ITMC403 Parallel and Distributed Computing

Tasks and Threads

Tasks and Threads

- A **task** is an abstraction of a series of steps
  - Might be done in a separate thread
  - Java libraries use the Runnable interface
  - work done by method run()
- **Thread**: a Java class for a thread
  - work done by method run()

- How to associate a task with a thread?
- How to start a thread?

New Steps to start a threads

To use the **Runnable interface** to create and start a thread, you have to do the following:

1) **Create** a class that implements Runnable.

2) **Provide** a run method in the Runnable class.

3) **Create an instance** of the Thread class and pass your Runnable object to its constructor as a parameter.

   A Thread object is created that can run your Runnable class.

1) **Call the Thread object's start method**.

   The run method of your Runnable object is called and executes in a separate thread.

Steps 1 & 2 are easy. Steps 3 & 4 you can complete them in several ways. Examples: your Runnable class is named RunnableClass:

```
RunnableClass rc = new RunnableClass();
Thread t = new Thread(rc);
t.start();
```

to be as concise as possible, so you often see this code compressed to something more like

```
Thread t = new Thread(new RunnableClass());
t.start();
```

or even just this:

```
new Thread(new RunnableClass()).start();
```

This single-line version works — provided that you don’t need to access the thread object later in the program.

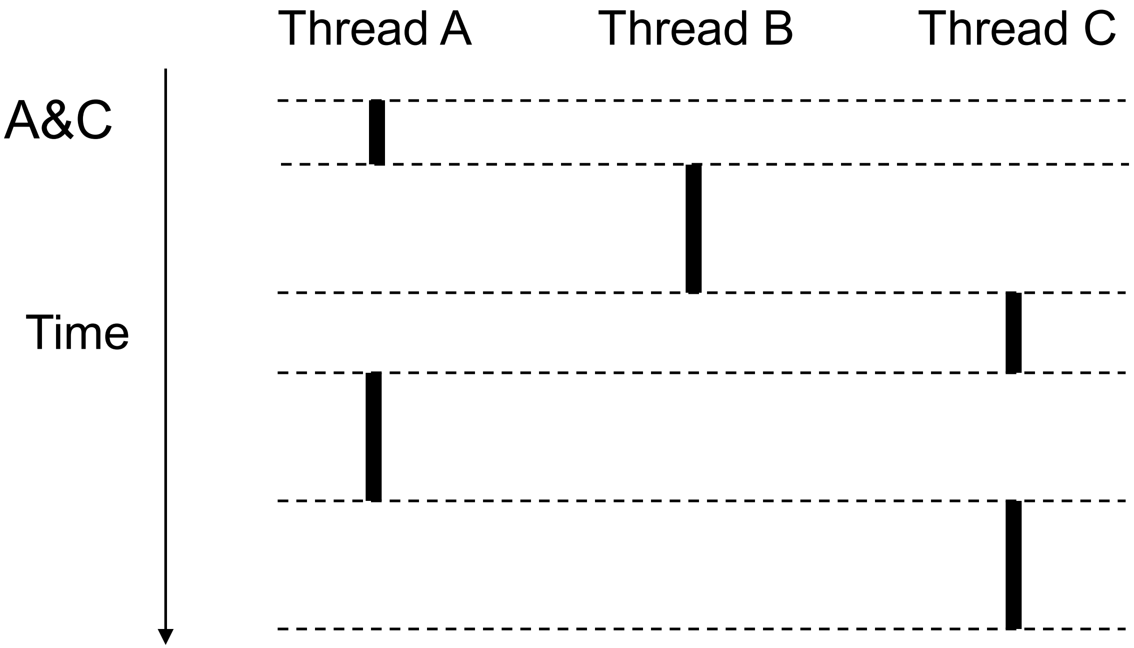
Concurrent Thread Execution

- Two threads run concurrently (are concurrent) if their logical flows overlap in time
- Otherwise, they are sequential (just like processes)

- Examples:
  - Concurrent: A & B, A&C

    True parallelism
  - Sequential: B & C

    Pseudo Parallelism



Threads in Java

- There are two ways to create a java thread:
  - By extending the **java.lang.Thread** class.
  - By implementing the **java.lang.Runnable** interface.
- The ***run()*** method is where the action of a thread takes place.
- The execution of a thread starts by calling its ***start()*** method.

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(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:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

Implementing the Runnable Interface

- In order to create a new thread we may also provide a class that implements the **java.lang.Runnable** interface.
- Preferred way in case our class has to subclass some other class.
- A Runnable object can be wrapped up into a Thread object:
  - **Thread**(Runnable target)
  - **Thread**(Runnable target, String name)
- The thread’s logic is included inside the **run()** method of the **runnable** object.

```
class ExClass
  extends ExSupClass
  implements Runnable {
    …
    public ExClass (String name) {
    }
    public void run() {
            …
    }
}
```

```
class A {
    …
    main(String[] args) {
        …
        Thread mt1 = new Thread(new ExClass("thread1”));
        Thread mt2 = new Thread(new ExClass("thread2”));
        mt1.start();
        mt2.start();
    }
}
```

Implementing the Runnable Interface

- Constructs a new thread object associated with the given ***Runnable*** object.
- The new Thread object's ***start()*** method is called to begin execution of the new thread of control.
- The reason we need to pass the runnable object to the thread object's constructor is that the thread must have some way to get to the ***run()*** method we want the thread to execute. Since we are no longer overriding the ***run()*** method of the ***Thread*** class, the default ***run()*** method of the ***Thread*** class is executed:

```
public void run() {
        if (target != null) {
                target.run();
        }
}
```

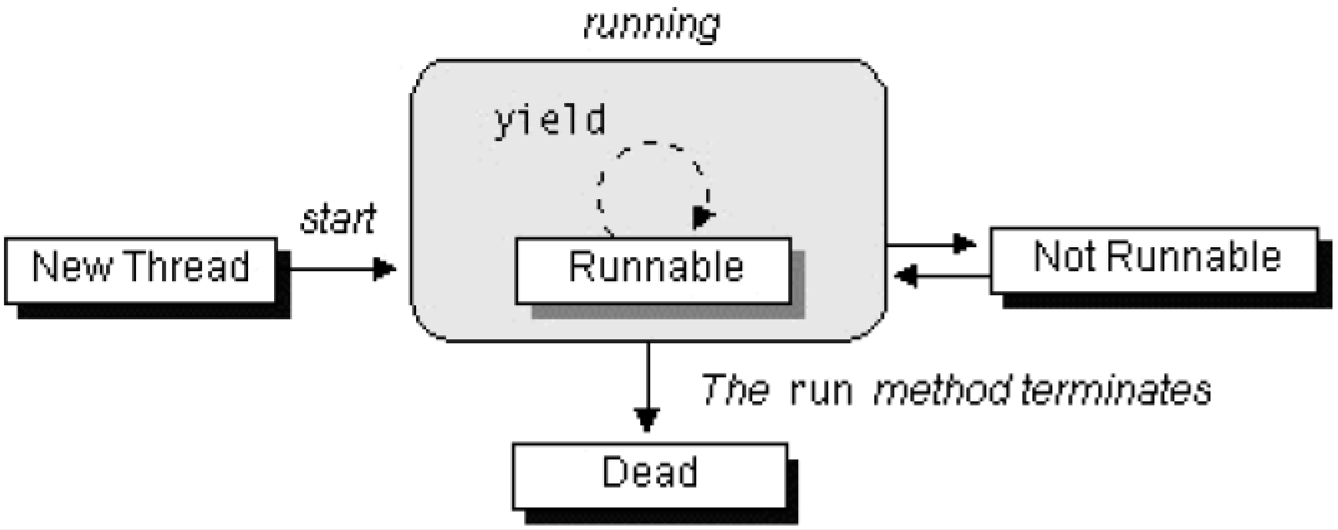
- Here, target is the runnable object we passed to the thread's constructor. So the thread begins execution with the ***run()*** method of the ***Thread*** class, which immediately calls the ***run()*** method of our runnable object.

Sleep, Yield, Notify & Wait Thread's Functions

- ***sleep(long millis)*** - causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- ***yield()*** - causes the currently executing thread object to temporarily pause and allow other threads to execute.
- ***wait()*** - causes current thread to wait for a condition to occur (*another thread invokes the **notify()** method or the **notifyAll()** method for this object*). This is a method of the ***Object*** class and must be called from within a **synchronized** method or block.
- ***notify()*** - notifies a thread that is waiting for a condition that the condition has occurred. This is a method of the ***Object*** class and must be called from within a **synchronized** method or block.
- ***notifyAll()*** – like the ***notify()*** method, but notifies all the threads that are waiting for a condition that the condition has occurred

The Lifecycle of a Thread

- The ***start()*** method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's ***run()*** method.
- A thread becomes **Not Runnable** when one of these events occurs:
  - Its ***sleep()*** method is invoked.
  - The thread calls the ***wait()*** method.
  - The thread is blocked on I/O operations.
- A thread dies naturally when the ***run()*** method exits.



Thread Priority

- On a single CPU, threads actually run one at a time in such a way as to provide an **illusion of concurrency**.

- Execution of multiple threads on a single CPU, in some order, is called **scheduling**.

- The Java runtime supports a very simple **scheduling algorithm** (fixed priority scheduling). This algorithm schedules threads based on their priority relative to other runnable threads.

- The runtime system chooses the runnable thread with the **highest** priority for execution.

Thread Priority

- If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a **round-robin fashion** - each process is guaranteed to get its turn at the CPU at every system-specified time interval.

- The chosen thread will run until:
  - A higher priority thread becomes runnable.
  - It yields (calls its ***yield()*** method), or its ***run()*** method exits.
  - On systems that support **time-slicing**, its time allotment has elapsed.

- You can modify a thread's priority at any time after its creation by using the ***setPriority()*** method.

Synchronization of Java Threads

- In many cases **concurrently** running threads share data and must consider the state and activities of other threads.
- If two threads can both execute a method that modifies the state of an object then the method should be declared to be ***synchronized***, those allowing only one thread to execute the method at a time.
- If a class has at least one ***synchronized*** method, each instance of it has a **monitor**. A monitor is an object that can **block** threads and **notify** them when the method is available.

Example:

```
public synchronized void updateRecord() {
//**** critical code goes here …
}
```

- Only one thread may be inside the body of this function. A second call will be blocked until the first call returns or ***wait()*** is called inside the synchronized method.

Synchronization of Java Threads

- If you don’t need to protect an entire method, you can synchronize on an object:

```
 public void foo() {
                synchronized (this) {
                //critical code goes here …
                }
                …
 }
```

- There are **two syntactic forms** based on the **synchronized** keyword - blocks and methods.
- **Block synchronization** takes an argument of which object to lock. This allows **any method to lock any object.**
- The most common argument to synchronized blocks is **this.**
- Block synchronization is considered more fundamental than method synchronization.

Applying Synchronization (Example)

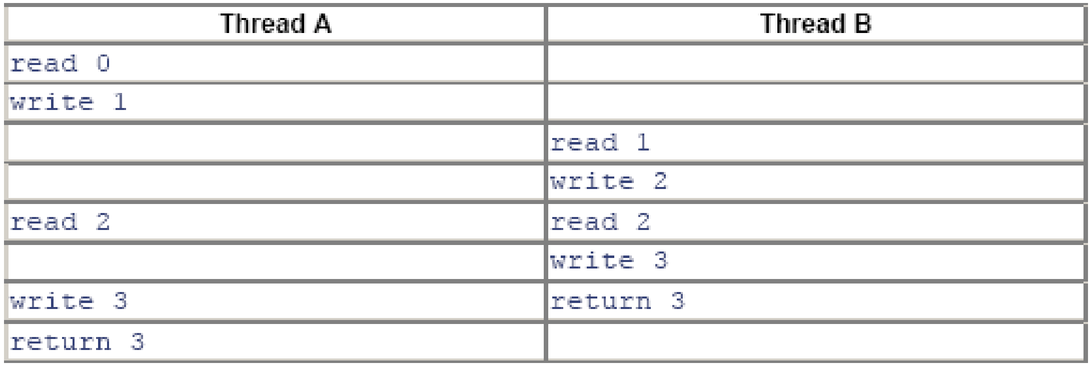
Consider the following class:

```
class Even {
        private int n = 0;
        public synchronized  int next(){
        ++n;
        ++n;
        return n; //**** next is always even
    }
}
```

Declaring the next method as synchronized would resolve such conflicting problems.

Without synchronizing, the desired postcondition may fail due to a storage conflict when two or more threads execute the next method of the same Even object.

Here is one possible execution trace:



Thread A	Thread B
read 0	
write 1	
	read 1
	write 2
read 2	read 2
	write 3
write 3	return 3
return 3	