Lab Notes
CMPU4021 Distributed Systems

Multithreading in Java

Concurrency

- The Java platform is designed from the ground up to support concurrent programming with basic concurrency support and high-level concurrency APIs.

- In concurrent programming, there are two basic units of execution: *processes* and *threads*.

- In the Java - concurrent programming
  - mostly concerned with threads

- *Concurrency in Java*
  - *`java.util.concurrent` packages*

Multiprocess applications

- Most implementations of the Java virtual machine run as a *single* process.

- A Java application can create additional processes using a `ProcessBuilder` object.

Threading with Java

- At least one thread
  - launched by JVM when `main` is started and ‘killed’ when `main` terminates.

- *Multithreading*
  - using one or more extra threads in order to ‘offload’ processing tasks onto them.
  - These threads need to be programmed explicitly.

Single processor threading

- Threads with the same priority are each given an equal **time-slice** or time **quantum** for execution on the processor;

- **Pre-emption**
  - – more urgent attention
  - – if thread is assigned higher priority.

Using threads in Java

- Multithreading directly accessible – no need to go through an operating system API
- System independent
- Each thread is associated with an instance of the class `java.lang.Thread.`
- There are two basic strategies for using Thread objects to create a concurrent application.
  - To directly control thread creation and management, simply *instantiate* Thread each time the application needs to initiate an asynchronous task.
  - To abstract thread management from the rest of your application, pass the application's tasks to an *executor*.
- We will concentrate on the first way

Defining and Starting a Thread

Two ways for creating threads:

- *Provide a `Runnable` object*
  - Create a class that does not extend `Thread` and specify explicitly that it implements interface `Runnable`.
  - The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor

- *Subclass `Thread`*
  - The `Thread` class itself implements `Runnable`, though its run method does nothing.
  - Create a class that extends `java.lang.Thread` class

Threading – Way 1: *Provide a Runnable object*

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }


}
```

Threading - Way 2: Subclass Thread

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Extend the `Thread` or implement the `Runnable`?

- If your class *must* be derived from some other class
  - then it should implement `Runnable`, otherwise it should extend `Thread`.

- If the class has a superclass (other than `Object`), it can not extend the `Thread` (no multiple inheritance).
  - A class that implements the `Runnable` interface can extend *any* class.

Extending the `java.lang.Thread` class

- The class that extends `Thread` should override the `run` method of class `Thread`.

- The `run` method serves the same purpose a the method `main` for a full application. Like `main`, `run` may not be called directly; it specifies the actions that a thread is to execute.

- The most common constructors used:
  - `Thread()` - allocates a new Thread object; the system generates a name of the thread as: *Thread-n*
  - `Thread(String name)` - allocates a new Thread object with the name *name*.

The Thread class (I)

- **getName()** method – retrieves the name of thread. E.g.:

```
Thread firstThread = newThread();
Thread secondThread() = new Thread(“namedThread”);
System.out.println(firstThread.getName());
System.out.println(secondThread.getName());
```

output:

```
Thread-0
namedThread
```

Pausing Execution with `Sleep`

- `Thread.sleep()` causes the current thread to suspend execution for a specified period

- An efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system

Pausing Execution with Sleep

- Method

  ```
  public static void sleep(long millis)
                 throws InterruptedException
  ```

  causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

  ```
  firstThread.sleep(1500); // pause for 1.5 seconds
  ```

- It allows other threads to be executed. When the sleeping time expires, the sleeping thread returns to a *ready* state, waiting for the processor.

- Sleep time very often determined by using the `java.lang.Math.random()` method in order to achieve a random sleeping time for threads.

Interrupts

- `The method:`

  `public void interrupt()`

  Interrupts an individual thread.

- It may be used by other threads to ‘awaken’ a sleeping thread before the thread’s sleeping time has expired.
  - Used very rarely.

- Static method `random` is often used to generate a random sleeping time for each threads

Example

```
public class ThreadShowName extends Thread {
   public static void main(String[] args){
      ThreadShowName thread1, thread2;
       thread1 = new ThreadShowName();
       thread2 = new ThreadShowName();

       thread1.start();
       thread2.start();
   }

   public void run(){
      int  pause;
      for (int i=0; i<10; i++){
         try{
           System.out.println(getName() + " being executed.");
           pause = (int)(Math.random() * 3000);
           sleep(pause);
         }
         catch (InterruptedException e){
           System.out.println(e.toString());
         }
      }
   }
}
```

Joins

- The join method allows one thread to wait for the completion of another, so that a thread will not start running until another thread has ended.
- If myThread is a Thread object whose thread is *currently* executing,

```
myThread.join();
```

- Causes the current thread to pause execution until *myThread's* thread terminates.
- Overloads of join allow the programmer to specify a waiting period.
- As with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.
- Like sleep, join responds to an interrupt by exiting with an `InterruptedException`.

Joins: Example

```
threadMessage("Waiting for MessageLoop thread to finish");
    // loop until MessageLoop thread exits
    while (t.isAlive()) {
            threadMessage("Still waiting...");
    // Wait maximum of 1 second for MessageLoop thread to finish.
    t.join(1000);
  if (((System.currentTimeMillis() - startTime) > patience)
                  && t.isAlive()) {
                threadMessage("Tired of waiting!");
                t.interrupt();
                // Shouldn't be long now -- wait indefinitely
                t.join();
            }
     }
threadMessage("Finally!");
```

Implementing the Runnable interface

1. Create an application class that explicitly implements the `Runnable` interface.

1. In order to create a thread – instantiate an object of the `Runnable` class and ‘wrap’ it in a `Thread` object (by creating a Thread object and passing the `Runnable` object as an argument to the `Thread` constructor). E.g.

```
Thread(Runnable<object>)
Thread(Runnable<object>, String<name>)
```

When either of these constructors is used, The `Thread` object uses the `run` method of the `Runnable` object in place of its own (empty) `run` method.

Implementing the Runnable interface (I)

```
class PrimeRun implements Runnable {
         long minPrime;
         PrimeRun(long minPrime) {
             this.minPrime = minPrime;
         }
         public void run() {
             // compute primes larger than minPrime
              . . .
         }
     }
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);
Thread thread1 = new Thread(p);
thread1.start();
```

Or, shorter:

```
Thread thread1 = new Thread(new PrimeRun(143));
thread1.start();
```

Extending the Thread / Implementing Runnable

```
class MyThread extends Thread {
   public void run() {
        System.out.println (“Hello World!”);
   }
}
class MyTest {
   public static void main(String Args[]) {
        new MyThread().start();
   }
}
--------------------------------------------------------------
---------------------------------------------------------------
class MyThread implements Runnable {
   public void run() {
        System.out.println (“Hello World!”);
   }
}
class MyTest {
   public static void main(String Args[]) {
        new Thread(new MyThread()).start();
   }
}
```

Multithreaded servers

- Servers can handle more than one client/connection at a time.

- Two stages:
  - the main thread (running in method `main`) allocates individual threads to incoming clients.

  - the thread allocated to each individual client then handles all subsequent interaction between that client and the server (via the thread's `run` method).

Multithreaded servers

- For each client-handling thread that is created, the main thread must ensure that the *client-handling thread is passed a reference to the socket* that was opened for the associated client.

(See the *MultiEchoServer.java* in Labs)

Daemon Threads

- Deamon threads: that exist for the purpose of providing a certain service.
  - E.g. A daemon thread, named Background Image Reader, of the HotJava web browser, reads images from the file system or the network for any object or thread that needs an image.
- The **run()** method for a deamon thread is usually an infinite loop that waits for a service request.
- **setDaemon() –** any thread can become a deamon or return from a deamon mode to a non-deamon.
- **isDaemon()**

Daemon Threads

- Normal thread and daemon threads differ in what happens when they exit.

- When the JVM halts any remaining daemon threads are *abandoned*: finally blocks are not executed, stacks are not unwound – JVM just exits.
  - Hence, daemon threads should be used sparingly and it is dangerous to use them for tasks that might perform any sort of I/O.

Parallel Execution of Threads

```
class PrintThread implements Runnable {
   String str;
   public PrintThread (String str) {
        this.str = str;
   }
   public void run() {
        for (;;) System.out.print (str);
   }
}
class ConcurrencyTest {
   public static void main (String Args[]) {
        new Thread(new PrintThread("A")).start();
        new Thread(new PrintThread("B")).start();
   }
}
```

Parallel Execution of Threads

- The output of the program above should look something like this (on multi-processor machines):

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
BBBBBBBBBBBBB
```

- It has nearly equal number of A’s and B’s.:

Preemptive versus Non-Preemptive multi-threading

- *Preemptive multi-threading* means that a thread may be preempted by another thread with an equal priority while it is running.

- The Java runtime will not preempt the currently running thread for another thread of the same priority. However, the underlying operating system implementation of threads may support preemption.

- Today, nearly all operating systems support preemptive multitasking, including the current versions of Windows, Mac OS, GNU/Linux, iOS and Android.

- If multi-tasking is not preemptive, the output of the previous example program (on an older OS) would like something like this:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA
```

Well behaved Threads

- A thread is supposed to be well behaved and give up the CPU periodically in order for other threads to be able to run.

- If your thread does not give up the CPU by suspending itself, waiting for a condition, sleeping or doing I/O operations then it should relinquish the CPU periodically by invoking the *Thread* class’s *yield()* method.

Preemptive versus Non-Preemptive multi-threading

```
class WellBehavedPrintThread implements
  Runnable {
  String str;
  public PrintThread (String str) {
      this.str = str;
  }
  public void run() {
      for (;;) {
      System.out.print (str);
      Thread.currentThread().yield();
      }
  }
}
```

Preemptive versus Non-Preemptive multi-threading

- The statement `Thread.currentThread().yield()` uses a public static method of the *Thread* class to get a handle to the currently running thread and then tells it to yield.
- The output of this example is:

```
ABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABAB...
```

- As a rule of thumb, threads should yield whenever possible, to allow others run.

Thread States

A thread can be in any of the following states:

- NEW
  - A thread that has not yet started is in this state.
- RUNNABLE
  - A thread executing in the Java virtual machine is in this state.
- BLOCKED
  - A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  - A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
  - A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
  - A thread that has exited is in this state.

Thread Priorities

- Each thread is assigned a priority, ranging from MIN_PRIORITY (equals 1) to MAX_PRIORITY (which is 10). A thread inherits its priority from the thread that spawned it.
- The scheduling algorithm always lets the highest priority runnable thread run. If at any time a thread with a higher priority than all other runnable threads becomes runnable, the runtime system schedules it for execution.
- The new higher priority thread is said to preempt the other threads.
- A lower priority thread can only run when all higher priority threads are non-runnable.

Thread Priorities

- The priority of the main thread (the thread that starts the `main()` method of your program) is NORM_PRIORITY (equals 5).

- You can set a thread’s priority by invoking the `setPriority()` method and get the thread’s priority by invoking `getPriority()`.

Thread synchronization

- Many threads - need to synchronize their activities.
- Need to prevent concurrent access to data structures in the program that are shared among the threads.
- Java provides mechanisms for synchronization and mutual exclusion (allowing only one thread to run through critical code sections in the program).

Thread Synchronization

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to.

- Two kinds of errors possible:
  - thread interference; and
  - memory consistency errors.

- The tool needed to prevent these errors is synchronization.

Thread Synchronization

- Synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even suspend their execution.

- *Starvation* and *livelock* are forms of thread contention.

Starvation

- The term *starvation* is used to denote situations where one thread is deprived of a resource (namely an access to a monitor). Unlike deadlock, in a starvation situation the calculation can continue in the system, it’s just the starved thread that can’t go on.
- Starvation can occur when a high priority thread starts running and never relinquish the CPU.
- Although the high priority thread can do great many things, all the lower priority threads are starved.

Livelock

- A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result.

- Livelocked threads are unable to make further progress.
  - However, the threads are not blocked — they are simply too busy responding to each other to resume work.
  - This is comparable to two people attempting to pass each other in a corridor:
    - John moves to his left to let Paul pass, while Paul moves to his right to let John pass.
    - Seeing that they are still blocking each other, John moves to his right, while Paul moves to his left. They're still blocking each other, so...

Deadlocks

- *Deadlock*
  - occurs when threads are waiting for events that will never occur
- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
- The two threads are in a deadlock if one of them waits for the value to change while blocking the other one from changing it.
- Deadlock occurs when two or more threads are waiting for some condition to change, while that condition is precluded from changing because of all threads that can change the condition are waiting.

Intrinsic locks or monitor locks

- Synchronization is built around an internal entity known as the intrinsic lock or monitor lock.
  - The API specification often refers to this entity simply as a "monitor."
- A monitor is associated with a specific data item and functions as a lock on that data.
- When a thread holds the monitor for some data item, other threads are locked out and cannot inspect or modify the data.
- A thread can acquire the monitor if no other thread currently owns it, and it can release it at will.
- A thread can re-acquire the monitor if it already owns it.
- A locking is achieved by using the Java keyword `synchronized`
  - *synchronized methods* and *synchronized statements*.

Synchronized methods

- Declaring a method **synchronized** means that only the thread holding the monitor can run through the synchronized method in that instance.

```
public synchronized void updateSum(int amount){
    sum+=amount;
}
```

- If *sum* is not locked when the above method is invoked, then the lock on *sum* is obtained, preventing any other thread from executing *updateSum*.

Synchronized statements

- Any code segment can be declared as synchronized:
  - performance issues (if too many)
- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Waiting for events: `java.lang.Object` methods

- They can just be called only:
  - When the current thread has a lock on the object (i.e. from within a `synchronized` method, or
  - From within a method that has been called by a `synchronized` method.)

- `wait()`
  - If a thread executing a synchronized method determines that it cannot proceed then it may put itself into a waiting state by calling method `wait()` .This releases the thread’s lock on the shared object and allows other threads to obtain the lock.

- `notify()`
  - A synchronized method reaches completion, then it may call `notify()` , which will ‘wake up’ a thread that is in the waiting state.

- `notifyAll()`
  - ‘wakes up’ all object waiting on a given object.

Problems with Java's Synchronized

- No backtracking
  - Once a thread hits the entrance to a synchronized block there is no way to go back.
  - The thread must wait around for the monitor to be released
- Only one thread can hold the monitor at a time
- A thread can only hold one monitor at a time
- A thread can not take a monitor and pass it into a different method

Deadlock

- A deadlock can occur if we use a busy-waiting loop inside a monitor, waiting for another thread to change a condition but never giving it the opportunity to obtain the monitor.

- Threads are locked in a deadlock if the calculation, or whatever it is the threads are doing, can’t continue.

Semaphores

- Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource.
- Each `acquire()` blocks if necessary until a permit is available, and then takes it.
- Each `release()` adds a permit, potentially releasing a blocking acquirer.

java.util.concurrent.Semaphore

- Semaphore object maintains a set of permits:

Constructors:

`Semaphore(int permits);`

  - Each acquire blocks til permit is available; Each release adds a permit
  - Just keeps a count of available permits

- Semaphore constructor also accepts a fairness parameter:

`Semaphore(int permits, boolean fair);`

- Creates a Semaphore with the given number of permits and the given fairness setting.

permits: initial value

fair: if true semaphore uses FIFO to manage blocked threads; if set false, class doesn’t guarantee order threads acquire permits.

- See `SemaApp.java` example in Labs

High Level Concurrency Objects

- Previous examples
  - Low-level APIs that have been part of the Java platform from the very beginning.
  - Important to understand
  - To build simple applications

- Higher-level building blocks are needed for more advanced tasks, such as
    - massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

High Level Concurrency Objects: `java.util.concurrent`

- Lock objects
  - support locking idioms that simplify many concurrent applications.
- Executors
  - define a high-level API for launching and managing threads.
  - Executor implementations provided by `java.util.concurrent` provide thread pool management suitable for large-scale applications.
- Concurrent collections
  - make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
- Atomic variables
  - have features that minimize synchronization and help avoid memory consistency errors.
- `ThreadLocalRandom`
  - for applications that expect to use random numbers from multiple threads

Interface Lock

- Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.

- Their biggest advantage over implicit locks is can back out of an attempt to acquire a Lock:
  - i.e. livelock, starvation & deadlock are not a problem

Interface Condition

- `Conditions` (also known as condition queues or condition variables) provide
  - a means for one thread to suspend execution (to "wait") until notified by another thread that some state condition may now be true.
- Because access to this shared state information occurs in different threads, it must be protected, so a lock of some form is associated with the condition.

Interface Lock

- `Lock` interface also supports a *`wait/notify`* mechanism, through the associated `Condition` objects

- `Lock` and `Condition`
  - replace basic monitor methods (`wait()`, `notify()` and `notifyAll()`) with specific objects:
    - Lock in place of *synchronized* methods and statements.
    - An associated Condition in place of Object's monitor methods.
    - A Condition instance is intrinsically bound to a Lock.

Executors

- In the previous examples
  - there's a close connection between the task being done by a new thread,
    - as defined by its Runnable object, and
  - the thread itself
    - as defined by a Thread object.

- In large-scale applications, it makes sense to separate thread management and creation from the rest of the application.
  - Executors.
    - Objects that encapsulate these functions

Reference

- Chapter 3, *Introduction to Network Programming in Java* by Jan Graba

- *Parallel Programming in Java*: A Tutorial, Arik Baratz, Dror Birkman, Ofir Carny, Shy Cohen, and Assaf Schuster

- Java Tutorial section on Threads
  - https://docs.oracle.com/javase/tutorial/essential/concurrency
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html