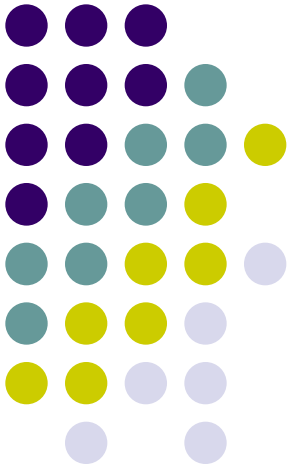
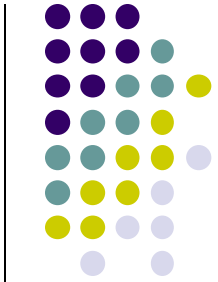
ITMC403 Parallel and Distributed Computing

Threads Scheduling





Intro.

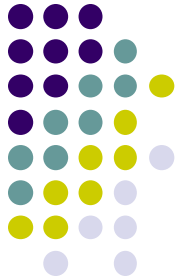
- The term "**thread scheduling**" covers a variety of topics.

- One of those topics, which is
  how a computer selects particular threads to run.

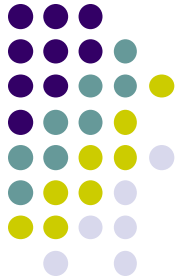
- The key to understanding **thread scheduling** is to realize that *a CPU is a scarce resource*.

- When two or more threads want to run on a single-processor machine, they end up competing for the CPU, and it's up to someone — either the programmer, the Java virtual machine, or the operating system — to make sure that the CPU is shared among these threads.

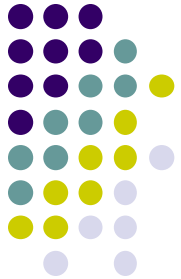
- The same is true whenever a program has more threads than the machine hosting the program has CPUs.



```
package javathreads.examples.itmc403;
import java.util.*;
import java.text.*;

public class Task implements Runnable {
        long n;
        String id;

        private long fib(long n) {
                if (n == 0)
                        return 0L;
                if (n == 1)
                        return 1L;
                return fib(n - 1) + fib(n - 2);
        }
```

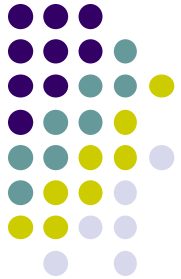


```
    public Task(long n, String id) {
            this.n = n;
            this.id = id;
    }

    public void run( ) {
            Date d = new Date( );
            DateFormat df = new SimpleDateFormat("HH:mm:ss:SSS");
            long startTime = System.currentTimeMillis( );
            d.setTime(startTime);
            System.out.println("Starting task " + id + " at " + df.format(d));
            fib(n);
            long endTime = System.currentTimeMillis( );
            d.setTime(endTime);
            System.out.println("Ending task " + id + " at " + df.format(d) +
            " after " + (endTime - startTime) + " milliseconds");
    }
}
```



```
package javathreads.examples.itmc403.example1;
import javathreads.examples.itmc403.*;

public class ThreadTest {
        public static void main(String[] args) {
                int nThreads = Integer.parseInt(args[0]);
                long n = Long.parseLong(args[1]);
                Thread t[ ] = new Thread[nThreads];

                for (int i = 0; i < t.length; i++) {
                        t[i] = new Thread( new Task(n, "Task " + i) );
                        t[i].start( );
                }

                for (int i = 0; i < t.length; i++) {
                        try {
                                t[i].join( );
                        } catch (InterruptedException ie) {}
                }
        }
}
```

Running this code with 3 threads produces this kind of output:

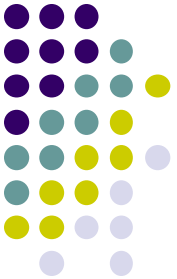


```
Starting task Task 2 at 00:04:30:324
Starting task Task 0 at 00:04:30:334
Starting task Task 1 at 00:04:30:345
Ending task Task 1 at 00:04:38:052 after 7707 milliseconds
Ending task Task 2 at 00:04:38:380 after 8056 milliseconds
Ending task Task 0 at 00:04:38:502 after 8168 milliseconds
```

Comments (notice that):

- Last thread we created and started (Task 2) was the first one that printed its first output.
- All threads started within 20 milliseconds of each other.
- The actual calculation took about 8 seconds for each thread.
- The threads ended in a different order than they started in.
- Task 2 started first, it took 349 milliseconds longer to perform the same calculation as Task 1 and finished after Task 1.

Certain virtual machines and operating systems, however, would produce this output:



```
Starting task Task 0 at 00:04:30:324
Ending task Task 0 at 00:04:33:052 after 2728 milliseconds
Starting task Task 1 at 00:04:33:062
Ending task Task 1 at 00:04:35:919 after 2857 milliseconds
Starting task Task 2 at 00:04:35:929
Ending task Task 2 at 00:04:37:720 after 2791 milliseconds
```

Comments (notice that):

- The total here takes about the same amount of time, but now they have run sequentially:
- The Task 2 did not begin to execute until the first Task 1 was finished.
- Another interesting fact about this output is that each individual task took less time than it did previously.