Recap

- Mutual exclusion
- Peterson's algorithm
  - s/w solution
  - Assume program order execution
- Synchronization instructions
  - Test&set

KU THE UNIVERSITY OF KANSAS

TestAndSet Instruction

- Pseudo code

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

Mutual Exclusion using *TestAndSet*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex)
{
    *mutex = 0;
}

void lock (int *mutex)
{
    while(TestAndSet(mutex))
        ;
}

void unlock (int *mutex)
{
    *mutex = 0;
}
```

KU THE UNIVERSITY OF KANSAS

CAS (Compare & Swap) Instruction

- Pseudo code

```
int CAS(int *value, int oldval, int newval)
{
        int temp = *value;
        if (*value == oldval)
           *value = newval;
        return temp;
 }
```

Mutual Exclusion using *CAS*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex) {
    *mutex = 0;
}

void lock (int *mutex) {
    while(CAS(&mutex, 0, 1) != 0);
}

void unlock (int *mutex) {
    *mutex = 0;
}
```

KU THE UNIVERSITY OF KANSAS

Roadmap

- Solutions for mutual exclusion
  - Peterson's algorithm (Software)
  - Synchronization instructions (Hardware)

- **High-level synchronization mechanisms**
  - **Mutex**
  - **Semaphore**
  - **Monitor**

Spinlock using TestAndSet

```
void init_lock (int *mutex)
{
   *mutex = 0;
}

void lock (int *mutex)
{
   while(TestAndSet(mutex))
      ;
}

void unlock (int *mutex)
{
   *mutex = 0;
}
```

What's Wrong With **Spin**locks?

- Very wasteful
  - Waiting thread continues to use CPU cycles
  - While doing absolutely nothing but wait
  - 100% CPU utilization, but no useful work done
  - Power consumption, fan noise, …
- Useful when
  - You hold the lock only *briefly*
- Otherwise
  - A better solution is needed

KU THE UNIVERSITY OF KANSAS

Mutex – Blocking Lock

- Instead of spinning
  - Let the thread sleep
    - There can be multiple waiting threads
  - In the meantime, let other threads use the CPU
  - When the lock is released, wake-up one thread
    - Pick one if there multiple threads were waiting

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
…
    while(TestAndSet(&lock->value)) {
…
…
…
…
…
    }
…
}

void mutex_unlock (mutex_t *lock)
{
…
    lock->value = 0;
…
…
…
}
```

More reading: mutex.c in Linux

Thread waiting list

To protect waiting list

More reading: mutex.c in Linux

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
…
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;
        list_add(&lock->wait_list, current);
…
        schedule();
…
    }
…
}

void mutex_unlock (mutex_t *lock)
{
…
    lock->value = 0;
…
…
…
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

More reading: mutex.c in Linux

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
    spin_lock(&lock->wait_lock);
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;
        list_add(&lock->wait_list, current);
…
        schedule();
…
    }
   spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
…
    lock->value = 0;
…
…
…
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

More reading: mutex.c in Linux

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
    spin_lock(&lock->wait_lock);
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;
        list_add(&lock->wait_list, current);
        spin_unlock(&lock->wait_lock);
        schedule();
        spin_lock(&lock->wait_lock);
    }
   spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
…
    lock->value = 0;
…
…
…
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

More reading: mutex.c in Linux

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
    spin_lock(&lock->wait_lock);
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;
        list_add(&lock->wait_list, current);
        spin_unlock(&lock->wait_lock);
        schedule();
        spin_lock(&lock->wait_lock);
    }
   spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
…
    lock->value = 0;
    if (!list_empty(&lock->wait_list))
        wake_up_process(&lock->wait_list)
…
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

Someone is waiting for the lock

Wake-up a waiting thread

More reading: mutex.c in Linux

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
    spin_lock(&lock->wait_lock);
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;
        list_add(&lock->wait_list, current);
        spin_unlock(&lock->wait_lock);
        schedule();
        spin_lock(&lock->wait_lock);
    }
   spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
    spin_lock(&lock->wait_lock);
    lock->value = 0;
    if (!list_empty(&lock->wait_list))
        wake_up_process(&lock->wait_list)
    spin_unlock(&lock->wait_lock);
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

Someone is waiting for the lock

Wake-up a waiting thread

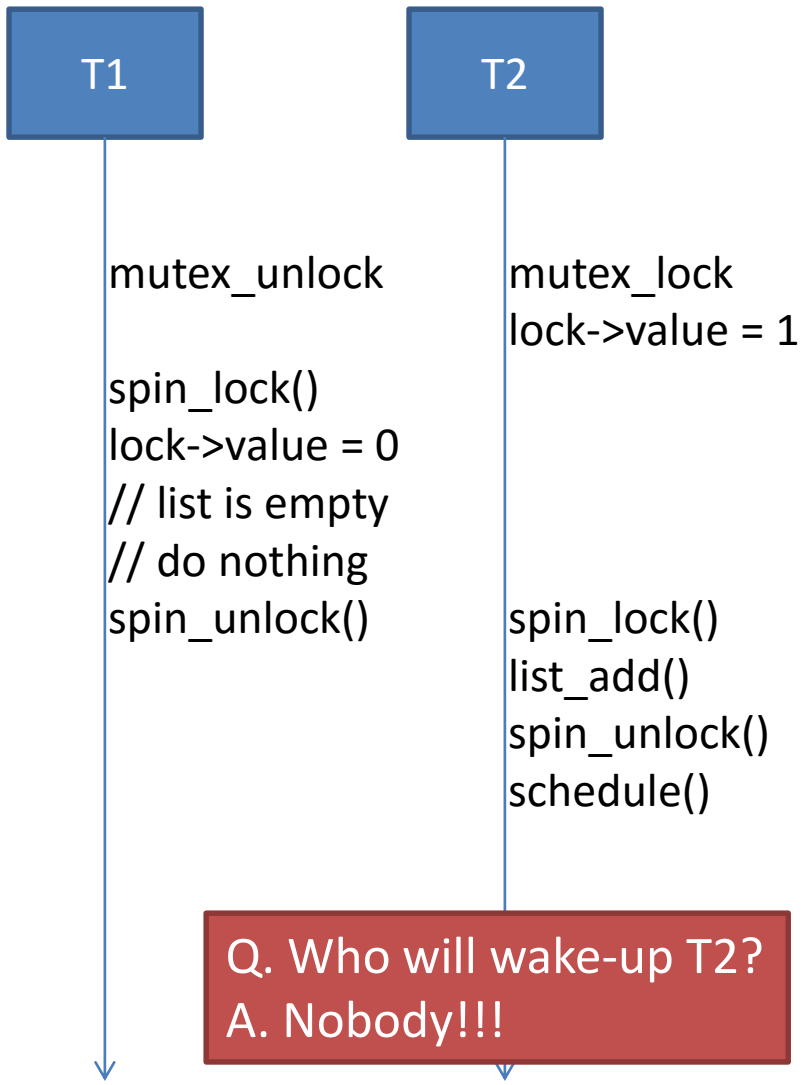
```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;
        spin_lock(&lock->wait_lock);
        list_add(&lock->wait_list, current);
        spin_unlock(&lock->wait_lock);
        schedule();
    }
}

void mutex_unlock (mutex_t *lock)
{
    spin_lock(&lock->wait_lock);
    lock->value = 0;
    if (!list_empty(&lock->wait_list))
        wake_up_process(&lock->wait_list)
    spin_unlock(&lock->wait_lock);
}
```

Correct?

More reading: [mutex.c in Linux]



Agenda

- High-level synchronization mechanisms
  - Mutex
  - **Semaphore**
  - **Monitor**

High-level Synchronization Primitives

- Lock (mutex) is great, but…
  - Too low-level primitive
  - Sometimes we need more **powerful primitives**

- Semaphore
  - Binary/integer semaphore
- Monitor
  - Condition variable

Semaphore

- High-level synchronization primitive
  - Designed by Dijkstra in 1960’
- Definition
  - Semaphore is an integer variable
  - Only two operations are possible:
    - P() or wait() or down()
    - V() or signal() or up()

Simple Semaphore Implementation

- P() operation

```
P(semaphore *S) {
        while(S–>value == 0) {
                S->list->addQ(P);
                schedule( );
        }
        S–>value—— ;
}
```

schedule ( ) – schedule another thread

- V() operation

```
V(semaphore *S) {
        if(!IsEmpty(&S->list)) {
                P = delQ(&S->list);
                wakeup(P);
        }
        S–>value++ ;
}
```

wakeup( ) – wake up a thread

What's wrong with the code?

Simple Semaphore Implementation

- P() operation

```
P(semaphore *S) {
    S->lock->Acquire();
    while(S–>value == 0) {
        addQ(&S->list, P);
        S->lock->Release();
        schedule( );
        S->lock->Acquire();
    }
    S–>value—— ;
    S->lock->Release();
}
```

- V() operation

```
V(semaphore *S) {
    S->lock->Acquire();
    if(!isEmpty(&S->list)) {
        P = delQ(&S->list);
        wakeup(P);
    }
    S–>value++ ;
    S->lock->Release();
}
```