CS 219 — Lecture 28

Quiz 2
25th Wed.
8am

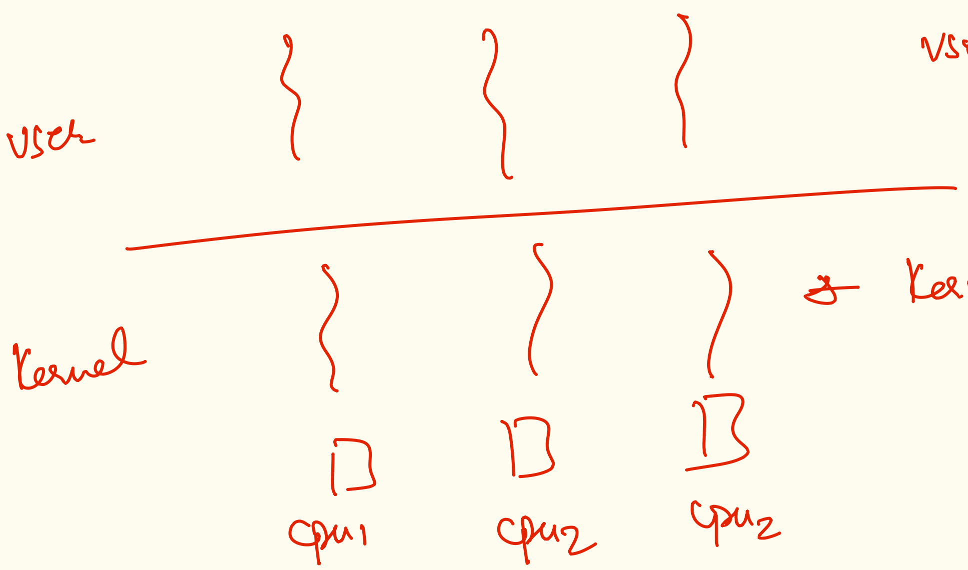
Lab 9 is available

more synchronization primitives.

\# threads of execution.

(1) multiple + shared memory access.

- user instances — threads/processes
- Kernel instances — threads of execution



user threads/processes

& kernel threads instances

Kernel daemon threads

[kswapd]
[kmigrationd]

user space threads ~ system call for kernel support

xchg LOCK,1
cmp R0,0

← futex — fast mutex

futex(uaddr, futex-op, val) ...

- uaddr ↑ sync. variable
- futex-op ↑ wait, wakeup, test & set
- val ↑ value to interpreted w/ futex-op

(2) spinlock in the Kernel

- implementation of spinlock
  - disable interrupts before xchg instruction and enables interrupt after lock obtained.
  - & same of spin unlock.
  - Q1: WHY?

H.W

- Q2: Spinlock implementation in user space does NOT need this disabling interrupts.
  - WHY?

---

(3) generalized synchronization primitives

condition variables, semaphores, reader-writer locks, . . .

- (conditions)
  generalized mutexes
  - mutexes: binary condition — lock 0 or 1
  - wait till child process terminates
  - waitN till 'N' child processes terminate
  - sleep/pause (N) — for N ticks/seconds

HW2 ~ 3 examples in kernel space that need condition variables.

semaphores

└ counting-based synch. primitive.

3 operations

init — initialize semaphore variable

down ~ if (s→val > 0)
        s→val --;
      else
        sleep(s);
        jmp
} mutually exclusive in execution.

up ~ s→val ++;
     wakeup(s);
} atomic

usage example

3 chairs ~

```
s.init(3);

down(s);
sit();
eat();
up(s);
```
} every thread

implementation of a semaphore.

```
struct semaphore {
    int val;
    spinlock L;
}
```

semaphore S;

```
down(S) {
    spinlock(S->L);
    while (S->val <= 0)
        sleep(&S, S->L);
    S->val --;
    spinunlock(S->L);
}
```

condition ↓ (pointing to `S->val <= 0`)

condition variable denoting this condition. (pointing to `&S`)

```
up(S) {
    spinlock(S->L);
    S->val ++;
    wakeup(&S);
    spinunlock(S->L);
}
```