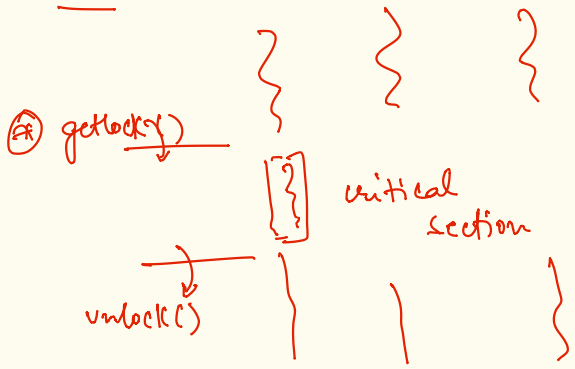
lecture #18 synchronization primitives.

* disable preemption
  disable interrupts



read-modify-update

(1) what is lock?
- in the prog. sense
  - variable object.
  - machine/process
    - memory address

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

```
int getlock (lock * l) {
a   while (l->locked == 0) {   //available
b       l->locked = 1;
    }
    return 1;
}
```

```
strut Lock {
    int locked;
    int id;
};
```

T_1 a	T_2 a	T_1 b	T_2 b
1	1	1	1
0	0	1	1

(*) issue

multiple instructions / c-statements check & update
- status of lock.
- non-atomic!

① multiple c-statements
② single c-statement
③ single ISA instruction

} lack atomic guarantees.

} pipelining of a set of instructions.

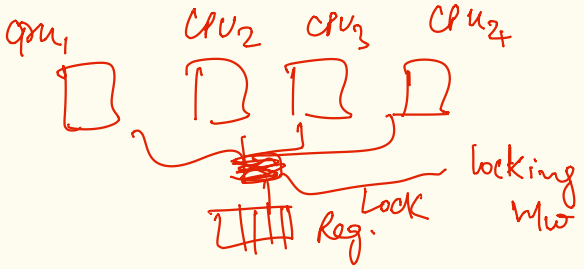
inc #R0

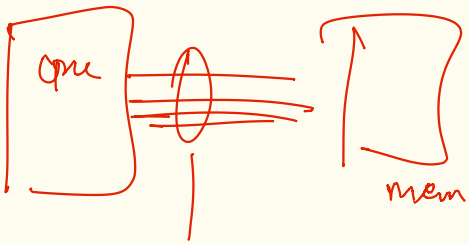
inc %mem

(*)

lock; inc %mem

prefix x86.





lock the address bus

(*) locks depend on Hw primitives

↳ ISA support for atomic test and set / Compare & swp instructions.

eg: TSL LOCK, R0 — atomic

test & set

xchng

- old value = LOCK
- LOCK = R0
- return oldvalue

Spinlock.

① threads are ~ spinning / in busy loop on the CPU till lock acquired.

```
getlock:  mov R0, 1
          TSL LOCK, R0
          CMP R0, 0
          JNZ getlock
          RET

unlock:   mov LOCK, 0
          RET
```

assuming return value is in R0.

while (xchng(LOCK, 1)==0)

② mutex / sleeplock

- if lock not available sleep till lock is available & recheck.
  - proc → STATE = BLOCKED
- on unlock wakeup all threads waiting on this lock.
  - proc → state = RUNNABLE;

```
mutex lock:  mov R0, 1
             TSL LOCK, R0
             CMP R0, 0
             JZ OK
             CALL YIELD
             jmp mutex lock

OK :         RET
```

proc → state = BLOCKED

proc → chan = &LOCK;

mutex unlock:

mov LOCK, 0

call wakeup (&LOCK) → iterate over all blocked processes { proc->chan == &LOCK proc->state = RUNNABLE;

(*)

```
struct mutex {
    int id;
    int locked;      // 0 ~ available, 1 ~ locked
}

mutex lock (mutex* m) {
    while (m->locked == 1)        // not atomic
        sleep (m->id);            // (xchng(m->locked, 1) != 0)

    m->locked = 1;
}

mutex unlock (mutex *m) {
    m->locked = 0;
    wakeup (m->id);
}
```

sleep — wakeup

wait — signal

(*) sleeplock

sleepunlock