CS4410

Operating Systems

Rachit Agarwal

Goal of Today's Lecture

- Understand the concurrency problem
- Understand the concurrency/synchronization terminology

Concurrency and Synchronization

Understanding the problem

Recall Example 1.1: The racing threads—one possibility

Two threads: Thread A and Thread B, operating on a shared variable value (initiated to 0)

```
value = value + 1;
If (value =/= -1)
      print ("Thread A wins");
```

```
value = value - 1;
If (value == -1)
      print ("Thread B wins");
```

Whats happening under the hood (inside the loop)?

(If threads were running concurrently)

```
rA = 0 <- load rA, value

rA = 1 <- add rA, rA, 1

value = 1 <- store rA, value
```

```
rB = 0 <- load rB, value

rB = -1 <- sub rB, rB, 1

value = -1 <- store rB, value
```

Time

Whats value after these executions?

Recall: Example 1.2: The racing threads—another possibility

Two threads: Thread A and Thread B, operating on a shared variable value (initiated to 0)

```
value = value + 1;
If (value =/= -1)
      print ("Thread A wins");
```

```
value = value - 1;
If (value == -1)
      print ("Thread B wins");
```

Whats happening under the hood (inside the loop)?

(If threads were running concurrently)

```
rA = 0 <- load rA, value

rA = 1 <- add rA, rA, 1



value = 1 <- store rA, value
```

```
rB = 0 <- load rB, value

rB = -1 <- sub rB, rB, 1
value = -1 <- store rB, value
```

Time

Whats value after these executions?

The crux of the problem

- Two concurrent threads (or processes)
  - Accessing a shared resource (account)
  - Without any coordination—with “synchronization"

- Lack of synchronization
  - Creates race conditions
  - Non-deterministic outputs, depending on thread scheduling

- In scenarios involving Shared resources + concurrent execution
  - We need mechanisms for synchronization
  - Ensure that we can reason about execution outputs
  - Ensure deterministic outputs

Example 3.1: The real-world ATM banking example

Shared bank account

Initial balance: $1000;

both of you execute **withdraw (account, 500) at the same time**

```
balance = read_balance (account);
balance = balance - amount;
write_balance (account, balance);
return balance;
```

```
balance = read_balance (account);
balance = balance - amount;
write_balance (account, balance);
return balance;
```

Time

- What is the final balance?
  - 500? 1000? 0?
  - Everyone is happy!

Recall: Example 3.2: The real-world ATM banking example

Shared bank account

Initial balance: $1000;

both of you execute **withdraw (account, 500) at the same time**

```
balance = read_balance (account);
balance = balance - amount;



write_balance (account, balance);
return balance;
```

```
balance = read_balance (account);
balance = balance - amount;
write_balance (account, balance);
return balance;
```

- What is the final balance?
  - 500? 1000? 0?
  - Bank goes berserk!

Time

Example 4: Too-much-milk problem

You in your lovely, cozy, non-shared apartment

3:00	Look in fridge. Out of milk.
3:05	Leave for store.
3:10	Arrive at store.
3:15	Buy milk.
3:20	Arrive home. Put milk in fridge.
3:25	
3:30	

Drink milk, be strong!

Example 4: Too-much-milk problem

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Time	You	Roommate
3:00	Look in fridge. Out of milk.	
3:05	Leave for store.	
3:10	Arrive at store.	Look in fridge. Out of milk.
3:15	Buy milk.	Leave for store.
3:20	Arrive home. Put milk in fridge.	Arrive at store.
3:25		Buy milk.
3:30		Arrive home. Put milk in fridge.

Too much milk!

Example 4: Too-much-milk problem

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

```
If (no Milk) {
	Buy milk;
}
```

```
If (no Milk) {
	Buy milk;
}
```

Too much milk!

Example 4: Potential solution? Attempt 1

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Attempt 1: Let us try the "freezing" idea

```
If (no Milk) {
    If (no Note) {
        Leave note;
        Buy milk;
        Remove note;
    }
}
```

```
If (no Milk) {
    If (no Note) {
        Leave note;
        Buy milk;
        Remove note;
    }
}
```

Does this work?

Example 4: Potential solution? Attempt 1

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

No!

```
If (no Milk) {


     If (no Note) {
          Leave note;
          Buy milk;
          Remove note;
     }
}
```

```
If (no Milk) {
     If (no Note) {





          Leave note;
          Buy milk;
          Remove note;
     }
}
```

Example 4: Potential solution? Attempt 2

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Attempt 2: Let us get smarter: freeze first

```
Leave note;
If (no Milk) {
      If (no Note) {
            Buy milk;
      }
}
Remove note;
```

```
Leave note;
If (no Milk) {
      If (no Note) {
            Buy milk;
      }
}
Remove note;
```

Does this work?

Example 4: Potential solution? Attempt 2

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

No!

```
Leave note;
If (no Milk) {
      If (no Note) {
            Buy milk;
      }
}
Remove note;
```

```
Leave note;
If (no Milk) {
      If (no Note) {
            Buy milk;
      }
}
Remove note;
```

Nobody ever buys milk!

Example 4: Potential solution? Attempt 3

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Attempt 3: May be different interpretations of notes

```
If (no Note) {
    If (no Milk) {
        Buy milk;
    }
    Leave note;
}
```

```
If (Note) {
    If (no Milk) {
        Buy milk;
    }
    Remove Note;
}
```

Does this work?

Example 4: Potential solution? Attempt 3

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

No! Starvation!

```
If (no Note) {
      If (no Milk) {
            Buy milk;
      }
      Leave note;
}
```

Example 4: Potential solution? Attempt 4

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Attempt 4: Perhaps two notes?

```
Leave noteA;
If (no noteB) {
      If (no Milk) {
            Buy milk;
      }
}
Remove noteA;
```

```
Leave noteB;
If (no noteA) {
      If (no Milk) {
            Buy milk;
      }
}
Remove noteB;
```

Does this work?

Example 4: Potential solution? Attempt 4

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Even worse! Lockup, deadlock, starvation!

```
Leave noteA;



~~If (no noteB) {~~
    ~~If (no Milk) {~~
        ~~Buy milk;~~
    }
}
~~Remove noteA;~~
```

```
Leave noteB;



~~If (no noteA) {~~
    ~~If (no Milk) {~~
        ~~Buy milk;~~
    }
}
~~Remove noteB;~~
```

Example 4: Potential solution? Attempt 5

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Attempt 5: What are we missing?

"If roommate is not doing something, I should do it"

"If roommate is doing something, I should not do it"

```
Leave noteA;
While (noteB) {
      Do nothing;
}
      If (no Milk) {
            Buy milk;
      }
Remove noteA;
```

```
Leave noteB;
If (no noteA) {
      If (no Milk) {
            Buy milk;
      }
}
Remove noteB;
```

Does this work?

Example 4: Potential solution? Attempt 5

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Case 1: While (noteB) "happens before" Leave noteB

B has **not** left a note

A buys milk;
A's note is up;
B cannot buy milk

```
Leave noteA;
While (noteB) {
    Do nothing;
}
    If (no Milk) {
        Buy milk;
    }
Remove noteA;
```

```
Leave noteB;
If (no noteA) {
    If (no Milk) {
        Buy milk;
    }
}
Remove noteB;
```

Example 4: Potential solution? Attempt 5

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Case 2.1:

While (noteB) happens after Leave noteB

If (no noteA) <u>happens before</u> Leave noteA

B's note is up
A does nothing

```
Leave noteA;
While (noteB) {
        Do nothing;
}
        If (no Milk) {
                Buy milk;
        }
Remove noteA;
```

```
Leave noteB;
If (no noteA) {
        If (no Milk) {
                Buy milk;
        }
}
Remove noteB;
```

If A's note is not up;
B buys milk;

Example 4: Potential solution? Attempt 5

You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

Case 2.2:

While (noteB) happens after Leave noteB

If (no noteA) <u>happens after</u> Leave nodeA

B's note is up
A does nothing

Once B removes note
A buys milk

```
Leave noteA;
While (noteB) {
        Do nothing;
}
        If (no Milk) {
                Buy milk;
        }
Remove noteA;
```

```
Leave noteB;
If (no noteA) {
        If (no Milk) {
                Buy milk;
        }
}
Remove noteB;
```

If A's note is up;
B does not buy milk;
Eventually removes B's note;

This generalizes to n threads …

Leslie Lamport's "Bakery Algorithm" (1974) generalizes this solution to n threads

Computer Systems

G. Bell, D. Siewiorek, and S.H. Fuller, Editors

A New Solution of Dijkstra's Concurrent Programming Problem

Leslie Lamport
Massachusetts Computer Associates, Inc.

A simple solution to the mutual exclusion problem is presented which allows the system to continue to operate

Discussion

- Our solution protects a single "**critical section**" piece of code for each thread

```
If (no Milk) {
      Buy milk;
}
```

- Our solutions works, but is really unsatisfactory
  - **Complexity**—even for this simple example
    - Hard to convince of correctness
  - **Asymmetric code**—You and your roommate have different codes
    - What if there are lots of threads
  - **While your thread is waiting, the thread is wasting CPU time**
    - This is called "busy-waiting"

- Is there a better way?
  - **Better hardware support**
    - what if hardware can support executing critical section in "**atomic**" steps
  - **Better higher-level programming abstractions**
    - Using whatever atomic operations hardware supports

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 four examples, each with different complexity
  - And with different set of operations

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
- Covered in 3410—please review
- Can be used to implement higher-level primitives

Building higher-level primitives using atomic operations

- We will study three primitives
    - Locks—mostly covered in 3410
    - Semaphores
    - Conditional variables
        - Monitors: locks + conditional variables

- Can be used to implement higher-level primitives

Recall: Locks

- **Lock: Used to restrict access to something important (shared data)**
  - Lock before accessing shared data
  - read/write shared data (critical section)
    - Other threads waiting at this point for the lock to be released
    - Important idea: synchronization requires waiting
  - Unlock

- Most operating systems offer two atomic operations on locks:
  - **lock.acquire()**
    - wait until lock is free, then mark it as busy atomically
    - After the call returns, calling thread holds the lock
  - **lock.release()**
    - releases the lock
    - Should be called only by the thread that holds the lock

Example 1: The racing threads with locks

- Two threads: **Thread A** and **Thread B**, operating on a shared variable **value (initiated to 0)**

```
Lock.acquire();
value = value + 1;
If (value != -1)
{
  print("Thread A wins");
}
Lock.release();
```

```
Lock.acquire();
value = value - 1;
If (value == -1)
{
  print("Thread B wins");
}
Lock.release();
```

The thread that acquires the lock first, wins!

Example 2: The complicated racing threads with locks

- Two threads: **Thread A** and **Thread B**, operating on a shared variable **value (initiated to 0)**

```
Lock.acquire();
while (value < 10)
{
  value = value + 1;
}
print("Thread A wins");
Lock.release();
```

```
Lock.acquire();
while (value > -10)
{
  value = value - 1;
}
print("Thread B wins");
Lock.release();
```

Again, the thread that acquires the lock first, wins!

Example 3: The real-world ATM banking example with locks

- Initial balance: $1000; two simultaneous withdrawals of $500;

```
int withdraw(account, amount) {
  lock.acquire();
  balance = read_balance(account);
  balance = balance - amount;
  write_balance(account, balance);
  lock.release();
  return balance;
}
```

```
int withdraw(account, amount) {
  lock.acquire();
  balance = read_balance(account);
  balance = balance - amount;
  write_balance(account, balance);
  lock.release();
  return balance;
}
```

Balance is always deterministic! (0 in this case)

Note: Always release before returning from the function call

Example 4: Too-much-milk problem with locks

- You and your (partly crazy) roommate in your not-so-lovely, not-so-cozy apartment

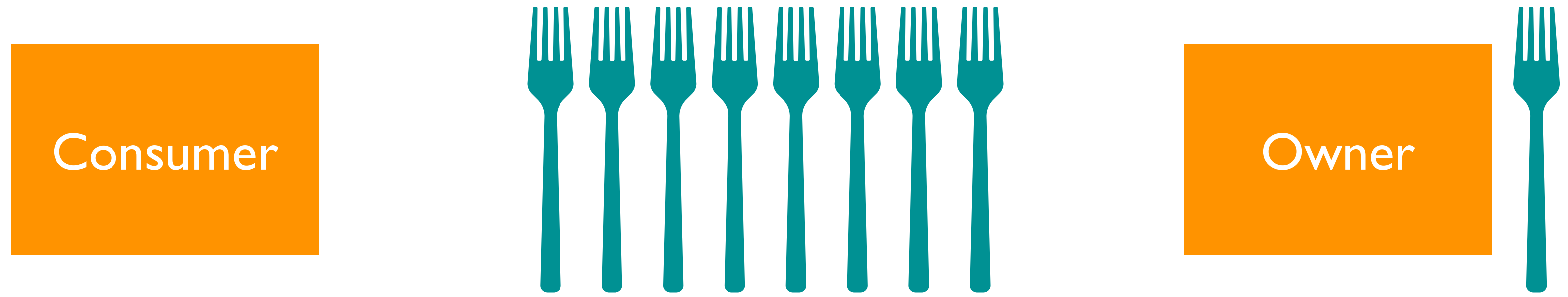
```
lock.acquire();
If (no Milk) {
  Buy milk;
}
lock.acquire();
```

```
lock.acquire();
If (no Milk) {
  Buy milk;
}
lock.acquire();
```

Drink milk and be strong without buying too much milk!

Do locks solve all problems?

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

```
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 == 1)
    {
      wakeup(consumer);
    }
  }
}
```

Are we done? Is this correct?

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

```
Consumers() {
  while(true)
  {
    if(forkCount == 0)
    {
```

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

```
      sleep();
    }
```

Time

Example 5: The producer consumer problem with locks

- Suppose we implement producer and consumer this way

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

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

Too many CPU cycles wasted by the while loop!!!

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**:
    - 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**:
    - 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
  - All problems solvable with locks can be solved with a binary semaphore