CSc 360 Assignment #1

Due date: 11:59pm, Wednesday, 22 May 2013

Overview

In this assignment, we will be introducing The **R**eal Time Java Based **O**perating **S**ystem **E**mulator (ROSE). Download a copy of "assign1.org.tgz", which includes everything you need for this assignment. Follow these instructions on how to install Cygwin for ROSE.

Objectives

You will learn the following issues:

1. Become familiar with ROSE and its development toolchain (GCC, javac, Cygwin, Makefile and our ClassLinker) and Doxygen .
2. Understand race conditions using java threads with preemptive scheduling.
3. Understand how the real-time clock and Java native methods are implemented in ROSE.
4. Understand how our Java threads are scheduled (i.e. our scheduling policy and how the scheduler API works).
5. Add functionality to the Thread class into the "uvic" package library and a new *native* method into ROSE (i.e. create "stubs" of the ROSE JVM).

Questions

There are four parts to this assignment. Each is independent of the other. They are increasingly more difficult. You should try them in the order listed below. Parts (a) and (b) will be done in Java. Part (c) and (d) will be done in C.

(**Important note**: "System.out.println" in Java is renamed to "System.println" in ROSE, actually in SimpleRTJ. ROSE doesn't implement the standard Java library; SimpleRTJ provides it.)

To do this assignment, you must study the design of ROSE. Make sure that you understand the scheduling levels as specified in our "uvic.posix.Thread" class under "lib-source/" directory.

a) Concurrent sharing of a single resource [10%]

Consider the problem of three concurrent processes (threads) incrementing a global shared integer variable. Each process increments the shared variable by 1, 2, and 3 respectively for 10,000 times. Initially, the shared variable has a value of 0. Its value is printed after all three processes have completed. We should expect the final result to be 60,000. *Surprise!*

Question: Sometimes, the results are not 60,000. Why?

In the code, go to folder **a** and work in *Race.java*. For this part, the main() program is created as a SYSTEM level thread in ROSE by default, which has the highest priority. It will create three instances of a *Race* thread, each of which is asked to increment this global shared integer variable. We'll create each *Race* thread at the REAL_TIME level. REAL_TIME level threads are scheduled in a round-robin fashion with a quantum size of 1 tick. A tick is specified at the command line to ROSE. For example,

```
../soln/rose.exe 3 Race.bin
```

specifies each tick to be 3 milliseconds.

Try setting the main thread's level to DAEMON before starting each thread:

```
    // create race threads here
    ...
    set( DAEMON ); // now, let the race threads run

    // Run race threads at REAL_TIME level
```

Question: The result turns out as 60,000. Why?

b) Concurrent sharing of multiple resources [10%]

On a symmetric mulitprocessor system, several CPUs share a common memory. Many concurrent processes may issue access to multiple shared resources simultaneously. To prevent simultaneous access to a shared resource, we often use a "spinlock" to control concurrent access. That is, a "lock" is associated with each shared resource, which is unlocked/free initially. Before a process is allowed to make access to this shared resource, it must lock it first by busy-waiting,

```
while (locked) do nothing; // wait until lock is free
locked = true;             // now, lock it
```

Let lock l1 be assocated with shared resource r1, and lock l2 be associated with shared resource r2. When a process wants to use r1 and r2, it must lock l1 and l2 first. When it is done, it must unlock l1 and l2.

When there are mulitple applications/processes which may share the same set of shared resources, they must lock all of them in some application dependent order.

For example, let us assume that we have two globally shared integers s1 and s2. There are two "race" threads (similar to Part a) increment s1 and s2 by 1, and by 2 respectively 10,000 times. There are two "spinlocks" l1 and l2 assocated with s1 and s2.

If they both lock l1 and l2 in the same order, then the final accumulated results in s1 and s2 should be always 30,000 each.

What if one of the race threads, for some reason unknown to the other, decided to lock s2 first and then s1? *Surprise!*

For this part, start editing the *LockRace* thread class in folder **b**. You should not need to modify *SpinLock*. Your main program will create two instances of *LockRace*, two Spinlocks and two shared integers. When you run this program, the final answer should always be 30,000.

Now, create a subclass *LockRace1* of *LockRace*. Redefine the run() method in class *LockRace1* so that it locks "l2" before "l1". In your new main program, now create an instance of *LockRace* and *LockRace1*. Try running this program several times (5 or 6 times).

Question: What do you discover? Could you explain why?

c) Pre-emptive and time-slice scheduling [30%]

We want our ROSE Java VM to support four scheduling levels (ordered by priority):

1. **SYSTEM**
    - Quantum: Infinite (or maximum possible number)
    - Strategy: FCFS
2. **REAL_TIME**
    - Quantum: 1
    - Strategy: Round-Robin
3. **USER**
    - Quantum: 2
    - Strategy: Round-Robin
4. **DAEMON**
    - Quantum: Infinite (or maximum possible number)
    - Strategy: FCFS

In the last part of this assignment, you are asked to implement this scheduling policy in our JVM. It is pre-emptive. That is, whenever a higher priority thread becomes ready (or runnable), it will pre-empt the current lower priority active thread immediately. There is no automatic promotion or demotion. But, a thread may request to change its running level during its life time by called "set()".

For this part, you need to study the ROSE JVM source code, which is written in C. The only part that you are concerned about is the "j_thread.c", which is the runtime multithreading support of this JVM. For the most part, you don't need to know. The only part that you need to study is the scheduler.

Our thread scheduler is defined by the following API:

```
InitROSE(); // initialize various internal data structures
Dispatch(); // select a new "active" thread to run
AddReady(); // insert a thread into the ready_q
PreemptIfNecessary(); // check for preemption condition
Reschedule(); // "active" gives up its share of CPU; select another
              // thread to run
AssignQuantum(); // assign a fresh quantum
VMTick();  // this is called by JVM whenever there is a "tick"
```

and two very important variables:

```
threat_t  *ready_q;  // a queue of READY threads
thread_t  *thr_active;  // a ptr to the current active thread
```

Your task is to modify these procedures to implement our scheduling policy. This aspect of the assignment is the most difficult and will likely take you the most amount of time. Dispatch() and AddReady() are partially done, but you may need to make some changes. InitROSE() is done for you; so you don't need to make any changes.

First and foremost, you must decide how you are going to represent your ready_q. That is, what data structure you are going to use? Then, you must understand what these API operations do as far as the process/ thread state transition diagram is concerned.

The structure "thread_t" is defined in "jvm.h" by:

typedef struct thread_T thread_t;

```
struct thread_T {
    uint16    state;   // current state of this thread, which may be
                           // DEAD, READY and RUNNING (defined in jvm.h)
    int32     ticks;   // remaining number of ticks in this quauntum
    int32     pri;     // priority level of this thread
    struct thread_t  *next;
}
```

The priority level are defined in "jvm_cfg.h", which are from 0 to 3.

d) Thread CPU time and Java native extension [20%]

In this part of the assignment we are going to add functionality from ROSE to our Java threads using *native* extension. This will require coding in both Java and C. Your test solution will be in folder **/d**, but most of the work will involve modifying **/rose** and **lib-source/uvic/posix/Thread.java**.

This part will involve you modifying multiple files concurrently. But it might be helpful to examine things in this order.

First, edit *lib-source/uvic/posix/Thread.java* and add a new native function:

```
public stati
```