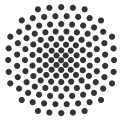

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2022

Exercise 6: Concurrency

(Deadline for uploading solutions: July 20, 2022, 11:59pm Stuttgart time)

The materials provided for this homework are:

- a pdf file with the text of the homework (this);
- a zip file with **the folder structure and the templates that must be used for the submission**.

The folder structure is shown in Figure 1.

```
exercise6.zip
├─ task_1/
│  └─ answers.csv  (Insert your solution in this file)
├─ task_2/
│  ├─ program_1/
│  │  ├─ data_races.csv  (Insert your solution in this file)
│  │  └─ possible_results.csv  (Insert your solution in this file)
│  ├─ program_2/
│  │  ├─ data_races.csv  (Insert your solution in this file)
│  │  └─ possible_results.csv  (Insert your solution in this file)
│  └─ program_3/
│     ├─ data_races.csv  (Insert your solution in this file)
│     └─ possible_results.csv  (Insert your solution in this file)
├─ task_3/
│  ├─ program_1.c  (Insert your solution in this file)
│  ├─ program_2.c  (Insert your solution in this file)
│  └─ stuttgart.pgm
└─ task_4/
   └─ src/
      ├─ Main.java
      └─ BankAccount.java  (Insert your solution in this file)
```

Figure 1: Directory structure in the provided .zip file and in your uploaded solution file.

The submission must be compressed in a **zip file** using the given folder structure. The names of folders and files must not be changed or moved, otherwise the homework will not be evaluated.

General tips for making sure your submission is graded as you expect:

- Use only the zip format for the archive (not rar, 7z, gz, or anything else).
- Your zip file should contain exactly one top-level directory *exercise6/*, as in the zip file provided by us.
- Do not rename files or folders, simply open the files provided and put your solutions.

1 Task I (10% of total points of the exercise)

This task is to check your understanding and revise terminology around concurrency. For that, we give you questions, some of which are accompanied by a code snippet. You have to **answer each question** by selecting **exactly one** correct answer out of multiple choices. (If in doubt, choose the one that you think matches best.)

To submit your answer, please fill in the file *exercise6/task_1/answers.csv*. One row corresponds to one question. E.g., fill in **A** in the second column of the CSV file to select option A as the answer, **B** to select option B, etc. Figure 2 shows an example. Make sure you submit a **comma**-separated file; do not use tabs, semicolons, or any other delimiter. For example, be careful when editing the CSV with Excel and double-check the file format with a plain text editor.

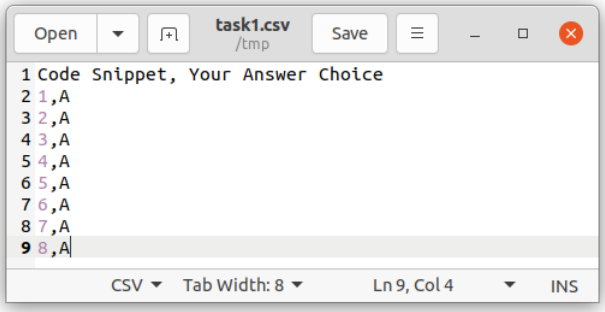

Figure 2: Example of the file format for the answers of Task I.

Question 1 Assume the following snippet is an excerpt of a web server, for example implemented in JavaScript and running on Node.js. It waits for incoming requests from the local machine on port 8080 and serves responses according to those requests. The server is running on a single machine with a single CPU core. All requests come from the same machine where the server is also running on.

JavaScript snippet of a web server

```
const http = require("http");
const requestListener = function (req, res) {
  switch (req.url) {
    case "/about":
      res.writeHead(200); res.end("About page"); break;
    // other pages...
    default:
      res.writeHead(404); res.end("Page not found"); break;
  }
};
http.createServer(requestListener).listen(8080);
```

Which of the following statements is correct?

A This program makes use of concurrency.

B Multiple requests will be processed in parallel, that is, at the exact same time.

C This program is part of a distributed system.

Question 2 Concurrency and parallelism are related, but distinct concepts. They are often confused with each other. Make sure your understanding is correct, then select the correct paragraph of the following two.

A Every concurrent program is always executed in parallel. Because every hardware is parallel, concurrency is an outdated concept. Thus, programming for concurrency requires no changes to your programs.

B In a concurrent program more than one execution context may be active at the same time. Different parts of the program are not necessarily executed at the exact same time. E.g., with the help of the operating system, a single CPU may make progress on multiple tasks, while neither is completed before the other.

Question 3 Assume the following snippet is an excerpt of a program written in the Go programming language. It creates a channel, starts a goroutine that waits for data from that channel, and sends data to that channel.

Go snippet of a concurrent program

```
func main() {
    // Create a channel c.
    c := make(chan int)
    // Start anonymous goroutine.
    go func() {
      fmt.Println("Received␣from␣channel:␣", <-c)
    }()
    // Send data via channel.
    c <- 1337
}
```

Which of the following statements is correct?

A This program uses shared memory for concurrency.

B This program uses explicit vector instructions for concurrency.

C This program uses message passing for concurrency.

Question 4 Make sure you understand the difference between processes and threads. Which of the following statements is correct?

A Multiple processes can execute on top of a single thread.

B Processes are provided by mainstream operating systems (e.g., Linux, Windows, MacOS) to run multiple programs concurrently and give each their own, separate address space.

C Threads are more heavyweight than processes, e.g., they require more resources and it takes a longer time switching between them.

Question 5 The following snippet is a simple and non-optimized implementation of a basic synchronization primitive in C++. (Do not use it in production, this is just for educational purposes.)

C++ snippet of a synchronization primitive

```
class sync_primitive {
  std::atomic<bool> lock_ = {false};
  void lock() { while(lock_.exchange(true)); }
  void unlock() { lock_.store(false); }
};
```

Which of the following statements is correct?

A This program uses an explicit memory barrier for concurrency.

B This program implements a spin lock, which is a form of busy waiting.

C This program is based on software transactional memory.

2 Task II (30% of total points of the exercise)

This task is about data races and volatile variables in Java. Below, we give three excerpts of concurrent Java programs. Assume that the declarations in the first two lines initialize the variables sequentially at the beginning of the program. Then, the methods *thread1* and *thread2* are executed in two different threads and the program waits until both are completed, e.g., via *join()*. This concludes the program execution.

Your task is twofold. First, you shall determine whether the programs suffer from data races, and if so, list each data race with the memory accesses that participate in the race. Second, you shall list all possible values that the program variables could take at the end of program execution, as per the Java language specification.

To submit your answers, please fill in the files *exercise6/task_2/program_N/data_races.csv* and *possible_results.csv*, where *N* is 1, 2, or 3, and *N* corresponds to the programs below.
For *data_races.csv*, one row in the CSV file corresponds to one data race. Please submit three columns for each data race: First, the name of the racy variable. Then, the two memory accesses that make it a data race, identified by their line numbers. E.g., if there is a data race on a variable *z* because of memory accesses in line 4 and line 7, you would add the line **z,4,7** to the CSV. The order of data races in the CSV file, and the order of accesses does not matter.
For *possible_results.csv*, one row in the CSV file corresponds to one possible final state of the program. Please submit columns for each variable in the program. E.g., if the program has three variables *a*, *b*, and *c* and one possible state at the end of execution is *a=1* and *b=2* and *c=3*, you would add the line **1,2,3** to the CSV.

Make sure you submit **comma**-separated files; do not use tabs, semicolons, or any other delimiter. For example, be careful when editing the CSV with Excel and double-check the file format with a plain text editor.

Program 1

```
1  int x = 2;
2  int y = 6;
3
4  void thread1() {
5      if (y < 6) {
6          x = x + y;
7      }
8  }
9  void thread2() {
10     y = 5;
11 }
```

Program 2

```
1  int x = 2;
2  volatile int y = 6;
3
4  void thread1() {
5      x = y;
6  }
7  void thread2() {
8      y = x + 1;
9  }
```

Program 3

```
1  volatile int x = 4;
2  int y = 1;
3
4  void thread1() {
5      x = x + y;
6  }
7  void thread2() {
8      x = x - y;
9  }
```

3 Task III (30% of total points of the exercise)

This task is about parallelizing existing sequential C programs with OpenMP. For that, you are given two small C programs, and your task is to insert the appropriate OpenMP pragmas to parallelize them. We indicate which regions of the program shall execute in parallel via comments in the original source code of the programs.

To submit your solution, please modify the files *exercise6/task_3/program_N.c*, where *N* is 1 or 2, corresponding to the program. A short description of the programs and which parts shall be parallelized is as follows.

- Program 1 prints several different hellos. Each individual operation is independent and resource intensive (here simulated with a call to *sleep*). In the parallelized version, each call to the *hello* function shall be executed in parallel (on a machine with a sufficient number of cores), such that on average less than 1 second passes between two outputs appearing on the console.
- Program 2 reads the *stuttgart.pgm* greyscale image from disk, inverts it, and saves it to a new file again. In the parallelized version, the outer loop in the body of the *invert* function shall be executed in parallel on different threads, and the inner loop shall be vectorized, i.e., make use of SIMD instructions.

To compile and run the original and the parallelized programs, run the following commands on a typical Linux system in the *exercise6/task_3/* directory (where *N* is 1 or 2, corresponding to the program):

```
gcc -O2 -fopenmp program_N.c -o program_N
./program_N
```

(Purely optional, for interested students:) You can also inspect the generated machine code to understand the effect of different OpenMP pragmas. For that, disassemble the compiled binary with:

```
objdump -d program_N > program_N.s
```

In the output file, you can search for the name of the function you are interested in.

Evaluation Criteria: The parallelized programs will be compiled with GCC version 9.4 with the flags `-O2 -fopenmp` and run on Ubuntu 20.04 LTS. The program outputs will be compared against the correct outputs (accounting for non-determinism due to parallel execution), and both the source code and generated machine code will be pattern matched against the correct solution. Changing the program semantics besides parallelization, and inserting/removing code besides OpenMP pragmas will lead to failing the task.

4 Task IV (30% of total points of the exercise)

This task is about fixing an insufficiently synchronized Java program. The program is a very simplified banking application, with *BankAccount* objects containing a balance and an account ID. Money can be transferred between accounts using the *transferToOther* method. Two accounts are set-up in the *main* method in class *Main*, and then many transfers are performed in two different threads.

The initial program is not properly synchronized. Your task is to modify the *exercise6/task_4/src/BankAccount.java* file. Insert the appropriate synchronization to ensure that the public API of this class can be used safely by multiple threads. In particular, ensure that there are no data races and that no money is accidentially ”lost” or ”created”. Do not insert *sleep* statements or otherwise change the semantics of the program, besides ensuring proper synchronization.

Hints: In case you have trouble fixing the program, read up on the difference between data races and race conditions, and on deadlocks and lock order inversion. Think about which variables are shared between threads, and whether they are properly protected by synchronization primitives. Think about which operations are atomic and which are not and can thus be interleaved with other operations.

Evaluation Criteria: Your program will be compiled and run with JDK version 8 (there is no need for using newer Java features). The output of running the main method will be compared against the correct output. The invariant that the global amount of money shall remain the same will be checked. The program will be run with a timeout of 30 seconds, a longer running program is assumed to suffer from a deadlock and hence not receive points. The source code and compiled class file will be pattern matched against the correct solution. Changing the program semantics besides parallelization, and inserting/removing code besides for the purpose of synchronization will lead to failing the task.