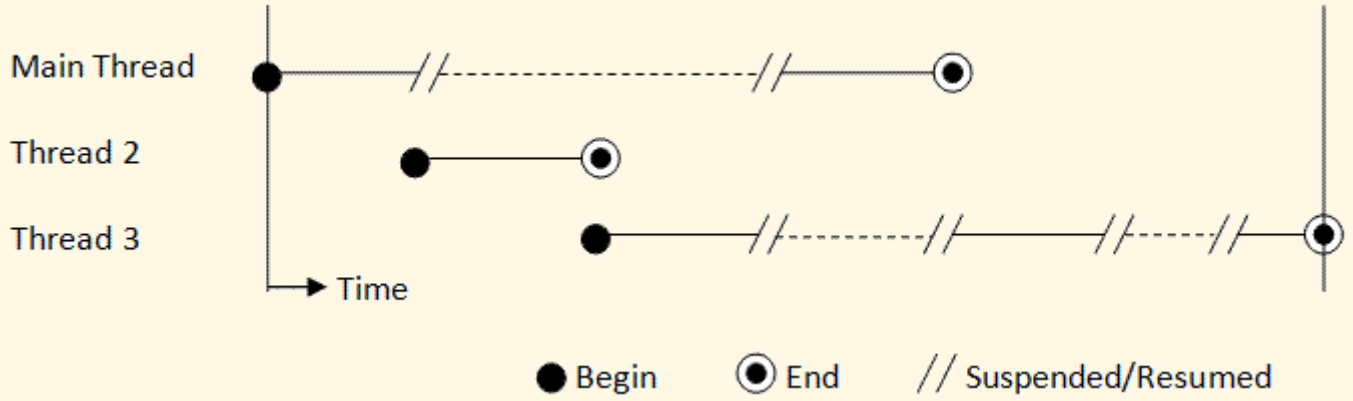
Concurrency

- Threads
- Synchronization
- Memory Consistency
- High-Level Concurrency

Threads

Introduction

- Even a single application is often expected to do more than one thing at a time → *concurrency programming*
  - A streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display
  - A word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display



Processes vs Threads

- A *process* or program has its own address space and control blocks
  - It is called heavyweight because it consumes a lot of system resources
  - Within a process or program, we can run multiple threads concurrently to improve the performance
- *Threads* are lightweight and run inside a single process
  - They share the same address space, the resources allocated and the environment of that process
  - It is lightweight because it runs within the context of a heavyweight process and takes advantage of the resources allocated for that program and the program's environment
  - A thread must carve out its own resources within the running process: its own stack, registers and program counter
- A program when started always has a default (or main) process and thread

Creating a Thread

- By implementing `Runnable` interface
  - ```
    Thread newThread = new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println("New thread...");
        }
    });
    newThread.start();
    ```
  - ```
    new Thread(() -> {
        System.out.println("New thread...");
    }).start();
    ```
- By extending `Thread` class
  - ```
    Thread newThread = new Thread() {
        @Override
        public void run() {
            System.out.println("New thread...");
        }
    };
    newThread.start();
    ```
    ```

Passing Parameters

```
for (int i = 0; i < 10; i++) {
    new Thread(new Runnable() {
        private int ii;

        @Override
        public void run() {
            System.out.println(ii);
        }

        public Runnable pass(int ii) {
            this.ii = ii;
            return this;
        }
    }.pass(i)).start();
}
```

```
for (int i = 0; i < 10; i++) {
    final int ii = i;
    new Thread(() -> System.out.println(ii)).start();
}
```

Thread Class

- Main methods

Method	Description
`void start()`	Starts a new thread
`void interrupt()`	Sets the `interrupted` flag, and triggers an `InterruptedException` in the thread if a supporting method is under execution
`void join()`	Allows one thread to wait for the completion of another
`void wait([long millis])`	Puts the current thread into sleep until another thread wakes it up
`void notify()` `void notifyAll()`	Wakes one/all other threads that is/are waiting
`void setPriority(int priority)`	Changes the priority of thread to the given value
`static void sleep(long millis, [long nanos])`	Pauses the thread for a given period
`static boolean interrupted()`	Checks if the `interrupted` flag has been set
`static Thread currentThread()`	Gets the `Thread` object for the current thread

Interrupting a Thread

- To response to the `interrupt()` request:
  - Handle `InterruptedException`
  - Check for the `interrupted` flag
- Example:

```
public void run() {
    for (int i = 0; i < 100000; i++) {
        // do something...
        if (Thread.interrupted()) return;
    }

    try {
        Thread.sleep(3000);
    } catch(InterruptedException e) {
        return;
    }
}
```

Thread's Priority

- Thread's priority is an integer in the range 1 to 10 (larger the higher priority), but 3 constants are commonly used:
  - `MIN_PRIORITY` = 1
  - `NORM_PRIORITY` = 5 (default)
  - `MAX_PRIORITY` = 10
- Example:
  - `thread.setPriority(Thread.MAX_PRIORITY);`

Exercise

- Write a program with 2 threads:
  - One thread regularly prints the current time
  - One thread reads numbers and calculate their squares

Synchronization

Overview

- There are a lot of reasons that the execution of tasks in different threads need to be synchronized:
  - A task in one thread need the result produced by another task in another thread
    - Let's think about multiple chefs in preparing dishes for a meal
  - A shared resource cannot be used by 2 (or more) different threads at the same time
    - Let's think about books in a library
- Synchronization is about the capability to control the execution of multiple threads

`join()` Method

- If invoked:
  - The current thread goes into the wait state and remains until the thread on which the `join()` method is invoked has achieved its dead state
  - If interruption of the thread occurs, then it throws the `InterruptedException`
- Useful when a thread wants to wait until the result from another is available
- Example:

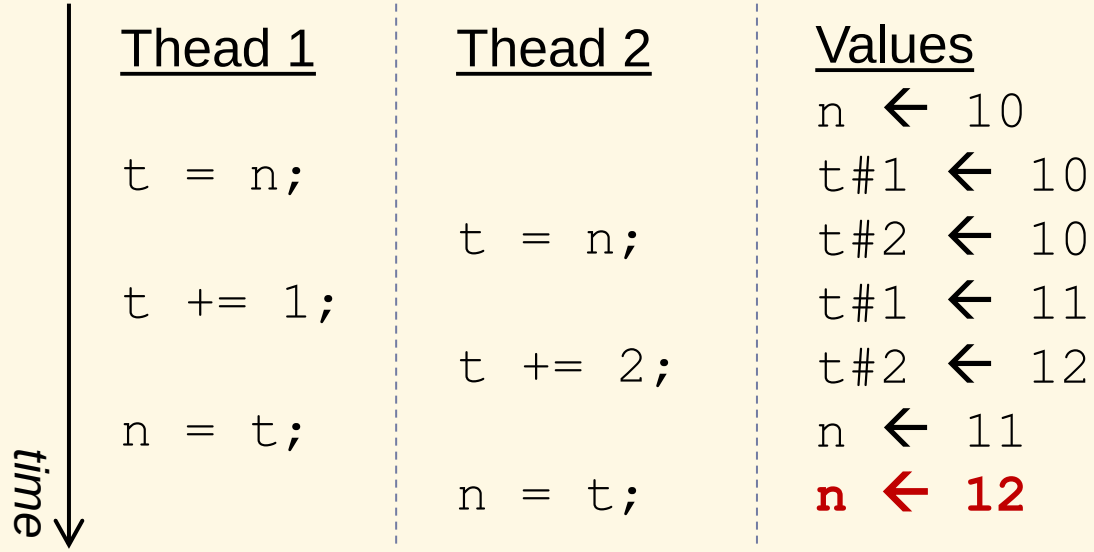
```
Thread t = new Thread(() -> {
    // do something in new thread...
    result = 1000;
});
t.start();

// do something in main thread...

t.join();
System.out.print(result);
```

Critical Sections

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to → resource access conflicts
  - Resources can be interpreted in a generalized meaning, which can be memory blocks, peripherals, files, network connections, or code segments,…
- Example:



- Access to resources need to be controlled
  ➔ *Critical sections (CS)*

Shared Memory

- Memory model:
  - Local variables, method parameters, exception handler parameters are stored in the stack memory
  - All objects and their instance fields, static fields are stored in the heap memory
- The heap memory is shared between threads, the stack is not

Synchronized Methods

- It is not possible for two invocations of synchronized methods on the same object to interleave
  - *Mutual exclusion (mutex)*: When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block until the first thread is done with the object
  - The code in a synchronized method is a critical section and needs to be protected for conflicts
  - Constructors cannot be synchronized
- Example:
  - ```
    class SyncCounter {
        private int counter = 0;

        synchronized void increase1() {
            counter += 1;
        }

        synchronized void increase2() {
            counter += 2;
        }

        synchronized int getValue() {
            return counter;
        }
    }
    ```
    ```

Synchronized Statements

- Possible to make more fine-grained synchronization for code segments
- Example:

```
class SyncCounter {
    private int counter = 0;

    void increase1() {
        synchronized(this) {
            counter += 1;
        }
        // do something else...
    }

    void increase2() {
        synchronized(this) {
            counter += 2;
        }
        // do something else...
    }

    // ...
}
```

Synchronized Statements using Multiple Locks

```
class SyncVars {
    private int var1 = 0, int var2 = 0;
    private Object lock1 = new Object(), lock2 = new Object();

    void increaseVar1() {
        synchronized(lock1) {
            var1 += 1;
        }
    }

    void increaseVar2() {
        synchronized(lock2) {
            var2 += 1;
        }
    }

    void increaseBoth() {
        synchronized(lock1) {
            var1 += 1;
        }

        // do something else...

        synchronized(lock2) {
            var2 += 1;
        }
    }
}
```

Atomic Actions

- An atomic action is one that effectively happens all at once
  - It cannot stop or be interleaved in the middle: it either happens completely, or it doesn't happen at all
  - No side effects of an atomic action are visible until the action is complete
- A synchronized block is atomic against all other blocks that are synchronized on the same object
- Attention:
  - `boolean`, `char`, `byte`, `short`, `int` assignments are atomic
  - `long` and `double` assignments are not atomic (because of word tearing)
  - Integer increment (`count++`) is not atomic
  - All complex expressions are not atomic

`wait()`, `notify()` and `notifyAll()` Methods (1)

- Calling `wait()` forces the current thread to wait until some other thread invokes `notify()` or `notifyAll()` on the same object
- Useful when a thread wants to wait until when a condition is met
- Example (see `LibraryBorrowing.java`):

```
void borrowBook(Book book) throws InterruptedException {
    synchronized(book) {
        while (book.isBorrowed()) {
            System.out.println(name + " is waiting for: " + book.getTitle());
            book.wait();
        }
        book.setReader(this);
        System.out.println(name + " borrowed: " + book.getTitle());
    }
}

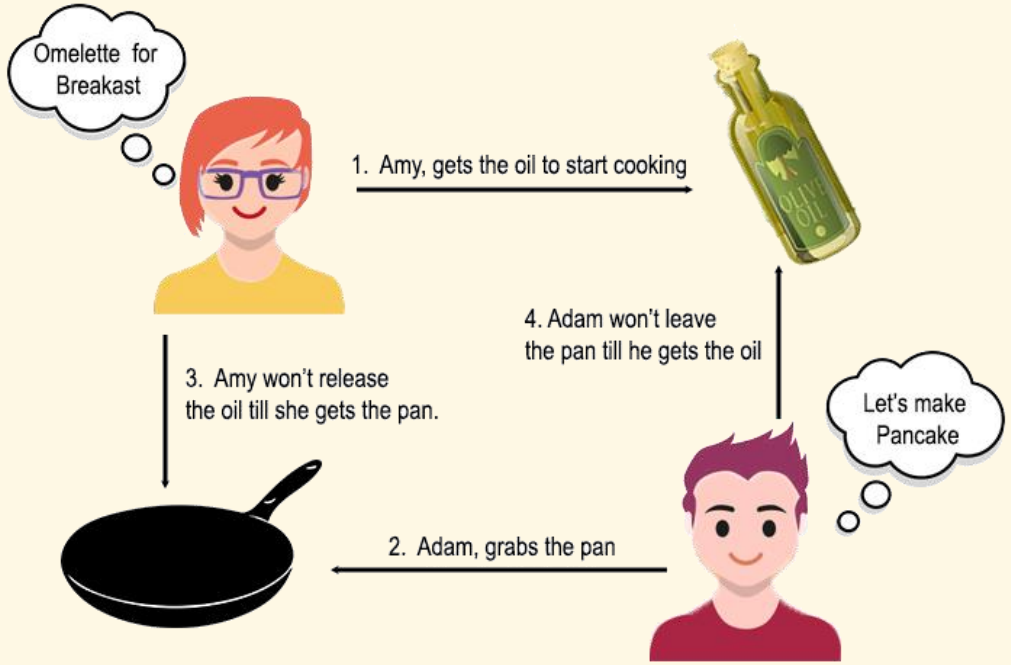
void returnBook(Book book) {
    synchronized(book) {
        book.setReader(null);
        System.out.println(name + " returned " + book.getTitle());
        book.notify();
    }
}
```

`wait()`, `notify()` and `notifyAll()` Methods (2)

- `notify()` wakes up only one of the waiting threads, while `notifyAll()` wakes up all
- These methods must always be called within a synchronized method or statement
- `wait()` must always be put in a loop which terminates only when the condition that the thread is waiting for holds

Deadlocks

- Deadlock occurs when multiple threads need the same locks but obtain them in different order
- Simplest case:
  - Thread 1 waits for Lock A while holding Lock B
  - Thread 2 waits for Lock B while holding Lock A



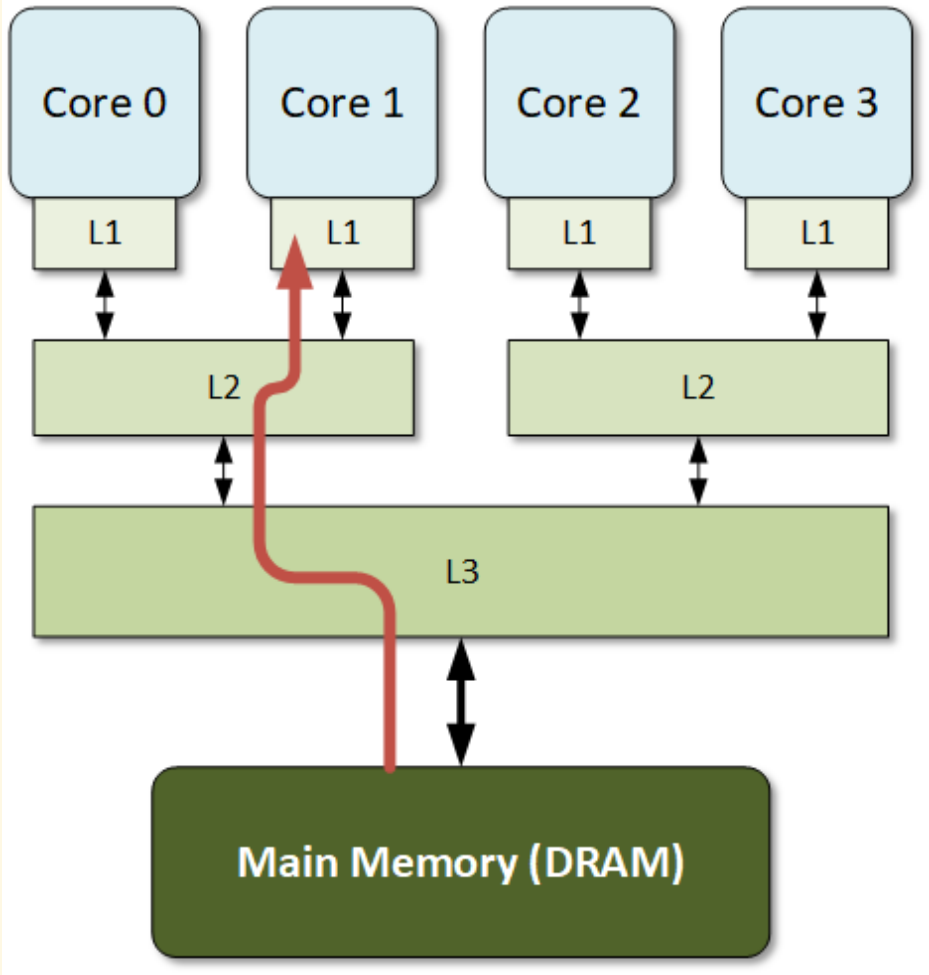
Exercises

1. Write a test program to make sure that the increment is not atomic in Java
2. From the `LibraryBorrowing.java` example, add more books and readers and see the result
3. Implement a blocking queue structure with 2 operations:
   - `put()`: adds an element to the queue (non-blocking)
   - `take()`: removes the first element from the queue, waits if none exists (blocking)
4. Implement a pool of worker threads which pick jobs from a blocking queue to process

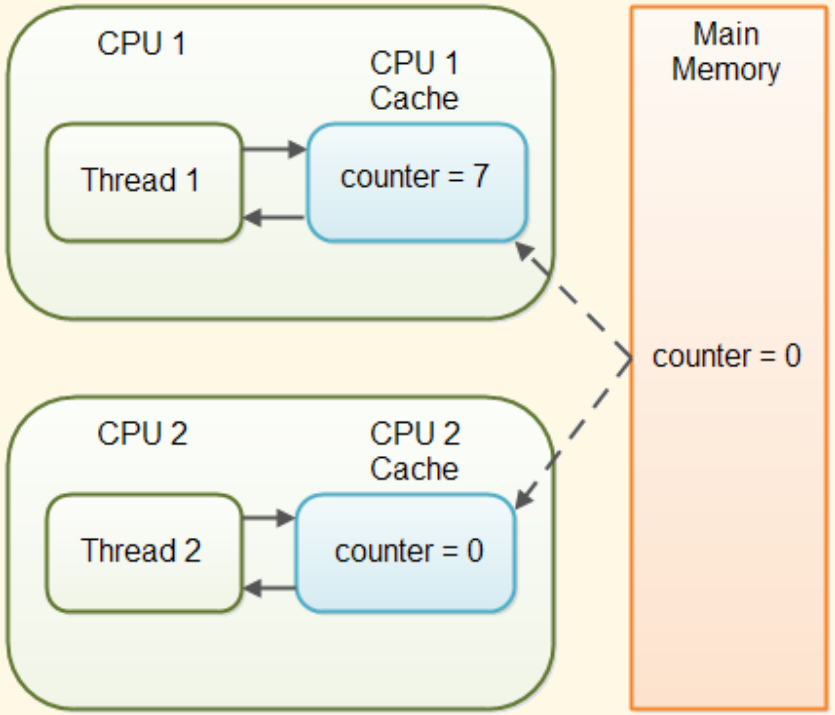
Memory Consistency

Introduction

- For performance optimization, a shared multiprocessor architecture frequently uses the following techniques:
  - Multi-level caching of variables and operations
  - Parallelization
  - Out-of-order execution
- For a single-thread code, it's not necessary to worry about the *cache coherence*
- However, in multi-threading context, the cache coherence may cause memory inconsistency when using shared variables



Variable Visibility Problem



- Each thread may copy the variables into the CPU cache of different CPUs
- Visibility problem: A thread does not see the latest value of a variable because it has not yet been written back to main memory by another thread

Visibility Guarantee

- Declaring a variable `volatile` to guarantee the visibility for other threads of writes to that variable
- `volatile` does not only affect the variable itself, but:
  - If Thread A writes to a `volatile` variable `v` and Thread B subsequently reads `v`, then all variables visible to A before writing to `v`, will also be visible to B after it has read `v`
  - If A reads `v`, then all all variables visible to A when reading `v` will also be re-read from main memory
  - ```
    class MyFraction {
        private volatile int nom;
        private int den;

        void update(int nom, int den) {
            this.den = den;
            this.nom = nom;  // den, nom are all written to the main memory
        }
        double value() {
            double f = nom;  // den, nom are all read from the main memory
            return f / den;
        }
    }
    ```
    ```

Instruction Reordering Problem and Happens-Before Guarantee

- JVM and the CPU are allowed to reorder instructions in the program for performance reasons, as long as the semantic meaning of the instructions remain the same
- But this may introduce problems when one of the variables is `volatile`
  - Example: Imagine if the order of the `update()` and `value()` methods of the `MyFraction` class were changed
- Luckily, the `volatile` keyword gives a *happens-before* guarantee:
  - The reads/writes before a write to a `volatile` variable `v` are guaranteed to "happen before" the write to `v`
    - Note that the reordering from after to before is still allowed
  - Reads from and writes to other variables after `v` cannot be reordered to occur before a read of `v`
    - The reordering from before to after is still allowed

volatile vs synchronized

- synchronized also gives visibility, so variables don't need to be volatile if they are completely guarded by synchronized methods or blocks → volatile can generally be done by synchronized
  - ```
    class MyFraction {
        private int nom, den;

        synchronized void update(int nom, int den) {
            this.den = den;
            this.nom = nom;
        }
        synchronized double value() {
            return (double)(nom) / den;
        }
    }
    ```
- But volatile is cheaper than using synchronized
- In general, use volatile to let variables be accessed by multiple threads but the atomicity is not required or has already been guaranteed (like in the MyFraction class, reads and writes are atomic)
    ```

Immutable Objects

- An immutable object is one that is no longer modified once it has been constructed
  - Since there is no write to the object, it's thread-safe (but its references are not)
  - Its class has no setters, every modification results in a new object
  - All its fields should be declared `final` and `private`
  - The class should also be declared `final` to disallow subclassing
  - Remind that `String` is immutable
- Example:

```
final class ImmutableFraction {
    private final int nom, den;

    ImmutableFraction(int nom, int den) {
        this.nom = nom;
        this.den = den;
    }

    ImmutableFraction invert() {
        return new ImmutableFraction(den, nom);
    }
}
```

High Lever Concurrency

Introduction

- Higher-level building blocks are needed for more advanced tasks
- Implemented in the `java.util.concurrent` packages

Atomic Types

- However, suspending and resuming a thread are very expensive and hit the performance
- Low-level atomic types in place of primitive types may help in many circumstances:
  - `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, `AtomicReference`
  - Not every primitive types have corresponding atomic versions
- High-level atomic types:
  - `ConcurrentHashMap` (in place of `HashMap`)
  - `ConcurrentSkipListMap` (in place of `TreeMap`)
  - `ThreadLocalRandom` for thread-safe random number generation

ReentrantLock

- `synchronized` can only be used if the lock acquire and release are close together
- For the case where these two operations are at different places, use `ReentrantLock`:
  - ```
    class SharedMemory {
        private byte[] mem = new byte[100];
        private Lock mutex = new ReentrantLock();

        byte[] open() {
            mutex.lock();
            return mem;
        }

        void close() {
            mutex.unlock();
        }
    }
    ```
    ```

Semaphore

- For a mutex, only one thread can access a CS, while `Semaphore` allows a fixed number of threads
- Example: A reading room with 20 reading spaces (see full example in `ReadingRoomExample.java`)
  - ```
    class ReadingRoom {
        final int PLACES = 20;
        private Semaphore sem = new Semaphore(PLACES);

        void welcome(Reader r) throws InterruptedException { // blocking
            sem.acquire();
            System.out.println(r.getName() + " came in");
        }

        void onLeave(Reader r) {
            sem.release();
            System.out.println(r.getName() + " left");
        }

        int getReaderCount() {
            return PLACES - sem.availablePermits();
        }

        int getQueueLength() {
            return sem.getQueueLength();
        }
    }
    ```
    ```

ExecutorService

- `ExecutorService` automatically provides a pool of threads and an API for assigning tasks to it
- Example: (see full example in `ExecutorServiceExample.java`)
  - ```
    ExecutorService exe = Executors.newFixedThreadPool(5);
    List<Callable<String>> tasks = new ArrayList<>();
    // create tasks...
    List<Future<String>> futures = exe.invokeAll(tasks);

    for (Future<String> f : futures) {
        // use results...
    }
    ```
    ```

Other Pools

- `ExecutorService exe = Executors.newSingleThreadPool()`
  - Create a pool with only a single thread
- `ExecutorService exe = Executors.newCachedThreadPool()`
  - Create a dynamic and unbounded pool that grows from zero thread
  - If a thread is idle for 1min, it may get terminated
- For more control, see `ThreadPoolExecutor` class

Exercises

1. Using `Semaphore`, extend the library example to allow each book title to have multiple instances
2. Use `ExecutorService` to implement a pool of job workers