Operating System: Design for Project 1

Instructed by *Wei Xu*

Due on October 29, 2019

Wei Zhang, Tian Ye and Yifan Tang YaoClass 70 2017011204 2017012514 2017012508

Task I & Task II & Task III

Answer: We built join2 and 3 to join join1 in KThread.java. Run "KThread.selfTest();" in the selftest function of ThreadedKernel.java will run our testing method for Task I. As for Task II and III, we implemented Runnable into test classes in Condition2.java and Alarm.java. The method to call them is to add a line:

Condition2.selfTest();

to ThreadKernel.java. We can tune the parameter in Condition2.java, currently we have a reader reading 2 chars forking first. Then, separated by some time, we let a writer write 2 symbols. We check whether the reader sleeps each time it reads all the remaining chars.

Task IV

Answer:

We designed two functions **speak** and **listen** in class **Communicator**, to implement one word communication. Please see test1 and test2 function at the end of Communicator.java for our experiments.

In the first experiment, speaker 1 first is ready to speak 123 and then wait for listener. Then listener 2 starts to listen and receives speaker 1's message. Later, listener 1 is ready to listen, and receives speaker 2's message when the latter one speaks 456. This suggests that speaker and listener will wait for each other.

The second experiment runs 3 speakers and 2 listeners, to test if they heard correct information when the threads join each other. The output shows our algorithm works appropriately.

Task V

Answer:

As for priority scheduling, we follow two principles: always choose a thread of the highest effective priority to dequeue and if multiple threads with the highest priority are waiting, choose the one that has been waiting in the queue for the longest time. The key point in this task is to implement *priority inversion* when necessary.

For the `PriorityQueue` class, we add methods `waitForAccess(KThread)`, `acquire(KThread)`, `nextThread()`, `updateThreadPriority(ThreadState)`, `getThreadPriority()`, `pickNextThread()` and `print()`. Their meaning and our design are shown below:

- `waitForAccess(KThread)` For an incoming thread, this method schedules it by checking whether it needs to donate its priority to the thread holding the queue and then assigning it to the waiting queue. The code is

Listing 1: Pseudo code for `waitForAccess`

```
public void waitForAccess(KThread thread) {
        Lib.assertTrue(Machine.interrupt().disabled());
        ThreadState ts = getThreadState(thread);
        ts.waitForAccess(this);
        if ((this.holder != null) &&
        (ts.effectivePriority > this.getThreadPriority())){
            waitingThreads.add(ts);
            if (transferPriority)
                holder.calcEffectivePriority();
        }else{
            waitingThreads.add(ts);
        }

        if (verbose) print();
    }
```

`getThreadPriority()` is a procedure to get the maximum of effective priority among all threads in `waitingThreads`.

- `acquire(KThread)` It acquires a thread in the queue. The pseudo code is

Listing 2: Code for `acquire`

```
public void acquire(KThread thread) {
       getThreadState(thread).acquire(this);
        this.holder = getThreadState(thread);
    }
```

- `nextThread()` It is used to pick the next thread in waiting with the largest effective priority or the earlier one when multiple threads' effective priority coincide. The pueudo code is

Listing 3: Pseudo code for `nextThread`

```
public KThread nextThread(){
    nextT = ThreadState in waitingThreads with largest effectivePriority,
    pick the first one when tied
    if (holder != null){
        this.holder.relinquish(this)
    }
    if (nextT == null){
        this.holder = null
        return null
    }
    nextT.acquire(this)
    this.holder = nextT
    return nextT.thread
}
```

- `updateThreadPriority(KThread)` Only recalculate a thread's effective priority when it has the possibility to change, by checking whether it held the lock and has transferred priority to and calling `calcEffectivePriority`. The pseudo code is

Listing 4: Pseudo code for `nextThread`

```java
public void updateThreadPriority(ThreadState){
    if (transferPriority && this.holder != null){
        this.holder.calcEffectivePriority();
    }
}
```

- `getThreadPriority()` As mentioned before, it is used to get the maximum of effective priority among all waiting threads.
- `pickNextThread()` It is used to pick the first thread with the highest priority to run. The code is

Listing 5: Code for `pickNextThread`

```java
protected ThreadState pickNextThread() {
    int maxPriority = 0;
    ThreadState argmax = null;
    for (ThreadState ts : waitingThreads){
        int tempPrior;
        if (transferPriority){
            tempPrior = ts.getEffectivePriority();
        }else{
            tempPrior = ts.getPriority();
        }
        if (tempPrior > maxPriority){
            maxPriority = tempPrior;
            argmax = ts;
        }
    }
    return argmax;
}
```

- `print()` It is used to print outcomes of testcases.

For the `ThreadState` class, we modify and add methods `calcEffectivePriority()`, `getEffectivePriority()`, `setPriority(int priority)`, `waitForAccess(PriorityQueue)`, `acquire(PriorityQueue)` and `relinquish(PriorityQueue)`. The noticeable method is `calcEffectivePriority()`:

- `calcEffectivePriority()` The effective priority of a thread is calculated from the maximum of those waiting in `acuiredResource` queue. Then we check the necessity to implement priority donation: if another thread in the `waitingResource` has a higher effective priority, then we update the the thread's effective priority to that value. The code is

Listing 6: Code for `calcEffectivePriority`

```java
public int calcEffectivePriority(){
    int maxPriority = priority;
    for (PriorityQueue pq : acquiredResource){
```

Task V continued on next page...

```
            int tempPrior = pq.getThreadPriority();
            if (tempPrior > maxPriority){
                maxPriority = tempPrior;
            }
        }
        if ((maxPriority != this.effectivePriority) &&
        (this.waitingResource != null)){
            this.effectivePriority = maxPriority;
            this.waitingResource.updateThreadPriority(this);
        }else{
            this.effectivePriority = maxPriority;
        }
        return maxPriority;
    }
```

Test case:

The test cases of this task consist of two tests. `test1` checks whether the two principles are followed in scheduling. `test2` checks whether priority donation works in the implementation. They are located from line 475 in `PriorityScheduler.java`.