# CS1550 Lab 2

# Locking

- Computes with multiple CPU executing independently
- Some data is shared (e.g. bank balance)
- Have to have a strategy to maintain the correctness
- Lock provide the mutual exclusion
  - Only one CPU can hold the lock for one shared data

# Code

```
struct list {
  int data;
  struct list *next;
};
struct list *list = 0;
void
insert(int data) {
        struct list *l;
        l = malloc(sizeof *l);
        l->data = data;
        l->next = list;
        list = l;
}
```

Assume execution isolation

```
struct list *list = 0;
struct lock listlock;

void
insert(int data)
{
        struct list *l;

        acquire(&listlock);
        l = malloc(sizeof *l);
        l->data = data;
        l->next = list;
        list = l;
        release(&listlock);
}
```

Execution with locks

# Spinlock

```
Void
acquire(struct spinlock *lk)
{
    for(;;) {
        if(!lk->locked) {
            lk->locked = 1;
            break;
        }
    }
}
```

Keep spin until find lock is released

Problem?

# Spinlock

```
Void
acquire(struct spinlock *lk)
{
    for(;;) {
        if(!lk->locked) {
            lk->locked = 1;
            break;
        }
    }
}
```

Keep spin until find lock is released

Problem?

In multiple CPUs environment, if two CPUs simultaneously reach line 25…

Both of them will get access to data

Need to let it execute in one atomic step.

xchg instruction  (each iteration atomatically set lock to 1)

# Xchg atomic hardware instruction

- Swap a word in memory with the contents of a register
- In acquire function:
  - loop xchg instruction
  - Each round atomically read lock and set the lock to 1

```
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to
avoid deadlock.
    if(holding(lk))
        panic("acquire");

// The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0);

    __sync_synchronize();

// Record info about lock acquisition for
debugging.
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}
```

# Test and Set

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = TRUE;
        return rv;
}
```

# Compare_and _swap

```
int compare _and_swap(int *value, int expected, int new_value)
{
        int temp = *value;
        if (*value == expected)
                *value = new_value;
        return temp;
}
```

# Sleep locks

- For code need to hold a lock for a long time
  - Read/write on the disk
- Efficient way is let the processor be yielded and let other threads execute
- Locked field protected by a spinlock
- Sleep function yield the CPU

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}

void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

# Sleep

- Put one process to sleep waiting for event
- Mark current process as sleeping
- Call sched() to release the processor
- Acquire ptable lock to make sure other process ‘s call wakeup not affect putting the process to sleep

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
  p->chan = chan;
  p->state = SLEEPING;

  sched();
  p->chan = 0
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}
```

# Wake up

- Wake up process when event happened
- Mark a waiting process as runnable

```
static void
wakeup1(void *chan)
{
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
}

void
wakeup(void *chan)
{
    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}
```

# Reference


- Cox, Russ, M. Frans Kaashoek, and Robert Morris. "Xv6, a simple Unix-like teaching operating system." *2013-09-05]. http://pdos. csail. mit. edu/6.828/2012/xv6. html* (2011).