CSE 120 - Discussion session 3

April 22, 2019

Logistics

- Project 1 - April 24, 11:59 pm
- Homework 2 - April 27, 11:59 pm
- Homework 1 - Grades are up on Gradesource and Gradescope

Homework 1 - Solutions

Question 3

Which of the following instructions should be privileged? Give a one-sentence explanation for why.

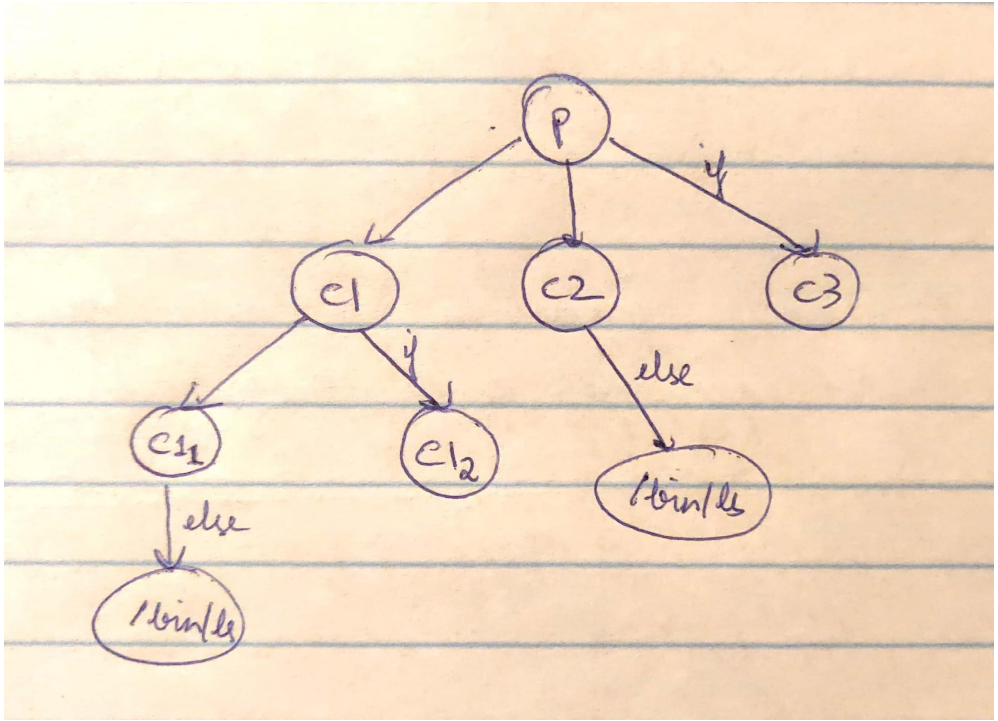
- Set value of timer - Privileged
- Read the clock - Not privileged
- Clear memory - Privileged
- Turn off interrupts - Privileged
- Switch from user to monitor (kernel) mode - Privileged

Question 8

```
#include <stdlib.h>

int main (int argc, char *arg[])
{
    fork ();
    if (fork ()) {
        fork ();
    } else {
        char *argv[2] = {"/bin/ls", NULL};
        execv (argv[0], argv);
        fork ();
    }
}
```

1. How many total processes are created?
2. How many times does the /bin/ls program execute?



Thus, we have 6 processes and /bin/ls is executed 2 times.

Project 1

Problems

1. Alarm - waitUntil() method
2. KThread - join() method
3. **Condition variables**
4. **Scheduled wait on Condition Variables**
5. **squadMatch**

Problem 3 - Condition Variables

3. Implement condition variables using interrupt enable and disable to provide atomicity. The class `Condition` is a sample implementation that uses semaphores, and your job is to provide an equivalent implementation in class `Condition2` by manipulating interrupts instead of using semaphores.

Implementation of Condition Variables

Two ways to implement Condition Variables:

1. Semaphores - Condition class
2. Interrupts - Condition2 class

Methods to implement:

1. sleep() - Wait() operation of Condition Variables
2. wake() - Signal() operation of Condition Variables
3. wakeAll() - Broadcast() operation of Condition Variables

Using Semaphores - Condition class

```
public Condition(Lock conditionLock) {
    this.conditionLock = conditionLock;

    waitQueue = new LinkedList<Semaphore>();
}
```

1. sleep()

```
public void sleep() {
    Lib.assertTrue(conditionLock.isHeldByCurrentThread());

    Semaphore waiter = new Semaphore(0);
    waitQueue.add(waiter);

    conditionLock.release();
    waiter.P();
    conditionLock.acquire();
}
```

2. wake()

```
public void wake() {
    Lib.assertTrue(conditionLock.isHeldByCurrentThread());

    if (!waitQueue.isEmpty())
        ((Semaphore) waitQueue.removeFirst()).V();
}
```

3. wakeAll()

```
public void wakeAll() {
    Lib.assertTrue(conditionLock.isHeldByCurrentThread());

    while (!waitQueue.isEmpty())
        wake();
}
```

Using Interrupts - Condition2 class

- What is the idea behind disabling and enabling interrupts?
- How can this be an alternative to using semaphores?
- Where do we need to disable/enable interrupts?
- Which instructions need to be executed atomically?

General idea to implement sleep() using interrupts:

- Disable interrupts
- Make thread wait on Condition Variable
- Put the thread to sleep
- Enable interrupts

Figure out where to release and acquire the lock!

Problem 4 - Scheduled wait on Condition Variables

4. Add support for a "scheduled wait" operation where threads can wait on a condition variable with a timeout. Implement the `sleepFor` method of `Condition2` and the `cancel` method of `Alarm` to provide this functionality (modifying other methods as necessary). With `sleepFor(x)`, a thread is woken up and returns either because another thread has called `wake` as with `sleep`, or the timeout `x` has expired.

Hints

- Does the sleepFor() functionality seem similar to something?
- How can we use waitUntil()?
- What happens once we call waitUntil()?
  - timeout occurs
  - Another thread wakes this thread up
- What if this thread is woken up before its timeout?
  - We need to ensure that this thread is removed from the wait queue of the Alarm class
- What do we do within the cancel() method?

Problem 5 - SquadMatch

- We have 3 types of players
  - Warrior
  - Wizard
  - Thief
- A squad can be formed only when we have one player of each type
- Goal - given a number of player threads, synchronize them to form squads correctly

Hints

- We can use locks and condition variables to solve the SquadMatch problem
- But, how?
- To figure this out, let us see what it is that we need to do with the help of an example

Consider the following scenario:

- A warrior thread is created first - call this w1
- What should happen?
- Since there are no wizard or thief threads, w1 shouldn’t proceed

- Say another warrior thread is created - w2
- Again, since there are no wizard or thief threads, w2 shouldn’t proceed

- Now, let a wizard thread be created - call this z1
- What should happen?
- Again, since there are only 2 warrior threads and no thief threads, z1 shouldn’t proceed

- Next, let a thief thread be created - call this t1
- What should happen?
- This thief thread can check if there exists a warrior and a wizard thread
- Since warrior and wizard threads do exist at this point, we can form a squad (i.e., release w1, z1)
- Note:
    - We could also release w2 rather than w1. Any of the existing w threads can be matched.
    - We could also release z1 followed by w1/w2

General clarifications

- Threads in each squad are put to the ready queue, but we can’t guarantee that they will execute in any particular order.
- In the above example, say no thief thread is ever created. The expected behaviour is that the wizard and warrior threads will remain blocked.