Operating System Concepts

Lecture 18: Synchronization Primitives

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

- Synchronization primitives
  - Mutex locks
  - Condition variables
  - Semaphores

Programming abstractions for synchronization

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

Higher-level API	locks	condition variables	semaphores	monitors
Hardware Support	atomic load and store	Interrupt disable	test_and_set	compare_and_swap

Mutex locks

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

```
while(test_and_set(&lock))
  ; /* spin */
/* critical section */
lock = false;
```

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

Blocking implementation of locks

```
class Lock {
public:
  void Acquire();
  void Release();
private:
  int locked;
  Queue Q;
}
```

waits until lock is free and then grabs it

releases the lock and wakes up any waiters

Blocking implementation of locks

```
class Lock {
public:
   void Acquire();
   void Release();
private:
   int locked;
   Queue Q;
}
```

waits until lock is free and then grabs it

releases the lock and wakes up any waiters

- mutual exclusion can be implemented using locks (symmetric solution)

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

initially free (not held by any process)

acquired before accessing shared data

critical section

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();
}
```

threads of a process share the heap

How to implement locks on uniprocessors?

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

Compare_and_swap

- test the value against some constant
- if the test returns true, set value in memory to different value
  - if [addr] == r1 then [addr] = r2;
- report the result of the test in a flag

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

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

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

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(Thread T) {
  while (test_and_set(&locked) == 1) {
    ; // if busy, do nothing
  }
}

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

Can we build test_and_set locks without busy-waiting?

- we can't eliminate busy waiting entirely but we can minimize the busy-waiting time
  - instead of busy waiting until lock is free, we busy wait to atomically check the lock value and give up the CPU if we find that the lock is busy

```
class Lock {
public:
  void Acquire(Thread T);
  void Release();
private:
  int locked;
  int guard;
  Queue Q;
}

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

Test_and_set — minimal waiting

```
Lock::Acquire(Thread T) {
  while(test_and_set(guard) == 1)
    ;
  if(locked != 0) {          // lock is busy
    queue_add(Q, T);
    thread_block(T, guard); // set guard to 0 before blocking thread
  } else {                   // lock is free
    locked = 1;
    guard = 0;
  }
}

Lock::Release() {
  while(test_and_set(guard) == 1)
    ;
  if(!queue_empty(Q)) {
    // take thread off wait queue and place on ready queue
    thread_unblock(queue_remove(Q));
  } else {
    locked = 0;
  }
  guard = 0;
}
```

Comparing to "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`

Going 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: producer puts things in a fixed-size buffer, consumer takes them out
  what are the constraints for 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 variables

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

Condition variables

- <u>Definition:</u> an abstraction that supports **conditional synchronization**
  - a queue of threads waiting for a specific **event** inside a critical section
    - free memory is getting low, run the garbage collector
    - new data has arrived in the I/O port, process it
  - the condition of the condition variable depends on data protected by mutex lock
- provide three operations
  - `Wait( )`
    - 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
  - `NotifyAll( )` — historically called `Broadcast( )`
    - wake up all waiting threads

Condition variables

- Definition: an abstraction that supports **conditional synchronization**
  - a queue of threads waiting for a specific **event** inside a critical section
    - free memory is getting low, run the garbage collector
    - new data has arrived in the I/O port, process it
  - the condition of the condition variable depends on data protected by mutex lock
- provide three operations
  - `Wait( )`
    - 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
  - `NotifyAll( )` — historically called `Broadcast( )`
    - wake up all waiting threads
- thread must hold the lock when doing these condition variable operations
  - 1st reason: these operations may update the state
  - 2nd reason: 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 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

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( );
    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();
    while(count == 0)
       notEmpty.wait(&lock); //release lock before blocking; reacquire when waking up
    remove coke from the machine;
    count--;
    notFull.notify();
    lock->release();
}
```

Semaphores

- semaphores are generalized locks invented by Dijkstra in 1965
  - they are a (non-negative) integer variable that supports two **atomic** operations: `wait` and `signal` (or `down` and `up`)

Semaphores

- semaphores are generalized locks invented by Dijkstra in 1965
  - they are a (non-negative) integer variable that supports two **atomic** operations: `wait` and `signal` (or `down` and `up`)
- **binary semaphore** (or mutex lock): used for mutual exclusion
  - guarantees mutually exclusive access to a resource (i.e., only one process is in the critical section at a time)
  - can vary from 0 to 1 – It is initialized to free (value = 1)

Semaphores

- semaphores are generalized locks invented by Dijkstra in 1965
  - they are a (non-negative) integer variable that supports two **atomic** operations: `wait` and `signal` (or `down` and `up`)
- **binary semaphore** (or mutex lock): used for mutual exclusion
  - guarantees mutually exclusive access to a resource (i.e., only one process is in the critical section at a time)
  - can vary from 0 to 1 – It is initialized to free (value = 1)
- **counting semaphore**: used for conditional synchronization
  - useful when multiple units of a specific resource are available
  - the initial count to which the semaphore is initialized is usually the number of resources
  - a process can acquire access so long as at least one unit of the resource is available

Atomic operations with semaphores

- each semaphore supports a queue of processes that are waiting to access the critical section (e.g., to check and buy milk)
- if a thread executes `Wait( )` and the semaphore is free (non-zero), it continues executing after decrementing the semaphore's variable. But if it is not free, the OS puts the thread on the wait queue for that semaphore
- `Signal( )` unblocks one thread on the semaphore's wait queue
- semaphores can be used to implement mutual exclusion:

Atomic operations with semaphores

- each semaphore supports a queue of processes that are waiting to access the critical section (e.g., to check and buy milk)
- if a thread executes `Wait( )` and the semaphore is free (non-zero), it continues executing after decrementing the semaphore’s variable. But if it is not free, the OS puts the thread on the wait queue for that semaphore
- `Signal( )` unblocks one thread on the semaphore's wait queue
- semaphores can be used to implement mutual exclusion:

```
Semaphore milksemaphore; // suppose value is set to 1

milksemaphore.Wait()
<critical section>
milksemaphore.Signal()
```

acquired before accessing shared data

released after accessing shared data

Implementing signal and wait by disabling interrupts

```
class Semaphore {
public:
  void Wait(Thread T);
  void Signal();
private:
  int value;
  Queue Q;
}

Semaphore::Semaphore(int val) {
  value = val; // initialized to the number of available resources
}

Semaphore::Wait(Thread T) {
  intr_disable();
  value = value - 1;
  if(value < 0) {     // |value| is the number of waiting threads
    queue_add(Q, T);
    thread_block(T);
  }
  intr_enable();
}

Semaphore::Signal() {
  intr_disable();
  value = value + 1;
  if(value <= 0)     // if there is a waiting thread
    thread_unblock(queue_remove(Q));
  intr_enable();
}
```

Implementing signal and wait by disabling interrupts

```
class Semaphore {
public:
  void Wait(Thread T);
  void Signal();
private:
  int value;
  Queue Q;
}

Semaphore::Semaphore(int val) {
  value = val; // initialized to the number of available resources
}

Semaphore::Wait(Thread T) {
  intr_disable();
  value = value - 1;
  if(value < 0) {     // |value| is the number of waiting threads
    queue_add(Q, T);
    thread_block(T);
  }
  intr_enable();
}

Semaphore::Signal() {
  intr_disable();
  value = value + 1;
  if(value <= 0)      // if there is a waiting thread
    thread_unblock(queue_remove(Q));
  intr_enable();
}
```

can you implement these atomic operations using test_and_set?