CS153: Semaphores and Monitors

Chengyu Song

Slides modified from
Harsha Madhyvasta, Nael Abu-Ghazaleh, and Zhiyun Qian
Administrivia

• Lab
  • Lab1 is due this Sunday
  • Demo sessions next week

• Homework
  • Homework1 is due this Friday
Recap: locks

• We looked at using locks to provide mutual exclusion
  • Spinlocks with test-and-set
  • Disabling interrupts
• Locks works but they have some drawbacks when critical sections are long
  • Spinlocks – inefficient
  • Disabling interrupts – can miss or delay important events
• Solutions
  • Wait queue
Higher-level synchronization

• We want synchronization mechanisms that
  • Block waiters
  • Leave interrupts enabled inside the critical section
  • Support more scenarios

• Look at two common high-level mechanisms
  • **Semaphores**: binary (mutex) and counting
  • **Monitors**: mutexes and condition variables
Semaphores

- Semaphores are an abstract data type that provide mutual exclusion to critical sections
  - 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()`
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

- **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 \( \text{count} = 1 \), counting has \( \text{count} = N \)
  - Represents a resource with many units available, or a resource allowing some unsynchronized concurrent access (e.g., reading)
Using semaphores

```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;
}
```

- `wait(S);` block
- `critical section`
- It is undefined which thread runs after a signal

```c
wait(S);
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
signal(S);

... signal(S);
```

```c
wait(S);

wait(S);
put_balance(account, balance);
signal(S);

... signal(S);
```
Why semaphores

• We've looked at a simple example for using synchronization
  • Mutual exclusion while accessing a bank account
  • Similar to our locks, just with different semantics
• Now we're going to use semaphores to look at more interesting examples
  • Readers/Writers
  • Producer consumer with bounded buffers
• Read more from the [Little book of semaphores](http://example.com/semaphores)
The readers/writers problem

• An object is shared among several threads
  • Some threads only read the object, others only write it
  • We can allow multiple readers but only one writer in the critical section

• Let \( r \) be the number of readers, \( w \) be the number of writers

• Safety: \((r \geq 0) \land (0 \leq w \leq 1) \land ((r > 0) \Rightarrow (w = 0))\)
Reader/Writers

• Use three variables
  
  • `int readcount` – number of threads reading object
  
  • Semaphore `mutex` – control access to `readcount`
  
  • Semaphore `w_or_r` – exclusive writing or reading
Readers/Writers (cont.)

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex = 1;

// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}

reader {
    wait(mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex); // unlock readcount
    Read;
    wait(mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex); // unlock readcount
}
Readers/Writers notes

- `w_or_r` provides mutex between readers and writers
  - Readers wait/signal when `readcount` goes from 0 to 1 or 1 to 0
- If a writer is writing, where will readers be waiting?
  - Once a writer exits, all readers can fall through
    - Which reader gets to go first?
    - Is it guaranteed that all readers will fall through?
- If readers and writers are waiting, and a writer exits, who goes first?
Readers/Writers notes (cont.)

- Why do readers use mutex?
- What if the `signal()` is above `if (readcount == 1)`?
- If reader in progress when writer arrives, when can writer get access?
  - Is there a starvation? If so, how to solve?
Bounded buffer (1)

- Problem: 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.
  - **Consumer** removes resources from the buffer set
    - Whatever is generated by the producer
Bounded buffer (2)

- Producer and consumer execute at different rates
  - No serialization of one behind the other
  - Tasks are independent (easier to think about)
  - The buffer set allows each to run without explicit handoff

- What is desired safety property?
  - Sequence of consumed values is prefix of sequence of produced values
  - If $nc$ is number consumed, $np$ is number produced, and $N$ the size of the buffer, then $0 \leq np - nc \leq N$
Bounded buffer (3)

Semaphore mutex = 1;  // mutual exclusion to shared set of buffers
Semaphore empty = N;  // count of empty buffers (all empty to start)
Semaphore full = 0;    // count of full buffers (none full to start)

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

consumer {
    while (1) {
        wait(full);     // wait for a full buffer
        wait(mutex);    // lock buffer list
        Remove resource from a full buffer;
        signal(mutex);  // unlock buffer list
        signal(empty);  // note an empty buffer
        Consume resource;
    }
}
Semaphore summary

- Semaphores can be used to solve any of the traditional synchronization problems
- However, they have some drawbacks
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
  - No control or guarantee of proper usage
Monitors

• A monitor is a programming language construct that controls access to shared data
  • Synchronization code added by compiler, enforced at runtime
  • Why is this an advantage?

• A monitor is a module that encapsulates
  • Shared data structures
  • Procedures that operate on the shared data structures
  • Synchronization between concurrent threads that invoke the procedures
Account example

Monitor `account` {
    double balance;

    double `withdraw`(amount) {
        balance = balance – amount;
        return balance;
    }
}

Threads block waiting to get into monitor

- `withdraw(amount)`
  - balance = balance – amount;
- `withdraw(amount)`
- `withdraw(amount)`
- `return balance (and exit)`

When first thread exits, another can enter. Which one is undefined.

- `balance = balance – amount`
- `return balance;`
- `balance = balance – amount;`
- `return balance;`
Condition Variables

• What if a thread wants to wait inside the monitor?

• Condition variables support three operations:
  • `wait()` – release monitor lock, wait for C/V to be signaled
    • So condition variables have wait queues, too
  • `signal()` – wakeup one waiting thread
  • `broadcast()` – wakeup all waiting threads

• Note: condition variables are *NOT* boolean objects
  • `if (condition_variable)` does not make sense
C/V != Semaphores

• Condition variables != semaphores
  • Similar names but entirely different semantics
  • However, they each can be used to implement the other
• Condition variable is protected by a lock
  • `wait()` blocks the calling thread, and **gives up the lock**
    • Why?
  • `signal()` causes a waiting thread to wake up
    • If there is no waiting thread, **the signal is lost**
Signal semantics

- **Hoare** monitors (original)
  - `signal()` immediately switches from the caller to a waiting thread
  - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
  - Signaler must restore monitor invariants before signaling

- **Mesa** monitors (Mesa, Java)
  - `signal()` places a waiter on the ready queue but signaler continues
  - Condition is not necessarily true when waiter runs again
Summary

• Semaphores
  • `wait()/signal()` implement blocking mutual exclusion
  • Also used as atomic counters (counting semaphores)

• Monitors
  • Language level support for synchronization

• Condition variables
  • Used by threads as a synchronization point inside monitors to wait for events
For next class ...

- Deadlock
- Little book of semaphores
- Textbook
  - Module 32