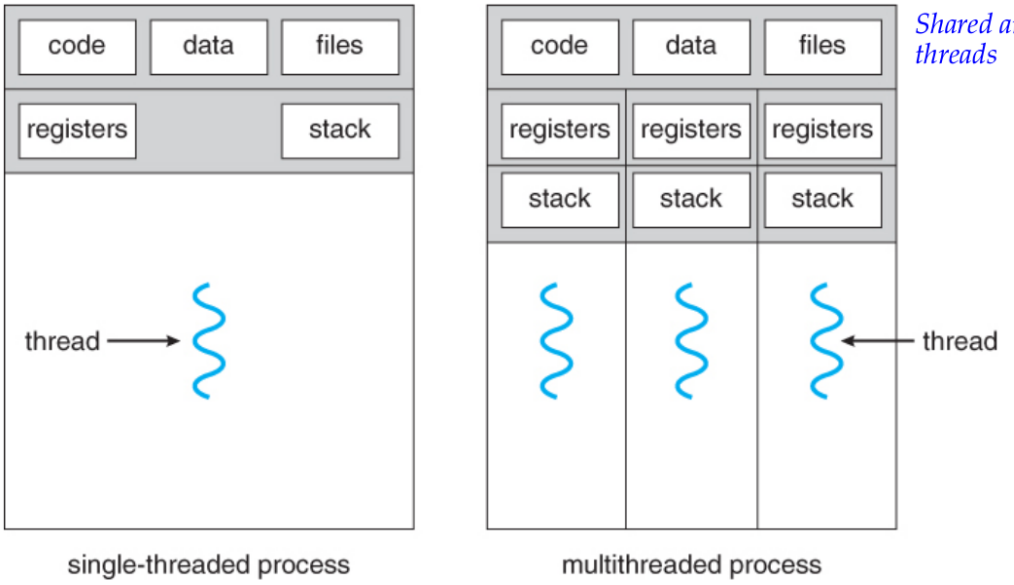
CSC 2620 Object Oriented Design — Spring 2024
Unit 7 – Threads and Syncronization

7.1 Multithreading

- **Thread:** Separate streams of execution of a process that share the same address space.
  - Allows one process to have multiple point of executions.
  - Scheduled (context-switched) by the OS and run as independent entities.
- A thread of execution is the smallest memory unit of processing that can be scheduled.
  - It's a light-weight process that allows a single process to share its computing time among several tasks it owns.
- A process can have multiple threads of execution.
  - If there are multiple threads, the PCB includes thread information, called thread control block (TCB).
- Multi-threaded applications have multiple threads within a single process.



- In theory there are two different ways to parallelize the workload
  - Data parallelism divides the data up amongst multiple cores (threads), and performs the same task on each subset of the data, e.g., histograms and edge detection, on each piece on different cores.
  - Task parallelism divides the different tasks to be performed among the different cores and performs them simultaneously, e.g., min and max.
  - In practice, some sort of hybrid combination is more common. We'll save the more complicated parallelization algorithms for future classes.
- In Java, each thread has a priority. Priorities are represented by number from 1 to 10. The default priority of `NORM_PRIORITY` is 5, `MIN_PRIORITY` is 1 and `MAX_PRIORITY` is 10.

```
    public static void main(String[] args) throws
  InterruptedException
    {
          ThreadPriority t1 = new ThreadPriority();
          ThreadPriority t2 = new ThreadPriority();
          ThreadPriority t3 = new ThreadPriority();

          t1.setPriority(Thread.MAX_PRIORITY);
          t2.setPriority(Thread.MIN_PRIORITY);
          t3.setPriority(Thread.NORM_PRIORITY);

          t1.start();
          t2.start();
          t3.start();
    }
```

7.1.1 Benefits of Multithreading

- Allows for **concurrency**
  - Better responsiveness overall
  - Necessary for applications such as web servers (handle multiple requests), browsers (to run GUI and network code concurrently), etc.
- Provides for easier overhead
  - There's no need to copy address space when creating a thread
  - There's no need to switch process address space
- It provides for scalability
  - Increase of resources will guarantee better performance of a threaded application
- The primary benefit we'll be focusing on in this class is concurrency, allowing our GUIs, sockets, and back-end code can run simultaneously

7.1.2 Threads in Java

- To create threads in Java, you need a thread object, or an object that will operate in a different thread than `Main`
- A thread class `implements` the interface `Runnable` which requires a definition for the `Run` method

```
public class Printer implements Runnable {
  public void Run(){
    System.out.println(``Printing stuff.'');
  }
}

public class Main {
  public static void main(String[] args)
  {
    Printer printer = new Printer();
    Thread childThread = new Thread(printer);
    childThread.start();
  }
}
```

7.1.3 Volatile Keyword

The `volatile` keyword can be added to *variables* to ensure that all threads access the same variable copy, thus ensuring data sharing and consistency

- For multithreaded applications, we need to ensure a couple of rules for consistent behaviour
  1. **Mutual Exclusion** – only one thread executes a critical section at a time
  2. **Visibility** – changes made by one thread to the shared data are visible to other threads to maintain data consistency
- *synchronized* methods and blocks provide both of the above properties at the cost of application performance.
- *volatile* is quite a useful keyword because it can help ensure the visibility aspect of the data change *without providing mutual exclusion*

Let's see a few examples.

7.2 Producer/Consumer Problem

- The producer-consumer problem is the classical concurrency of a multi process synchronization problem. It is also known as bound-buffer problem.
- The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer generates a piece of data, put it into the buffer and starts again.
- The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
- A correct solution and implementation can be seen here:

```
public class Unit6_ProducerConsumer {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}
```

```
class CubbyHole {

    private int contents;
    private boolean available = false;

    // "synchronized" means only one thread can access
    //    this method at a time.
    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        available = false;
        return contents;
    }

    // "synchronized" means only one thread can access
    //    this method at a time.
    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        contents = value;
        available = true;
    }
}
```

```
class Consumer extends Thread {

    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #"
                       + this.number + " got: " + value);
        }
    }
}
```

```
class Producer extends Thread {

    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #"
                       + this.number + " put: " + i);
            try {
                sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

7.3 Practice

1. Create a new Java project.
2. Create a canvas and draw the following using Java 2D Graphics:
   (a) Create two small squares, different colors, that appear together on the canvas
   (b) Animate the two squares moving, each in a random direction, each on their own thread
   (c) When the two squares collide, they should randomly switch directions (make the squares big enough that they collide pretty regularly)
3. Create a "particle system"
   - When the two squares collide, there should be 25 small 'particles' (squares) created with random colors
   - Have them "explode" from the collision (shoot away in random directions)