# Operating System Concepts

## Lecture 18: Synchronization Primitives

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

# Today’s class

- Synchronization primitives
  - Mutex locks
  - Condition variables

# Programming abstractions for synchronization

- With low-level hardware support, programming languages provide atomic operations for synchronization
  - locks: can be held by at most one process/thread at a time; it is obtained before entering a critical section and released after exiting it
  - condition variables: provide conditional synchronization
  - semaphores: more general version of locks
  - monitors: connect shared data to synchronized primitives

# Mutex locks

- A high-level programming abstraction; an object that only one thread can hold at a time
  - can be implemented as a **spinlock** which does busy waiting

```
void acquire() {
    while(test_and_set(&lock))
      ; /* spin */
    /* critical section */
    lock = false;
}
```

  - can be implemented as a blocking operation (next slide)

# Blocking implementation of locks

```
class Lock {
public:
  void Acquire();
  void Release();
private:
  int locked;   // lock state
  Queue Q;      // lock waiters queue
}
```

Acquire(): waits until lock is free and then grabs it

Release(): releases the lock and wakes up any waiters

Mutual exclusion can be supported using locks (symmetric solution)

```
Lock milklock;
...
milklock.Acquire( )
if(milk == 0)
  buy_milk();
milklock.Release( )
```

Lock milklock: initially free (not held by any process)

milklock.Acquire( ): acquired before accessing shared data

if(milk == 0) buy_milk(): critical section

milklock.Release( ): released after accessing shared data

# Example of using locks (loose syntax)

```
void *malloc(size_t size) {
  heaplock.acquire();
  p = allocate memory of the specified size
  heaplock.release();
  return p;
}

void free(void *p) {
  heaplock.acquire();
  deallocate memory & put it back on free list
  heaplock.release();
}
```

**recall that threads of a process share the heap section**

# How to implement locks on uniprocessors?

```
Lock::Acquire() {
    intr_disable();
    if (locked == 0) {    // lock is free
        locked = 1;
    } else {              // lock is held by another thread
        queue_add(Q, gettid());
        thread_block();   // put this thread to sleep
    }
    intr_enable();
}

Lock::Release() {
    intr_disable();
    if(queue_empty(Q)) {
        locked = 0;                          // release the lock
    } else {
        thread_unblock(queue_remove(Q)); // put on ready queue
    }
    intr_enable();
}
```

**CLI and STI (privileged) instructions are used to clear and set interrupts respectively**

# How to implement locks on multiprocessors?

- A thread/process executing a CLI instruction does not disable interrupts on other processors!
- So we have to use other hardware support to implement `acquire` and `release` methods
  - test_and_set
  - compare_and_swap

# Implementation with compare_and_swap

- Compare the value against some expected value (in register), if they are the same, set the value in memory to a different value (in register)
  - if [addr] == r1 then [addr] = r2;
- Report either a boolean response or the old value
  - there are two variants

```
Lock::Lock {
  locked = 0;
}

Lock::Acquire() {
  while(compare_and_swap(&locked, 0, 1) != 0)
     ; // if busy, do nothing
}

Lock::Release() {
  locked = 0;
}
```

# Implementation with test_and_set

- If lock is free (value = 0), test&set reads 0, sets value to 1, and returns 0
  - the Lock is now busy: the test in the while fails (Acquire is complete)
- If lock is busy (value = 1), test&set reads 1, sets value to 1, and returns 1
  - continues to loop until a Release is executed

```
Lock::Lock {
  locked = 0;
}

Lock::Acquire() {
  while (test_and_set(&locked) == 1) {
    ; // if busy, do nothing
  }
}

Lock::Release() {
  locked = 0;
}
```

# Avoiding busy-waiting as much as possible

- We can't eliminate busy waiting entirely but we can minimize its use to build a more efficient lock
  - instead of busy-waiting until lock is free, we busy-wait to atomically check the lock state and give up CPU if we find that the lock is busy
  - updating the lock state is a short critical section than the critical section in which shared data are updated and protected by the lock

```
class Lock {
public:
  void Acquire();
  void Release();
private:
  int locked; // lock state
  int guard;  // safe to check the lock state
  Queue Q;
}

Lock::Lock {
  locked = 0; // lock is free initially
  guard = 0;
}
```

# Test_and_set — minimal waiting

```
Lock::Acquire() {
  while(test_and_set(guard) == 1)
    ;   // spin until guard can be acquired
  if(locked != 0) {         // lock is busy
    queue_add(Q, gettid());
    guard = 0;              // set guard to 0 before blocking thread
    thread_block();         // block this thread
  } else {                  // lock is free
    locked = 1;             // lock is acquired
    guard = 0;
  }
}

Lock::Release() {
  while(test_and_set(guard) == 1)
    ;
  if(!queue_empty(Q)) {
    // take a thread off the queue and pass the lock directly to it
    thread_unblock(queue_remove(Q));
  } else {
    locked = 0; // let go of lock as there is no waiting thread
  }
  guard = 0;
}
```

# Observations

- Why does a thread set guard to 0 before blocking itself?
  - so that another thread can obtain guard to release the lock (**liveness issue**)
- Why does a thread that is releasing the lock pass it to next waiting thread (if any) rather than just releasing it and putting the awakened thread on the ready queue?
  - the awakened thread doesn't hold guard when it wakes up so it can grab the lock before entering the critical section (**mutual exclusion issue**)
- What if there's a context switch right before calling thread_block and the next thread releases the lock?
  - the first thread would be blocked for ever (**wakeup/waiting race**)
  - one solution is to ensure that two operations, i.e., releasing guard and blocking thread, are implemented as one atomic operation

# Comparing to the “interrupt disable” solution

```
Lock::Acquire(Thread T) {
    intr_disable();
    if (locked == 0) {
        locked = 1;
    } else {
        queue_add(Q, T);
        thread_block(T);
    }
    intr_enable();
}

Lock::Release() {
    intr_disable();
    if(queue_empty(Q)) {
        locked = 0;
    } else {
        thread_unblock(queue_remove(Q));
    }
    intr_enable();
}
```

Replace:

- `intr_disable()` with `while(test&set(guard));`
- `intr_enable()` with `guard = 0;`

# Two-phase locks

- Spin for a small amount of time (spin phase) and if the lock cannot be acquired then put caller to sleep (sleep phase)
- What’s the advantage?

# Beyond locks

- Locks provide mutual exclusion but sometimes a thread has to wait only if a certain condition is true (synchronizing on a condition)
- Example: A producer thread puts data in a bounded buffer, a consumer thread takes them out.
  What are the constraints for the bounded buffer?

  1. only one thread can manipulate buffer queue at a time (*mutual exclusion)*
  2. consumer must wait for producer to fill buffers **if all empty** (*scheduling constraint)*
  3. producer must wait for consumer to empty buffers **if all full** (*scheduling constraint)*

# Condition variable

- An abstraction that supports **conditional synchronization**
  - a queue of threads waiting for a specific **event inside the critical section**
    - free memory is getting low, run the garbage collector
    - new data has arrived in a port, process it
  - the condition of the condition variable is defined based on the data protected by a mutex lock

# Operations on condition variables

- Support three operations
  - `Wait( )` usually takes a lock (to be released) as a parameter
    - atomically release lock and go to sleep (block the thread until signalled)
    - reacquire lock upon waking up
  - `Notify( )` — historically called `Signal( )`
    - wake up a waiting thread (if any) and put it on the ready queue (may not run immediately)
  - `NotifyAll( )` — historically called `Broadcast( )`
    - wake up all waiting threads and put them on the ready queue
- A thread **must hold the lock** when doing these condition variable operations because
  - first, these operations may update the state
  - second, to ensure signal and wait operations are not interleaved (by two threads)

# Protocol for using condition variables

- Acquire the lock to enter the critical section
- Check the condition inside the critical section
  - if condition is true: block the thread and release the lock
  - if condition is false: only release the lock

# Example: the coke machine (loose syntax)

Condition variables are used with a mutex lock and in a loop (to check the condition)

```
Class CokeMachine{
    …
    storage for cokes (buffer)
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}

CokeMachine::Deposit(){
    lock->acquire( );  // entering the critical section
    while(count == n)
      notFull.wait(&lock); // release lock before blocking; reacquire when waking up
    add coke to the machine;
    count++;
    notEmpty.notify();
    lock->release();
}

CokeMachine::Remove(){
    lock->acquire();  // entering the critical section
    while(count == 0)
      notEmpty.wait(&lock); // release lock before blocking; reacquire when waking up
    remove coke from the machine;
    count--;
    notFull.notify(); // always hold a lock while signalling to avoid a race condition
    lock->release();
}
```