CS 347 m Lecture 18

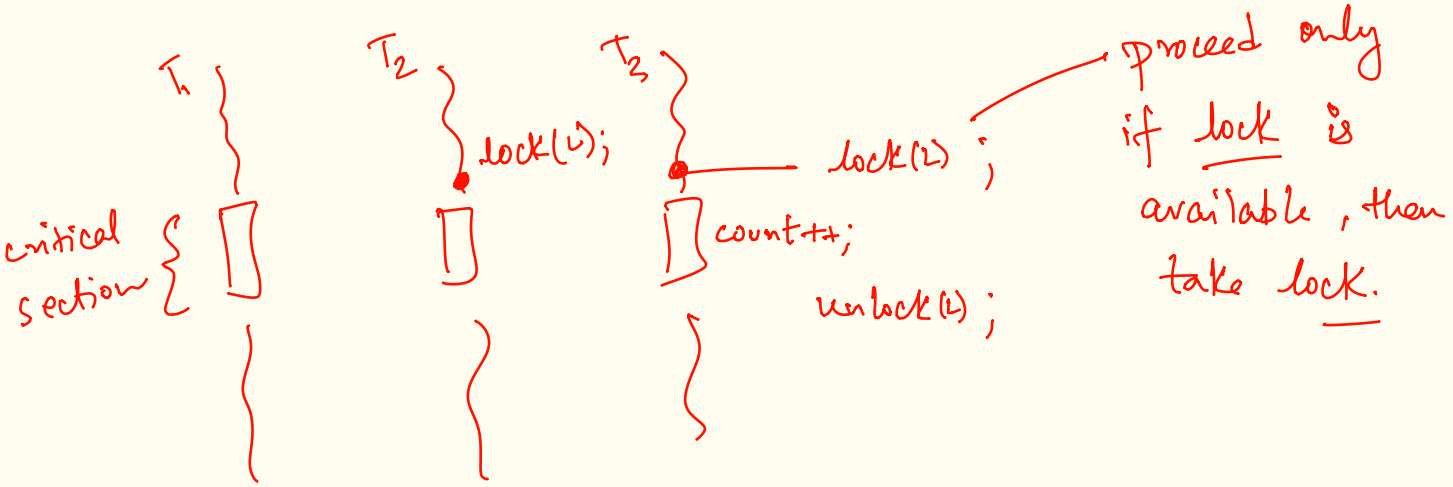
Design of synchronization primitives.

recap: race condition, critical section, atomicity, locks

- multi-instance execution
- access shared memory/objects
- read - modify - write } - multi-part sequence on shared object
  update
  compare - decide

(*) disable-preemption, disable interrupts.

mutual exclusion via locks



Q how to design locks?

operations on lock should be atomic

lock should enforce synchronization constraint.

~Q what is a lock?

value at a memory address / variable

0 — available

1 — taken

```
lock(L) {
    if (L==0) {
        L = 1;
        return TRUE;
    else
        return FALSE;
}
```

non atomic

comp } non-atomic
set }

not proceeding into critical section

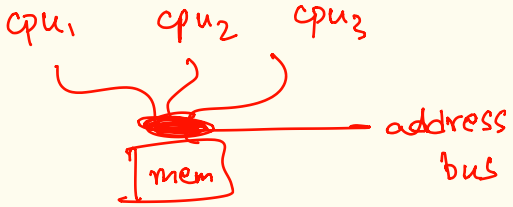
(i) multiple C-style instructions
(ii) single C-style instruction
(iii) single ISA instruction

} non-atomic

pipeling of instruction sub components (fetch-decode-execute)

```
inc eax;          — increments value of eax rgs. by 1
dec %(memaddr)
```

x86 provides locking of the address bus till completion of instruction.




```
LOCK;  inc %(memaddr)
```

↑ tag to indicate lock address bus

Locks need ISA (new) abstractions / instructions to provide atomic compare and set capabilities.

test

examples: instructions

- TSL — test & set lock

x86.

xchg (arg1, arg2)

TSL dest, SRC

dest } % mem addr
src } reg.

- old value = dest
- dest = src
- src = old value / return old value

} ATOMIC

(1) **spin lock** ~ spins / keep checking lock till available on the CPU till lock is available

LOCK ← lock variable

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

zero flag be set if R0 = 0

LOCK	R0	TSL
0	1	0
1	1	1

spinlock(L); spinlock(L); spinlock(L);

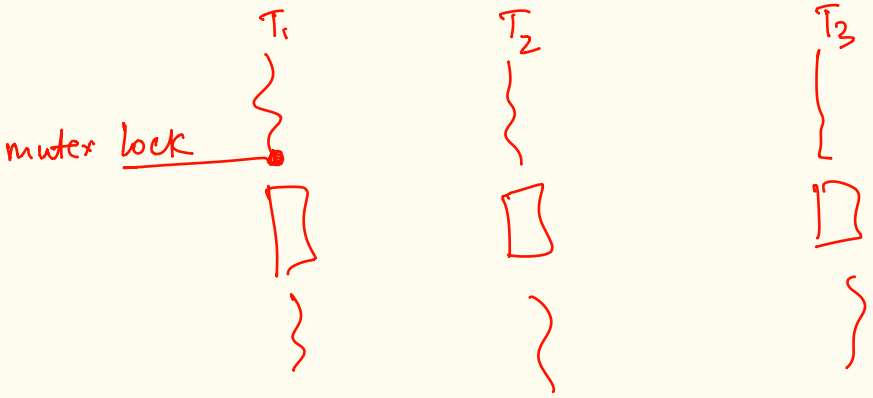
```
Spinunlock: mov R0, 0
            TSL LOCK, R0
```

x86 ≈ while(xchg (lock, 1) != 0)

---

(2) **mutex**: mutex exclusion lock

- if lock not available do not burn the CPU — spin on CPU
- on unlock signal lock available
- recheck when lock is available.



T_1	T_2	T_3
lock	Blocked sleeping	Blocked sleeping
unlock	Ready	Ready
	Block sleeping	lock

LOCK ← lock variable

```
mutex lock :   mov R0, 1
               TSL LOCK, R0
               JZ  OK        ← jump on zero || got lock
               CALL YIELD
               jmp mutexlock
OK :           RET
```

flag set

yield / give up the CPU

proc → state = BLOCKED;

proc → chan = LOCK;

```
mutex unlock :  mov LOCK, 0
atomic! { CALL wakeup(LOCK)
```

wakeup / set proc → state = READY for all processes blocked on LOCK.

(*) spinlock ~ integer / memory address

mutex ~

```
struct mutex {
    int lock;    ← status of mutex lock
    int id;      ← identifier
    spinlock s;
};

struct mutex m;
```

```
mutex_lock (mutex * m) {
while  if (m->lock == 1)
           sleep (m->id);
       m->lock = 1;
}
```

non-atomic

(xchg (m->lock, i) != 0)

```
mutex_unlock (mutex *m) {
        m->lock = 0;
        wakeup (m->id);
}

sleep (m->id)  {
```

```
wakeup (m->id) {
  for all processes p {
    if (p->state == BLOCKED
      && p->chan == m->id)
    { p->state = READY
      p->chan = 0
    }
}
```

```
mutex_lock (mutex *m)
{
    spinlock (m->s);
    while (m->lock == 1)
        sleep (m);
    m->lock = 1;
    spinunlock (m->s);
}
```

```
mutex_unlock (mutex *m)
{
    spinlock (m->s);
    m->lock = 0;
    wakeup (m);
    spinunlock (m->s);
}
```

```
sleep (mutex *m)
{
    p = myproc ();          // process calling sleep
    p->state = BLOCKED;
    p->chan = m->id;
    spinunlock (m->s);
    sched ();               // invoke scheduler   yield!
    spinlock (m->s);
}
```

```
wakeup (m->id) {
    for all processes p
        if (p->chan == m->id) {
            p->chan = 0;
            p->state = READY;        RUNNABLE
        }
}
```