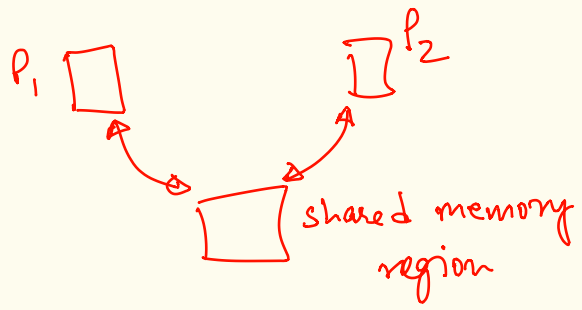
CS347m Lecture #17

concurrency & synchronization



- shmget, shmat } Linux-like OS system calls to setup & attach address space region to shared areas

* fork()
  - shares / copies
    - CoW on all memory regions

* ~~all~~ address space abstraction provides isolation guarantee

* both P1 & P2 access some common memory / variable / object

P1 & P2 execute → C-code:

count = count + 1;

- single 'C' instruction

count = 3

P1	P2
4	5

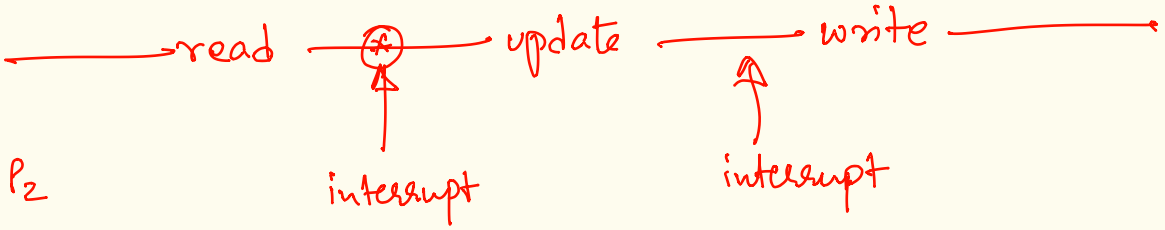
ISA speak:

```
mov  ebx, (%maddr)
add  ebx, 1
mov  (%maddr), ebx
```

move contents at maddr to ebx regs.

read → update → write

this is the complete logical action



P_1	P_2	
4	5	✓
5	4	✓
4	4	✗ inconsistent

the all-or-nothing property on a sequence of instructions is violated.

atomicity.

leads to race condition

program instance race to execute common instructions & the order of ~~elements~~ instances in the race lead to inconsistent outcomes.

(*) race condition happens when —

(i) shared state/objects/variables

(ii) multi-instance execution

(iii) read-update-write semantics on shared state

(*) set of instructions that can lead to race conditions is the critical section.

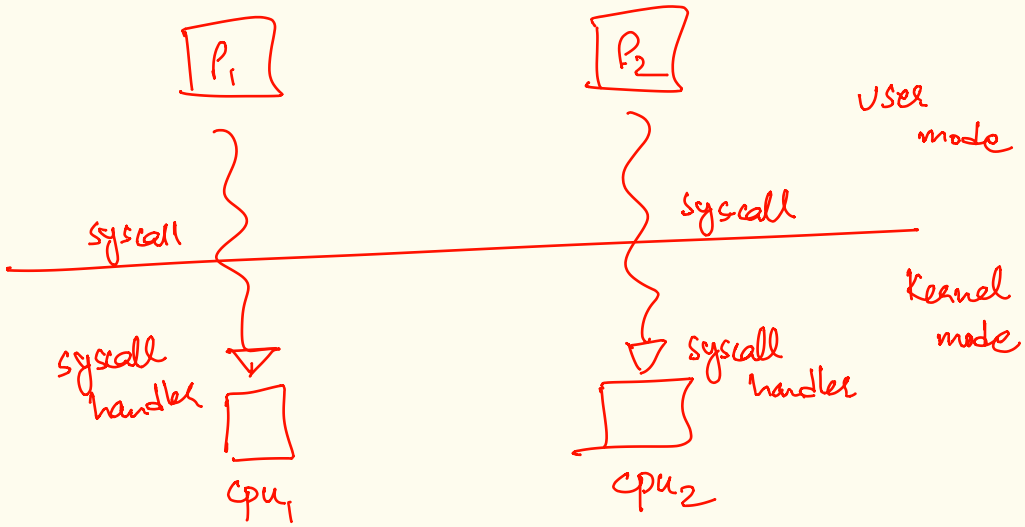
goal: is to serialize (execute one at a time) the critical section of process/program logic to ensure atomicity

Q when/which execution instances does race conditions happen?.

(1) multi-process + shared memory

(2) multi-threaded (processes)

(3) multi-instances of ~~the~~ Kernel execution!



eg:

(i) fork()

↓

xv6 ~ array of PCBs

↳ allocproc ~ find a free PCB

if (proc->state == UNUSED)

(ii) scheduler as last instruction of system call

↳ np = getnextprocess( ); ← go through list of READY/RUNNABLE processes & decide which to schedule next.

checks for READY/RUNNABLE

select a process

(np->state == RUNNABLE)

np->state = RUNNING;

schedule(np); // on cpu

(iii) kalloc

↳ which fetches a free page from the freelist of memory pages to build v2p mapping in the page table.

race conditions in the Kernel mode are possible when ——— following cases.

(i) system call + system call

(ii) system call + interrupt

(iii) interrupt + interrupt

★ synchronization primitives / techniques to avoid race conditions

① disable pre-emption

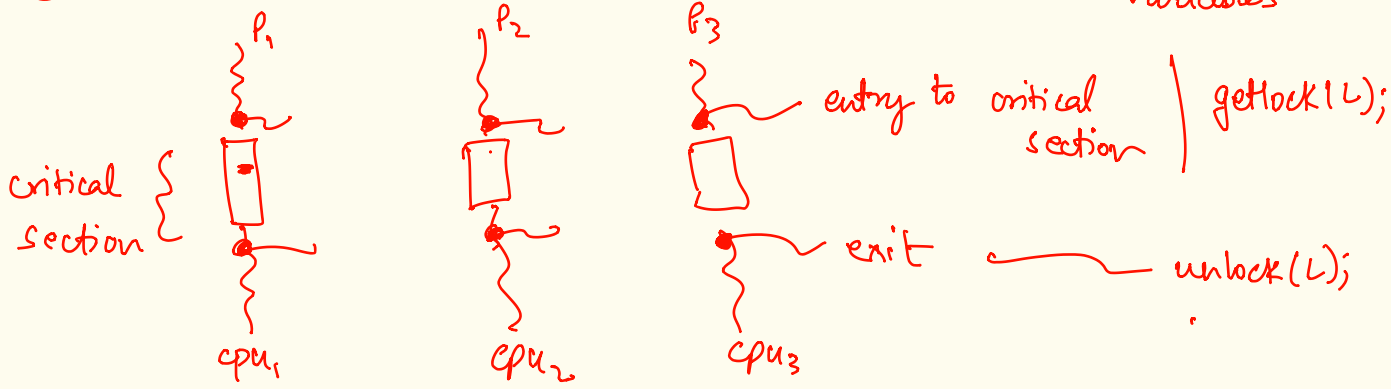
~ ensure run-to-completion when in Kernel mode! (okay for syscalls)

-ves:
- okay of syscalls
- interrupts land in Kernel mode (multi-instance of execution)
- inefficient!
- not applicable / feasible with multiple CPUs.

② ⊕ disable interrupts:
- ensures run to completion in Kernel mode

-ves
- lost interrupts!
- not feasible multi-cpu setup.

③ mutual exclusion via locks / semaphores / condition variables



Q how to design these mutual exclusion primitives ?

↳ what is a lock?

↳ variable / struct / object in memory.

```
int lock;
    lock => 0 ;      // available
    lock => 1 ;      // taken!
```