TDT4186 Operating Systems

Exercise 3

Stian Hvatum (hvatum)
MTDT

24. januar 2012

Innhold

1 Description of Code

1.1 CPU

The CPU-class is based on the Memory class, using an internal queue for queueing processes beeing processed. I have passed along the cpuQueue from the simulator, the max amount of CPU time pr. round, the statistics-object, the event queue and the GUI. The most important method in the CPU-class is the *work*-method, wich evaluates the processes and drives the Round-Robin-algorithm.

```
public void work(long clock) {
  if (currentProcess != null) {
    long timeNeeded = currentProcess.getCpuNeeded();
    long nextIO = currentProcess.getCPUTimeToNextIO();
    if (timeNeeded > maxCpuTime && nextIO > maxCpuTime) {
      currentProcess.giveCpuTime(maxCpuTime);
      eventQueue.insertEvent(new Event(Simulator.SWITCH_PROCESS, clock↩
          + maxCpuTime));
      statistics.cpuTimeSpent += maxCpuTime;
      statistics.processShifts++; } else if (timeNeeded > nextIO && ↩
          nextIO <= maxCpuTime) {
      eventQueue.insertEvent(new Event(Simulator.IO_REQUEST, clock + ↩
          nextIO));
      eventQueue.insertEvent(new Event(Simulator.SWITCH_PROCESS, clock↩
          + nextIO + 1));
      statistics.cpuTimeSpent += nextIO;
    } else if (timeNeeded <= maxCpuTime) {
      currentProcess.giveCpuTime(timeNeeded);
      eventQueue.insertEvent(new Event(Simulator.END_PROCESS, clock + ↩
          timeNeeded));
      eventQueue.insertEvent(new Event(Simulator.SWITCH_PROCESS, clock↩
          + timeNeeded + 1));
      statistics.cpuTimeSpent += timeNeeded;
    }
  }
}
```

This method takes the current process, if any, and check if it can be finished within this timeshare. If cannot, and does not have to do I/O within this timeshare, it will be processed for as long as the algorithm allows, and then we force a process change. Else if there is an I/O-request within this timeshare, then we dispatch a request for the I/O-operation, wich goes back to the Simulator. Else, the time needed to finish the process is less than current timeshare, and we can finish the process, and dispatch a "end_process_event".

The *switchProcess*-method checks if the current process exists, if so, we queue it, and we take the next process from the queue and puts it as the active process. We also update the GUI accordingly.

```
public void switchProcess() {
 if (cpuQueue.getQueueLength() > 0) {
    if (currentProcess != null) {
      cpuQueue.insert(currentProcess);
    }
  currentProcess = (Process) cpuQueue.removeNext();
  gui.setCpuActive(currentProcess);
  }
}
```

The last important method in the CPU is the *endCurrentProcess*-method. This method removes the current process from the CPU, if the process is finished,

and returns it to the caller (most likely the Simulator).

```
public Process endCurrentProcess() {
  Process endedProcess = null;
  if (currentProcess != null && currentProcess.getCpuNeeded() <= 0) {
    endedProcess = currentProcess;
    currentProcess = null;
  } else {
    System.err.println("BUG!");
  }
    gui.setCpuActive(null);
    return endedProcess;
  }
```

The *fetchCurrentProcess*-method does the exact same thing, but does not check if the process is finished. I use this method to extract the processes that request I/O.

1.2 IO

The IO-class is the class representing the I/O-device. This class is rather simple, compared to the CPU. This class implements a FIFO-queue, a method for "doing" I/O, and a method for releasing the completed processes.

We insert a process to the queue like this, if there are no processes in the queue from before, we start doing I/O.

```
    public void insertProcess(Process p, long clock) {
        ioQueue.insert(p);
        p.addedToIoQueue();
        statistics.insertsIOQueue++;
        if (currentProcess == null) {
            gui.setIoActive(p);
            currentProcess = (Process) ioQueue.removeNext();
            doIo(clock);
        }
    }
```

We do I/O like this, mostly we use time and update statistics. We also dispatch a *END_IO*-event when the I/O-request is done.

```
    public void doIo(long clock) {
        if (currentProcess != null) {
            long ioTimeNeeded = currentProcess.doIO();
            eventQueue.insertEvent(new Event(Simulator.END_IO, clock
                    + ioTimeNeeded));
            statistics.ioOps++;
            statistics.ioTimeSpent += ioTimeNeeded;
        }
    }
```

We extract the completed processes like this. This method is called from the Simulator when a *END_IO*-event is processed. If there are more processes in the I/O-queue, we start processing the next one.

```
    public Process getCompletedProcess() {
        Process completed = currentProcess;
        if (ioQueue.getQueueLength() > 0) {
            currentProcess = (Process) ioQueue.removeNext();
        } else {
            currentProcess = null;
        }
        gui.setIoActive(currentProcess);
```

```
        return completed;
    }
```

1.3 Simulator

1.3.1 Variables

In the Simulator class, I added three variables,

1. private CPU cpu
2. private long avgIoTime
3. private IO io

These variables represents the CPU, the I/O unit and the average I/O-time for each operation. The I/O-time is given as a system parameter, but it was never keept nor used (as I could find).

I initialized the CPU and I/O just like the memory-unit, but I also gave them a handle for the eventQueue and the GUI, so that they could dispatch events and display "what's going on" to the user.

Code:

```
public Simulator(...) {
  ...
  memory = new Memory(memoryQueue, memorySize, statistics);
  cpu = new CPU(cpuQueue, maxCpuTime, statistics, eventQueue, gui);
  io = new IO(ioQueue, statistics, eventQueue, gui);
  this.avgIoTime = avgIoTime;
  ...
}
```

1.3.2 Event Handling

I extended the event handling model by implementing the four unimplemented functions, *switchProcess*, *endProcess*, *processIoRequest* and *endIoOperation*.

Switch process simply tells the processor to change what proccess that is currently active, and starts working on this one.

```
/**
     * Simulates a process switch.
     */
    private void switchProcess() {
        cpu.switchProcess();
        cpu.work(clock);
    }
```

End process fetces the current active process, wich is suppoesed to be done. The process is removed from the processor in this step. Then we free the memory, update statistics and just forgets about this process.

```
    /**
     * Ends the active process, and deallocates any resources ↩
         allocated to it.
     */
```

```
    private void endProcess() {
        Process p = cpu.endCurrentProcess();
        memory.processCompleted(p);
        p.updateStatistics(statistics);
    }
```

Process I/O Request start with fetching current process from CPU (the difference between fetch and end current process is that end current process return null if process is not completed), then update some statistics, and enters I/O-queue.

```
    /**
     * Processes an event signifying that the active process needs to
     * perform an I/O operation.
     */
    private void processIoRequest() {
        Process p = cpu.fetchCurrentProcess();
        p.leftCpuQueue(clock);
        io.insertProcess(p, clock);
    }
```

When an I/O operation is finished, we take the process wich completed the operation out of the I/O-device, tells the I/O-device that it may continue on if there are more processes left, do some statistics, and put the process back into the CPU-queue.

```
    /**
     * Processes an event signifying that the process currently doing ↩
       I/O
     * is done with its I/O operation.
     */
    private void endIoOperation() {
        Process p = io.getCompletedProcess();
        io.doIo(clock);
        p.leftIoQueue(clock);
        cpu.insertProcess(p, clock);

    }
```

1.3.3 Statistics

I also added a call to the *timePassed*-method of *cpu* and *io* in the same way as was done for *memory*. This is only for statistics.

```
public void simulate() {
  ...
  while (clock < simulationLength && !eventQueue.isEmpty()) {
    Event event = eventQueue.getNextEvent();
    ...
    long timeDifference = event.getTime()-clock;
    ...
    memory.timePassed(timeDifference);
    ...
    cpu.timePassed(timeDifference);
    io.timePassed(timeDifference);
    ...
  }
  ...
}
```

1.4 Statistics

1.5 Event

I added a toString-method to the Event-class, for easier use of debugger, stack traces and outputs. This has nothing to do with the program logic, only for my own understanding of the code.

2 Experimenting with the system

2.1 System parameters and their impact on performance

I have done some experimenting on the parameters of the system, and found that both memory and Round-Robin-timeshare is not that important to overall system throuhput. But if the I/O-hunger of the processes increase, we get a huge performance hit, since the processor has to do more waiting for processes, the processes stay longer in the system, and thereby uses more memory-time, and again blocks new processes.

If we have a lot of memory, we can get the CPU running most of the time even with high I/O, but we still have to wait a long time before any process finish, and the thouhput is stil bad. I think that if we are to get a significantly higher throuhput with I/O-hungry processes, we need to change the Round-Robin-algorithm with something else.

Some of my test-runs are in the Excel-file delivered with this report.

2.2 Alternative algorithms

I think that if we used an algorithm that sorted the processes so that the processes with least CPU-time left would be handled first, we could get rid of more processes at a faster pace, and the system would speed up. We would still have to wait for those processes using I/O, so maybe we should implement some more efficent algorithm as I/O-queue as well. I have not tried to implement a new algorithm, as I neither have more time left for this task, nor does the task tell me to do so.