Chalmers | GÖTEBORGS UNIVERSITET
Alejandro Russo, Computer Science and Engineering

Concurrent Programming TDA382

Monday, August 25, 14:00, Väg och vatten.

Raúl Pardo (Alejandro Russo), tel. 0762 744561 (0705 110896)

- The maximum amount of points you can score on the exam: 68 points. To pass the course, you need to pass each lab, and get at least 24 points on the exam.

  The grade for the exam is as follows:

  Chalmers: grade 3: 24 - 38 points, grade 4: 39 - 53 points, grade 5: 54 - 68 points.
  GU: Godkänd 24-53 points, Väl godkänd 54-68 points

  The grade for the whole course is based on the points obtained in the exam and the labs. More specifically, the course grade (exam + lab points) is determined as follows.

  Chalmers: grade 3: 40 - 59 points, grade 4: 60 - 79 points, grade 5: 80 - 100 points.
  GU: Godkänd if passed the labs and get Godkänd in the exam. Väl godkänd is you passed the labs and got Väl godkänd in the exam.

- Results: within 21 days.
- **Permitted materials (Hjälpmedel):** Dictionary (Ordlista/ordbok)
- **Notes:**
  - Read through the paper first and plan your time.
  - Answers preferably in English, some assistants might not read Swedish yet.
  - 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.
  - As a recommendation, consider spending around 45 minutes per exercise. However, this is only a recommendation.
  - To see your exam: *by appointment* on Sept 4th, 12:00-13:00, room 5480 (5th floor), EDIT building.

Question 1. Answer the following questions. You should always justify your reply when answering a Yes/No question. Make it short and to the point.

a) In which situations unique references, created by calling `make_ref()`, are needed in Erlang? *(3p)*

b) In a Java 5 monitor, is it guarantee that the interaction between the boundary queue and the queues associated with conditional variables provides *fairness* for threads? *(3p)*

c) Why workers are useful? *(3p)*

d) Are semaphores as expressive as monitors (signal and continue semantics)? In other words, can you take any program written using monitors and rewrite it using semaphores so that you get the same synchronization behavior as with the monitor? *(3p)*

e) From the Erlang's programmer point of view, is there any different between implementing concurrent systems or systems which can use parallelism (i.e. several cores)? In other words, if your Erlang program wants to use as many cores as possible, do you need to use some special primitives in our code? *(3p)*

Question 2. The first solution presented for the shared-update problem was Peterson's algorithm:

```
private int turn = 0;
private boolean flag[] = {false, false};
process CS ((int i=0;i<2;i++)) {
  other = (i+1)%2;
  while (true) {
    flag[i] = true;
    turn = other
    while (flag[other] && turn==i) ;
    // Critical section
    flag[i] = false;
  }
}
```

a) In case that *both processes simultaneously want to enter the critical section*, how does the algorithm decide which process will succeed and which one will wait? *(5p)*

b) Assume you replace the inner **`while`**-loop by the following line:

```
while (flag[other] && turn==other) ;
```

Does this modification of the algorithm provide a fair solution? In other words, if a process wants to enter the critical section, will it eventually do it? *(8p)*

Question 3. **The Problem** Even though this winter is not very snowy this year, still there is a considerable amount of people who have decided to go skiing. When we go there, it is very common to use a cable car to go up to the mountain. There are quite a lot of different types of cable cars, for the exercise we assume one which has the following features:

- There is only one car in the line.
- Skiers only take it from the bottom to the top. They go down skiing!
- The car is not at the bottom initially. So, it has to get to the bottom station and wait for the skiers.

- The car has space for N skiers, where N is a positive natural number.
- Skiers can get in the cable car at the same time. It is important that a beginner skier (who takes quite a lot of time to get in) does not slow down another more proficient skiers when they want to get in.
- The cable car only departs when is full of skiers.
- We assume $N * P$ skiers per day, where P is a positive natural number.

Your assignment

a) You have to implement the simulation of a cable car used by skiers to go up the top of the mountain. The cable car have to wait at the bottom of the mountain until it is full of skiers. Then it goes up and once it reaches the top the skiers can get off. You can assume that the skiers get off instantly from the cable car once it reaches the top. The cable car goes down when is empty. *(8p)*

b) Extend the previous solution without the assumption that the skiers get instantly off the cable car. Therefore, the cable car has to wait until all the skiers get off before it goes down. It is also possible that the beginner skiers also take more time for getting off and it should not interfere when the more proficient skiers want to get of. *(5p)*

You should use Java 5 monitors as synchronization primitive. No other synchronization constructs are allowed. Every skier in the simulation must execute the same code.

Here is a short reference of what you will need from `java.util.concurrent.locks`.

```
class ReentrantLock {
 public ReentrantLock();
 public Condition newCondition();
 public void lock();
 public void unlock();
}
class Condition {
 public void await();
 public void signal();
 public void signalAll();
}
```

Question 4. Consider a savings account shared by several people. Each person associated with the account may deposit or withdraw money from it. The current balance in the account is the sum of all deposits to date less the sum of all withdrawals to date. Clearly, the balance must never become negative. A person making a deposit never has to delay (except for mutual exclusion), but a withdrawal has to wait until there are sufficient funds.

Your assignment You are giving the task to write up a Java class to implement shared accounts. More specifically, you need to provide the following interface.

```
class SharedSavingsAccount {
   public SharedSavingsAccount(long initialBalance) ;
   public void deposit(int amount) ;
   public withdraw(int amount) ;
}
```

Your solution must fulfill the following criteria:

- You must use Java
- You must use semaphores for synchronization or mutual exclusion. No other synchronization constructs are allowed.
- Do not care about fairness in your solution.

(13p)

Question 5. **The Problem** Haskell utilizes special variables, called MVars, as means for synchronization among threads. For simplicity, we assume that we have *only one* MVar that stores integer numbers. We are going to learn how this MVar works by showing its primitives:

- `putMVar(int x)`: This primitive stores the value of *x* into the MVar only if the MVar is empty. Once a value is stored into the MVar, we say that the MVar is full.
- `int takeMVar()`: This primitive reads the value stored into the MVar and empties it.

As usual, if any of these operations cannot be completed because some condition was not fulfilled (e.g., the MVar is empty), then the thread should block until that condition is satisfied.

Your assignment You should implement the notion of MVar in Erlang. For that, you should provide the following primitives:

- `e_initMVar()`, which initializes the MVar
- `e_putMVar(X)`, which implements `putMVar(int x)` in Erlang
- `e_takeMVar()`, which implements `takeMVar()` in Erlang

Your solution must fulfill the following criteria:

- You must use Erlang
- You must use Erlang's message passing model. No other synchronization constructs are allowed.

(14p)