Concurrency:
State Models & Design Patterns

Practical Session

Week 05

Exercises 04

<u>Discussion</u>

Exercise 04 - Task 1

a) Why are immutable classes inherently safe?

Because the state of an instance of an immutable class cannot be changed after its creation.

b) What is “balking”?

Balking describes a design pattern that is used in conjunction with state-dependent actions. This pattern is very common in data access classes such as URLConnection, where the connection needs to be in a certain state before data can be exchanged.

Exercise 04 - Task 1

c) When is partial synchronization better than full synchronization?

- When objects encapsulate both mutable and immutable data
- When code parts can be organized such that not all methods have to deal with critical sections

d) How does containment avoid the need for synchronization?
By embedding unsynchronized objects inside other objects that have at most one active thread at a time.

Exercise 04 - Task 1

e) What liveness problems can full synchronization introduce?
When one method contains two separate critical sections a potential deadlock may occur. Moreover, the nested monitor lockout problem can arise as well.

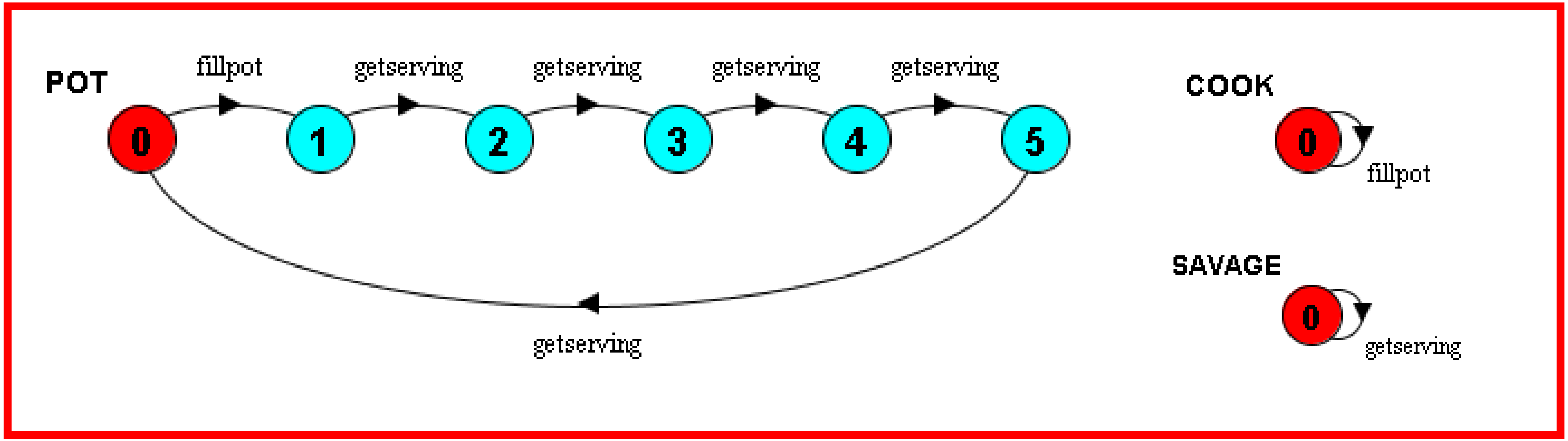
f) When is it legitimate to declare only some methods as synchronized?

When methods exist that access only immutable data.

Exercise 04 - Task 2

The dining savages: A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. (...)

Model the behavior of the pot as an FSP process.



Exercise 04 - Task 3

Please consider the FSP below and answer the questions:

```
property LIFTCAPACITY = LIFT[0],
     LIFT[i:0..8] = (enter -> LIFT[i+1]
     |when(i>0) exit -> LIFT[i-1]
     |when(i==0)exit -> LIFT[0]).
```

a) Which values can the variable i take?

The variable i can either be one of the values {0, 1, 2, 3, 4, 5, 6, 7, 8}

b) What kind of property is used and what does it guarantee?

A safety property is used and it enforces the variable to be within the predefined range (i = 0..8).

Exercise 04 - Task 3

Please consider the FSP below and answer the questions:

```
property LIFTCAPACITY = LIFT[0],
     LIFT[i:0..8] = (enter -> LIFT[i+1]
     |when(i>0) exit -> LIFT[i-1]
     |when(i==0)exit -> LIFT[0]).
```

c) Provide an action trace that violates the provided property.

enter, enter, enter, enter, enter, enter, enter, enter, enter (9 times)

d) Provide an action trace that does not violate the provided property.

enter, enter, enter, exit, exit, exit

Exercise 04 - Task 4

Implement a thread-safe MessageQueue class in Java using one of the safety patterns. You have to justify your choice of the safety pattern.

- The MessageQueue has the following specifications:
- It provides FIFO (first-in-first-out) access
- (...)
- There are (at least) two methods: `add(String msg)` and `remove()`
- Write a unit test for your MessageQueue to demonstrate the thread-safety of your implementation

Exercise 04 - Task 4

```java
public class MessageQueueTest {

    @Test
    public void queueStressTest() throws InterruptedException {
        int threadCount = 10;
        final MessageQueue mq = new MessageQueue(5);
        List<Thread> threads = new ArrayList<Thread>();
        for (int i=0; i < threadCount; i++) {
            threads.add(new MessageProducer(mq, 100));
            threads.add(new MessageConsumer(mq, 100));
        }
        for (Thread thread: threads) { thread.start(); }

        for (Thread thread: threads) { thread.join(); }

        assertEquals(true, mq.isEmpty());
    }
}
```

```java
public class MessageQueue {
    private final Lock mutex = new ReentrantLock();
    private final Condition notFull = mutex.newCondition();
    private final Condition notEmpty = mutex.newCondition();

    private String[] messages;
    private int head = 0;
    private int tail = 0;
    private int size = 0;

    public MessageQueue(int maxSize) {
        assert maxSize > 0;
        this.messages = new String[maxSize];
    }

    public int size() {□

    public int maxSize() {□

    public boolean isFull() {□

    public boolean isEmpty() {□

    public void add(String s) {
        this.mutex.lock();
        try {
            while (this.isFull()) {
                this.notFull.await();
            }

            this.messages[this.tail] = s;
            this.tail = (this.tail + 1) % this.messages.length;
            this.size++;
            this.notEmpty.signal();
        } catch (InterruptedException e) {
        } finally { this.mutex.unlock(); }
    }
```

Exercises 05

Preview

Exercise 05 - Task 1

Answer the following questions:

a) **What is a guarded method and when should it be preferred over balking?**

b) **Why must you re-establish the class invariant before calling wait()?**

c) **What is, in your opinion, the best strategy to deal with an InterruptedException? Justify your answer!**

d) **How can you detect deadlock? How can you avoid it?**

e) **Why it is generally a good idea to avoid deadlocks from the beginning instead of relying on deadlock detection techniques?**

f) **Why is progress a liveness rather than a safety issue?**

g) **Why should you usually prefer notifyAll() to notify()?**

h) **What is the difference and what are the similarities between a nested monitor lockout (a.k.a. nested deadlock) and a classical deadlock?**

Exercise 05 - Task 2

You have seen the dining philosophers problem during lecture. Please describe four different solutions to this deadlock problem and explain the concept and fairness characteristics of each.

Exercise 05 - Task 3

Consider the FSP model for the dining philosphers:

```
PHIL = ( sitdown
-> right.get -> left.get -> eat
-> left.put -> right.put -> arise -> PHIL ).
FORK = ( get -> put -> FORK).
||DINERS(N=5) =
forall [i:0..N-1]( phil[i]:PHIL
|| {phil[i].left,phil[((i-1)+N)%N].right}::FORK ).
```

Modify the FSP specification, in a different manner than seen in the lecture, to remove the deadlock.

Exercise 05 - Task 4

In a FSP model of a process a deadlocked state is a state with no outgoing transitions, a terminal state so to say. A process in such a state can engage in no further actions. In FSP, this deadlocked state is represented by the local process STOP. By performing a breadth-first search of the LTS graph (in CHECK PROGRESS), the LTSA tool guarantees that a sample trace is the shortest trace to the deadlock state.

Consider the maze depicted in Figure 1. Write a description of the maze as FSP process which, using deadlock analysis, finds the shortest path out of the maze starting at any square in the maze.

You may check your solution by inspecting the trace to deadlock from specific squares.

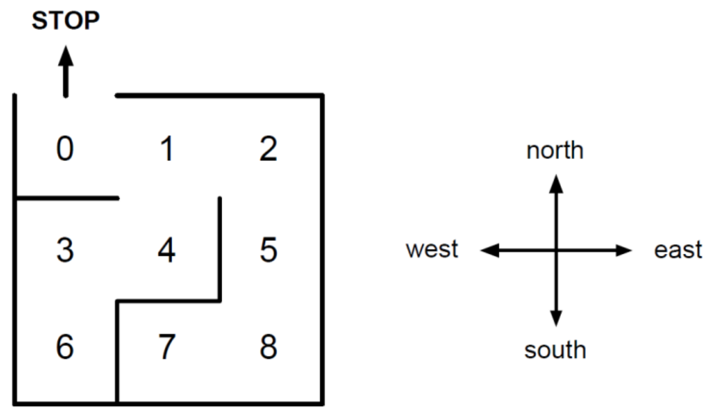


Figure 1: A maze.

… one more thing

Lab 01

Sneak Peak

Lab 01 - Overview

Concept

- *Eclipse + Java based tasks*
- *Concurrency code that needs work*
- *Work in groups of two*
- *The lab starts at 10:15 and ends at 12:00*
- *Location will be <u>announced later through Piazza</u>*

Grading

- *5 bonus points for 100% correctness*

Requirements

- *Notebook with power adapter*
- *Latest version of Eclipse installed*
- *Latest version of the Java SDK installed*
- *Optional: a mouse*

Lab 01 - Sections

1) Pull from public GitHub repository
2) Fix the four provided sample apps
3) Submit your zipped project solutions by mail to gadient@inf.unibe.ch

Allowed:

- Lecture slides and books

- Internet access

NOT ALLOWED:

- COPYING OLD SOLUTIONS