CS 162: Operating Systems and Systems Programming

Lecture 7: Semaphores, Monitors, Reader/Writer

July 3, 2019
Instructor: Jack Kolb
https://cs162.eecs.berkeley.edu
Logistics

• Project 1: Design Doc due Tonight
• No class or office hours tomorrow
  • Resume normal schedule on Friday
• Groups assigned to TAs earlier today
• Pintos Overview Session: Monday
  • 11am-1pm, Wozniak Lounge
Recall: Atomic Operations

• Atomic load/store not good enough to build a lock

• Instead: Hardware instructions that atomically read a value from (shared) memory and write a new value

• Hardware responsible for making this work in spite of caches
Recall Locks with test\&set

```c
int guard = 0;
int value = 0;

Acquire() {
    // Short busy-wait time
    while(test\&set(guard));
    if (value == 1) {
        put thread on wait-queue;
        go to sleep()& guard = 0;
    } else {
        value = 1;
        guard = 0;
    }
}
```

Remember:
```
Lock.Acquire();
<critical section>
Lock.Release();
```
Recall Locks with test&set

Release() {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
        take thread off wait-queue
        Place on ready queue;
    } else {
        value = 0;
    }
    guard = 0;
}
Recall: Semaphore

• **Definition**: A non-negative integer value with two possible operations
  • `P()` or `down()` or `wait()`*: atomically wait for semaphore to become positive, then decrement it by 1
  • `V()` or `up()` or `signal()`*: atomically increment semaphore (waking up a `P()` thread)
Recall: Multithreaded Web Server

- **Bounded** pool of worker threads
  - Allocated in **advance**: no thread creation overhead
  - **Queue** of pending requests

![Diagram of multithreaded web server]

- Client sends a request to the **Master Thread**.
- The Master Thread processes the request and adds it to the **Queue**.
- The **Thread Pool** contains a bounded number of worker threads that process the requests from the queue.
- The **Queue** ensures that requests are processed in a safe and managed manner.
- The **Client** receives a response after the request is processed.
Producer/Consumer Problem

• Shared buffer of fixed size
  • Producer inserts items
  • Consumer removes items
• Producer/consumer don't need to work in lockstep
• Example: Worker thread processes web request independent of master thread
Producer/Consumer Correctness

• Scheduling constraints:
  • Consumer waits for producer if buffer is empty
  • Producer waits for consumer if buffer is full

• Mutual Exclusion: Only one thread manipulates the buffer at a time

• One semaphore per constraint
  1. Mutex (mutual exclusion)
  2. Filled Slots (consumer waits if necessary)
  3. Empty Slots (producer waits if necessary)
Producer/Consumer Code

Semaphore fullSlots = 0; // Buffer empty to start
Semaphore emptySlots = bufSize; // All slots empty
Semaphore mutex = 1; // No one in critical sect.

Producer(item) {
    emptySlots.P(); // Wait for free slot
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V(); // Tell consumers about new data
}
Producer/Consumer Code

Semaphore fullSlots = 0; // Queue empty to start
Semaphore emptySlots = bufSize; // All slots empty
Semaphore mutex = 1; // No one in critical sect.

Consumer() {
    fullSlots.P(); // Wait for item
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V(); // Tell producers about new slot
    return item;
}
Producer/Consumer Code

Semaphore fullSlots = 0; // Queue empty to start
Semaphore emptySlots = bufSize; // All slots empty
Semaphore mutex = 1; // No one in critical sect.

Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
Discussion

• What if we wrote the following?

Producer(item) {
   mutex.P();
   emptySlots.P();
   Enqueue(item);
   mutex.V();
   fullSlots.V();
}

Consumer() {
   fullSlots.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   emptySlots.V();
   return item;
}

Deadlock
Discussion

• What about this?

Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    fullSlots.V();
    mutex.V();
}

Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}

Correct, possibly less efficient
Problems with Semaphores

• More powerful than simple locks
• But the textbook really doesn't like them

• Argument: Clearer to have separate constructs for
  • Mutual Exclusion: One thread can do something at a time
  • Waiting for a condition to become true

• Need to make sure a thread calls P() for every V()
  • Other tools are more flexible than this (as we'll see)
Condition Variables

• A queue of threads waiting *inside* a critical section

• Operations:
  • `wait(&lock)`: Atomically release lock and go to sleep. Re-acquire the lock before returning.
  • `signal()`: Wake up on waiting thread (if there is one)
  • `broadcast()`: Wake up all waiting threads

• **Rule**: Hold lock when using a condition variable
Monitors

• Locks for mutual exclusion
• Condition variables for waiting

A monitor is a lock and zero or more condition variables with some associated data and operations
  • Java provides this natively
  • POSIX threads: Provides locks and condvars, have to build your own
• **Lock**: protects access to shared data
• Always acquire lock when accessing
• Queue of threads waiting to enter the monitor
Monitors

- **Condition Variables**: queue of threads waiting for something to become true inside critical section.
- Atomically release lock and start waiting
  - Another thread using the monitor will signal them
- The condition: Some function of monitor's data
Implementing a Queue

Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
    lock.Acquire();    // Get Lock
    queue.enqueue(item); // Add item
    dataready.signal(); // Signal any waiters
    lock.Release();     // Release Lock
}

RemoveFromQueue() {
    lock.Acquire();   // Get Lock
    while (queue.isEmpty()) {
        dataready.wait(&lock); // If nothing, sleep
    }
    item = queue.dequeue();    // Get next item
    lock.Release();           // Release Lock
    return(item);
}
Why the while Loop?

- When a thread is woken up by `signal()`, it is simply put on the ready queue.
- It may or may not reacquire the lock immediately!
  - Another thread could be scheduled first and "sneak in" to empty the queue.
  - Need a loop to re-check condition on wakeup.

Why not:

```java
while (queue.isEmpty()) {
    dataready.wait(&lock);
}
```

```java
if (queue.isEmpty()) {
    dataready.wait(&lock);
}
```
Why the while Loop?

- Can we "hand off" the lock directly to the signaled thread so no other thread "sneaks in?"
  - Yes. Called **Hoare-Style Monitors**
  - Many textbooks describe this scheme

- Most OSs implement **Mesa-Style Monitors**
  - Allows other threads to sneak in
  - Much easier to implement
  - Even easier if you allow "spurious wakeups"
  - `wait()` can return when no signal occurred, in rare cases
  - POSIX allows spurious wakeups
Interlude: Concurrency Is Hard

• Even for practicing engineers trying to write mission-critical, bulletproof code!
• Therac-25: Radiation Therapy Machine with Unintended Overdoses (reading on course site)
• Mars Pathfinder Priority Inversion (JPL Account)
• Toyota Uncontrolled Acceleration (CMU Talk)
  • 256.6K Lines of C Code, ~9-11K global variables
  • Inconsistent mutual exclusion on reads/writes
Comparing High-Level Synchronization

- Semaphores can implement locks
  - `Acquire()` { `semaphore.P();` }
  - `Release()` { `semaphore.V();` }
- Monitors are a superset of locks
- Can monitors implement semaphores?
Semaphores With Monitors

Lock lock;
int count = initial value of semaphore;
CondVar atOne;

P() {
  lock.acquire();
  while (count == 0) {
    atOne.wait(&lock);
  }
  count--;
  lock.Release()
}

V() {
  lock.acquire();
  count++;
  if (count == 1) {
    atOne.Signal();
  }
  lock.Release()
Comparing High-Level Synchronization

• Semaphores can implement locks
  • `Acquire()` { semaphore.P(); }
  • `Release()` { semaphore.V(); }

• Monitors are a superset of locks

• Can monitors implement semaphores? Yes

• Can semaphores implement monitors?
CVs with Semaphores (Attempt 1)

```java
Wait(Lock lock) {
    lock.Acquire();
    semaphore.P();
    lock.Release();
}

Signal() {
    semaphore.V();
}
```

Problem:
- `Signal` affects threads in the future (semaphores are stateful)
- What about doing a `broadcast()`?
CVs with Semaphores (Attempt 2)

Wait(Lock lock) {
    lock.Acquire();
    semaphore.P();
    lock.Release();
}

Signal() {
    Atomically {
        if semi queue nonempty
        semaphore.V();
    }
}

Problem: "is queue empty" not a semaphore operation
• But there is a solution
CVs with Semaphores

• Key Idea: Queue of semaphores
  • Protected by a binary semaphore
  • Each waiting thread has its own semaphore
  • Call V on waiter's semaphore and remove it from queue

• More in our textbook
Comparing High-Level Synchronization

- Semaphores can implement locks
  - `Acquire() { semaphore.P(); }`
  - `Release() { semaphore.V(); }`

- Monitors are a superset of locks

- Can monitors implement semaphores? Yes
- Can semaphores implement monitors? Yes
Break
Reader/Writer Problem

- Shared Database
  - Many readers – never modify the database
  - Few writers – read and modify database

- Single lock sufficient?
Reader/Writer Correctness

• Readers can access when no writers
• Writers can access when no readers and no other writers

• A lock will satisfy these requirements
  • But we want to allow multiple readers
  • Better efficiency
Reader/Writer with Monitors

Reader() {
    Wait until no active writers
    Access database
    Maybe wake up a writer
}

Writer() {
    Wait until no active readers or writers
    Access database
    Maybe wake up reader or writer
}

Lock (for mutual exclusion)
int activeReaders, condVar okToRead
int activeWriters, condVar okToWrite
Reader Version 1

Reader() {
    // First check self into system
    lock.Acquire();
    while (AW > 0) { // Is it safe to read?
        okToRead.wait(&lock); // Sleep on cond var
    }
    AR++; // Now we are active!
    lock.release();
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    lock.Acquire();
    AR--; // No longer active
    if (AR == 0) // No other active readers
        okToWrite.signal(); // Wake up one writer
    lock.Release();
}
Writer Version 1

Writer() {
    // First check self into system
    lock.Acquire();
    while (AR > 0 || AW > 0) { // Is it safe to write?
        okToWrite.wait(&lock); // Sleep on cond var
    }
    AW++; // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--; // No longer active
    okToWrite.signal(); // Wake up one writer
    okToRead.broadcast(); // Wake up all readers
    lock.Release();
}
If there are always readers, this is always true! Writer starves.
Writer Version 1: Conflict

Writer() {
    // First check self into system
    lock.Acquire();
    while (AR > 0 || AW > 0) { // Is it safe to write?
        okToWrite.wait(&lock); // Sleep on cond var
    }
    AW++; // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--; // No longer active
    okToWrite.signal(); // Wake
    okToRead.broadcast(); // Wake
    lock.Release();
}

Relies on waiting threads double-checking condition
Writer Version 1: Conflict

Writer()
{
    // First check self into system
    lock.Acquire();
    while (AR > 0 || AW > 0) {
        // Is it safe to write?
        okToWrite.wait(&lock); // Sleep on cond var
    }

    AW++; // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--; // No longer active
    okToWrite.signal(); // Wake
    okToRead.broadcast(); // Wake
    lock.Release();
}

Everyone races, all but I thread just goes back to sleep
**Reader/Writer with Monitors v2**

Reader() {
    Wait until no active or waiting writers
    Access database
    Maybe wake up a writer
}

Writer() {
    Wait until no active readers or writers
    Access database
    If waiting writer, wake it up;
    Otherwise, wakeup readers;
}

int waitingWriters
Reader Version 2

Reader() {
    // First check self into system
    lock.Acquire();
    while (AW > 0 || WW > 0) { // Is it safe to read?
        okToRead.wait(&lock);  // Sleep on cond var
    }
    AR++;  // Now we are active!
    lock.release();
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    lock.Acquire();
    AR--;  // No longer active
    if (AR == 0) // No other active readers
        okToWrite.signal();  // Wake up one writer
    lock.Release();
}
Writer Version 2

Writer() {
    // First check self into system
    lock.Acquire();
    while (AR > 0 || AW > 0) { // Is it safe to write?
        WW++;
        okToWrite.wait(&lock); // Sleep on cond var
        WW--;
    }
    AW++; // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--; // No longer active
    if (WW > 0)
        okToWrite.signal(); // Wake up one writer
    else
        okToRead.broadcast(); // Wake up all readers
    lock.Release();
}
Simulation of Reader/Writer

• Sequence of arrivals: R1, R2, W1, R3

• On entry each reader checks

  while (AW > 0 || WW > 0) {
    // Is it safe to read?
    okToRead.wait(&lock);
    // Sleep on cond var
  }

• First R1 enters (no waiting)
  • \textbf{AR} = 1, AW = 0, WW = 0

• Then R2 enters (no waiting)
  • \textbf{AR} = 2, AW = 0, WW = 0
Simulation of Reader/Writer

- Sequence of arrivals: R1, R2, *W1, R3
- R1, R2 still running (AR = 2)
- W1 does a check: AR > 0, waits on okToWrite
  while (AR > 0 || AW > 0) {
    WW++;
    okToWrite.wait(&lock); // Sleep on cond var
    WW--;
  }
- Now AR = 2, AW = 0, WW = 1
- R3: WW > 0, waits on okToRead
Simulation of Reader/Writer

- R1 finishes, does not wake anyone up
  - AR = 1, AW = 0, WW = 1
- R2 finishes
  - AR = 0, AW = 0, WW = 1
  - Wakes up W1 (signals okToWrite)
- W1 runs and finishes
  - AR = 1, AW = 1 then 0, WW = 0
  - Wakes up R3 (okToRead.Broadcast())
Reader/Writer Design Choices

• Reader starvation:

\[
\text{while (AW > 0 || WW > 0) \{ // Safe to read?}
\text{okToRead.wait(&lock); // Sleep on cond var}
\]

• "Writer-biased" Lock
  • Can favor readers by changing conditions on wait loops
  • Other possibilities, e.g. track readers waiting since before current writer started
Summary

• Semaphores: More general than locks, but used in two different ways (mutual exclusion, waiting)

• Alternative: Monitors
  • One lock, zero or more condition variables

• Reader/Writer Synchronization
  • Treat readers differently from writers for efficiency