CSE 120 Discussion

Week 3

Edward Wong

Testing

You can define the static function selfTest() which just executes your own made tests. You can separate out the test cases with other static functions.

```
public static void selfTest() {
    testCase1();
    testCase2();
    testCase3(); // etc
}
```

Then you can just put ClassName.selfTest() in ThreadedKernel.selfTest or something like that.

Project 1 - Threads

Implementing Join. What is join though?

- Lets assume there are 2 threads.
- Let’s call them Thread A and Thread B.
- What happens if Thread A executes the line “B.join()”?
- It means Thread A will wait until Thread B has finished.
- So how do we implement that?

High Level Idea

1. If Thread A calls “B.join()”, we want a way to BLOCK thread A until thread B has finished.
2. Once Thread B finishes, we want to unblock Thread A

Thats it!

Pieces of the Puzzle

Question 1: How do we block a thread?

Answer 1: Let’s take a look at KThread.sleep()

Question 2: How do we unblock a process?

Answer 2: Let’s take a look at ready()

Question 3: How do I know when a process is finished?

Answer 3: KThread.finish()

Pieces of the Puzzle

Question 4: If A calls B.join. What is the currentThread? What is ‘this’?

Answer: ‘this’ = B, currentThread = A

Question 5: What if A calls B.join and B is a thread that has already finished?

Some Other Details

Whats the deal with “boolean intStatus = Machine.interrupt().disable();“ and “Machine.interrupt().restore(intStatus);“? Let’s take a look at sleep()

Also, keep in mind that for any given Thread X. “X.join()” can be executed at most ONE time.

We have to associate a thread A with a thread B where A executed “B.join”, so that when B finishes, we can unblock the thread A. How do we do that?

Project 1 - Threads

Implementing Alarm

- What does it mean for a thread A to call alarm.waitUntil(X)?
  - Thread A must be blocked until at least X ticks has passed.
- There can only be ONE instance of Alarm at any given time
- With the code given, the code for Alarm is busy waiting.
  - We want to have the same functionality without busy-waiting
  - Why is busy-waiting probably not ideal?

Alarm - waitUntil

```
    * @param x the minimum number of clock ticks to wait.
    *
    * @see nachos.machine.Timer#getTime()
    */
   public void waitUntil(long x) {
```

- If the current time is Y and the current running thread is A, then we want A to be blocked until **at least** time Y + x.

Alarm - timerInterrupt

```
public void timerInterrupt() {
```

- Add logic here to check to see all the blocked threads that have called alarm.waitUntil(X) and see if the time for them to be unblocked has passed.
- Let's say thread A has called waitUntil(X) at time Y, then we unblock thread A if the current time is greater than X + Y

Alarm - Details

1. If we're going to block a thread, we need to be able to associate that blocked thread to a wake up time.
    a. In waitUntil() we calculate the wake up time and store that information
    b. In TimerInterrupt(), we use the information that was stored from waitUntil(). We check to see if the current time is past that wake up time. If the current time is past the wake up time, then we unblock that read.
2. Which of these data structures work for storing the information needed?
    a. HashMap
    b. List
    c. Set

What is a Condition variable? How is it used?

```
while (the festival is closed()){
    // do nothing
}
goToFestival()

// vs

if (festival is closed()) {
    cond.sleep()
    // some other thread/process will call
    // cond.wakeAll() when festival is open
}
goToFestival()
```

Project 1 - Implementing Condition2

The goal is to implement a Condition variable using enabling/disabling interrupts.

There is an example implementation of a Condition variable using Semaphores instead. You can use this as a reference. Let's go over that real quick.

In Condition2.java, we should not be using a Semaphore but provide the exact same functionality as Condition.java. We will enable/disable interrupts instead of using a Semaphore.

Condition2 - Details

- Sleep
  - Puts the current thread to sleep and remember that thread.
    - Release the lock before you block the current thread
  - Acquire the lock again after it’s awoken/unblocked
- Wake
  - Wake up/ready() one of the threads you blocked in sleep
- WakeAll
  - Wake up/ready() all of the threads you blocked in sleep

Consider what resources are under contention!

Project 1 - Game Match

- numPlayersInMatch required before a game can be “played”
  - Threads that call play will be blocked until numPlayersInMatch have called “played” for the same ability.
  - Return the match ID (Starts a 1 and increases by 1 each time a match is successfully created)
- 3 different abilities (abilityBeginner, abilityIntermediate, abilityExpert)
  - Logic should be the symmetric for all of them.
- This is an example of a synchronization problem. You use synchronization primitives (locks/condition variables) to implement this.

Demo - SimpleGameMatch

1. Let’s try to do something similar but simpler.
2. There is only 1 ability.
3. We don’t have to return the matchID

Things to consider

1. For my instance/class variables. When would I use static or non-static?
    a. What happens when I have multiple GameMatch objects?