Multithreaded applications

Advanced Programming

Why multithreaded programs

- JRE uses multiple threads for better use of CPU cycles
- A **single threaded** program uses event loop with **polling** mechanism
  - Decision mechanism under which an event is given the **priority** to be handled by the **event handler**
  - Until the event handler returns **nothing else can run**
  - When the event handler is waiting for a specific resource the overall execution is **blocked waiting for that resource**

Multithreaded programs

- In multithreaded programs when one thread is blocked the other can run
- In **single core processors** the threads use different **slices** of the CPU whereas in **multicore** the use CPU in **parallel** over the cores

Physical memory (RAM)

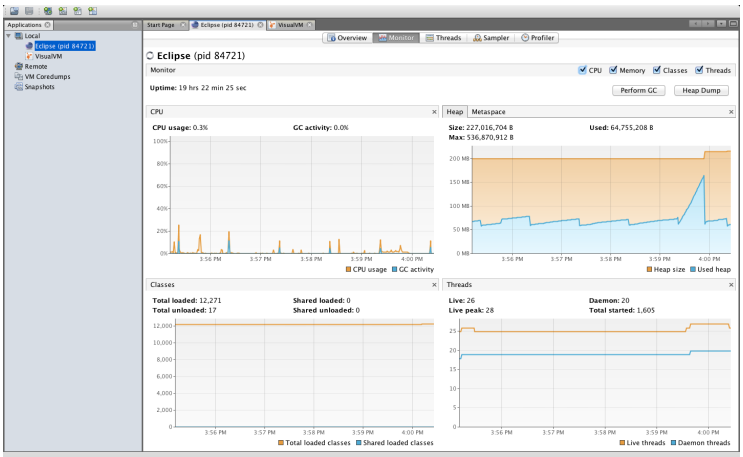
- Do you remember? We saw that:
  - Main memory is wired directly to the processor, addressable by physical address
  - Access to main memory may take tens or even hundreds of cycles

Thread

- The OS allocates a copy of the physical memory (called **virtual memory**) for each Java process (called **thread**)
- One can have at least as many threads in parallel as the number of CPUs (modern computers are multi processors and multi cores)

Visualising memory consumption

- Go to your bin folder and search for jvisualvm
- In the shell window type jvisualvm



The Class Thread

- Java uses the class Thread to instantiate and manage threads

Methods of class Thread

- **getName (setName)** Obtain/set a thread's name
- **getPriority (SetPriority)** Obtain/set a thread's priority
- **isAlive** Determine if a thread is still running
- **join** Wait for a thread to terminate
- **run** Entry point for the thread
- **sleep** Suspend a thread for a period of time
- **start** Start a thread by calling its run method

The main thread

- To get he current thread

  static Thread **currentThread**( )

Managing the current Thread

```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

Source: reference book

Thread.sleep(long)

- Thread.sleep causes the current thread to suspend execution for a specified period
- To processor time available to the other threads of an application or other applications that might be running on a computer system
- The sleep method can also be used for pacing and waiting for another thread

Reflecting on code

- The **main()** declares that it throws **InterruptedException**
- This is an exception that **sleep() throws** when **another thread interrupts** the current thread **while sleep is active**
- If this application has not defined another thread to cause the interrupt, it doesn't bother to catch InterruptedException

Sleep time

- Two overloaded versions of sleep are provided:
  - one that specifies the sleep time to the **millisecond** and one that specifies the sleep time to the **nanosecond**
- It is not guaranteed that invoking sleep will suspend the thread for precisely the time period specified:
  - They are limited by the OS
  - The sleep period can be terminated by interrupts

Two ways to create a thread

- Either extend **Thread** (C) or
  - Thread contains other methods besides run()
- Implement **Runnable** (I)
  - has only the run() method but the implementing class can be further extended

Runnable object

- Override run() for the code to be executed in the Thread
- The Runnable object is then passed to the Thread constructor
- Then the Thread object calls start()

```
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();
        }

    }
```


Subclassing Thread

- The Thread class itself implements Runnable, though its **run method does nothing**
- Subclass Thread and override run()
- An object of the subclass calls start()

```
public class HelloThread extends Thread {

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

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

}
```

Note

- In both cases is an object of type Thread that invokes start()

Create a second thread

```
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

```
class ThreadDemo {
  public static void main(String args[ ] ) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {

      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Thread constructor

- **this** passes the the current object of type (Runnable) NewThread to the constructor of the Thread object "t"
- "Demo Thread" is the name of such object
- Next, **start**( ) is called, which starts "t" with the overridden method of the current object of type NewThread
- The thread of execution beginning at the run( ) method

- After calling start( ), NewThread's constructor returns to main( ).
- When the main thread resumes, it enters its for-loop Both threads continue running, sharing the CPU in single-core systems, until their loops finish

Output

it prints out the existing threads
and the priority of the current thread;
in our case the current thread is main
with priority 5 (max default)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

After the new thread object has been created
in the heap, the execution returns to
the main method and starts the for loop there

When sleep is invoked in the main method,
the demo thread runs and the for loop in
the newThread object executes
until the sleep time of the main thread finishes

See the NewThread and
ThreadDemo classes in the sample code

- When object in the heap are created and the start method has been invoked, they are alive, but they might not be running

Sequence of threads

- Generally we want the main thread to finish last
  - We used sleep() with a larger number of milliseconds
- Other two ways to determine whether a thread has finished:
  - Call the boolean method **isAlive**( ) on the thread
  - Use the **join**( ) on the specific thread object that **waits for this thread to die**

join()

- The join method allows one thread to wait for the completion of another.
- If t is a Thread object whose thread is currently executing

```
t.join();
```

- causes the current thread to pause execution until t's thread terminates

join()

- Overloads of join allow the programmer to specify a waiting period
- Join is dependent on the OS for timing, so join might not wait exactly as long as one specifies

Example

- See example DemoJoin class in code sample

Threads prioritization

- In class Thread use setPriority( ) method to set a thread's priority

  final void setPriority(int level)

- Levels from 1 to 10. Static final variable of Thread:

  MIN_PRIORITY=1

  MAX_PRIORITY=10

  NORM_PRIORITY=5

- Use getPriority() to get the priority of one thread

Problem

```
class Foo {
  private Helper helper;
  public Helper getHelper() f
  if (helper == null) {
      helper = new Helper();
  }
  return helper;
}
```

Synchronization

- A lock must be obtained in case two or more threads call getHelper() simultaneously
- Otherwise, either they may both try to create the object at the same time, or one may wind up getting a reference to an incompletely initialized object

```
class Foo {
     private Helper helper;
     public synchronized Helper getHelper() f
     if (helper == null) {
         helper = new Helper();
     }
     return helper;
}
```


Synchronization

- The first call to getHelper() creates the object and only the few threads trying to access it during that time need to be synchronized
- After that, all calls just get a reference to the member variable.
- Since **synchronizing a method** can **decrease performance** by a factor of 100 or higher, the overhead of acquiring and releasing a lock every time this method is called is unnecessary

Double-checked locking w. synchronized statement

```
class Foo {
   private Helper helper;
   public synchronized Helper getHelper() {
   if (helper == null) {
      synchronized(this){
         if (helper == null) {
            helper = new Helper();
         }
      }
   }
   return helper;
   }
}
```


Double-checked locking w. synchronized statement

- Check that the variable is initialized (without obtaining the lock)
- If it is initialized, return it immediately. Otherwise, obtain the lock
- Double-check whether the variable has already been initialized: if another thread acquired the lock first, it may have already done the initialization
- If so, return the initialized variable
- Otherwise, initialize and return the variable
- **A synchronized block can choose which object it synchronizes on!**

Problem

- Consider the classic **queuing problem**, where one thread is **producing** some data and another is **consuming** it and the producer has to wait until the consumer is finished before it generates more data and vicersa

Producer and Consumer metaphor

- We can implement a **loop** to check some condition repeatedly. Once the condition is true, appropriate action is taken. To implement this loop Java uses polling: CPU cycling until the condition is satisfied. **This wastes CPU time**
- **Multi-threads does not make use of polling**

Inter-thread Communication

- Java uses wait( ), notify( ), and notifyAll( ) methods(final methods of Object).
- All three methods can be called only from within a **synchronized** context

```
Object mon = new Day();
synchronized (mon) {
    mon.wait();
}
```


Inter-thread Communication

- Use wait(), notify() to communicate between running threads so that the execution follow the path wanted
- **wait**() causes the current thread to wait until another thread invokes notify() or notifyAll() for this object
- **notify**() wakes up a single thread that is waiting
- If more threads are waiting on this object, one of them is chosen arbitrary
- **notifyAll**( ) wakes up all threads that are waiting

Use wait within a loop

- In very rare cases the waiting thread could be awakened with no apparent reasons. In this case, a waiting thread resumes without notify( ) or notifyAll( ) having been called.
- Oracle recommends that calls to wait( ) should take place **within a loop** that checks the condition on which the thread is **waiting**

```
synchronized {
    while (!condition) { mon.wait(); }
}
```



```
public class Queue {
    // An incorrect implementation of a producer and consumer.
    int n;
    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }
     synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}
```

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
```

see the code sample

Correct implementation

```
boolean valueSet = false;
      synchronized int get() {
            while(!valueSet)
                  try {
                        wait();
                  } catch(InterruptedException e) {
                        System.out.println("InterruptedException caught");
                  }
            System.out.println("Got: " + n);
            valueSet = false;
            notify();
            return n;
      }

synchronized void put(int n) {
            while(valueSet)
                  try {
                        wait();
                  } catch(InterruptedException e) {
                        System.out.println("InterruptedException caught");
                  }
            this.n = n;
            valueSet = true;
            System.out.println("Put: " + n);
            notify();
      }
}
```

Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3