CS4410

Operating Systems

Lecture 11:
Condition variables, and atomic primitives

Rachit Agarwal
Announcements

• “Missed class” emails

• Last lecture: max number of “missed class” emails received :)

• Many of those were within the 1-hour limit announced earlier
  • [1] Last minute things come up
  • [2] Defines “are we true friends?” moments
    • We are having a quiz; you don’t seem to be here :)

• No way for me to differentiate between people in [1] and [2]

• If care about fairness, follow the principle of sticking with rules
  • People in [1] may miss out, but assuming this is rare
  • People in [2] do not benefit unfairly

• So, I am going to stick with the rule
  • Sorry if you happen to be in [1]
Goal of today’s lecture

• Wrap up synchronization and concurrency
• Wrap up Semaphores
• Condition variables, and Monitors
• Atomic instructions, and implementing locks
Examples that we have seen so far

• The racing threads
• The complicated racing threads
• The ATM banking
• Too-much-milk
• Producer-consumer
Recall: Example 5: The producer-consumer problem

• Suppose we want to build a **fork dispenser** for a cafe

• The dispenser (shared resource) has limited capacity

• Consumers pull out forks on one end of the dispenser
  • `removeFromDispenser()`
  • Error if tries to pull out a fork from an empty dispenser
  • Error if cannot pull out a fork when there is one

• Owner adds forks on the other end of the dispenser
  • `addToDispenser()`
  • Error if tries to add a fork to a full dispenser
Recall: Semaphores

- Semaphores are a kind of generalized lock

- A semaphore is “stateful”
  - Has a non-negative value associated with it
  - Value is incremented and decremented atomically

- Semaphore has a positive value initially, and offers two atomic operations
  - **Down()** or **P()**—stands for “proberen” (to test) in Dutch:
    - Thread “waits” for the semaphore value to become positive
    - When so, atomically decrement it by 1
  - **Up()** or **V()**—stands for “verhogen” (to increment) in Dutch:
    - Thread “waits” for the semaphore value to become less than “max”
    - When so, atomically increment the semaphore value by 1
    - Wake up a thread waiting on P, if any
Recall: Producer consumer problem with semaphores

Split binary semaphore: at most one of the semaphore is released

```java
enoughRoom = semaphore(dispenser_capacity);
count = semaphore(0);
```

Consumers() {
    while(true)
    {
        count.down();
        lock.acquire();
        Fork = removeFromDispenser();
        forkCount = forkCount - 1;
        lock.release();
        enoughRoom.up();
        use(Fork);
    }
}

Owner(fork) {
    while(true)
    {
        Fork = newFork();
        enoughRoom.down();
        lock.acquire();
        addToDispenser(Fork);
        forkCount = forkCount + 1;
        lock.release();
        count.up();
    }
}

Complicated sequence of semaphore locks
easy to make mistakes!!
Example 5: The producer-consumer problem

- Suppose we want to build a **fork dispenser** for a cafe
- The dispenser (shared resource) has limited capacity
- Consumers pull out forks on one end of the dispenser
  - `removeFromDispenser()`
  - `sleep()`—**consumer blocks until the producer wakes it up**
  - Error if tries to pull out a fork from an empty dispenser
  - Error if cannot pull out a fork when there is one
- Owner adds forks on the other end of the dispenser
  - `addToDispenser()`
  - `wakeup()`—a **routine for producer to wake up a consumer**
  - Error if tries to add a fork to a full dispenser
Example 5: The producer-consumer problem: Attempt 2

• Suppose we implement producer and consumer this way

```java
Consumers() {
    while(true) {
        if(forkCount == 0) {
            sleep();
        }
        Fork = removeFromDispenser();
        forkCount = forkCount - 1;
        if(forkCount == dispenserCapacity - 1) {
            wakeup(owner);
        }
        use(Fork);
    }
}

Owner(fork) {
    while(true) {
        Fork = newFork();
        if(forkCount == dispenserCapacity) {
            sleep();
        }
        addToDispenser(Fork);
        forkCount = forkCount + 1;
        if(forkCount == dispenserCapacity - 1) {
            wakeup(owner);
        }
        if(forkCount == 1) {
            wakeup(consumer);
        }
    }
}
```

Wrong: inconsistent forkcount
Example 5: The producer-consumer problem: Attempt 2

- Suppose we implement producer and consumer this way

```java
Consumers() {
    while(true) {
        lock.acquire();
        if(forkCount == 0) {
            lock.release();
            sleep();
            lock.acquire();
        }
        Fork = removeFromDispenser();
        forkCount = forkCount - 1;
        if(forkCount == dispenserCapacity - 1) {
            wakeup(owner);
        }
        use(Fork);
        lock.release();
    }
}

Owner(fork) {
    while(true) {
        Fork = newFork();
        lock.acquire();
        if(forkCount == dispenserCapacity) {
            lock.release();
            sleep();
            lock.acquire();
        }
        addToDispenser(Fork);
        forkCount = forkCount + 1;
        if(forkCount == 1) {
            wakeup(consumer);
        }
        lock.release();
    }
}
```

Deadlocks!
Example 5: The producer-consumer problem: Attempt 2

- Can lead to “deadlocks”
  - Step 1: The consumer reads forkCount (=0); about to enter if
  - Step 2: Just before calling sleep()
    - Consumer interrupted
    - Producer adds a fork, puts it into dispenser, forkCount=1
    - Since forkCount=1, tries to wake up the consumer
    - But the consumer isn’t sleeping yet—wakeup call lost
  - Step 3: The consumer calls sleep()
    - Goes to sleep;
    - Never wakes up, since wakeup call only when forkCount=1
  - Step 4: Producer fills up the dispenser
    - Goes to sleep
    - Never wakes up, since wakeup call only from consumer
What we really need for synchronization

- We need higher-level synchronization mechanism that provides
  - Mutual exclusion
    - Easy to create critical sections
  - Scheduling
    - Block threads until some desired event occurs
Condition variables

- Synchronization mechanisms need more than just mutual exclusion
  - Also need a way to wait for another thread to do something
  - e.g., wait for a fork to be added to the dispenser

- Condition variable: A mechanism to enable threads to wait inside a critical section
  - Achieved by releasing a lock

Three operations on condition variables (condition x;)

- wait(condition, lock):
  - Atomically: Release lock; put thread to sleep until condition is signaled
  - When thread wakes up again, re-acquire lock before returning

- signal/notify(condition, lock):
  - If any threads waiting on condition, wake up one of them
  - Caller must hold lock: must be the same as the lock used in the wait call

- broadcast/notifyall(condition, lock):
  - Same as signal/notify, except wake up all waiting threads
Condition variables

- Three operations on condition variables (condition x;)
  - x.wait()
  - x.signal() or x.notify()
  - x.broadcast() or x.notifyall()

- Only call the above operations when holding a lock

- Condition variables (unlike semaphores) are stateless
Condition variables—notify semantics

• When a thread calls `x.notify()`, it is signaling “waiting” threads
  • There is some task that can be done by the waiting threads
  • The thread calling `notify()` can continue doing its tasks
  • Which threads executed once `notify()` is called?

• If no thread waiting on condition variable, notifier continues

• If one or more threads waiting on condition variable
  • At least two “ready” threads: those waiting, and the notifier; which one runs?

• Mesa (or Brinch Hansen semantics)
  • Waiting thread moved to ready queue; but not guaranteed to run right away

• Hoare semantics:
  • Thread calling `notify()` suspended, and
  • atomically: ownership of the lock passed to one of the waiting threads
    • The thread getting the ownership resumes execution immediately
  • Thread calling `notify()` is resumed if the above thread exits critical section
    • Or if the above thread goes to wait again
notify() versus notifyall()

• Signal versus broadcast
  • Signals wakes up one of the waiting threads
  • Broadcast wakes up all of the waiting threads

• It is always safe to use notifyall() instead of notify()
  • But performance may be affected

• notify() is preferable when
  • At most one waiting thread can make progress (e.g., with mutual exclusion)
  • Any of the threads waiting on condition variable can make progress

• notifyall() is preferable when
  • Multiple waiting threads may be able to make progress
  • Some of the waiting threads can make progress, others cannot
Condition variables versus Semaphores

- **wait() versus down()**
  - `down()` blocks threads only if value=0
  - `wait()` always blocks, and gives up lock

- **notify() versus up()**
  - `up()` is stateful
    - if no waiting thread, `up()` ensures future thread does not wait on `down()`
  - `notify()` is stateless
    - If no waiting thread, `notify()` is a no op

- **Condition variables are stateless, making code easier to read**
  - Conditions for which threads are waiting are **explicit**
Monitors

- When locks and condition variables are used together like the above
  - The result is called a monitor

- Monitor
  - A collection of procedures manipulating a shared data structure
  - One lock that must be held whenever accessing the shared data
    - Typically each procedure acquires the lock at the very beginning
    - And releases the lock before returning
  - One or more condition variables used for waiting
Example 5: Producer-consumer with condition variables

```java
enoughRoom = condition();
count = condition();
```

Consumers() {
    while(true) {
        lock.acquire();
        while(forkCount == 0) {
            count.wait(lock);
        }
        Fork = removeFromDispenser();
        forkCount = forkCount - 1;
        if (forkCount == dispenserCapacity-1) {
            enoughRoom.signal();
        }
        lock.release();
        use(Fork);
    }
}

Owner(fork) {
    while(true) {
        lock.acquire();
        Fork = newFork();
        while(forkCount == dispenserCapacity) {
            enoughRoom.wait(lock);
        }
        addToDispenser(Fork);
        forkCount = forkCount + 1;
        if (forkCount == 1) {
            count.signal();
        }
        lock.release();
    }
}

Can sleep within critical section and simpler code!
One last remaining bit
What is atomic, and what is not?
Recall: Atomic Operations

- “Indivisible operations” supported by hardware
  - Indivisible: An operation that always runs to completion or not at all
  - No interruptions
    - It cannot be stopped in the middle
    - And state cannot be modified by someone else in the middle

- Fundamental building block
  - If no atomic operations, then have no way for threads to work together

- What atomic operations should the hardware support?
  - We have studied five examples, each with different complexity
    - And with different set of operations
  - We have also studied three different higher-layer primitives
    - Locks, Semaphores, condition variables
    - Are these atomic? What else is atomic?
Atomic Operations

• Most modern processors support a basic set of atomic operations
  • Atomic read-write
  • Atomic swap
  • test-and-set
  • fetch-and-add
  • compare-and-swap
  • store-conditional

• Can be used to implement higher-level primitives
  • E.g., locks, semaphores, condition variables
Atomic test and set

- Hardware offers an instruction which
  - Sets the value of a memory location to 1
  - Returns the previous value

- Hardware executes both operations atomically

- Caller uses return value to see if the instruction changed the state

```c
int test_and_set(int* x)
{
    old = *x;
    *x = 1;
    return old;
}
```
Locks using test and set

• Suppose we implement locks this way

```c
int x = 0;
while(test_and_set(x)) {} // acquire lock
x = 0; // release lock
```

1. While loop wastes CPU cycles if wait is long!!
2. Efficient only when wait is short?
Atomic compare and swap

- Hardware offers an instruction which
  - Compares a given value with a given expected value
  - If equal, changes it to given new value
    - Return true
  - Else, return false
- All operations are executed atomically

```c
int compare_and_swap(int* p, int expected, int new) {
    if(*p != expected) {
        return false;
    }
    *p = new
    return true;
}
```
Atomic add using compare and swap

- Suppose we implement atomic add this way

```c
atomic_add(int* p, int x) {
    done = false;
    while(!done) {
        value = *p;
        done = compare_and_swap(p, value, value + x);
    }
    return value + x;
}
```

Atomically adds x to the value present at p
Some final thoughts on synchronization

• One of the hardest topics in operating systems
  • It is okay if you had hard time grasping some of the ideas
  • All of us have struggled with synchronization (for a very long time!)

• It is important to understand the problem
  • We have done many examples
  • Many more examples in books/Internet

• Synchronization primitives require practice
  • Many problems in HW2
  • Some more problems in HW3
  • More problems in the book
    • Try to solve them
    • Come to office hours to ask questions
  • Practice, practice, practice