Lab 1

Introduction to the work environment

The programming during the concurrency labs will be done using Java. The programming can be done using Eclipse on the lab machines and the sources can be further tested on the T2000 SunFire multicore machine to better observe the multithreading effects.

Logging in via SSH

To log in via SSH you can use either *ssh* on Linux or *putty* on Windows. The basic syntax for Linux is:

```
ssh accountname@serveraddress
```

For windows the *putty* interface is self-explanatory. The program can be downloaded from here.

Basic commands

The multicore machine runs Sun Solaris 10 as operating system. The shell configured for the accounts will be BASH. Detailed information for running a command can be obtained with *man commandname* (to exit the manual screen press "q"). Below are short descriptions for the main commands available.

- **ls** - Lists the files from a directory, implicitly the current one. By using the *-l* option the file entries are displayed in detailed form. The *-a* option results in displaying also the files that have a name starting with . (hidden by default).
- **cp [source] [destination]** - Copies the source file to destination. To copy entire folders recursively it is used with the *-r* option.
- **mv [source] [destination]** - Moves the source file to destination.
- **rm [target]** - Removes the file specified as target. To remove folders by deleting the contents recursively the *-r* option must be specified.
- **cd [path]** - Changes the current directory to the specified path.
- **mkdir [folder]** - Creates the directory specified as argument.
- **less [file]** - One of the commands that can be used to display the contents of a file.
- **mc** - launches Midnight Commander, a graphical file manager
- **mcedit [filename]** - launches one simple to use text editor for writing a file.
- **ps** - Displays the list of current running processes at the users terminal containing the process ids in the first column. By adding the *-u username* option, all the processes belonging to the specified user are displayed.
- **kill [-signal number] [process id]** - Sends the specified signal to the specified process. To terminate a process the signal number option would be *-9* .

Transferring files via SCP

Files can be transferred to and from the multicore machine via SCP. If the local machine is running Linux the following can be used at the command line:

- `scp localfilepath useraccount@serveraddress:remotefilepath`

  to transfer the local file to the remote file path on the target machine (if only the target file name without any path is specified it will be transferred in the user's home account folder)

- `scp useraccount@serveraddress:remotefilepath localfilepath`

  to transfer the remote file to the current host

If the local machine is running Windows various SCP clients are available. A graphical one is WinSCP available here and a command-line one is PSCP available here.

Using Java from command line

The Java compiler *javac* can be used to compile sources at the command line. A typical usage will be:

```
javac Program.java
```

For compiling more sources at once wildcards can be used:

```
javac *.java
```

For specifying the path for additional sources needed the -sourcepath option can be used.

The compilation process results in creating .class files for the compiled classes. As example, for the above Program.java (considering it contains a main function in the class Program) to execute you should run:

```
java Program
```

Note that only the main class name is specified, not the entire filename (without the .class extension).

Basic Java Thread notions

The Concurrency API was first introduced with the release of Java 5 and then progressively enhanced with every new Java release. The majority of concepts shown here also work in older versions of Java. However, the code samples focus on Java 8 and sometimes use lambda expressions and other new features. If you're not yet familiar with lambdas it's recommended to read the Java 8 Tutorial first.

All modern operating systems support concurrency both via processes and threads. Processes are instances of programs which typically run independent to each other, e.g. if you start a java program the operating system spawns a new process which runs in parallel to other programs. Inside those processes we can utilize threads to execute code concurrently, so we can make the most out of the available cores of the CPU.

An informal definition for a thread would be a sequential series of instructions executing inside a process. An application has at least one thread - the main thread. Running a Java application implies also other threads for different operations like memory management, event delivery, etc, but these are usually transparent from the programmer's point of view. When programming an application in Java the main thread is the thread started by the *main* function and ended along with it and implicitly with the program execution.

Creating and starting a thread

Java supports Threads since JDK 1.0. Before starting a new thread you have to specify the code to be executed by this thread, often called the *task*.

There are two main possibilities for creating a task and starting a Thread with that task in Java. The first is by extending the `java.lang.Thread` class and overriding the `run()` method:

```
public class MyThread extends Thread {

   public void run (){

        System.out.println("This is my thread!");
   }

}
```

The code of the task that will be executed by the new thread (in parallel with the thread that will start it) is contained inside the `run()` method. To create a thread, you first create a new object from the extended thread class. Doing this alone does not also start the thread. To actually get the thread running the `start()` method must be called. Notice that the `run()` method is not called directly[1]:

```
MyThread newThread = new MyThread();
newThread.start();
```

[1] running it would cause the task's code to be executed in the same thread, and not the newly created `newThread`

The second possibility for creating a thread is by first creating the task that needs to run concurrently; then the new thread is started using the created task. This is done by implementing `Runnable` - a functional interface defining a single void no-args method `run()` as demonstrated in the following example:

```
Runnable task = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello the task is running in thread named " +
threadName);
};

task.run();
```

The `run()`method needs to be implemented, just like in the previous example, although using a lambda expression. To actually start the thread after instantiating a new object, it's again the `start()` method from the Thread class the one that should be called. To do so, a Thread instance is created based on the Runnable one:

```
Thread thread = new Thread(task);
thread.start();

System.out.println("Done!");
```

Since `Runnable` is a functional interface we can utilize Java 8 lambda expressions to print the current threads name to the console. First we execute the runnable directly on the main thread before starting a new thread.

The result on the console might look like this:

```
Hello the task is running in thread named main
Hello the task is running in thread named Thread-0
Done!
```

Threads can be put to sleep for a certain duration. This is quite handy to simulate long running tasks in the subsequent code samples of this article:

```
Runnable runnable = () -> {
    try {
        String name = Thread.currentThread().getName();
        System.out.println("Foo " + name);
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Bar " + name);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Thread thread = new Thread(runnable);
thread.start();
```

When you run the above code you'll notice the one second delay between the first and the second print statement. `TimeUnit` is a useful enum for working with units of time. Alternatively you can achieve the same by calling `Thread.sleep(1000)`.

Working with the `Thread` class can be very tedious and error-prone. Due to that reason the **Concurrency API** has been introduced back in 2004 with the release of Java 5. The API is located in package `java.util.concurrent` and contains many useful classes for handling concurrent programming.

Thread Pools

After processing a task in the manually created thread, that thread ceases to execute and dies --- it cannot be reused to other tasks. This implies that once a thread dies it cannot be started again. As creating a thread is expensive, there exist thread pools, in which threads are started and lay dormant, until they receive task to be processed. The Concurrency API introduces the concept of an `ExecutorService` to implement the concept of a thread pool.

Executors are capable of running tasks, which are `submit()`-ted to it, and manage a pool of threads, so we don't have to create new threads manually. The thread pool distributes the submitted tasks among available threads. All threads of the internal pool will be reused under the hood for incoming tasks, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service.

This is how the first thread-example looks like using executors:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});

// => Hello pool-1-thread-1
```

The class `Executors` provides convenient factory methods for creating different kinds of executor services. In this sample we use an executor with a thread pool of size one.

The result looks similar to the above sample but when running the code you'll notice an important difference: the java process never stops! Executors have to be stopped explicitly - otherwise they keep listening for new tasks.

An `ExecutorService` provides two methods for that purpose: `shutdown()` waits for currently running tasks to finish while `shutdownNow()` interrupts all running tasks and shut the executor down immediately.

Interacting with a Thread

A Thread instance can be treated like any other objects with respect to calling methods inside its class. For instance, this can imply setting or getting the values for various members custom defined in the Thread class, setting other Thread specific properties like a Thread name (check the Java API for more info), etc. Note that calling a method for a Thread instance is executed inside the caller thread, like for any other object, and not in the thread for which it's called.

```
SomeThread someThread = new SomeThread();

someTread.setSomeStuff();  // <-- this is executed in the current thread;
                           //     the someThread object is modified
                           //     but the change in the object state
                           //     is not performed by the thread itself
```

After starting a thread to check if it is still running, the *isAlive()* method can be called. This will return *true* in case the Thread for which it's called did not finish its *run()* method.

```
SomeThread someThread = new SomeThread();
someThread.start();

if (someThread.isAlive()) {
        System.out.println("someThread is not dead yet");
}
```

Pausing a thread for a specific amount of time can be achieved by calling *sleep()* having the duration as argument. This must be called by the thread that is supposed to be paused and cannot be called for it by another thread. Another method for temporarily interrupting a thread's execution is *yield()*. As *sleep()* this also must be called by the thread itself, but the effect is just a switch in the thread priorities, the thread that yields letting other threads that might be waiting for processor time to be executed.
A thread can wait for another thread to finish its *run()* by calling *join()* for its instance

```
SomeThread someImportantThread = new SomeThread();
someImportantThread.start();
someImportantThread.join(); // <-- waiting in the current
                            // thread for someImportantThread to finish
```

Stopping a thread

Certain thread APIs were introduced to facilitate thread suspension, resumption, and termination but were later deprecated because of inherent design weaknesses. For example, the `Thread.stop()` method causes the thread to immediately throw a `ThreadDeath` exception, which usually stops the thread.

Invoking `Thread.stop()` results in the release of all locks a thread has acquired, potentially exposing the objects protected by those locks when those objects are in an inconsistent state. The thread might catch the `ThreadDeath` exception and use a finally block in an attempt to repair the inconsistent object or objects. However, doing so requires careful inspection of all synchronized methods and blocks because a `ThreadDeath` exception can be thrown at any point during the thread's execution. Furthermore, code must be protected from `ThreadDeath` exceptions that might occur while executing catch or finally blocks. Consequently, programs must not invoke `Thread.stop()`.

The most simple and correct way to stop a thread can be by setting a flag that is verified periodically inside the thread's *run()* method, and according to its value the thread stops its execution. (Note that this approach can be used to alter the thread behavior externally also for some other purposes like telling the thread to sleep for example).

```
public class MyThread extends Thread {

   boolean volatile[2] alive = true;

   public void run (){
        while (alive) {
                System.out.println("spending time doing nothing...");
                Thread.sleep(10);

        }
   }
   public void killThread() {
        alive = false;
   }
}

... // in main

MyThread lazyThread = new MyThread();
lazyThread.start();
Thread.sleep(2);         // the main thread sleeps for a bit
lazyThread.killThread(); // <-- at the next while iteration
// inside run after alive is set to false the lazyThread will stop
```

Another way to stop a thread can be calling *interrupt()* for the thread's instance. This call sets an internal flag for the thread that can be checked by the thread itself with the *isInterrupted()* method. In addition, calling *interrupt()* makes also any blocking method like *sleep()* or *join()* to stop their execution and throw an exception.

```
public class MyThread extends Thread {

   public void run (){
        while (!isInterrupted()) {
                System.out.println("spending time doing nothing...");
                Thread.sleep(10);

        }
   }

}

... // in main

MyThread lazyThread = new MyThread();
lazyThread.start();
lazyThread.interrupt(); // <-- if this is executed while the thread is
                        // sleeping an exception will be
                        // thrown; if it is executed at
                        // another moment the internal interrupt flag will
                        // be set and at the next while iteration the
                        // thread will terminate
```

[2] we elaborate on the need to mark `alive` as `volatile` in later sections

Source Examples[3]

- PrinterThread - code for a thread that prints letters from a string
- SumThread - code for a thread that computes the sum of an interval
- Test - code including test cases functions using the above classes

Basic Java synchronization notions

The "synchronized" keyword

In java every object instance has associated an internal lock. This lock can be used by threads to achieve synchronization when trying to access the object using the keyword *synchronized* in two ways.
The first possibility is by declaring methods of the objects class as synchronized. When a thread executes a synchronized method it will get the lock for the object. Any other thread trying to execute any other synchronized method on the same object instance will wait until the current thread will release the lock by finishing its method invocation.

```
public class SharedObject {

    int x = 0;

    public synchronized void inc {
        x++;
    }

}

public class MyThread extends Thread {

   SharedObject shared;

   public void MyThread(SharedObject shared) {
        this.shared = shared
   }

   public void run (){
        int i = 0;
        while (i < 100) {
               shared.inc();   // <---------------------------|
               Thread.sleep(10);              //              |
        }                                     //              |
   }                                          //              |
                                              //              |
}                                             //              |
                                              //              |
...                                           //              |
                                              //              |
SharedObject myShared = new SharedObject();   //              |
MyThread thread1 = new MyThread(myShared);    //              |
MyThread thread2 = new MyThread(myShared);    //              |
```

[3] The source code is on ILIAS

```
                                                //               |
thread1.start();                                // <--When each of
thread2.start();                                // <--these threads
// executes shared.inc(), it attempts to get the myShared's
// internal lock, obtaining it immediately if it is free or waits for it to
// be released
```

The other way of using *synchronized* is by explicitly specifying the object for which the lock is taken and the section of code that is locked. This can be useful either for synchronizing access only for part of some method's code or to use different locks for different portions of code. Using different locks for different portions of code of the same class through *synchronized* is achieved by using locks belonging to other objects than the current one. This permits different groups of threads simultaneously executing different code sections but still being synchronized for each section designated to be so.

```
public class SharedObject {

    int x = 0;
    int y = 0;
    SomeObject lockObject = new SomeObject(); //<-- it doesn't
        //matter what type of class we are using we are interested
        // only in its internal lock

    public synchronized void inc {
        //<-- synchronized with the lock of the current object
                       x++;
    }

    public void dec {
        //<-- not synchronized as a method, threads can start
        // executing dec while another thread is executing inc
        int i = 0;

        synchronized(lockObject) {//<-- one thread reaching here
        // takes the lock for the lockObject if available; if the
        // lock is not available will wait until it is released;
        // because the lock is for another object executing this
        // doesn't block other threads in executing inc() which
        // is accessed based on the current instance lock
        y--;
        }
        while (i < 100000) { //<-- we are taking advantage of
        // this type of synchronized usage to keep this portion
        // of code not synchronized;
        System.out.println("spending long time");
        Thread.sleep(10);
        i++;
        // if it would be one thread will block the others from
        // modifying the shared data y for an unnecessary amount
        // of long time during while y is not accessed
        }
    }
}

public class MyThreadInc extends Thread {

   SharedObject shared;
   public void MyThread(SharedObject shared) {
        this.shared = shared
   }

   public void run (){
        int i = 0;
        while (i < 100) {
                shared.inc();
                Thread.sleep(10);

        }
   }

}

public class MyThreadDec extends Thread {

   SharedObject shared;

   public void MyThread(SharedObject shared) {
        this.shared = shared
   }

   public void run (){
        int i = 0;
        while (i < 100) {
                shared.dec();
                Thread.sleep(10);

        }
   }

}

...

SharedObject myShared = new SharedObject();
MyThreadInc thread1 = new MyThreadInc(myShared);
MyThreadInc thread2 = new MyThreadInc(myShared);
MyThreadDec thread3 = new MyThreadDec(myShared);
MyThreadDec thread4 = new MyThreadDec(myshared);

thread1.start();
thread2.start();
thread3.start();
thread4.start();
```

Keep in mind that using synchronized on methods is really just shorthand (assume class is `SomeClass`):

```
synchronized static void foo() {
    ...
}
```

is the same as

```
static void foo() {
    synchronized(SomeClass.class) {
        ...
    }
}
```

and

```
synchronized void foo() {
    ...
}
```

is the same as

```
void foo() {
    synchronized(this) {
        ...
    }
}
```

You can use any object as the lock. If you want to lock subsets of static methods, you can

```
class SomeClass {
    private static final Object LOCK_1 = new Object() {};
    private static final Object LOCK_2 = new Object() {};
    static void foo() {
        synchronized(LOCK_1) {...}
    }
    static void fee() {
        synchronized(LOCK_1) {...}
    }
    static void fie() {
        synchronized(LOCK_2) {...}
    }
    static void fo() {
        synchronized(LOCK_2) {...}
    }
}
```

(for non-static methods, you would want to make the locks be non-static fields)

Visibility

Visibility is subtle because the things that can go wrong are so counterintuitive (as seen in the previous example). In a single-threaded environment, if you write a value to a variable and later read that variable with no intervening writes, you can expect to get the same value back. This seems only natural. It may be hard to accept at first, but when the reads and writes occur in different threads, *this is simply not the case*. In general, there is *no* guarantee that the reading thread will see a value written by another thread on a timely basis, or even at all. In order to ensure visibility of memory writes across threads, you must use synchronization.

NoVisibility illustrates what can go wrong when threads share data without synchronization. Two threads, the main thread and the reader thread, access the shared variables ready and number. The main thread starts the reader thread and then sets number to 42 and ready to true. The reader thread spins until it sees ready is true, and then prints out number. While it may seem obvious that NoVisibility will print 42, it is in fact possible that it will print zero, or never terminate at all! Because it does not use adequate synchronization, there is no guarantee that the values of ready and number written by the main thread will be visible to the reader thread.

```
public class NoVisibility {
private static boolean ready;
private static int number;
private static class ReaderThread extends Thread {
      public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
}
public static void main(String[] args) {
      new ReaderThread().start();
      number = 42;
      ready = true;
      }
}
```

NoVisibility could loop forever because the value of ready might never become visible to the reader thread. Even more strangely, NoVisibility could print zero because the write to ready might be made visible to the reader thread *before* the write to number, a phenomenon known as *reordering*. There is no guarantee that operations in one thread will be performed in the order given by the program, as long as the reordering is not detectable from within *that* thread—*even if the reordering is apparent to other threads*. When the main thread writes first to number and then to ready without synchronization, the reader thread could see those writes happen in the opposite order—or not at all.

NoVisibility is about as simple as a concurrent program can get—two threads and two shared variables—and yet it is still all too easy to come to the wrong conclusions about what it does or even whether it will terminate. Reasoning about insufficiently synchronized concurrent programs is prohibitively difficult.

This may all sound a little scary, and it should. Fortunately, there's an easy way to avoid these complex issues: *always use the proper synchronization whenever data is shared across threads.*

NoVisibility demonstrated one of the ways that insufficiently synchronized programs can cause surprising results: *stale data*. When the reader thread examines ready, it may see an out-of-date value. Unless synchronization is used *every time a variable is accessed*, it is possible to see a stale value for that variable. Worse, staleness is not all-or-nothing: a thread can see an up-to-date value of one variable but a stale value of another variable that was written first.

When food is stale, it is usually still edible—just less enjoyable. But stale data can be more dangerous. While an out-of-date hit counter in a web application might not be so bad, stale values can cause serious safety or liveness failures. In NoVisibility, stale values could cause it to print the wrong value or prevent the program from terminating. Things can get even more complicated with stale values of object references, such as the link pointers in a linked list implementation. *Stale data can cause serious and confusing failures such as unexpected exceptions, corrupted data structures, inaccurate computations, and infinite loops.*

MutableInteger is not thread-safe because the value field is accessed from both get and set without synchronization. Among other hazards, it is susceptible to stale values: if one thread calls set, other threads calling get may or may not see that update.

We can make MutableInteger thread safe by synchronizing the getter and setter as shown in SynchronizedInteger. Synchronizing only the setter would not be sufficient: threads calling get would still be able to see stale values.

```
// Non-thread-safe mutable integer holder.
@NotThreadSafe
public class MutableInteger {
      private int value;
      public int get() { return value; }
      public void set(int value) { this.value = value; }
}

@ThreadSafe
public class SynchronizedInteger {
      @GuardedBy("this")
      private int value;
      public synchronized int get() { return value; }
      public synchronized void set(int value) { this.value = value; }
}
```

The "volatile" keyword

The *volatile* keyword is used when declaring a variable to set off some compiler optimization that might cause memory inconsistencies. It also provides visibility guarantees, as mentioned in the previous section. The value for a variable that is not declared as volatile might be kept in the thread's local memory, and the thread might not be able to observe the changes made on it by a different thread. By declaring the variable as volatile any read or write operations on it are performed in the main memory having the cost of lower speed.

```
// Stop thread example .. revisited

public class MyThread extends Thread {

 boolean alive = true;
 public void run (){
  while (alive) {                                          //<----|
      System.out.println(" doing nothing...");                //|
            Thread.sleep(10);                                 //|
   }                                                          //|
  }                                                           //|
                                                              //|
   public void killThread() {                                 //|
      alive = false;                                          //|
   }                                                          //|
}                                                             //|
                                                              //|
...                                                           //|
                                                              //|
MyThread lazyThread = new MyThread();                         //|
lazyThread                                .start();    //|
lazyThread.killThread(); // <-- at the next while         <-//|
// iteration inside run after alive is set to false the
// lazyThread will stop
//     ... or not
// the variable is set by the the current thread and
// checked by lazyThread, which might not see
// the change relying on its own memory value

//Solution: add volatile to the alive variable declaration
//this will ensure reading and writing to main memory
//and real value visibility for all threads

boolean volatile alive = true;
```

Credits:

[1] http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/

[2] Java concurrency in practice, Brian Goetz, Tim Peierls, Addison-Wesley, 2006

[3] https://www.securecoding.cert.org/confluence/display/java/THI05-J.+Do+not+use+Thread.stop%28%29+to+terminate+threads

[3] https://stackoverflow.com/questions/578904/how-do-synchronized-static-methods-work-in-java

[4] https://dzone.com/articles/java-volatile-keyword-0