AI224: OPERATING SYSTEMS

Lab 5

Description. In the first task of this lab, you will learn how to create Unix **cron jobs** to schedule arbitrary tasks for execution at specific dates and times. In the second task, you will learn how to use Java **threads** to develop multi-threaded programs. In the last task, you will learn how to use **pipes** for inter-process communication.

Task 1. **cron** is a Unix "deamon" program that allows you to schedule and to automate tasks (known as **cron jobs**) that need to be executed periodically to maintain a good working and flexible system. Such tasks include, but are not limited to, deleting old log files, sending out notification emails (e.g., newsletters and password expiration emails), and regular clean-up of cached data. Most Unix-like operating systems provide the **crontab** utility to create and schedule **cron jobs**.

crontab is the file (Located at **/var/spool/cron/crontabs/<username>**) where you define (in a tabular style) what tasks to run and how often to run them. The **crontab** also represents the command to be used to open the **crontab** file. A **cron job** is an entry in the **crontab** file. It is defined in the form:

minute	hour	day	month	week	command
0-59	0-23	1-31	1-12	1-7	script

You can start by checking the status of the **cron** process as it should run in the background (as a deamon): **sudo systemctl status cron**. In the case where the service is not running, you may enable it with: **sudo systemctl enable cron** and then start the service with **sudo systemctl start cron**.

To create a **corn job**, use the **crontab -e** command to edit the crontab file and add the following two lines:

```
* * * * * ps -aux >> ~/processes_log
0 8 * * 1-5 firefox https://www.news.google.com
0 0 * * 1 curl https://www.google.com >> page.html
```

In the case **cron** is not installed, install it with **sudo apt install cron -y**.

Task 2. In this task, you will write your first multi-threaded Java program.

As we have seen in Lecture 4, a `thread` is a lightweight process. It can be seen as 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.

Threads allow you to develop multi-threaded applications to run on multi-core CPUs. This would improve applications' response and execution time, used memory space, and performance.

In Java, the simplest way to use threads 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.

```
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) and then call the method `start()` as shown below (`ThreadExample.java`)

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

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

1. `public void run () {...}`: Inside this method, you can type the body code that will be executed by each thread belonging to the main process. Then, for each thread (identified by an `id` — passed through the customized constructor), you can define a piece of code that would be executed by a specific thread.

2. `start()`: This method allows launching the execution of a specific thread (e.g., starting thread `t` is done through `t.start();`).

3. `stop()`: This method allows stopping the execution of a given thread (e.g., stopping thread `t` is done through `t.stop();`).

4. `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., `Thread-0, Thread-1, ..., Thread-n`).

5. `getPriority()`: This method allows retrieving the priority value of a given thread (e.g., `t.getPriotity();`)

6. `setPriority()`: This method allows setting the priority value for a given thread (e.g., `t.setPriotity(5);`)

7. `join()`: When we invoke the `join()` method on a thread, the calling thread goes into a waiting state. It remains in a waiting state until the referenced thread terminates (e.g., the main thread would wait for the termination of its child thread).

8. `destroy()`: This method allows to destroy a given thread. It is highly recommended to call this method after stopping the thread.

☞ Browse the course website and download the two Java source files, `Tmain.java` and `Tchildren.java` of Lab 5. Skim over and try to understand their content.

☞ Modify these source files so that the main process (thread) creates five (05) threads and orders them to do something (you are free to choose some tasks and distribute them to the threads). The main thread (process) would display a message (`All done!`) once all created threads terminate (you may use `join()`).

Task 3. Pipes (a.k.a., pipeline) are software communication means that processes can locally use to communicate and share information among each other within a Unix-like operating system. It is represented by an in-memory file that requires two file descriptors, one to read another one to write to the file. Hence, the concept is straightforward: when a process writes a certain amount of bytes to the pipe, then another process reads the same sequence of bytes from the other end of the pipe.

If a process tries to read from a pipe in which no other process has written yet in it, the read operation would block until something is written into the pipe. The read operation would then return the number of bytes it has read (+ eof). Also, the write operation returns the number of bytes that were written to the pipe.

In order to use pipes, you must include `unistd.h` header file. To create a pipe, we use the following syntax (where `Ext` is an array of two integers — `int Ext[2];`):

```
pipe (Ext);
```

On success, the `pipe()` function returns `0`, otherwise, it returns `-1`. The `pipe()` function takes an array of two integers as an argument. The first element `Ext[0]` is used for reading from the pipe, whereas the second element `Ext[1]` is used for writing to the pipe. The latter should be closed by the reader process when about to read. Similarly, the writer process would close `Ext[0]` when about to write to the pipe.

If a process wants to write a message `msg` (a string) to the pipe, it calls:

```
write(Ext[1], msg, strlen(msg));
```

If a process wants to write an integer `n` to the pipe, it calls:

```
write(Ext[1], &n, sizeof(n));
```

If it wants to read a message from the pipe, it calls (where `buffer` is an array of characters of size `buffer-size`):

```
read(Ext[0], buffer, buffer-size);
```

For reading an integer:

```
read(Ext[0], &n, sizeof(n));
```

☞ Use POSIX pipes to address the limitation encountered in Exercise 1 of Lab 4.