
R2
RB
RA
R1
Mixed up

Scheduler

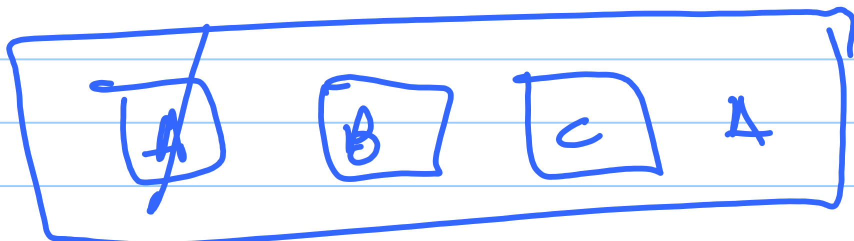


Running

yield → • save context. top of Q

• del top, add it @ end

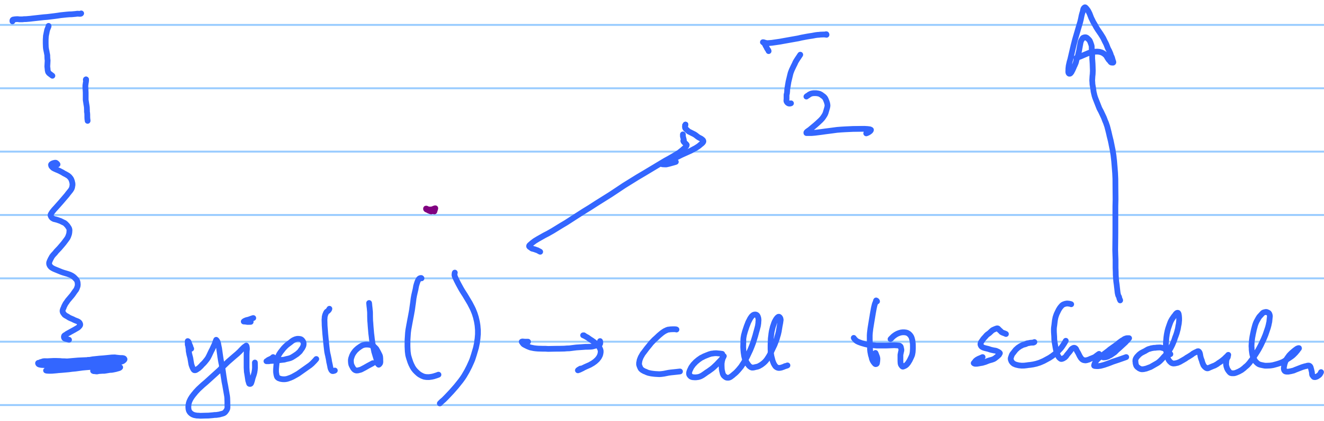
• load context from top



head

A B C

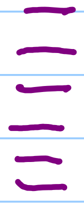
- non preemptive scheduling
  - ↳ A thread executes till it requests to be blocked

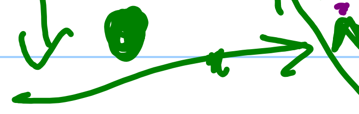


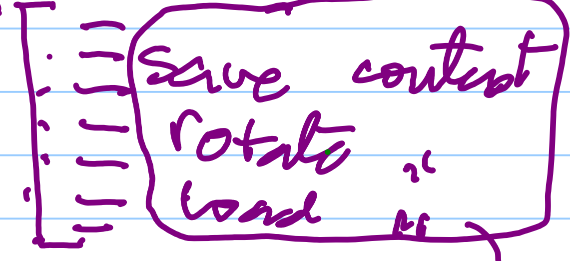
User implemented threads

- non preemptive - easy
- pre emptive - possible

Control flow in non preemptive scheduling





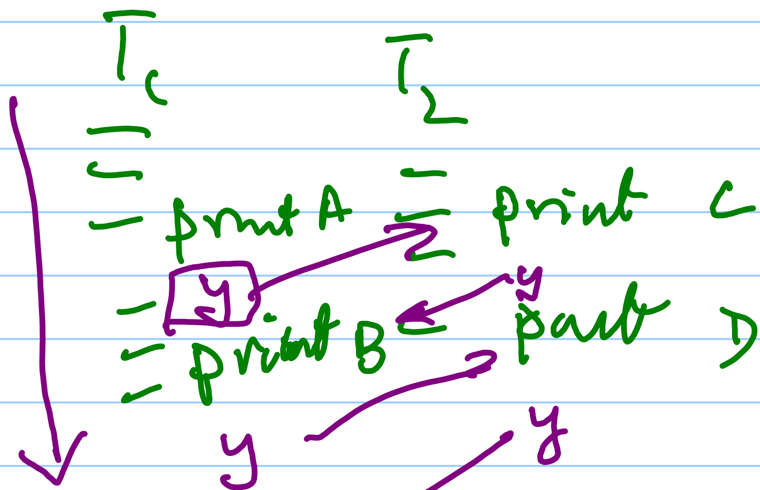


T_1
yield()
save context
rotate
load
return
SP
CPU
T_2
yield()
yield
T_3
yield
return from the call to yield in T_1.

non-pre → deterministic

@ yield points

→ follows the same path for every execution



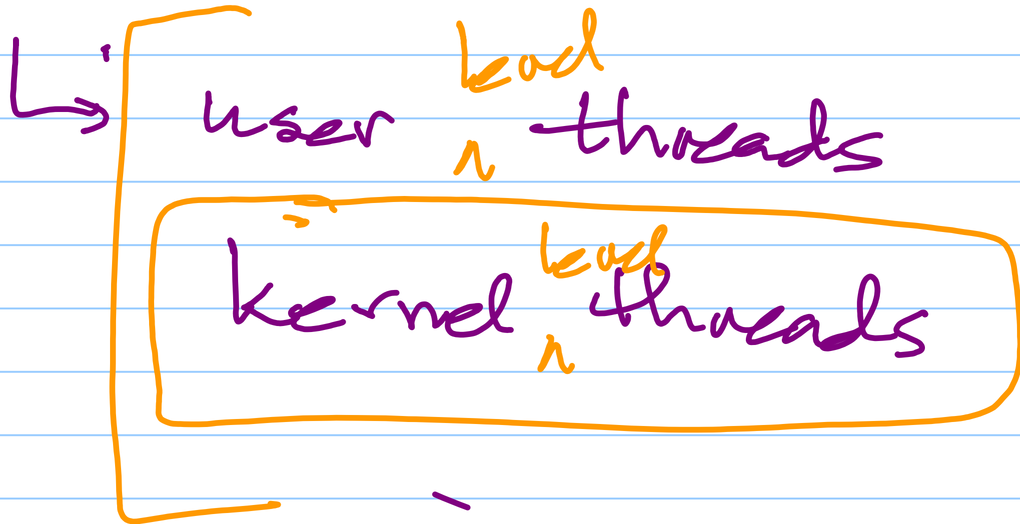
AB CD

A C B D

A C D B

→ A C B D

- user level scheduling
- kernel level scheduling



User level sched

kernel " "

→ takes one thread

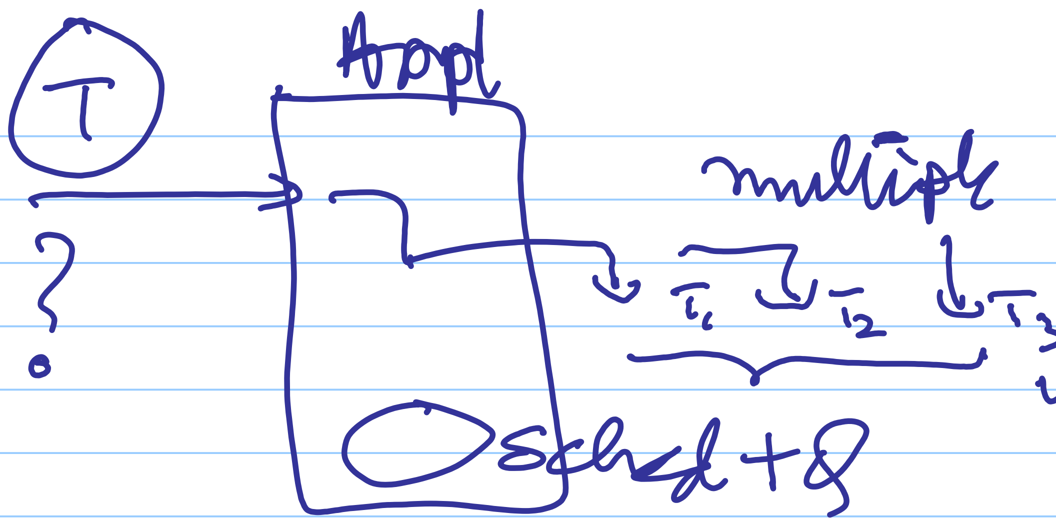
→ converts to multiple threads

→ Run/sched in appl

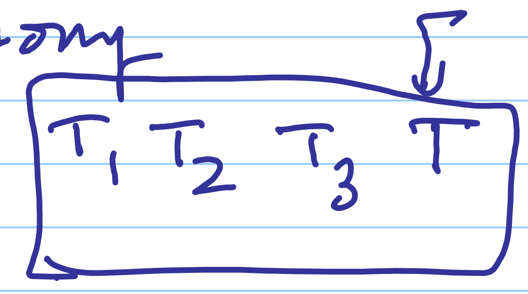
User level T

or User Threads

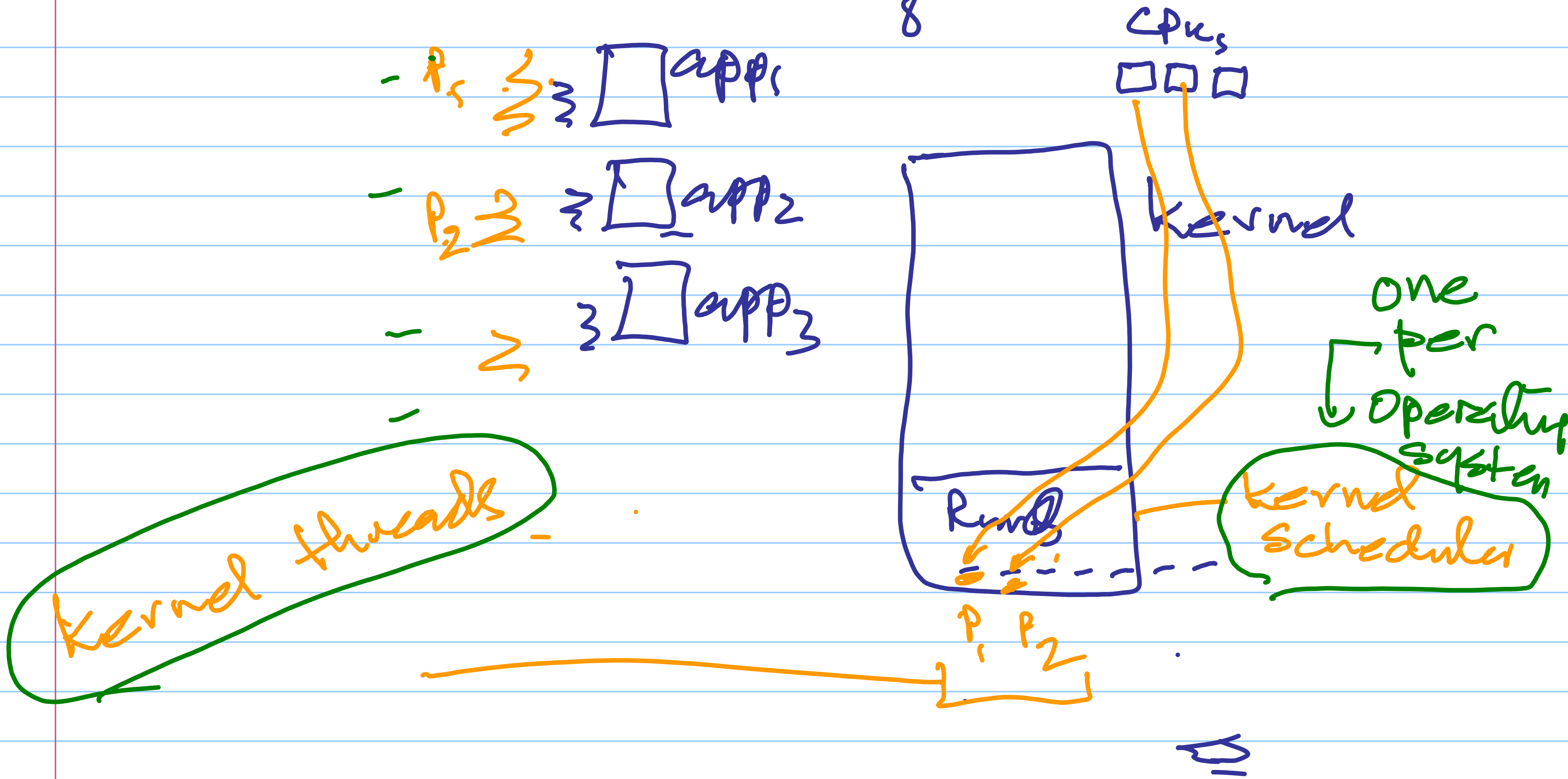
- light weight processes
- fibers



always share memory with



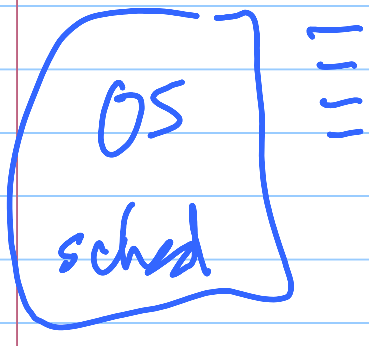
Kernel level scheduling



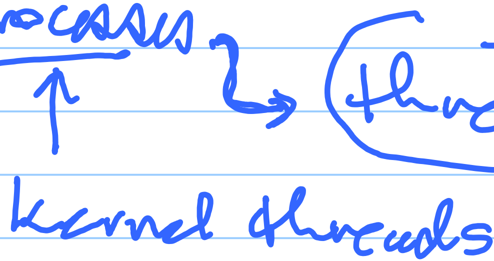
kernel thread

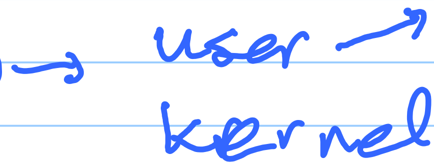
↳ kernel scheduled thread via kernel scheduler

↳ running user code + kernel code

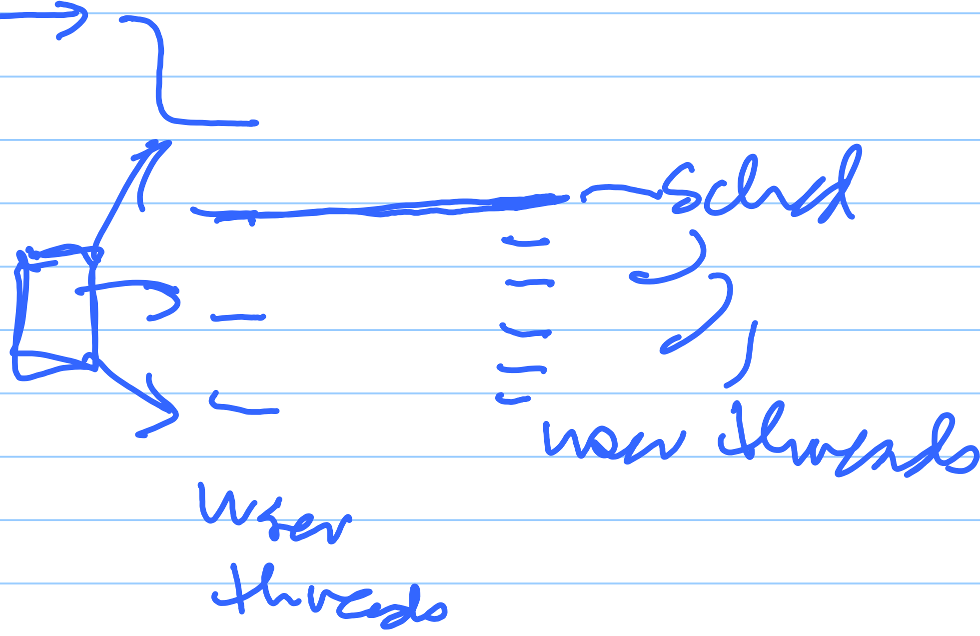


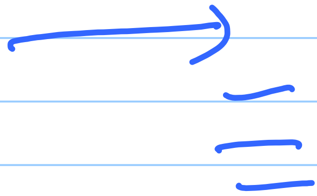
≡ process

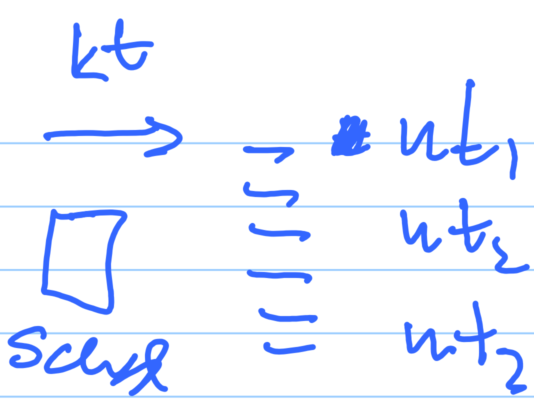


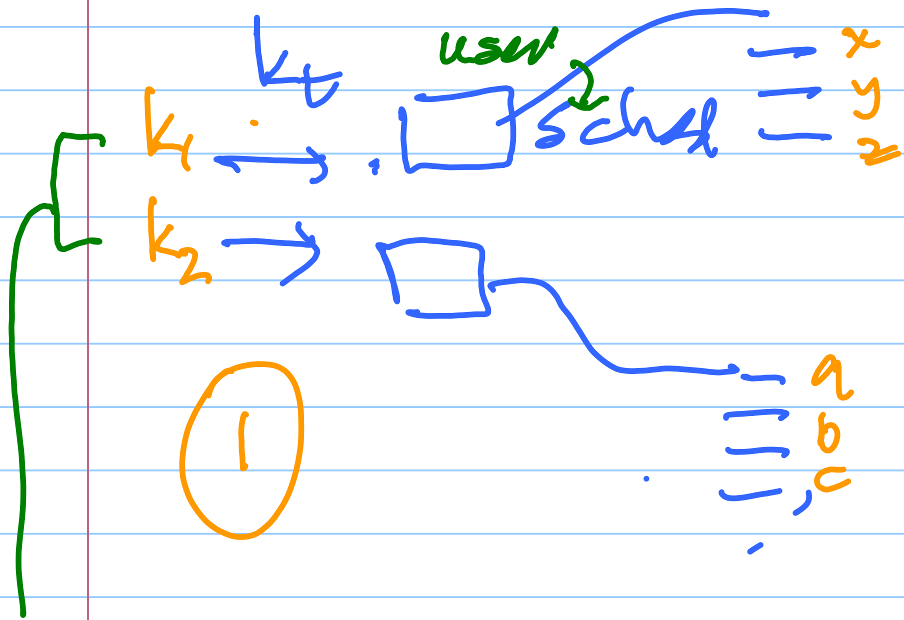


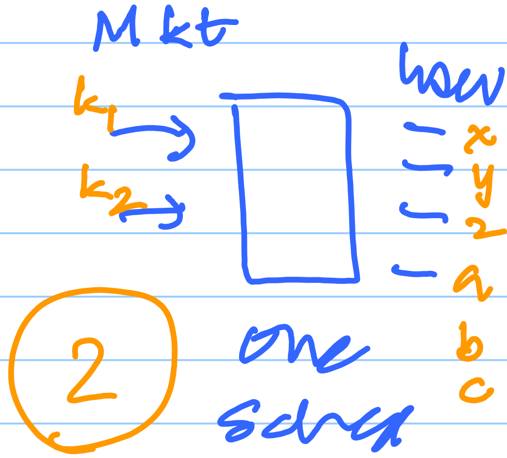
kernel thread

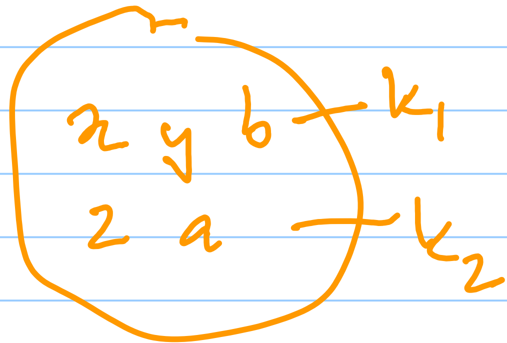





kt
sched
ut_1
ut_2
ut_2


kt
user
k_1
k_2
sched
x
y
z
a
b
c
1
kernel
sched


M kt
k_1
k_2
uow
x
y
z
a
b
c
2
one
sched



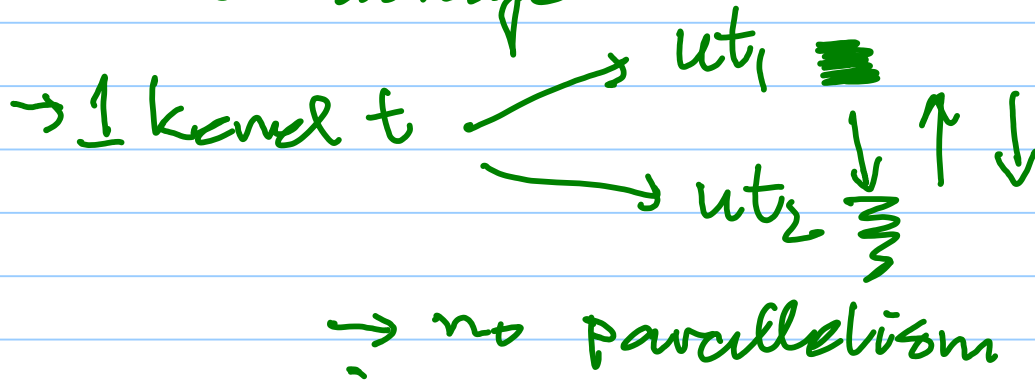
1 Runq
k_1
k_2
Sched
x y z a b c
yield
y
x y b
2 a
k_1
k_2

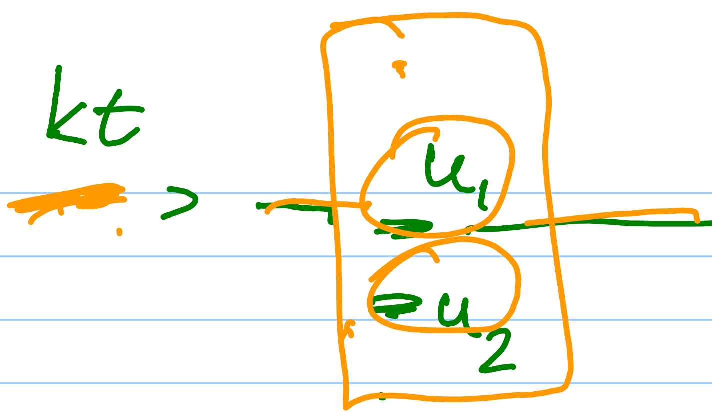
User level sched (& thread)

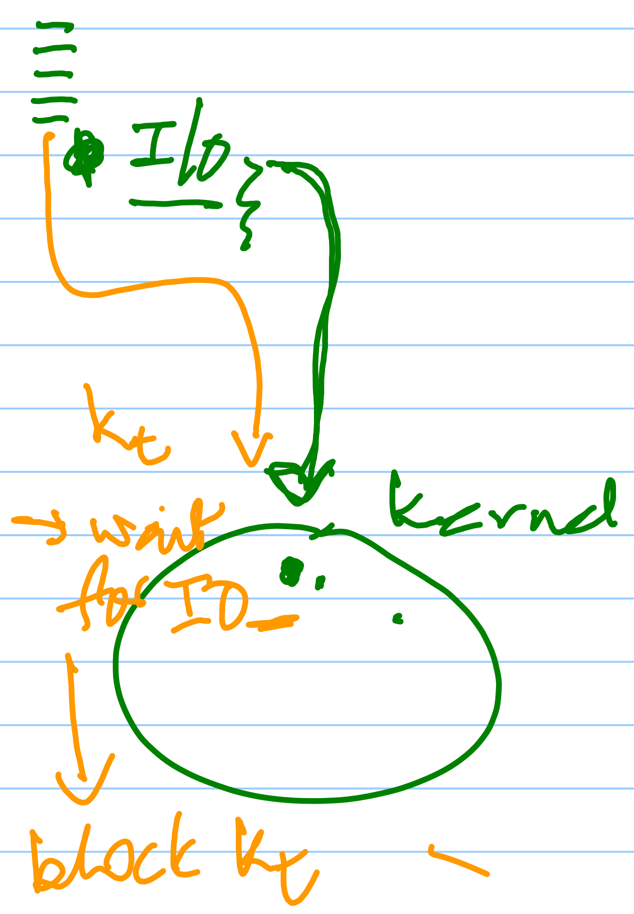
✓ Adv → faster

→ can be done

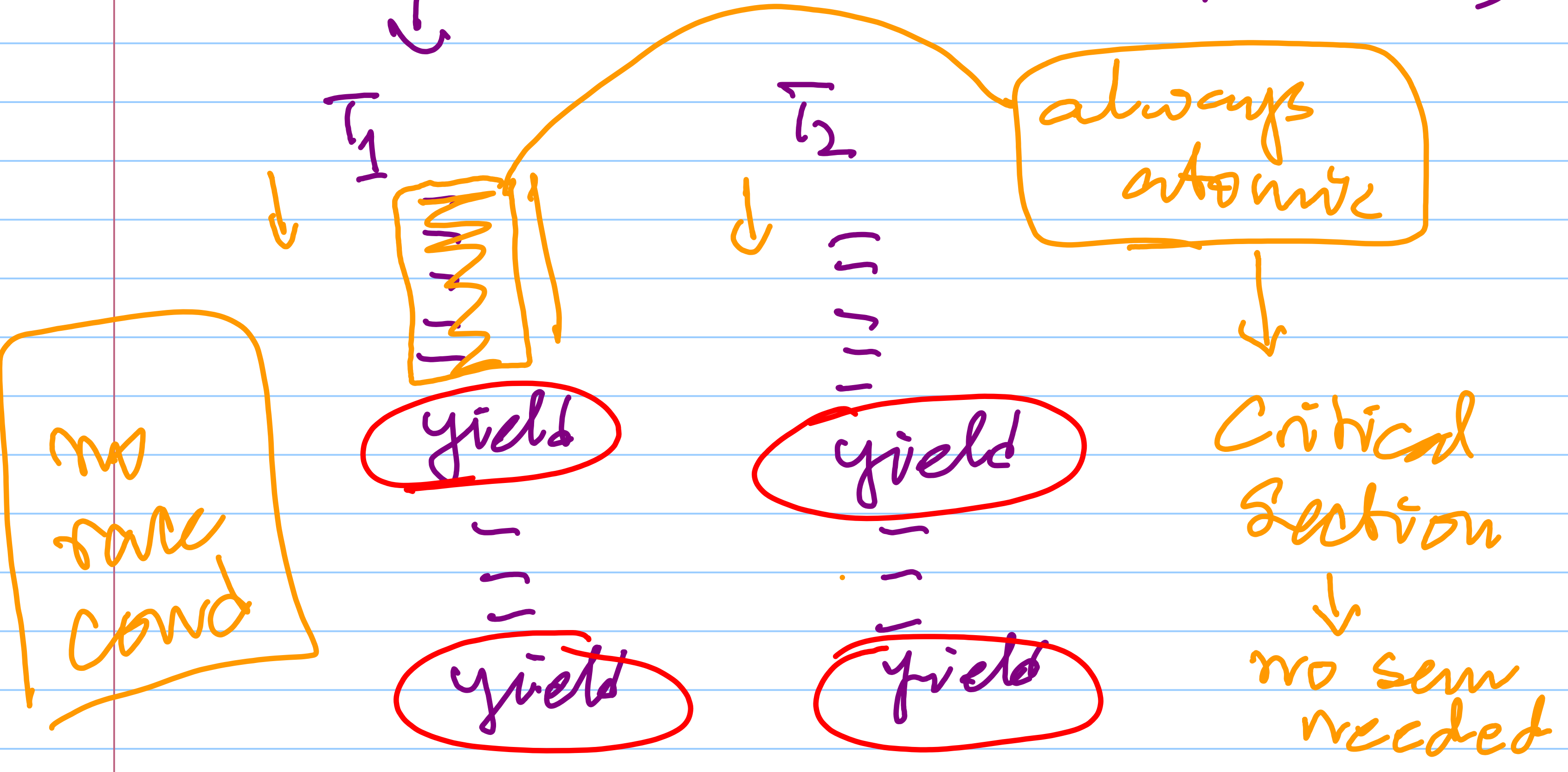
Disadvantage







if u_1 gets blocked then u_2 is also blocked


non preemptive (1 kernel thread
multiple user th)
T1
T2
yield
yield
yield
yield
always atomic
critical section
no sem needed
no race cond

Convert threads written with yields

↓

eliminate yield!

use preemption

use semaphores