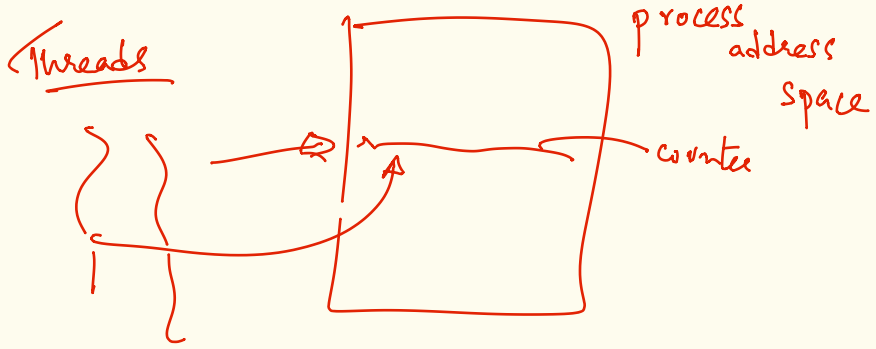
Concurrency / Synchronization.



C-statement

```
counter = counter + 1;
```

```
mov ebx, (% mem addr)      ; mov content from memory addr to ebx
add ebx, 1
mov (% mem addr), ebx
```

read - update - write

non-deterministic sequence ⇒ interrupt, interrupt

⇒ race condition!

threads of execution racing to read-update-write operations.

- outcome depends on when access is obtained.

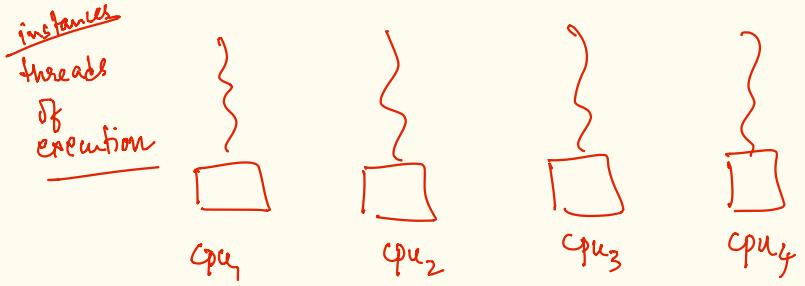
non-~~fa~~. atomicity = fails!

all or nothing property!

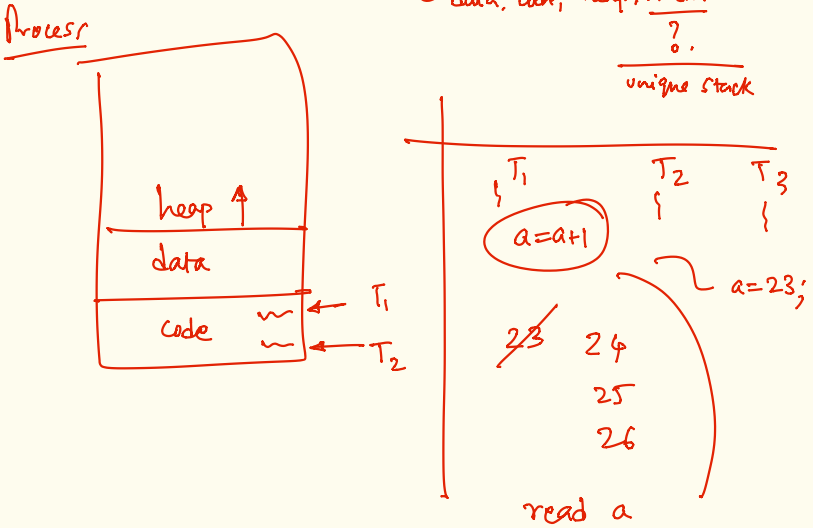
on what?

critical section.

Lecture 17 — Synchronization / Concurrency

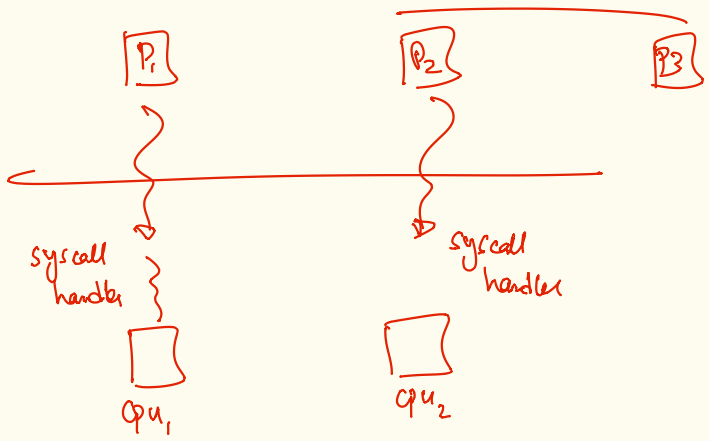


- multi- process
- multi- threaded: thread — execution context in a process
  - shared process address space
    - data, code, heap, stack, ?, ...
    - unique stack



multiple instances of execution on shared data in the Kernel (Kernel mode execution).

e.g.:



(i) P1 & P2 exit()
↓ p->state = ZOMBIE;
yield();
↓
sched();
↓
Scheduler();
↓
np = getnextprocess();
↓
get a process from the ready queue
↓
(proc->state = RUNNABLE)

(ii) simultaneous kalloc
↓
updates freelist

(iii) fork
↓
shared PCB/state list
- some free PCB is a problem
- same PID is a problem.

race condition

(i) multiple instances of execution w/ shared data/state

(ii) read – modify – update actions on shared state.

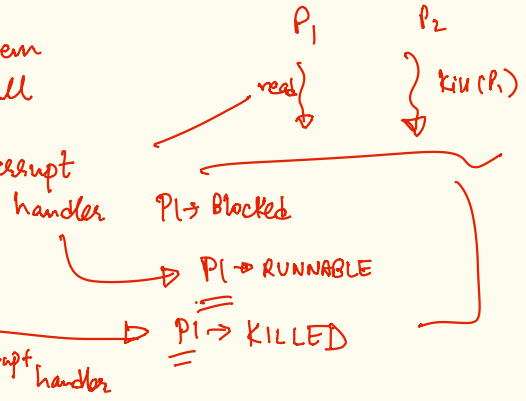
---

eg: of race conditions in kernel mode.

(i) system call + system call

(ii) system call + interrupt handler

(iii) interrupt handler + interrupt handler



Synchronization techniques avoid ~~cor~~ undesirable outcomes due to race conditions.

(1) system call + system call ←

– disable preemption, run system calls to completion.

-ves: inefficiency & multi-cpus can still have multiple system calls/instance

(2) system call + interrupts.

– disable interrupts — works well single-CPU systems.

– multiple cpus, is still a problem.

– losing interrupts is disastrous!

(3) locking / mutual exclusion.

critical section { [ ] } read-modify-update decision

atomic operation! ⇓ Key to build locks!