CS4410

Operating Systems

Lecture 10:
Semaphores and Monitors

Rachit Agarwal
Announcements

• Office hours
  • Priority to students who signed up (Calendly link on webpage)
  • You are welcome to walk in, but strict prioritization

• Homework submission
  • You are required to “mark” pages for individual answers
  • We will deduct 10% if you do not mark pages

• Prelims
  • Prelim1: 14th October; Prelim2: 23rd November
  • In-class: there should be no conflicts; no make up
  • Open notes, open book, open everything except:
    • The Internet
    • Other students
  • Infinite time: we want to test you on your knowledge, not speed
Goal of today’s lecture

- Wrap up synchronization and concurrent programming
- Semaphores, Condition variables, and Monitors
Examples that we have seen so far

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

Suppose we implement producer and consumer in the following manner:

```java
Consumer() {
    while(true)
    {
        if(forkCount > 0)
        {
            Fork = removeFromDispenser();
            forkCount = forkCount - 1;
            use(Fork);
        }
    }
}

Owner(fork) {
    while(true)
    {
        if(forkCount < dispenserSize)
        {
            Fork = newFork();
            addToDispenser(Fork);
            forkCount = forkCount + 1;
        }
    }
}
```

Is this correct?
Example 5: The producer consumer problem

- $t=0$, dispenserSize = 5, forkCount = 5

```java
if(forkCount > 0)
{
    Fork = removeFromDispenser();
    forkCount = forkCount - 1 ;
    use(Fork);
}
```

```java
if(forkCount < dispenserSize)
{
    Fork = newFork();
    addToDispenser(Fork);
}
```

```java
if(forkCount > 0)
{
    Fork = removeFromDispenser();
    forkCount = forkCount - 1 ;
    use(Fork);
}
```

Inconsistent forkCount!!

```java
forkCount = forkCount + 1;
```
Example 5: Producer consumer problem with Locks

• Let’s try locks

```java
Consumer() {
    while(true)
    {
        lock.acquire();
        if(forkCount > 0)
        {
            Fork = removeFromDispenser();
            forkCount = forkCount - 1 ;
            use(Fork);
        }
        lock.release();
    }
}

Owner(fork) {
    while(true)
    {
        lock.acquire();
        if(forkCount < dispenserSize)
        {
            Fork = newFork();
            addToDispenser(Fork);
            forkCount = forkCount + 1;
        }
        lock.release();
    }
}
```

CPU cycles may be wasted:
Consumer/producer may repeatedly acquire and release locks!!!
**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

- Binary Semaphore: Semaphore with initial value 1
  - Mutual exclusion like locks
Example 5: Producer consumer problem with semaphores

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

```java
enoughRoom = semaphore(1);
count = semaphore(0);
```

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

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

• Problem?
• Only works for dispenser size = 1
Example 5: Producer consumer problem with semaphores

Count semaphore: at most one of the semaphore is released

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

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

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

Problem?

Does not work: number of consumers/producers > 1
forkCount can become inconsistent with multiple threads in critical section
Example 5: Producer consumer problem with semaphores

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

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

**Problem?**

Deadlock:

consumer takes lock, executes down(), producer cannot update if forkcount=0; or, forkcount=dispenser-size and producer gets the lock;
Example 5: Producer consumer problem with semaphores

- Let’s use binary semaphores which are similar to locks

```
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

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

```plaintext
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();
    }
}
```

```java
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
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!
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 mechanisms to wait for* a condition to become true

- **Three operations on condition variables (condition x;)**
  - **wait(condition, lock):**
    - Release lock; put thread to sleep until condition is signaled
    - When thread wakes up again, re-acquire lock before returning
  - **signal(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(condition, lock):**
    - Same as signal, except wake up all waiting threads
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

```
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!