Lecture 9: Semaphores and Monitors

Some slides from Matt Welsh
Summarize Where We Are

- Goal: Use mutual exclusion to protect critical sections of code that access shared resources
- Method: Use locks (spinlocks or disable interrupts)
- Problem: Critical sections can be long

Spinlocks:
- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted

Disabling Interrupts:
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)

\[
\begin{align*}
\text{acquire(} & \text{lock}\text{)} \\
\ldots & \text{Critical section}\ldots \text{release(} & \text{lock}\text{)}
\end{align*}
\]
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held))
        thread_yield();
}
void release (lock) {
    lock->held = 0;
}
Implementing Locks (4) -- (no spin)

mutex_lock:
- TSL REGISTER, MUTEX | copy mutex to register, set mutex to 1
- CMP REGISTER, #0 | was mutex zero?
- JNE ok | if zero, mutex was unlocked, so return
- CALL thread_yield | mutex busy, schedule another thread
- JMP mutex_lock | try again later
- ok: RET | return to caller; CR entered

mutex_unlock:
- MOVE MUTEX, #0 | store a 0 in mutex
- RET | return to caller
Implementing Locks (5)  
-- Mutex (true blocking)

System-wide

```c
struct lock {
    int held = 0;
}

void acquire (lock) {
    if(test-and-set(&lock->held))
        // block the thread;
        // send it to a waiting queue
}

void release (lock) {
    lock->held = 0;
    // move on thread from the waiting
    // queue to ready queue
}```
Higher-level synchronization primitives

- We have looked at one synchronization primitive: locks
- Locks are useful, but may not satisfy all program needs
- Examples? Reader/Writer problem
  - Say we had a shared variable where we wanted any number of threads to read the variable, but only one thread to write it.
  - How would you do this with locks? What's wrong with this code?

```c
Reader() {
    lock.acquire();
    local_copy = shared_var;
    lock.release();
    return local_copy;
}

Writer() {
    lock.acquire();
    shared_var = NEW_VALUE;
    lock.release();
}
```
Semaphores

- Semaphores are an **abstract data type** that provide mutual exclusion to critical sections
  - Block waiters, interrupts enabled within critical section
  - Described by Dijkstra in THE system in 1968

- Semaphores are integers that support two operations:
  - **wait(semaphore)**: decrement, block until semaphore is open
    - Also P(), after the Dutch word for test, or down()
  - **signal(semaphore)**: increment, allow another thread to enter
    - Also V() after the Dutch word for increment, or up()
  - That's it! No other operations – not even just reading its value – exist

- Semaphore safety property: the semaphore value is always greater than or equal to 0
Blocking in Semaphores

- Associated with each semaphore is a queue of waiting threads/processes
- When `wait()` is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then `signal()` opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread
Semaphore Types

- Semaphores come in two types
  - **Mutex** semaphore (or **binary** semaphore)
    - Represents single access to a resource
    - Guarantees mutual exclusion to a critical section
  - **Counting** semaphore (or **general** semaphore)
    - Multiple threads pass the semaphore determined by count
    - » mutex has count = 1, counting has count = N
    - Represents a resource with many units available
    - or a resource allowing some unsynchronized concurrent access (e.g., reading)
Using Semaphores

- Use is similar to our locks, but semantics are different

```c
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```

Threads
- block
- critical section
- It is undefined which thread runs after a signal
Using Semaphores

- We’ve looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account

- Now we’re going to use semaphores to look at more interesting examples
  - Producer consumer with bounded buffers
  - Readers/Writers
Producer-Consumer Problem / Bounded Buffer

- Problem:
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization for coordinating producer and consumer

- Example
  - Coke machine
Problem: There is a set of resource buffers shared by producer and consumer threads
  - **Producer** inserts resources into the buffer set
    - Output, disk blocks, memory pages, processes, etc
  - Buffer between producer and consumer allows them to
    - operate somewhat independently (execute at different rates)
  - Otherwise must operate in lockstep
    - producer puts 1 thing in buffer, then consumer takes it out
    - then producer adds another, then consumer takes it out, etc

What is desired safety property?
  - Sequence of consumed values is prefix of sequence of produced values
  - If $nc$ is number consumed, $np$ number produced, and $N$ the size of the buffer, then $0 \leq np - nc \leq N$
First Try: Sleep and Wakeup

#define N 100    // # of slots in the buffer  
int count=0;  // # of items in the buffer

producer {  
  while(TRUE) {  
    // produce new item  
    if (count==N) sleep(inf); // wait for buffer  
    // insert item  
    count=count+1;  
    if(count==1) // just filled an empty buffer  
      wakeup(consumer);  
  }  
}

consumer {  
  while(TRUE) {  
    if (count==0) sleep(inf); // no more item  
    // remove item  
    count=count-1;  
    if(count==N-1) // have spaces now  
      wakeup(producer);  
    // consume resource;  
  }  
}
What are the problems?

- Producer-consumer problem with fatal race condition
  - Access to “count” is a race condition
  - Access to “buffer” is a race condition
  - Wakeup call could get lost

- `count = count + 1;`  
  `mov eax, count`  
  `inc eax`  
  `mov count, eax`

- `count = count - 1;`  
  `mov eax, count`  
  `dec eax`  
  `mov count, eax`

- Obviously, we need synchronization!
#define N 100  // # of slots in the buffer
int count=0;  // # of items in the buffer
Semaphore mutex = 1;  // mutual exclusion

producer {
    while(TRUE) {
        // produce new item
        wait(mutex); // lock for shared data access
        if (count==N) sleep(inf); // wait for buffer
        // insert item
        count=count+1;
        if(count==1) // just filled an empty buffer
            wakeup(consumer);
        signal(mutex); // unlock
        count=count-1;
        if(count==N-1) // have spaces now
            wakeup(producer);
    }
}

cconsumer {
    while(TRUE) {
        wait(mutex); // lock for shared data access
        if (count==0) sleep(inf); // no more item
        // remove item
        count=count-1;
        if(count==N-1) // have spaces now
            wakeup(producer);
        signal(mutex); // unlock
        // consume resource;
    }
}
Bounded Buffer (2)

- \[0 \leq np - nc \leq N\]
- Use three semaphores:
  - filled – count of filled buffers
    - Counting semaphore
    - filled = ?
      - \((np - nc)\)
  - empty – count of empty buffers
    - Counting semaphore
    - empty = ?
      - \(N - (np - nc)\)
  - mutex – mutual exclusion to shared set of buffers
    - Binary semaphore
Last Try: Semaphores

Semaphore mutex = 1; // mutual exclusion to shared buffer
Semaphore empty = N; // count of empty buffer slots (all empty to start)
Semaphore filled = 0; // count of filled buffer slots (none to start)

producer {
    while (1) {
        Produce new resource;
        wait(empty); // wait for empty slot
        wait(mutex); // lock buffer list
        Add resource to an empty slot;
        signal(mutex); // unlock buffer list
        signal(filled); // note a filled slot
    }
}

consumer {
    while (1) {
        wait(filled); // wait for a filled slot
        wait(mutex); // lock buffer list
        Remove resource from a filled slot;
        signal(mutex); // unlock buffer list
        signal(empty); // note an empty slot
        Consume resource;
    }
}
Bounded Buffer (4)

- Why need the mutex at all?
  - The pattern of signal/wait on full/empty is a common construct often called an interlock
  - Producer-Consumer and Bounded Buffer are classic examples of synchronization problems
Next time...

- Scheduling
  - Read Chapter 6