Chalmers | GÖTEBORGS UNIVERSITET
Josef Svenningsson, Computer Science and Engineering

Concurrent Programming TDA381/DIT390

Saturday, October 23, 8.30 – 12.30, M.

(including example solutions to programming problems)

Josef Svenningsson, tel. 070 5455542

- Grading scale (Betygsgränser):

  Chalmers: 3 = 20–29 points, 4 = 30–39 points , 5 = 40–50 points
  Chalmers ETCS: E = 20–23, D = 24–29, C = 30–37, B = 38–43, A = 44–50
  GU: Godkänd 20–39 points, Väl godkänd 40–50 points

  Total points on the exam: 50

- Results: within 21 days.
- **Permitted materials (Hjälpmedel):**
  - Dictionary (Ordlista/ordbok)
- **Notes:**
  - Read through the paper first and plan your time.
  - Answer in either Swedish or English.
  - If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
  - Start each of the questions on a new page.
  - The exact syntax of each programming language you are going to use is not so important as long as the graders can understand the intended meaning. If you are unsure just put in an explanation of your notation.
  - Points will be deducted for solutions which are unnecessarily complicated.

Question 1. What does it mean for a library to be thread-safe? What can cause a library to not be thread-safe? *(2p)*

Question 2. **Background** This question is based on the cigarette smokers problem originally proposed by Suhas Patil in 1971 to point out some weaknesses with semaphores as proposed by Edsger Dijkstra. Patil's argument was however rather weak and the criticism never stuck.

The Problem Assume a cigarette requires three ingredients to smoke:

1 Tobacco

2 Paper

3 A match

Assume there are also three chain smokers around a table, each of whom has an infinite supply of one of the three ingredients – one smoker has an infinite supply of tobacco, another has an infinite supply of paper, and the third has an infinite supply of matches.

Assume there is also a non-smoking arbiter. The arbiter enables the smokers to make their cigarettes by selecting two of the smokers arbitrarily (randomly), taking one item out of each of their supplies, and placing the items on the table. He then notifies the third smoker that he has done this. The third smoker removes the two items from the table and uses them (along with his own supply) to make a cigarette, which he smokes for a while. Meanwhile, the arbiter, seeing the table empty, again chooses two smokers at random and places their items on the table. This process continues forever.

The lifetime of the arbiter is as follows. He repeats the following sequence of instructions indefinitely:

- Wait for the table to become free of items.
- Select a smoker who should smoke. Take ingredients from the other two smokers and put them on the table.

The lifetime of a smoker is also assumed to go on indefinitely (unrealistic, yes, I know). Smokers issue the following instructions:

- Wait for the arbiter to select me as the new smoker.
- Take all the items off the table and make a cigarette.
- Smoke.

The smokers do not hoard items from the table; a smoker only begins to roll a new cigarette once he is finished smoking the last one. If the arbiter places tobacco and paper on the table while the match man is smoking, the tobacco and paper will remain untouched on the table until the match man is finished with his cigarette and collects them.

Your assignment Your task is to implement a simulation of the cigarette smokers problem. Your implementation does not have to model passing the actual ingredients on the table. The important thing to capture is the synchronization between the arbiter and the three smokers.

To get full points your solution must fulfill the following criteria:

- You can use either Java or JR

- You must use semaphores for synchronization. No other synchronization constructs are allowed.
- The three different smokers must all execute the same code.

(10p)

A solution in JR.

```
class Smokers {

  sem table = 1;
  sem[] smokers = {0,0,0};

  process arbiter {
    while(true) {
      P(table);                 //Put stuff on the table unless there's anything there already.
      int k = random(0,2);      //Choose a random smoker to do the smoking
      V(smokers[k]);            //Signal the smoker to start smoking
    }
  }
  process smoker(int i=0; i < 3; i++) {
    while(true) {
      P(smokers[i]);  //Wait for arbiter to let me smoke
                      //Make a cigarette
      V(table);       //Signal that all the ingredients are off the table
                      //Smoke the cigarette
    }
  }
}
```

Question 3. The Problem The search-insert-delete problem is a generalization of the readers/writers problem.

Three kinds of threads share access to a singly-linked list: searchers, inserters and deleters. Searchers merely examine the list; hence they can execute concurrently with each other. Inserters add new items to the end of the list; insertions must be mutually exclusive to preclude two inserters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and deletion must also be mutually exclusive with searches and insertions.

Your assignment Your task is to implement the search-insert-delete problem using Java 5 monitors. You do not have to think about the linked list mentioned in the above paragraph, your only concern is the synchronization between different threads. Use the following interface when implementing your solution:

```
public interface SID {
  public void startSearch ()
  public void endSearch()
  public void startInsert ()
```

```
  public void endInsert ()
  public void startDelete ()
  public void endDelete()
}
```

There is no need to think about fairness for this assignment. *(10p)*

```
public class SIDImpl implements SID {
  private final Lock lock = new ReentrantLock();
  private final Condition okToSearch = lock.newCondition();
  private final Condition okToInsert = lock.newCondition();
  private final Condition okToDelete = lock.newCondition();
  int ns, ni, nd;
  public void startSearch () {
    lock.lock();
    try {
      while (nd > 0)
        okToSearch.await();
      ns++;
    } finally
      lock.unlock();
  }
  public void endSearch() {
    lock.lock();
    try {
      ns--;
      if (ns == 0 && ni == 0) {
        okToDelete.signal();
      }
    } finally
      lock.unlock();
  }
  public void startInsert () {
    lock.lock();
    try {
      while (ni > 0 || nd > 0)
        okToInsert.await();
      ni++;
    } finally
      lock.unlock();
  }
  public void endInsert () {
    lock.lock();
    try {
      ni--;
      okToInsert.signal();
      if (ns == 0)
        okToDelete.signal();
```

```
    } finally
      lock.unlock();
  }
  public void startDelete() {
    lock.lock();
    try {
      while (ns > 0 || ni > 0 || nd > 0)
        okToDelete.await();
      nd++;
    } finally
      lock.unlock();
  }
  public void endDelete() {
    lock.lock();
    try {
      nd--;
      okToSearch.signalAll();
      okToInsert.signal();
      okToDelete.signal();
    } finally
      lock.unlock();
  }
}
```

Question 4. **The Problem** Imagine an assembly line which creates water molecules. There is a machine that given two hydrogen atoms and one oxygen atom will put them together into a water molecule. Unless there are sufficient number of hydrogen or oxygen atoms the machine has to wait.

Your assignment Your task is to implement the synchronization that the water machine has to do. You must use JR and message passing to solve the problem. Assume that there are two operations declared as follows:

```
op void hydrogen();
op void oxygen();
```

These are called whenever a hydrogen or an oxygen becomes available to the machine, you don't have to worry about calling these operations. Whenever the machine produces a water molecule two messages on the hydrogen channel and one message on the oxygen channel should be removed. However, and this is important, *no message should be removed unless there are sufficient number of atoms of each sort.* *(8p)*

```
while (true) {
  inni void oxygen() st hydrogen.length() >= 2 {
    receive hydrogen();
    receive hydrogen();
    // Make molecule
  }
}
```

Question 5. **The Problem** The objective of this assignment is to design and implement an Erlang module which implements a Linda-style tuple space and provides some of the standard tuple-space primitives.

The Interface The interface to the module must consist of the following collection of functions:

−**module**(linda).
−**export**([new/0, in/2, out/2]).

new creates a new (empty) tuple space.in takes two arguments, a tuple space and a pattern and returns a tuple matching the pattern from the tuple space. If there is no such tuple the operation will block. out takes two arguments, a tuple space and a tuple. It puts the tuple into the tuple space.

In the interface functions, tuples are naturally represented by Erlang tuples. The representation of patterns – in the definition of the in function – should use a single "wild card" pattern any.

You may assume the existence of a function match which takes two arguments, a pattern and a tuple. match returns **true** if the tuple matches the pattern and **false** otherwise.

It is important that your solution works for *any* tuples – do not assume that client tuples will never contain certain atoms. A client should not be able to break the implementation by inserting "bad" tuples. *(5p)*

Question 6. **CCR** In this assignment you will use a synchronization primitive called Conditional Critical Regions, or CCR for short. For the purpose of this assignment we are going to assume that Java is equipped with CCR as its only synchronization primitive. Therefore you may use any Java syntax and its libraries freely except for when it comes to synchronization.

The syntax of CCRs look as follows:

```
atomic (b) {
    ...
}
```

A CCR is a block of code that begins with the keyword **atomic**. After the **atomic** keyword comes a boolean condition which is written b in this example. The boolean condition can be left out if it is always true. After the boolean condition comes the code which should be executed. We will refer to these code blocks as atomic blocks.

Semantically CCRs work as follows: Whenever a thread executes inside an atomic block one can safely assume that no other thread is executing in any other atomic block. In other words, atomic blocks provide mutual exclusion. If there is a thread executing in an atomic block and another thread tries to enter another atomic block this other thread will block until the first one has exited its atomic block.

When a thread is trying to enter an atomic block and no other threads are executing inside an atomic block, the first thing that happens is that the boolean condition is evaluated. If it evaluates to true then the thread gets the mutex and enters the atomic block. If, however, the boolean expression evaluates to false the thread is blocked and will resume some time in the future when the condition becomes true. Therefore, the boolean expression in the atomic block provides a means for conditional synchronization. This completes the description of CCRs.

Your assignment This assignment is in two parts. In both of them you implement a solution to the barrier synchronization problem. Your solutions should implement a class with the following interface:

```
class Barrier {
  public Barrier(int n)
  public void await()
}
```

The constructor takes an integer which indicates how many processes that should be waited for before the barrier is released. The await method is called when a process arrives at the barrier and wants to synchronize with other processes. The call blocks until all processes has arrived at the barrier.

Part a) In the first part your solution should solve the barrier synchronization problem for a one-shot barrier, meaning that it only has to work for a single barrier. The object does not have to reset for subsequent usages. *(5p)*

Part b) In the second part your solution must be a cyclic barrier, meaning that once all processes have arrived the barrier is reset and should be usable again. *(10p)*

Solution. First the one-shot barrier.

```
class Barrier {
  int n;
  int arrived = 0;
  public Barrier(int n) { this.n = n; }
  public void await() {
    atomic {
      arrived++;
    }
    atomic (arrived == n) {
    }
  }
}
```

Here's the cyclic barrier, which is surprisingly subtle.

```
class Barrier {
  int n;
  int arrived = 0;
  int turn = 0;
  public Barrier(int n) { this.n = n; }
  public void await() {
    int myTurn;
    atomic (arrived < n) {
      arrived++;
      myTurn = turn;
    }
    atomic (arrived == n || turn /= myTurn) {
      if (myTurn == turn) {
```

```
            arrived = 0;
            turn++;
        }
    }
  }
}
```