Threads and Concurrency

CS2112 Fall 2021

What is a Thread?

A sequence of computer instructions that can perform a computational task independently and concurrently with other threads

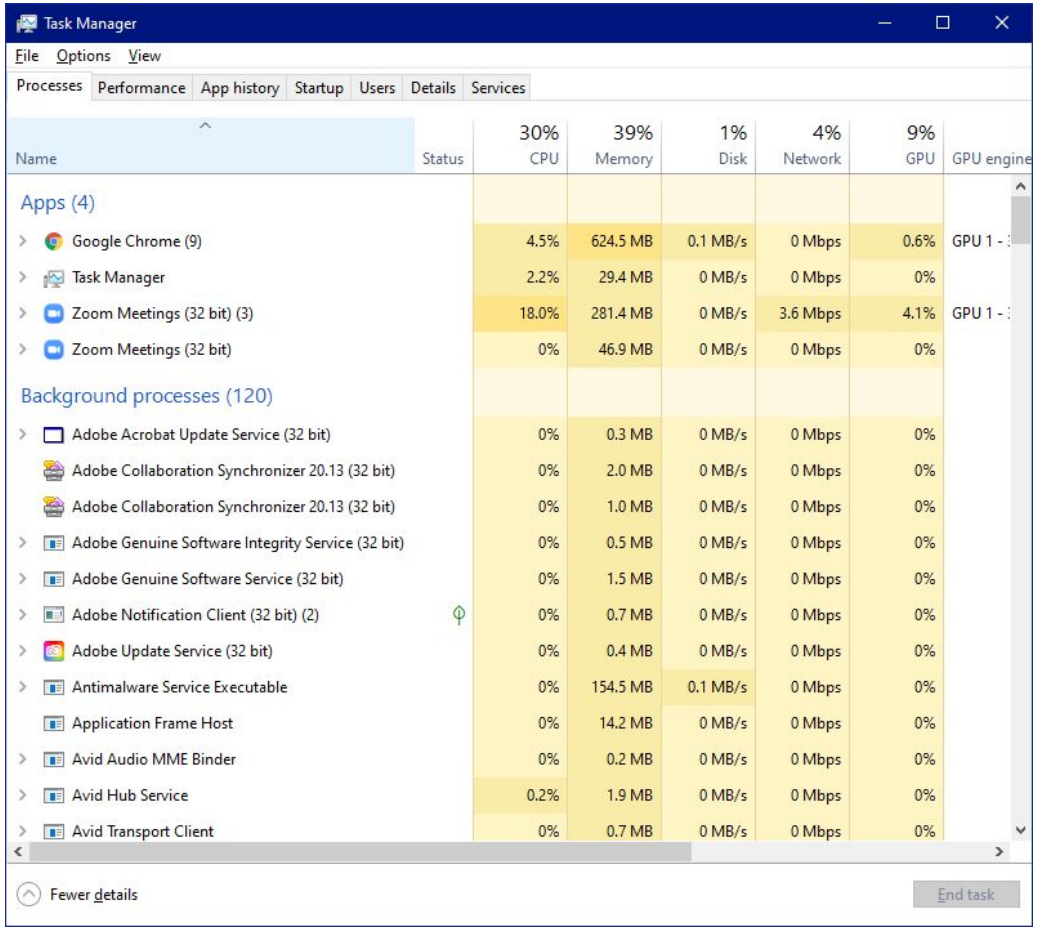
- Most programs have only one thread:
  - *main thread*
- GUIs have two other threads:
  - *application (or event dispatch) thread*
  - *rendering thread*
- A program can have many threads
- Threads share access to memory

What is a Thread?

In reality, threads are an abstraction

- Separate threads can sometimes be on the same core
  - E.g. you can have 100 threads on a 4-core CPU
- We let the operating system worry about how threads actually work

- The operating system provides support for multitasking
- In reality there is one processor doing all this
    - Any actual concurrency is across the *cores* of the processor
- But this is an abstraction too – at the hardware level, lots of multitasking
    - memory subsystem
    - video controller
    - buses
    - instruction prefetching



Concurrency (aka Multitasking)

- Refers to situations in which several threads are running simultaneously
- Special problems arise
  - race conditions
  - deadlock

Threads in Java

- Threads are instances of the class **`Thread`**
  - you can create as many as you like

- The Java Virtual Machine permits multiple concurrent threads
  - initially only one thread (executes **`main`**)

- Threads have a priority
  - higher priority threads are usually executed preferentially
  - a newly created Thread has initial priority equal to the thread that created it (but can change)

Race Conditions

- A *race condition* can arise when two or more threads try to access data simultaneously
- Thread B may try to read some data while thread A is updating it
  - Updating may not be an atomic operation
  - Thread B may sneak in at the wrong time and read the data in an inconsistent state
- Results can be unpredictable!

Example – A Lucky Scenario

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
    if (stack.isEmpty()) return;
    String s = stack.pop();
    //do something with s...
}
```

Suppose threads A and B want to call **`doSomething()`**, and there is one element on the stack

1. thread A tests **`stack.isEmpty()`** ⇒ false
2. thread A pops ⇒ stack is now empty
3. thread B tests **`stack.isEmpty()`** ⇒ true
4. thread B just returns ⇒ nothing to do

Example – An Unlucky Scenario

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
    if (stack.isEmpty()) return;
    String s = stack.pop();
    //do something with s...
}
```

Suppose threads A and B want to call **`doSomething()`**, and there is one element on the stack

1. thread A tests **`stack.isEmpty()`** ⇒ false
2. thread B tests **`stack.isEmpty()`** ⇒ false
3. thread A pops ⇒ stack is now empty
4. thread B pops ⇒ Exception!

Solution – Locking

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
     synchronized (stack) {
          if (stack.isEmpty()) return;
          String s = stack.pop();
     }
     //do something with s...
}
```

`synchronized` block

- Put critical operations in a **`synchronized`** block
- The **`Stack`** object acts as a lock
- Only one thread can own the lock at a time

Solution – Locking

- You can lock on any object, including **this**

```
public synchronized void doSomething() {
    ...
}
```

is equivalent to

```
public void doSomething() {
    synchronized (this) {
        ...
    }
}
```

File Locking

- In file systems, if two or more processes could access a file simultaneously, this could result in data corruption
- A process must *open* a file to use it – gives exclusive access until it is *closed*
- This is called *file locking* – enforced by the operating system
- Same concept as **`synchronized(obj)`** in Java

Deadlock

- The downside of locking – *deadlock*

- A *deadlock* occurs when two or more competing threads are waiting for the other to relinquish a lock, so neither ever does

- Example:
  - thread A tries to open file X, then file Y
  - thread B tries to open file Y, then file X
  - A gets X, B gets Y
  - Each is waiting for the other forever

Necessary Conditions for Deadlock

- Bounded resources
- No preemptions
- Hold & Wait
- Circular waiting

```
T1:
synchronized (lock1) {
  synchronized (lock2) {
    // ...
  }
}

T2:
synchronized (lock2) {
  synchronized (lock1) {
    // ...
  }
}
```

Necessary Conditions for Deadlock

- Bounded resources
  - Limited number of threads can use a resource
  - In this example, there are only two threads

```
T1:
synchronized (lock1) {
  synchronized (lock2) {
    // ...
  }
}

T2:
synchronized (lock2) {
  synchronized (lock1) {
    // ...
  }
}
```

Necessary Conditions for Deadlock

- No preemptions
  - No other thread can take the resource until the other thread is done

```
T1:
synchronized (lock1) {
  synchronized (lock2) {
    // ...
  }
}

T2:
synchronized (lock2) {
  synchronized (lock1) {
    // ...
  }
}
```

Necessary Conditions for Deadlock

- Hold & Wait
  - A thread holds one resource while it is waiting for a second one
  - T1 holds lock1 while waiting for lock2

```
T1:
synchronized (lock1) {
  synchronized (lock2) {
    // ...
  }
}

T2:
synchronized (lock2) {
  synchronized (lock1) {
    // ...
  }
}
```

Necessary Conditions for Deadlock

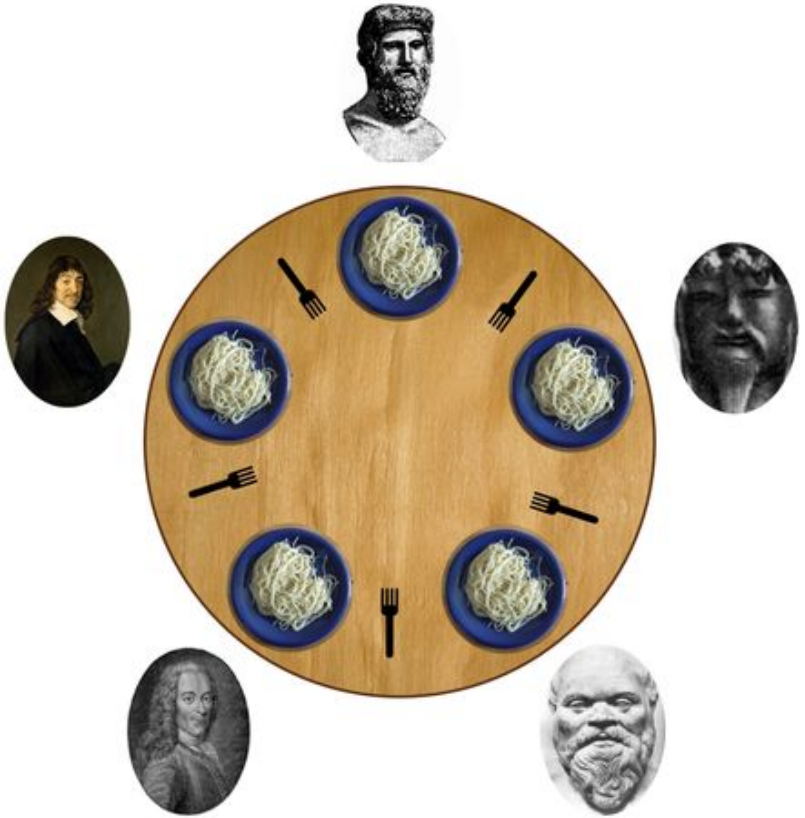
- Circular waiting
  - Threads wait for each other in a circular way
  - T1 waits for T2 who waits for T1

Why is this condition not sufficient on its own?

```
T1:
synchronized (lock1) {
  synchronized (lock2) {
    // ...
  }
}

T2:
synchronized (lock2) {
  synchronized (lock1) {
    // ...
  }
}
```

Dining Philosophers Problem:



Source: Wikipedia

Problem Description

"Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher at a time and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can only take the fork on their right or the one on their left as they become available and they cannot start eating before getting both forks."

Source: Wikipedia

Can you design a procedure to avoid deadlock, and ensure that each philosopher gets to eat?

Approach 1: Resource Hierarchy

- Assign the forks an order (1, 2, … 5)
- Resources must be requested in order
- Ensures that one philosopher will be able to pick up both forks and eat
- Avoids deadlock

Approach 2: Arbitrator

- Have an arbitrator who decides which philosophers get to pick up their forks
- Arbitrator only gives forks to one philosopher at a time
- Can be implemented as a mutex
- Can decrease parallelism

`wait/notify`

- A mechanism for event-driven activation of threads

- Animation threads and the GUI event-dispatching thread in can interact via **`wait/notify`**

wait/notify

animator:

```
boolean isRunning = true;

public synchronized void run() {  while (true) {
          while (isRunning) {
               //do one step of simulation


          }
          try {
               wait();
          } catch (InterruptedException ie) {}  isRunning = true;

     }
}
```

relinquishes lock on **animator** - awaits notification

```
public void stopAnimation() {
     animator.isRunning = false;
}

public void restartAnimation() {
     synchronized(animator) {
          animator.notify();
     }
}
```

notifies processes waiting for **animator** lock

When to use parallel programming

For long-running computations, single-threaded programming means wasting 3/4ths or 7/8ths of your compute resources

Multi-threading has high startup costs (constructing threads, synchronization, etc.) -> not worth it for short computation

Ideally have embarrassingly parallel applications: little or no dependency between parallel workers -> Communication between parallel workers is slow

Finding Concurrency Bugs

- Extremely difficult
  - Sometimes the addition/removal of a print statement can decide whether the program functions
  - Could only happen on certain machines
  - All up to timing!
- The only real way to find a concurrency issue is through reasoning about your program

Java-style multi-threading not only form of parallel programming available

Concurrency to hide I/O lag (storage is much slower than CPU; can perform 10,000s of math operations while waiting for data) -> Python, JavaScript

Programming with multiple processes, each with own memory (no race conditions, but higher communication costs)

Functional Programming using Promises and Callbacks (CS 3110)

SIMD Vector Parallelism (special instructions that operate on whole vectors) -> numpy

GPU Parallelism (CUDA) -> heavily used in Machine Learning

Tips for A6

- *Platform.runLater()* can be used to run drawing code on the JavaFX thread
- Show your GUI design to other people -> watch how they use it. Do they find the UX intuitive?
- Let your friend / parent play with your GUI for testing. We tend to avoid unfinished sections and don’t test them that much. They won’t.