CSS 430 Operating Systems
Program 1B: ThreadOS Shell

0. Attention

This series of programming assignments, 1B, 2B, and 3B, is a step-by-step implementation of a Java-based OS simulator, named *ThreadOS*. In contrast to 1A-4A in C++, this Java programming intends to work on the logical design of process, scheduling, and memory management in OS. Since 1B-3B will be all coded in Java, you may use any computing infrastructure: Linux, MacOS, or Windows. While other CSS430 sections may use *ThreadOS*, each instructor has a different version. You are not supposed to reuse your previous *ThreadOS* work in the section you couldn't make it; to look at any work of the students; nor to collaborate with your classmates on 1B-3B, any of which is considered plagiarism.

1. Purpose

This assignment is designed to help you understand that, **from the kernel's viewpoint, the *shell* is simply an application program** that uses system calls to spawn and to terminate other user programs. **You will also get familiar with our *ThreadOS* operating system simulator through the assignment**.

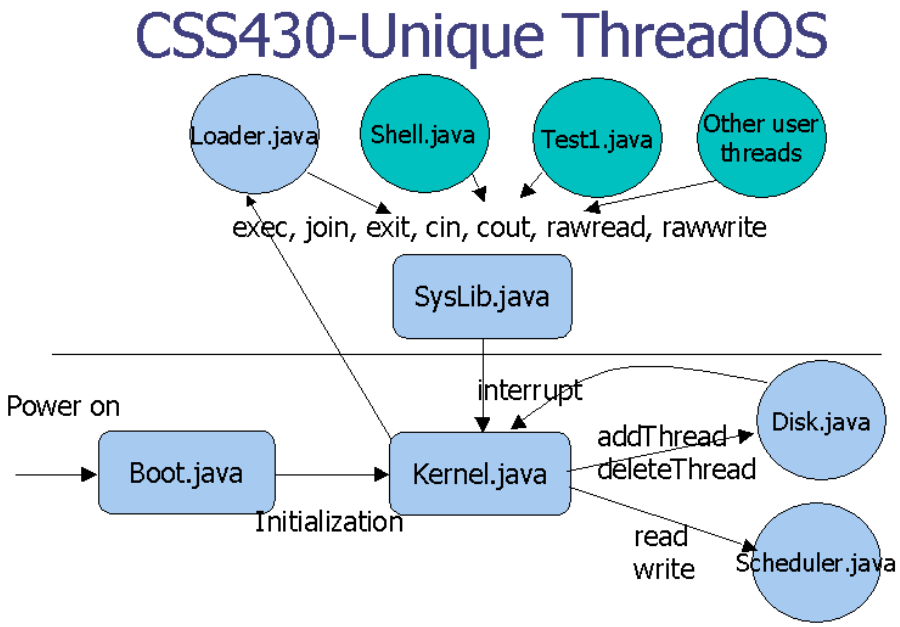
2. ThreadOS

ThreadOS loads into memory a Java program (which has been coded as an extension of the *Thread* class), manages it as a user-level active execution entity (which is the same as a *process* in Unix, a *task* in Linux/Windows, and a *job* in IBM mainframes), and provides it with basic operating system services. Throughout all 1B-3B programming assignments, we simply call this entity a *thread*. **A thread invokes ThreadOS system functions to receive various services:** spawning a child, sleeping the calling thread, terminating the thread, operating onto disk, and even interacting with a user (through standard input/output). **Such a system call is passed to ThreadOS as a form of software trap.** ThreadOS handles each trap and returns a status value to the thread that made this system call.

2.1. Structure

ThreadOS consists of several key components:

Components	Java Classes	Descriptions
Boot	Boot.java	Invokes a *BOOT* system call to have *Kernel* initialize its internal data, power on *Disk*, start the *Scheduler* system thread, and finally spawn the *Loader* system thread.
Kernel	Kernel.java	Receives and services traps from user threads and interrupts from HW (i.e., disk). Some traps are forwarded to *Scheduler* or *Disk* if necessary. A completion status will be returned to each trap.
Disk	Disk.java	Simulates a slow disk device composed of 1000 blocks, each containing 512 bytes. Those blocks are divided into 10 tracks, each of which thus includes 100 blocks. The disk has three commands: *read*, *write*, and *sync* detailed in program 3B-5B.
Scheduler	Scheduler.java TCB.java	Receives a Thread object that *Kernel* instantiated upon receiving an *EXEC* system call, allocates a new *TCB(Thread Control Block)* to this thread, enqueues the *TCB* into its ready queue, and schedules its execution in a round robin fashion.
SysLib	SysLib.java	Is a utility that provides user threads with a convenient style of system calls and converts them into corresponding traps to be passed to *Kernel*.



To start *ThreadOS*, simply type:

```
csslabXXX$ java Boot
ThreadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->
```

Note that the command in ThreadOS is case sensitive. Type "Boot" instead of "boot".

"Boot" initializes *Kernel* data, powers on *Disk*, and starts *Scheduler*. It finally launches *Loader* that then prints out the prompt --> and carries out one of the following commands.

?	Prints out its usage.
l user_program	Starts *user_program* as an independent user thread and waits for its termination.
q	Writes back in-memory file system data into disk and terminates *ThreadOS*.

Note that *Loader* is not a *shell*. It simply launches and waits for the completion of a user program (which is not so intelligent as *shell* that can launch multiple threads in parallel). From *ThreadOS*' point of view, there is no distinction between *Loader* and the other user programs.

2.2. User Program

A user program **must be a Java thread**. Java threads are execution entities concurrently running in a Java application. They maintain their own stacks and program counter but share static variables in their application. At least one thread, (i.e., the *main* thread) is automatically instantiated when an application starts a *main* function. Threads other than the *main* thread can be dynamically created in a similar way to instantiate a new class object using *new*. Once their *start* method is called, they keep executing their own *run* method independently from the calling function such as *main*. Java threads can be defined as a subclass of the *Thread* class. The following Java thread prints out a word given in *args[0]* repeatedly every *loop* number of dummy iterations. The *loop* is given in *args[1]*.

```
public class PingPong extends Thread {
    private String word;
    private int loop;
    public PingPong( String[] args ) {
       word = args[0];
       loop = Integer.parseInt( args[1] );
    }
    public void run( ) {
       while ( true ) {
          System.out.print( word + " " );
          for (int i = 0; i < loop; i++ );
       }
    }
 }
```

If you write the following *main* function,

```
public class ThreadDriver {
    public static void main( String[] args ) {
      String args[2];
      args[0] = "ping"; args[1] = "10000";
      new PingPong( args ).start( );

      args[0] = "PING"; args[1] = "90000";
      new PingPong( args ).start( );
    }
 }
```

it will instantiate two *PingPong* threads, one printing out "ping" every 10000 dummy iterations and the other printing out "PING" every 90000 dummy iterations.

ThreadOS Loader takes care of this thread-instantiating part of the *main* function. Once you invoke *ThreadOS*, *Loader* waits for a *l* command, say *"l PingPong ping 10000"*. Then, it will load your *PingPong* class into the memory, instantiate its object, pass a String array including *ping* and *10000* as arguments to this thread, and wait for its termination. **Note that general Java threads can receive any type of and any number of arguments, however, *ThreadOS* restricts its user programs to receive only a String array as their argument.**

Java itself provides various classes and methods that invoke real OS system calls such as *System.out.println* and *sleep*. Since *ThreadOS* is an operating systems simulator, user programs running on *ThreadOS* are prohibited to use such real OS system calls. Prohibited classes include but are not limited to:

- java.lang.System
- java.lang.Thread
- java.io.*

Instead, user programs are provided with ***ThreadOS*-unique system calls** including standard I/O, disk access, and thread control. **Therefore, *System.out.print(word + " ");* should be replaced with one of *ThreadOS*-unique systems calls:**

```
SysLib.cout( word + " " );
```

While Java threads can be terminated upon a simple return from their *run* method, *ThreadOS* needs an explicit system call to terminate the current user thread:

```
SysLib.exit( );
```

This is because thread termination is a part of thread control, thus one of *ThreadOS* services. Since the above example of user thread falls into an infinitive loop, we need to revise it so that this example code safely terminates the invoked thread and resumes *Loader*.

```
public class PingPong extends Thread {
    private String word;
    private int loop;
    public PingPong( String[] args ) {
       word = args[0];
       loop = Integer.parseInt( args[1] );
    }
    public void run( ) {
       for ( int j = 0; j < 100; j++ )
          SysLib.cout( word + " " );
          for (int i = 0; i < loop; i++ );
       }
       SysLib.cout( "\n" );
       SysLib.exit( );
    }
 }
```

2.3. System Calls

***ThreadOS Kernel* receives requests from each user thread as traps (i.e., software interrupts) to it.** Such an interrupt is performed by calling:

```
Kernel.interrupt( int interruptRequestVector,
                  int trapNumber,
                  int parameter,
                  Object args );
```

where *interruptRequestVector* may be 1: INTERRUPT_SOFTWARE, 2: INTERRUPT_DISK, and 3: INTERRUPT_IO; *trapNumber* specifies a request type of software interrupt such as 0: BOOT, 1: EXEC, 2: WAIT, 3: KILL, etc.; *parameter* is a device-specific value to control each device; and *args* are arguments of each interrupt request.

Since this interrupt method is not an elegant form to a user program (in other words, it's too deep into the Kernel), *ThreadOS* provides a user program with its system library, called *SysLib* that includes several important system-call functions as shown below. (Unless otherwise mentioned, each of these functions returns 0 on success or -1 on error.)

1. **SysLib.exec(String args[])** loads the class specified in args[0], instantiates its object, simply passes the following elements of String array, (i.e., args[1], args[2], ...), and starts it as a child thread. It returns a child thread ID on success, otherwise -1.
2. **SysLib.join( )** waits for the termination of one of child threads. It returns the ID of the child thread that has woken up the calling thread. If it fails, it returns -1.
3. **SysLib.sleep(long millis)** sleeps the calling thread for given milliseconds.

4. **SysLib.exit( )** terminates the calling thread and wakes up its parent thread if this parent is waiting on join( ).
5. **SysLib.cin(StringBuffer s)** reads keyboard input to the StringBuffer s.
6. **SysLib.cout(String s)** prints out the String s to the standard output. Like C's *printf*, it recognizes '\n' as a new-line character.
7. **SysLib.cerr(String s)** prints out the String s to the standard error. Like C's *printf*, it recognizes '\n' as a new-line character.
8. **SysLib.rawread(int blkNumber, byte[] b)** reads one block data to the byte array b from the block specified by blkNumber.
9. **SysLib.rawwrite(int blkNumber, byte[] b)** writes one block data from the byte array b to the block specified by blkNumber.
10. **SysLib.sync( )** writes back all on-memory data into a disk.

In addition to those system calls, the system library includes several utility functions. One of them is:

1. **public static String[] SysLib.stringToArgs(String s)** converts a space-delimited string into a String array in that each space-delimited word is stored into a different array element. This call returns such a String array.

2.4. Other Components

Those components include *Scheduler* and *Disk*. They will be explained in detail as you have to hack them in assignments 2B - 3B. What you need to know for 1B is only how to get started and finished with *ThreadOS*, all of which have been introduced above.

3. Statement of Work

Important: You can find "Shell_hw1b.java" from the Canvas File folder: **Files/code/prog1/Shell_hw1b.java. This is the template file you need to work on.**

Complete the implementation of *Shell.java*, a Java thread that will be invoked from *ThreadOS Loader* and behave as a shell command interpreter as follows.

```
csslab1h$ java Boot
ThreadOS ver 1.0:
Type ? for help
-->l Shell
l Shell
threadOS: a new thread (thread=Thread[Thread-6,2,main] tid=1 pid=0)
shell[1]% TestProg1 & TestProg2 &
    ... A concurrent execution of TestProg1 and TestProg2
    ...
shell[2]% TestProg1 ; TestProg2
    ... A sequential execution of TestProg1 and TestProg2
    ...
shell[3]% exit
-->q
csslab1h$
```

Once your *Shell.java* is invoked, it should print out a command prompt:

```
shell[1]%
```

In the above example, "TestProg1" and "TestProg2" are your own test programs, they are not provided. See below about how to use the provided PingPong test class to test your Shell.java implementation.

When a user types in multiple commands, each delimited by '&' or ';', your Shell.java executes each of them as an independent child thread with a SysLib.exec( ) system call. Note that the **symbol '&' means a concurrent execution, while the symbol ';' means a sequential execution**. Thus, when encountering a delimiter ';', your Shell.java needs to call SysLib.join( ) system call(s) to wait for **this** child thread to be terminated. Since SysLib.join( ) may return the ID of any child thread that has been previously terminated, you must repeat calling SysLib.join( ) until it returns the exact ID of the child thread that you want to wait for.

You do not need to implement standard I/O redirection or pipes. You do not need to provide shell variables nor programming constructs, either. **The only required functionality of your Shell.java is handling an arbitrary number of commands in one line.** You may assume that commands, arguments, and even delimiters are separated by arbitrary amounts of spaces or tabs.

To test your Shell.java, use *PingPong.class* that is found in the same directory as *ThreadOS*. Your test should be

```
Shell[2] PingPong abc 100 & PingPong xyz 100 & PingPong 123 100 &
Shell[1] PingPong abc 100 ; PingPong xyz 100 ; PingPong 123 100 ;
```

Important: ThreadOS class files can be found in Canvas Files: Files/code/ThreadOS

- Copy all compiled ".class" files into your working directory and thereafter compile your implementation of Shell.java (you need to rename the provided Shell_hw1b.java into Shell.java at first):
  javac Shell.java

- Do not try to compile the ThreadOS source code, some portions of which cannot be accessed. Needed ThreadOS source code will be released gradually. In this assignment, you only need to compile your implementation of Shell.java.

Hints: In order to read a command line, you should use SysLib.cin(StringBuffer s) that returns a line of keyboard input to the StringBuffer s. Parsing and splitting the line into words can be performed with the *SysLib.stringToArgs(String s)* utility function.

4. What to Turn in

Total 20pts.

	Materials	Points	Note
1	Shell.java a. Code organization: **+2pts** • Well organized: 2pts, • Poor comments or bad organization: 1pt, or • No comments and horrible code: 0pts b. Correctness: **+13pts** • Correct implementation: 13pts, • Minor bugs (still runnable): 12pts, • Major bugs (crashed): 11pts, • Incomplete (not even runnable): 10pts, or • No code: 0pts	15	Submit this java file individually
2	Execution snapshots a. Tests for & were successfully done: **+2pts,** b. Tests for ; were successfully done: **+ 2pts, and** c. Each command can accept an arbitrary number of arguments: **+ 1pt**	5	Include the snapshots in a pdf file named: FirstNameLastName_prog1B.pdf. Submit this pdf file individually. Please clearly label each snapshot.