COM00005I

BSc, BEng, MEng, MMath Examinations, 2013–2014

All Undergraduate Computer Science Programmes
Stage Two

Principles of Programming Languages (POPL)

Open Examination

Issued at:
Noon: Sum/1/Mon (21st Apr)

Submission due:
Noon: Sum/4/Wed (14th May)

Your attention is drawn to the Guidelines on Mutual Assistance and Collaboration in the Students' Handbook and the Department's 'Guide to Assessment Policies and Procedures' (http://www.cs.york.ac.uk/exams/statementonassessment/).

All queries on this assessment should be addressed to
Alan Wood (Part 1) `amw11@york.ac.uk`**, or**
Andy Wellings (Part 2), `andy.wellings@york.ac.uk`.

There are two parts in this assessment. You must complete both parts, which carry equal marks. Each part is divided into tasks, and the relative weight for each task is indicated as a percentage of the total marks for the assessment. All tasks must be attempted.

Submission

- All submissions should be done *electronically*, using the Department's online submission system.[1] *Hardcopy submissions will not be accepted*.
- You should submit a zip file that contains the following:
  1. A directory called `PartI` and a directory called `PartII`.
  2. The directory `PartI` must include A `pdf` [2] file of your answer to Task 1 called `Task1.pdf`
  3. In the directory `PartII`: the following should be included:

[1] `https://www.cs.york.ac.uk/submit/index.php`

[2] PDF output from a Word document is available by using the Departmental Word installations. PDF output from a LaTeX document is obtained by using `pdflatex`.

- A directory called `Task2` that contains the source code for your solution to Task 2. Do not use any subdirectories.
- A directory called `Task3` that contains the source code for your solution to Task 3. It should also contain a pdf file that describes your approach to implementing Java monitors using semaphores. Do not use any subdirectories.
- A directory called `Task4` which contains a pdf file with your answer to Task 4.

Part I.

There is one task in this part:

Task 1 Write an essay, of approximately 2500 words[3] (not including program code), with the following specification: 50%

Title:

Stateful *versus* Stateless Programming

Abstract The real-world is a 'stateful' place: cars move, food gets eaten, tress grow etc. In every case some property of the object *changes*, while its other properties stay the same. We think of this as a change in the object's *state*.

It is obvious, therefore, that any programming language that can be used to model the real-world *must* have means for changing the state of the representations of the objects that it is modeling. However, there are many programming languages that *do not* have explicit state-mutation constructs, and yet are Turing-complete, implying that they can compute *precisely* the same things as stateful languages.

This essay will discuss the pros and cons of stateless and stateful programming languages and styles, analysing the 'obvious' statement given above. Examples of computations which are better programmed in one style than the other will be given, ending with a conclusion about how to determine which style is most appropriate in various programming applications.

Notes

1. Start your essay with the *exact* title and abstract as given above.
2. Use the principles of writing that were taught in the Stage 1 SKIL module.
3. Full references, properly formatted, to sources of material *must* be given.[4]
4. No more than half of the items in your bibliography can be to URLs alone — the others must be to 'printed' sources.

[3] About 5 typed (10pt), single-spaced, A4 pages.
[4] Remember, we can Google as easily as you!

5. You *must* include appropriate program illustrations from at least *three* different programming languages, and these must all be syntactically correct. These illustrations should *not* be counted as part of the 2500 words of the essay.

6. In order to pitch the essay at the correct level, you should assume the reader has the technical knowledge of a typical CS student just starting Stage 2, who has *not* yet attended the POPL module.

Marking

Points for which marks will be given include:

- Clarity of explanation.
- Quality and choice of references.
- Appropriate program examples.
- Quality of argument.
- Clarity of essay structure.
- Quality of the conclusion.

A sliding penalty will be applied if the essay is too long or too short. Word counts within $\pm 10\%$ will recieve no penalty.

Part II.

A semaphore is a very flexible synchronization mechanism. It is often claimed that most synchronization problems can be solved using semaphores. There are various types of semaphores, including:

- a general semaphore, which is a non-negative integer: its value can rise to any positive number
- a binary semaphore which only takes the value 0 and 1: the signalling of a semaphore which has the value 1 has no effect - the semaphore retains the value 1.

A monitor is a higher level synchronization mechanism. One way to argue that a monitor is an effective synchronization mechanism is to show how it can implement semaphores.

Task 2 Show how you would use Java monitors to implement the functionality of a general and a binary semaphore. You should submit a Java program that implements a bounded integer buffer and has two producers, two consumers and a buffer size of 2 (but *test* your program with a larger number of producers and consumers). The bounded buffer should use your monitor-implementations of binary and general semaphores for its synchronization. The problem can be solved with one binary and two general semaphores (plus some other non-semaphore variables). Do not use any pre-defined Java utilities (such as those in the `java.util` packages). Note, the only synchronized methods that should appear in your program are those needed to implement the semaphore classes. 10%

Task 3 Show how you would use two binary semaphores (and any other objects you might need, e.g. integer counters) to implement the functionality of a Java monitor. You will not be able to produce a Java program to directly show this. Instead you will have to illustrate how you would "compile" a Java monitor into a sequence of calls on one or more binary semaphores. So for example, you would:

For each call to `wait()` in a synchronized method: Indicate how you would get the synchronization properties of `wait()` using one or more binary semaphores (and any other objects you might need).

For each call to `notify()` in a synchronized method: Indicate how you would get the synchronization properties of `notify()` using one or more binary semaphores (and any other objects you might need).

For each call to `notifyAll()` in a synchronized method: Indicate how you would get the synchronization properties of `notifyAll()` using one or more binary semaphores (and any other objects you might need).

You should submit a Java program that implements a bounded integer buffer and has two producers, two consumers and a buffer size of 2 but test your program with a larger number of producers and consumers. The bounded buffer should illustrate how your monitor-based solution to the bounded buffer, when compiled, results in calls on the Java semaphores (that you implemented in Task 2).

Note 1: With a bounded buffer, the producer and consumer tasks cannot both be waiting. With other synchronization problems, for example the readers-writers problems, threads may be waiting for different events to occur. Hence, your translation from monitors to semaphores should not use any knowledge of the characteristics of the bounded buffer problem.

Note 2: The only synchronized methods that should appear in your program are those needed to implement the general semaphore class.

Note 3: Marks are divided equally between your overall translation approach, and the bounded buffer code.

25%

Task 4 Assume that you are a concurrent programmer who is confident and competent using semaphores and that you are asked to program in a language that ONLY supports monitors. Discuss the advantages and disadvantages of creating semaphores using monitors and then solving all your synchronization problems using semaphore-style solutions. 15%

Use of JavaPathfinder

I recommend that you use Java Pathfinder to help convince yourself that your programs are correct. However, model checking is very slow and memory intensive. You will only be able to model check small programs.