Block 6

Block 6

6-OV Overview

6-OV.1 Contents of This Block

CP This block discusses identification of parallel tasks, handling exceptions and errors, and memory models.

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

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

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

- 6 hours self-study to prepare the CP exercises.

6-OV.3 Materials for this Block

- **CP**, from JCIP: Chapters 6-8, 16.
- **CC**, from EC: Chapter 6.

6-CP Concurrent Programming

Fork/Join. The *Fork/Join* framework is an implementation of the `ExecutorService` interface that helps to take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The center of the Fork/Join framework is the `ForkJoinPool` class. A `ForkJoinPool` object is used to execute `ForkJoinTask` processes. A `ForkJoinTask` is a thread-like entity that is much lighter weight than a normal thread. The efficiency of `ForkJoinTask`s stems from a set of restrictions reflecting their main use as *computational tasks* calculating pure functions or operating on purely isolated objects. Two important subclasses of `ForkJoinTask` are the classes `RecursiveAction` and `RecursiveTask`.

The primary coordination mechanisms of a `ForkJoinTask` are `fork`, that arranges asynchronous execution, and `join`, that does not proceed until the task's result has been computed. Fork/Join's logic is simple:

(i) separate (`fork`) each large task into smaller tasks, (ii) process each task in a separate thread (separating those into even smaller tasks if necessary), and (iii) `join` the results.

A `ForkJoinPool` differs from other kinds of `ExecutorService` mainly by virtue of employing *work-stealing*: all threads in the pool attempt to find and execute subtasks created by other active tasks (eventually blocking waiting for work if none exist).

Exercise 6-CP.1 In this exercise you are asked to develop a multi-threaded program which computes a number from the Fibonacci sequence. The Fibonacci sequence is defined as:

```
f(0) = 1
f(1) = 1
f(n) = f(n-1) + f(n-2)
```

Instead of using an efficient algorithm which computes the sequence bottom-up, i.e., from `f(0)` to `f(n)`, the computation of `f(n)` should be splitted into two independent tasks: `f(n-1)` and `f(n-2)`. Although this is highly inefficient, the exercise itself is instructive.

1. Write a Java program which computes the Fibonacci number `f(n)` using the `ExecutorService` class. Each Fibonacci task has to be represented by a subclass of `Callable<Integer>`.
2. Write a Java program which computes the Fibonacci number `f(n)` using the Fork/Join framework. Each Fibonacci task has to be represented by a subclass of a `RecursiveTask<Integer>`.
3. Which of the two implementations is more effecient?

□

Exercise 6-CP.2 Merge sort is a `O(n log n)` recursive sorting algorithm, which lends itself to a multi-threaded implementation. Conceptually, merge sort works as follows:

- Divide the unsorted list into two sublists of about half the size.
- Sort each of the two sublists (e.g., using merge sort).
- Merge the two sorted sublist back into one sorted list.

1. Write a Java program to sort an array of `int`s. Your implementation should use the Fork/Join framework. The sort task should be represented by a subclass of a `RecursiveAction`.
2. Test and compare your `ForkJoin` implementation against Java's `Array.sort` and `Array.parallelSort` implementations. What can you conclude from your results?

□

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 6-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 (answer) 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

Description of Functional programming, block 6

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 refernece.
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). □