Object-oriented programming and data-structures

CS/ENGRD 2110
SUMMER 2018

Lecture 16: Concurrency
http://courses.cs.cornell.edu/cs2110/2018su
The CPU is the part of the computer that executes instructions.

**Java**: `x = x + 2;`

Suppose variable `x` is at Memory location 800, Instructions at 10

**Machine language**:
10: `load register 1, 800`
11: `Add register 1, 2`
12: `Store register 1, 800`

Basic uniprocessor-CPU computer. Black lines indicate data flow, red lines indicate control flow.
Clock rate

- Clock rate “frequency at which CPU is running”
  Higher the clock rate, the faster instructions are executed.

- First CPUs: 5-10 Hz (cycles per second)
- Today MacBook Pro 3.5 GHz

- Your OS can control the clock rate, slow it down when idle, speed up when more work to do
Why multicore?

- Moore’s Law: Computer speeds and memory densities nearly double each year
But a fast computer runs hot

- Power dissipation rises as square of the clock rate

- Chips were heading toward melting down!

- Put more CPUs on a chip: with four CPUs on one chip, even if we run each at half speed we can perform more overall computations!
Today: Not one CPU but many

Processing Unit is called a core.

- Modern computers have “multiple cores” (processing units)
  - Instead of a single CPU (central processing unit) on the chip 5-10 common. Intel has prototypes with 80!

- We often run many programs at the same time

- Even with a single core (processing unit), your program may have more than one thing “to do” at a time
  - Argues for having a way to do many things at once
Running processes on my laptop

>100 processes are competing for time. Here’s some of them:
Programs can have several “threads of execution”

We often run many programs at the same time
And each program may have several “threads of execution”

Process graphics  Process user inputs
Distributed Systems

A **distributed system** is one in which the failure of a computer you didn't even know existed can render your own computer.

Modern systems like Google, Facebook run applications that are distributed across **thousands of machines** in **large datacenters**.

My own personal record is 342 machines.
Abstract View

Program

Task 1

Task 2

Task 3

Machine 1

Machine 2

Machine 3
Abstract View

Program

Task 1
Task 2
Task 3

Core 1
Core 2

Machine 1
Machine 2
Machine 3
Abstract View

Program

Task 1

Task 2

Task 3

Core 1

T1

T1

T1

T1

Core 2

T1

T1

T2

T1

Task 1

Task 2

Task 3

Machine 1

Machine 2

Machine 3
Concurrency

- Concurrency refers to a single program in which several processes, called threads, are running simultaneously.

- Special problems arise.

- They see the same data and hence can interfere with each other, e.g. one process modifies a complex structure like a heap while another is trying to read it.
  - In CS2110, we’ll look at:
    - Race Conditions
    - Deadlocks

- We’ll refer to any sequential execution chunk as a task.
A **thread** is an object that “independently computes”

- Needs to be created, like any object
- Then “started” --causes some method to be called. It runs side by side with other threads in the same program; they see the same global data

- The actual executions could occur on different CPU cores, but don’t have to
- We can also simulate threads by *multiplexing* a smaller number of cores over a larger number of threads
Java Class Thread

- Threads are instances of class Thread
  - Can create many, but they do consume space & time

- The Java Virtual Machine creates the thread that executes your main method.

- Threads have a priority
  - Higher priority threads are executed preferentially
  - By default, newly created threads have initial priority equal to the thread that created it (but priority can be changed)
Java Class Thread

- Threads are objects in Java, just like everything else

- There’s two ways to create a thread:
  - By extending the class `Thread`
  - By implementing the interface `Runnable`

- Which one do you think is better?
Creating a new Thread (Method 1)

class MaxThread extends Thread {
    private int[] array;

    MaxThread(int[] array) {
        this.array = array;
    }

    @Override
    public void run() {
        // Computes max of array
        ...
    }
}

MaxThread p = new MaxThread(array);
p.start();

overrides Thread.run()
Do this and Java invokes run() in new thread

Call run() directly?
no new thread is used: Calling thread will run it
Creating a new Thread (Method 2)

```java
class MaxRun implements Runnable {
    private int[] array;

    MaxRun(int[] array) {
        this.array = array;
    }

    public void run() {
        //compute max of an array
        ...
    }
}

MaxRun p = new MaxRun(array);
new Thread(p).start();
```
Threads can pause

- When active, a thread is “runnable”. It may not actually be “running”. For that, a CPU must schedule it. Higher priority threads could run first.

- A thread can pause
  - Call Thread.sleep(k) to sleep for k milliseconds
    - Suspends the execution of a thread
  - Doing I/O (e.g. read file, wait for mouse input, open file) can cause thread to pause
  - Java has a form of locks associated with objects. When threads lock an object, one succeeds at a time.

- A thread can offer another thread the CPU
  - Call yield()
Thread States
How do I wait for threads to finish?

- Calling `join()` on a thread will cause another thread to wait until the first thread is finished.
- Can be used to determine when the output of a computation is ready!
- For instance, let’s modify our run method to store the final max value in to an additional result array that is shared across all threads.
- Want to know when it’s safe to check the result!
How do I wait for threads to finish?

- Calling `join()` on a thread will cause another thread to wait until the first thread is finished.
- Can be used to determine when the output of a computation is ready!
  - For instance, let’s modify our run method to store the final max value in to an additional result array that is shared across all threads.

```
MaxRun p1 = new MaxRun(array1, result, 0);
MaxRun p2 = new MaxRun(array2, result, 1);
MaxRun p3 = new MaxRun(array3, result, 2);
MaxRun p4 = new MaxRun(array4, result, 3);
Thread t1 = new Thread(p1).start();
Thread t2 = new Thread(p2).start();
Thread t3 = new Thread(p3).start();
Thread t4 = new Thread(p4).start();
t1.join();
t2.join();
t3.join();
t4.join();
System.out.println("Result " + result[0] + " " + result[1] + " ");
```
How do I wait for threads to finish?

- Calling `join()` on a thread will cause another thread to wait until the first thread is finished.

- Can be used to determine when the output of a computation is ready!

- For instance, let’s modify our run method to store the final max value in to an additional result array that is **shared** across all threads.

- Want to know when it’s safe to check the result!

```java
MaxRun p1= new MaxRun(array1, result, 0);
MaxRun p2= new MaxRun(array2, result, 1);
MaxRun p3= new MaxRun(array3, result, 2);
MaxRun p4= new MaxRun(array4, result, 3);
Thread t1 = new Thread(p1).start();
Thread t2 = new Thread(p2).start();
Thread t3 = new Thread(p3).start();
Thread t4 = new Thread(p4).start();
t1.join();
t2.join();
t3.join();
t4.join();
System.out.println("Result " + result[0] + " " + result[1] + " " ...);
```
Memory Consistency Errors

- Threads often operate on **shared data**

- If not careful, however, concurrent access to shared data can break the correctness of the program

- Race conditions arise both
  - at the memory level
    - memory consistency errors
  - At the program level
    - Invariants can be violated due to concurrent updates
What if threads share data?

- Threads often operate on shared data

- If not careful, however, concurrent access to shared data can **break** the correctness of the program

- Race conditions arise both
  - at the memory level
    - memory consistency errors
  - At the program level
    - Invariants can be violated due to concurrent updates

- Code is said to be **thread-safe** if it remains correct when accessed concurrently
Race conditions

- A **race condition** arises if two or more processes access the same variables or objects concurrently and at least one does updates.

- If the updates are not **atomic**, the end state can be inconsistent.
  - An operation is atomic if it happens “all at once” without being interrupted by other events.
A race condition arises if two or more processes access the same variables or objects concurrently and at least one does updates.

If the updates are not atomic, the end state can be inconsistent. An operation is atomic if it happens “all at once” without being interrupted by other events.

Very few operations in modern systems are atomic!
Race conditions

- Suppose x is initially 5

  Thread t1

  ```java
  x = x + 1
  System.out.println(x);
  ```

  Thread t2

  ```java
  x = x + 1
  System.out.println(x);
  ```

- What do you think will be the end value?
Race conditions

Suppose x is initially 5

Thread t1
- LOAD x
- ADD 1
- STORE x

Thread t2
- ...
- LOAD x
- ADD 1
- STORE x

... after finishing, x = 6! We “lost” an update
Race conditions

- Suppose \( x \) is initially 5

**Thread t1**
- LOAD \( x \)
- ADD 1
- STORE \( x \)

**Thread t2**
- ... 
- LOAD \( x \)
- ADD 1
- STORE \( x \)

- Machine level implementation of increment is not atomic!
Race conditions

Suppose $x$ is initially 5

Thread t1
- LOAD $x$
- ADD 1
- STORE $x$

Thread t2
- ...
- LOAD $x$
- ADD 1
- STORE $x$

Second store happens after first store -> we lost an update!
What if we want to insert a new element to a linked list?
What if we want to insert a new element to a linked list?

```java
void add(V v) { // to tail
    Node newNode = new Node(v);
    if (tail!=null) {
        tail.next = newNode;
        newNode.prev = tail;
        tail = newNode;
    } else {
        head = new Node(v);
        tail = head;
    }
}
```

```java
V poll() { // from head
    V v = null;
    if (head!=null) {
        if (head == tail) { // list is one el
            tail = tail.prev;
        }
        elif (head.next == tail) { list is two el
            tail.prev = null
        }
        v = head.value;
        head = head.next;
        if (head!=null) {
            head.prev = null;
        }
    }
    return v;
}
```
What if we want to insert a new element to a linked list?

```java
void add(V v) { // to tail
    Node newNode = new Node(v);
    If (tail!=null) {
        tail.next = newNode;
        newNode.prev = tail;
        tail = newNode;
    } else {
        head = new Node(v);
        tail = head;
    }
}
```

```java
V poll() { // from head
    V v = null;
    If (head!=null) {
        if (head == tail) { // list is one el
            tail = tail.prev;
        } else if (head.next == tail) { // list is two el
            tail.prev = null
        }
        v = head.value;
        head = head.next;
        if (head!=null) {
            head.prev = null;
        }
    }
    return v;
}
```

What could go wrong?
Race conditions

- Race conditions are bad news
  - Race conditions can cause many kinds of bugs, not just the example we see here!
  - Common cause for “blue screens”: null pointer exceptions, damaged data structures
  - Concurrency makes proving programs correct much harder!
Yes, this kid really just deleted $300 MILLION by messing around with Ethereum’s smart contracts.
Race conditions
Synchronization

- To prevent race conditions, one often requires a process to “acquire” resources before accessing them, and only one process can “acquire” a given resource at a time.

- This process is called **synchronization**

- Different languages provide more/less native support for synchronisation. Java provides
  - Synchronized primitive
  - Locks
  - Semaphores (don’t look at this here)
Synchronized Keyword

- Synchronized keyword in Java acquires exclusive ownership of a given resource.

- Exists in two contexts:
  - Synchronized methods
  - Synchronized blocks

- Every object in Java has an **intrinsic lock** (or **monitor lock**).
Synchronized Methods

- To make a method synchronized, simply add the synchronized keyword to its declaration

```java
synchronized void add(V v) { // to tail
    Node newNode = new Node(v);
    if (tail!=null) {
        tail.next = newNode;
        newNode.prev = tail;
        tail = newNode;
    } else {
        head = new Node(v);
        tail = head;
    }
}
```
Synchronized Methods

- To make a method synchronized, simply add the synchronized keyword to its declaration.

- A synchronized method acquires exclusive ownership of the current instance of the object (monitor lock).
  - Ownership lasts from the beginning of the method until the end.

- No two synchronized methods on the same object can execute concurrently.

```java
synchronized void add(V v) { // to tail
    Node newNode = new Node(v);
    if (tail!=null) {
        tail.next = newNode;
        newNode.prev = tail;
        tail = newNode;
    } else {
        head = new Node(v);
        tail = head;
    }
}
```
Synchronized Methods

- Synchronized methods are great when modify only a single object

- And when are ok with locking the entire object during execution
  - Ex: currently locking the entire linked list, even if looking at different nodes

```java
synchronized void add(V v) { // to tail
    Node newNode = new Node(v);
    if (tail != null) {
        tail.next = newNode;
        newNode.prev = tail;
        tail = newNode;
    } else {
        head = new Node(v);
        tail = head;
    }
}
```
Synchronized Blocks

- Unlike synchronized methods, **synchronized blocks** must specify the **object** that they wish to lock.

- As in synchronized methods,
  - Might have to wait if other thread has acquired object.
  - While this thread is executing the synchronized block, the object is **locked**. No other thread can obtain the lock.

```java
void add(V v) {
  Node newNode = new Node(v);
  synchronized (this) {
    if (tail != null) {
      tail.next = newNode;
      newNode.prev = tail;
      tail = newNode;
    } else {
      head = new Node(v);
      tail = head;
    }
  }
}
```
Revisiting the DLL

- What if we added a third method: \textit{traverse}, that prints out all the nodes of the DLL
- Can we achieve better performance using synchronized blocks?
Revisiting the DLL

- What if we added a third method: **traverse**, that prints out all the nodes of the DLL
  - Can we achieve better performance using synchronized blocks?

- What if we locked individual nodes in the DLL instead of locking the DLL itself?
  - How should we lock those?
Revisiting the DLL (head/tail not null)

```java
void add(V v) { // to tail
    Node newNode = new Node(v);
    synchronized(head) {
        synchronized(tail) {
            if (tail != null) {
                tail.next = newNode;
                newNode.prev = tail;
                tail = newNode;
            } else {
                head = tail;
                tail = new Node(v);
            }
        }
    }
}

V poll() { // from head
    synchronized (head) {
        synchronized (tail) {
            V v = null;
            if (head != null) {
                if (head == tail) { // list is one el
                    tail = tail.prev;
                } else if (head.next == tail) { // list is two el
                    tail.prev = null
                }
                v = head.value;
                head = head.next;
                if (head != null) {
                    head.prev = null;
                }
            }
        }
    }
    return v;
}
```
What happens if null?

- Null objects will throw a null pointer exceptions when calling synchronized
- But if we don’t call synchronize, two threads could try to set head to non-null concurrently!
- What can we do?!
Null objects will throw a null pointer exceptions when calling synchronized.

But if we don’t call synchronize, two threads could try to set head to non-null concurrently!

What can we do?!
- One option: make add/poll acquire a lock on the linked list to check whether head/tail is null
- If not null, acquire lock on object, then release lock on linked list.
What happens if null?

- Null objects will throw a null pointer exceptions when calling synchronized

- But if we don’t call synchronize, two threads could try to set head to non-null concurrently!

- What can we do?!
  - One option: make add/poll acquire a lock on the linked list to check whether head/tail is null
  - If not null, acquire lock on object, then release lock on linked list.
  - **BUT synchronized blocks** can only be nested
What happens if null?

- Null objects will throw a null pointer exceptions when calling synchronized

- But if we don’t call synchronize, two threads could try to set head to non-null concurrently!

- What can we do?!  
  - One option: make add/poll acquire a lock on the linked list to check whether head/tail is null
  - If not null, acquire lock on object, then release lock on linked list.
  - Better option, use Java **Locks**, that can never be null
Lock objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time.

Benefits:
- decide when to acquire/release lock
- Can “give up” on trying to acquire lock

Lots of different types of lock in
java.util.concurrent.locks

Read up!
With great power ...

- ... Comes great responsibility
- Current locking, as we’ve seen, is hard!
  - Hard to understand what/when we should lock
- Locking in the wrong order can lead to deadlocks
  - What if we synchronize/lock first on head then on tail in one method, but on tail then on head in another method?
To prevent race conditions, one often requires a process to “acquire” resources before accessing them, and only one process can “acquire” a given resource at a time.

But if processes have to acquire two or more resources at the same time in order to do their work, deadlock can occur. This is the subject of the next slides.
Dining philosopher problem

Five philosophers sitting at a table.

Each repeatedly does this:
1. think
2. eat.

Need TWO forks to eat spaghetti!
Dining philosopher problem

Five philosophers sitting at a table.

Each repeatedly does this:
1. think
2. Eat.

Only brought 5 forks!

Need TWO forks to eat spaghetti!
Dining philosopher problem

Five philosophers sitting at a table.

To eat, they first pick up the **left fork**, then the **right fork**, then **eat**, then put the **left fork** down, then put the **right fork** down.

Need TWO forks to eat spaghetti!
Dining philosopher problem

Five philosophers sitting at a table.

At one point they all pick up their left fork!

Need TWO forks to eat spaghetti!
Dining philosopher problem

Five philosophers sitting at a table.

At one point they all pick up their *left fork*!

*We have a deadlock!*

Need TWO forks to eat spaghetti!
Dining philosopher problem

Simple solution to deadlock:
Number the forks. Pick up smaller one first
1. think
2. eat (2 forks)
eat is then:
pick up smaller fork
pick up bigger fork
pick up food, eat
put down bigger fork
put down smaller fork
Correct Locking is Hard!

- Locking objects in different orders in different functions will cause deadlock!
- Exceptions that occur in the middle of the program may cause locks to not be released
- Insufficient locking may lead to race conditions!
Good practices

- Prefer the use of `Lock` locks over synchronized.
- When in doubt, use a lock for the whole method!
  - Only optimise when you need to
  - Correct code is always faster than incorrect code :-)
- Always acquire locks at the beginning of the method unless a good reason not to
- Always try to release locks in a finally clause
Thread Coordination

- Threads often have to **coordinate** their actions.
  - Example 1: Thread 1 (thread that monitors user input) must **notify** thread 2 (the GUI thread) that there are new characters to draw. Thread 2 is **waiting** for Thread 1’s notification.

- Example 2: producer/consumer pattern

![Diagram of producer/consumer pattern]

- Blocking Queue
  - Producer-Thread: Put element in the queue and wait till space is available if queue is full.
  - Consumer-Thread: Retrieve element from the queue and wait till element is available if queue is empty.
Option 1: Busy waiting

while (dll.isEmpty()) {
    // Do nothing
}
// If exited the loop, means
// element was in thread
V v = dll.poll();

Loop until condition is satisfied. Only then do you exit the loop.
Option 1: Busy waiting

```java
while (dll.isEmpty()) {
    // Do nothing
}
// If exited the loop, means
// element was in thread
V v = dll.poll();
```

Loop until condition is satisfied. Only then do you exit the loop.

**Inefficient.** Not necessary to constantly re-check condition. Keeping thread busy for no reason.
Option 2: Guarded Blocks on Objects

```java
synchronized(this) {
    while (dll.isEmpty()) {
        // Do nothing
        this.wait();
    }
    V v = dll.poll();
}
```

*Suspend the current thread* until condition is satisfied by calling *Object.wait()*. The invocation of *wait* does not return until another thread has issued a *notification* that some special event may have occurred — though not necessarily the event this thread is waiting for:
Option 2: Guarded Blocks on Objects

```java
synchronized(dll) {
    while (dll.isEmpty()) {
        // Do nothing
        dll.wait();
    }
    V v = dll.poll();
}
```

Calling `wait()` blocks the current thread. Thread releases the `monitor` lock of the `dll` object. It will re-acquire it when receiving the notification.
Option 2: Guarded Blocks on Objects

```java
synchronized(dll) {
    while (dll.isEmpty()) {
        // Do nothing
        dll.wait();
    }
    V v = dll.poll();
}
```

Calling `wait()` blocks/suspends the current thread. Thread releases the monitor lock of the `dll` object. It will re-acquire it when receiving the notification.
Option 2: Guarded Blocks on Objects

synchronized(dll) {
    while (dll.isEmpty()) {
        // Do nothing
        dll.wait();
    }
    V v = dll.poll();
}

One should always invoke `wait` inside a loop that tests for the condition being waited for. Threads can be woken up for a number of reasons. Notification may not be for the particular condition that current thread was waiting for.
Option 2: Guarded Blocks on Objects

```java
synchronized(dll) {
    while (dll.isEmpty()) {
        // Do nothing
        dll.wait();
    }
    V v = dll.poll();
}
```

Notifications are sent using the `notify` or `notifyAll` keywords

- `notifyAll` wakes up all threads waiting on that lock that something important happened.
- `notify()` wakes up a single thread.
Option 3: Guarded Blocks with Locks

Locks are associated with Conditions that support `await` and `notify/notifyAll` methods.

Can associate as many conditions per locks as desired

**Main benefit**: more flexibility. Make it explicit what condition you are awaiting on
Java Concurrent Collections

- **BlockingDeque<E>**
  - A Deque that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.

- **BlockingQueue<E>**
  - A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

- **ConcurrentMap<K,V>**
  - A Map providing thread safety and atomicity guarantees.
Concurrency

- Brief overview of concurrency in Java!
- More formal treatment in higher level courses.
- Remember: **monitor locks, synchronized, wait, notify/All, conditions, race conditions, deadlocks**