CISC324: Operating Systems

Lab 2
January 5, 2019
This lab has been prepared to be conducted on any operating system that runs the Java Virtual Machine. Please make sure to have a well set up environment e.g., a computer, operating system, Java Virtual Machine, and a Java IDE like Eclipse [https://www.eclipse.org].

**Objective**

At the end of this lab, you will be able to develop multi-threaded programs using Java programming language. You will also learn how different threads belonging to the same main process (which created them) can share data and collaborate within the framework of an application. Below are some definitions as well as Java methods to be used during this lab:

1. **Thread**: A thread is a lightweight process. It is an execution flow of a given program. A program may have multiple execution flows (multi-threaded program). A thread shares with the main process (as well as with other threads from the same process) the code section, global data section, heap section, as well as the resources such as open files. Yet, it has a private stack section and private CPU-register values.

The simplest way to use threads in Java, is to start by defining a Java class that inherits from the class `java.lang.Thread`. In the example below (`MyThread.java`), the new Java class is called `MyThread`. Next, you should redefine the body of the method `run()` to include the code that will be executed by the thread.

```java
class MyThread extends Thread
{
    public void run()
    {
        // code to be executed by any thread
    }
}
```

To create and execute a thread, you must create an instance of the class `MyThread` (see class above) then call the method `start()` as shown below (`ThreadExample.java`):

```java
class ThreadExample
{
    public static void main(String[] args)
    {
        MyThread t_1 = new MyThread();
        t_1.start();
    }
}
```

Following are some useful Java thread method definitions, where `t` is a thread object:

Copyright © Karim Lounis, 2019.
2. `public void run () {...}`: Inside this method you can type the body code that will be executed by each thread belonging to the main process.

3. `start();`: This method allows to start the execution of a given thread (e.g., `t.start();`).

4. `stop();`: This method allows to stop the execution of a given thread (e.g., `t.stop();`).

5. `getName();`: This method returns the name of a thread `t`. If no name is given at the constructor, then a default name will be given to the thread. That name has a suffix `Thread-` followed by an integer that increments each time a thread is created (e.g., `t.getName();`).

6. `int getPriority();`: This method allows to retrieve the priority value of a given thread (e.g., `t.getPriority();`)

7. `int setPriority();`: This method allows to set the priority value for a given thread (e.g., `t.setPriority(5);`)

8. `destroy();`: This method allows to destroy a given thread. It is highly recommended to call this function after stopping the thread (e.g., `t.stop() then t.destroy();`).

Exercise

In IT Security (Information Technology Security), authentication is a security service that allows at least two parties to prove to each other (mutual authentication) or one party proves to another party (one way authentication) that it is what it pretends to be by proving the possession of the correct password, biometric information, or cryptographic keys. In this exercise, we assume that a challenge-response-based authentication protocol is being adopted by two processes `P_c` and `P_s` such that `P_c` proves to `P_s` that it is the real `P_c` (one way authentication) by proving the possession of the correct password. In this protocol, the process `P_s` sends a challenge (a random string `c`) to process `P_c`. Upon the reception of the challenge `c` by process `P_c`, the later concatenates that challenge `c` with a secret password `p` that is shared between `P_s` and `P_c`, to create a response `r = Hash(p||c)`, where `Hash(.)` is a hashing function (i.e., a function that takes as in input a sequence of bytes and outputs a unique value out of that). Then, process `P_c` sends back the computed response `r` to process `P_s`. The process `P_s` performs the same computation by computing `r' = Hash(p||c)` and checks whether `r' = r`. In the later case, process `P_s` authenticates process `P_c` as the real process `P_c` that knows the secret password. Otherwise, if `r' ≠ r`, process `P_s` aborts the communication.

An attacker, represented by a process `P_a`, wants to learn (crack) the password to impersonate process `P_c` later on. It eavesdrops the communication between `P_s` and `P_c` during their authentication, and captures the challenge string `c` that was sent by process `P_s` to process `P_c` as well as the response `r = Hash(p||c)` sent by process `P_c` to `P_s`. Furthermore, the attacker, via another source, learns that the password `p` that was used is a 5-characters word that uses only lower-case alphabetical letters (a, b, c, ..., z) and starts with letter i, t, or v. The attacker also learns that the hashing function that was used, is the function `Hash(.)`. 

---

Copyright © Karim Lounis, 2019.
An attacker, which is implemented as a multi-threaded process (composed of multiple threads), performs a brute force attack (i.e., tries all possible combinations of letters) to crack the password $p$. For that end, and to speed up the cracking, the attacker creates three threads $t_1$, $t_2$ and $t_3$. Each thread has access to the challenge $c$ and the response $r$. The attacker process (main thread) starts all children threads ($t_1$, $t_2$ and $t_3$) together and assigns to each thread a character to work on i.e., $i$, $t$, or $v$. In this way, each thread $t_i$ takes the challenge $c$, fix the first letter to $i$, $t$, or $v$ (depending on the assigned value by the main process), constructs a 5-characters password $p'$, computes the response $r'$, and compare the result $r'$ with the real response $r$. If they are equal (i.e., $r' = r$), then the thread stops its execution and displays the password $p'$. At the same time, the other remaining threads, through a shared variable, which they check each time, get notified that one of the threads has cracked the password. This makes them stop as well. Bellow is a small example for a better understanding:

Assuming that thread $t_i$ has been assigned the letter $w$. Then, during its first try, it creates a password $p' = waaaa$ then concatenates that password with the challenge $c$ (that was captured by the attacker) to generate the response $r' = Hash(p'||c)$, then compares that response with the correct response $r$ (i.e. test whether $r' = r$). If they are equal, thread $t_i$ notifies the other thread $t_{j \neq i}$ through a shared variable that the password has been cracked, then stops. The remaining threads $t_{j \neq i}$ will automatically stop doing unnecessary brute forcing once they get notified. Yet, if $r' \neq r$, then thread $t_i$ continues with the next try $p' = taaab$ and so one and so forth. The other threads $t_{j \neq i}$ will perform the same operations but on a different initial character.

We want to implement this attack using Java threads. The attacker Java source file is given in ThreadAttacker.java and the threads class definition in ThreadBot.java. Let us assume that the correct password is the word virus and the used challenge is challenge-sequence. Let us also assume that the attacker process only knows the challenge challenge-sequence, the response (variable named captured), and the hashing function (x.HashCode()). Even though you can see that the attacker source code contains the password. It is just there to help you create (simulate) the attacker having captured the challenge and the response. On the other side, the threads have access to the challenge variable (challenge-sequence) as well as the response (captured). The first process is assigned the letter $k$, the second $q$, and the third the letter $v$.

1. Complete the thread class definition to show that the attacker can crack the password. Which process cracked the password?

2. In the normal circumstances, one of the threads will be able to crack the password. Explain how the cooperation between the three threads took place.

3. Which solution do you think is faster: the attacker implemented as a multi-threaded process or as a single-threaded process?