Block 6

Block 6

6-OV Overview

6-OV.1 Contents of This Block

CP This block discusses lock-free algorithms and data structures, as well as the basics behind memory models.

FP This block has supervised lab time to work on the FP project.

CC This block continues and finishes the discussion of the procedure abstraction and memory layout started in Block 5.

LP This block finalises the LP strand with the project.

6-OV.2 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study to prepare the CP exercises;
- 8 hours of unsupervised work on the FP project.

6-OV.3 Materials for this Block

- **CP**, from JCIP: Chapters 14.5, 14.6, 15 and 16.
- **CC**, from EC: Chapter 6.

6-CP Concurrent Programming

Lock-free programming. Besides solving many concurrency-related problems (e.g., data races), locks can also introduce a wide range of problems (e.g., deadlocks and performance degradations). However, there are ways to make containers thread safe without the use of locks, so-called *lock-free* programming (aka lockless program). A program is called lock-free if it satisfies the following property: as long as the program is able to keep calling lock-free operations, the number of completed lock-free operations keeps increasing, no matter what. It is algorithmically impossible for the system to lock up during those operations. As always, a new method of solving existing problems is prone to introduce new problems.

WARNING. Lock-free programming is incredibly tricky to get right. It is strongly recommended to go online, look up sample implementations, possible problems (look for the ABA problem) and their possible solutions. The correctness of lock-free containers is even more difficult to check and guarantee than when using locks – the best way to learn is to look up the mistakes and solutions of others who have spent a lot of time thinking about the consequences of their design choices and tried to make sure their implementations were thread-safe. Some example information to get you started:

- A quick overview of lock-free programming problems
- Some more elaboration on the ABA problem.

Exercise 5-CP.1 In this exercise, we will look at lock-free stack implementations.

1. One of the most simple data structures to make lock-free is a stack. During the lecture we discussed Treiber's stack. Implement a lock-free stack yourself, featuring both a lock-free `push` and `pop` operation. Make sure that your stack implements the following interface:

   ```
   public interface LockFreeStackInterface <T> {
       public void push(T x);
       public T    pop();
       public int  getLength();
   }
   ```

2. As always: devise a test, and apply it to make sure that your lock-free implemenation works as expected.
3. Can you adjust your implementation to change it into a bounded stack? If yes, explain how you do this. If not, why is this not possible?
4. Make note of the issues that can arise from lock-free programming and try to identify ways to detect and prevent these problems. Note: some of the problems you encounter will be solved by the garbage collector as they only revolve around occasionally losing references, but often the problems are more complex; you also have to take them into account when programming in Java.

□

Barrier implementation. Last week you have seen an application of a barrier – as part of a MAPREDUCE framework. Often, especially when writing *parallel* programs (in contrast to writing *concurrent* programs) the need for barriers will arise because of the need to synchronize the progress of different threads. As with locks, there are many ways to implement a barrier, in both locked and lock-free ways.

Consider the interface `Barrier`:

```
public interface Barrier {
    public int await() throws InterruptedException;
}
```

Exercise 5-CP.2 In this exercise, we will look at various ways to implement barriers.

1. Implement a cyclic barrier with locks and a `wait/notify` or `await/signal` system. The use of `wait/notify` has been discussed during the lecture of block 4. Furthermore, some additional resources on writing code with `wait/notify` can be found here:
   - Why `wait`, `notify` and `notifyAll` must be called from synchronized block or method in Java.
2. Implement a cyclic barrier using lock-free programming. Similarly to Exercise 6-CP.1, think very carefully about your design as lock-free programming can introduce very subtle bugs. However, by using some of the classes in `java.util.concurrent` you can simplify your design a lot – decreasing the odds of concurrency bugs.

□

Memory Models. To understand the unexpected behaviours of a program with data races when it is executed with a relaxed memory model, here are a few exercises.

Exercise 5-CP.3 Motivate your answers!

1. If initially `x` and `y` are `0`, then what are the possible final values of `i` and `j`?

Thread 1	Thread 2
`x = 1`	`y = 1`
`j = y`	`i = x`

2. If initially `answer` = `0`, what can be printed after executing the following threads?

Thread 1	Thread 2
`answer = 42`	`if (ready) then`
`ready = true`	`    println(answer)`

   What would change if `answer` be declared `volatile`?

3. When executing the following threads, can this result in `r1 = r2 = 1`?

Thread 1	Thread 2
`r1 = x`	`r2 = y`
`if r1 > 0 then`	`if r2 > 0 then`
`    y = 1`	`    x = 1`

□

6-FP Functional Programming

The work of Block 6 consists of doing the FP project.

6-CC Compiler Construction

Consider the following PASCAL program:

```
Program fib;

Function fib(n: Integer): Integer;
    Begin
        If n <= 1
        Then fib := 1
        Else fib := fib(n-2) + fib(n-1)
    End;

Var arg, result: Integer;
Begin
    In("Argument? ", arg);
    result := fib(arg);
    Out("Result: ", result)
End.
```

The following PASCAL-specific points should be noted:

- Keywords are case-insensitive;
- Function names and variable names cannot overlap;

- The return value of a function is determined by assigning it to the function name, as if it were a variable.

As before, **`In`** and **`Out`** are special functions introduced for the purpose of this course to provide a limited form of user interaction; they correspond one-to-one to the ILOC instructions **`in`** and **`out`**.

Exercise 6-CC.1 You are asked to write an ILOC program that faithfully simulates the above PASCAL program, in particular including the recursive calls of `fib`.

1. What is the layout of your activation records, i.e., what do you need to store there, and at what relative position? Take EC Fig. 6.4 as a reference for the possible components of an activation record. *(This question is intended to guide you for the next steps; do not invest too much time in a very precise answer.)*
2. Where will you place your activation records: dynamically allocated on the heap, dynamically allocated on the stack, or statically allocated? See pages 283–284 of EC for reference.
3. Write the ILOC code and test it using the ILOC `Simulator`. You may freely take advantage of the ILOC extensions described in Appendix B (in this reader); in particular the usage of labels to refer to line numbers (and to store the return address for a procedure call) and the built-in stack functionality.
4. At around $n = 20$ and above, the simulation starts to get very slow. Explain this phenomenon. □

Challenge exercise. The following exercise does not need to be signed off; however, it is recommended for everyone who plans to include procedures/functions in the language they plan to design for the final project.

Among the provided files you will find `SimplePascal6.g4`, which is the same grammar as in the last block, and `FuncPascal6.g4`, which extends it with function declarations and calls. Note that functions can only be declared on the top level.

Issues you have to address when extending your solution to cope with functions are:

- You should type check your function calls. To enable this, the provided `Type` class also contains a `Function` type. Optionally, you may also want to test if every function always assigns a return value.
- The simple `Scope` class is no longer sufficient, as you have to distinguish local variables (of the functions) from global variables (of the main program). For this purpose, you can use symbol tables as seen before. You should consider carefully at which moment you open the scope of a function: the function parameters should be part of the inner scope.

Exercise 6-CC.2 *This exercise is optional and does not have to be signed off.* Program classes `pp.block6.cc.func.Checker` and `pp.block6.cc.func.Generator` analogous to the ones of last Block (Exercises 5-CC.6 and 5-CC.7) for the grammar `FuncPascal6.g4`. Test your result using the provided example programs (in the lab files). □

6-LP Logic Programming

The work of Block 6 consists of finishing and submitting the LP project.