University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Winter 2024/2025

Exercise 6: Concurrency and Control Abstraction

Strict deadline for uploading solutions: **Feb 5, 2025, 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:

```
exercise6.zip
├─ task_1
│  ├─ example.txt
│  ├─ program_1.txt
│  ├─ program_2.txt
│  ├─ program_3.txt
│  ├─ program_4.txt
│  └─ program_5.txt
├─ task_2
│  └─ main.c
├─ task_3
│  ├─ part_1
│  │  ├─ answers.txt
│  │  ├─ array_multiplication.c
│  │  ├─ dot_product.c
│  │  └─ matrix_addition.c
│  └─ part_2
│     ├─ array_multiplication.c
│     ├─ dot_product.c
│     └─ matrix_addition.c
└─ task_4
   ├─ program_1.js
   ├─ program_2.js
   └─ program_3.js
```

The submission must be compressed in a .zip file (not .rar, .7z, .g.z, or any other format) using exactly the folder structure and names shown above.

1 Task 1 (20% of total points)

In this task, you are presented with five files containing excerpts of concurrent Java programs. It is assumed that the declarations in the initial lines sequentially initialize the variables at the beginning of the program. Subsequently, the methods "thread1" and "thread2" are executed in separate threads, and the program awaits their completion, typically through the use of the "join()" method.
Each file includes four questions after the code. The first question inquires whether a data race exists, to which you should respond with either "YES" or "NO". Following that, you are requested to indicate the line numbers where the data race occurs (i.e, the lines causing the data race). Additionally, you should identify the variable names involved in the race. Lastly, provide all the potential values for the given variable at the end of execution. Use the file example.txt as a reference for the format of the answers only.

2 Task 2 (35% of total points)

In this task, you are provided with the code for a producer-consumer system (`task_2/main.c`), and the objective is to introduce synchronization using mutexes to ensure proper functionality of the code.
The system is a producer-consumer simulation involving factories, bakeries, and consumers. Each component operates as a separate thread, interacting with shared resources (deposits) to simulate real-world production, distribution, and consumption of goods.

2.1 Components

1. **Factories:** Responsible for producing raw ingredients: flour and eggs. These resources are deposited into specific locations, which serve as storage points for bakeries.
   - **Flour Factory:** Produces flour at a configurable rate. The flour is deposited into:
     - Flour deposit A: Reserved for Bakery A.
     - Flour deposit B: Reserved for Bakery B.
     - Flour deposit C: Shared between both bakeries.
   - **Egg Factory:** Similar to the Flour Factory, the Egg Factory produces eggs at a configurable rate and deposits them into:
     - Egg deposit A: Reserved for Bakery A.
     - Egg deposit B: Reserved for Bakery B.
     - Egg deposit C: Shared between both bakeries.
2. **Bakeries:**

   The bakeries use ingredients from the deposits to produce baked goods. Each bakery has its own rules for ingredient access and the types of products it creates.
   - **Bakery A:** Produces baguettes using flour and eggs. Each baguette requires 1 unit of flour and 2 units of eggs, yielding 5 baguettes per production cycle.
     - Primary Ingredient Access: Bakery A first attempts to use ingredients from Deposit A.
     - Fallback Access: If Deposit A is empty or insufficient, Bakery A may access Deposit C.

     All baguettes produced are placed into a shared Baguette Deposit, which is accessible to individual customers (not restaurants).
   - **Bakery B:** Produces petit-pain (small bread rolls) using flour and eggs. Each production cycle requires 1 unit of flour and 2 units of eggs, yielding 4 petit-pain per cycle.
     - Primary Ingredient Access: Bakery B first attempts to use ingredients from Deposit B.
     - Fallback Access: If Deposit B is empty or insufficient, Bakery B may access Deposit C.

     The Petit-pain produced are distributed as follows:
     - Deposit 1: Accessible to regular customers (individuals).
     - Deposit 2: Accessible to restaurants that serve petit-pain.
3. **Final Consumers**

   The final stage of the system involves individual customers and restaurants consuming the baked goods produced by the bakeries.

- **Individual customers (n=5):** Consume baguettes and petit-pain. Each customer acts as an independent thread.
- **Restaurants (n=2):** Consume petit-pain from Deposit 2. Each restaurant acts as an independent thread.

2.2 Production and Consumption Workflow

Factories continuously produce flour and eggs, adding them to the appropriate deposits. On the other hand, Bakeries consume flour and eggs from their respective deposits to produce baked goods. If their primary deposits are empty, they access the shared backup deposit (Deposit C). Customers consume baguettes directly from the shared Baguette Deposit and consume petit-pain from Deposit_1 of petit-pain. Restaurants consume petit-pain from Deposit_2. Both types of consumers operate independently, simulating varying consumption rates. The system maintains a live view of the current state of all deposits and production statistics. A periodic summary is displayed (on the terminal), providing a snapshot of the system's state for monitoring and debugging purposes.

2.3 Your Task

You are given the implementation of the system without proper synchronization. Your task is to introduce the necessary synchronization using mutexes to ensure that the different threads do not cause an incoherent state of the system (when adding and taking from different deposits). It is important to note that you are not permitted to modify the system's logic or make changes to the existing lines of code. **You are only allowed to add synchronization lines, i.e., declarations and usages of mutexes.**
Hint: Use `pthread_mutex_t` from the `pthread` library. To compile the code, use the command: `gcc main.c -o main -lpthread`

3 Task 3 (30% of total points)

This task involves parallelizing existing sequential C programs using OpenMP and POSIX threads.

3.1 Part 1: OpenMP-Based Parallelization

You are provided with three small C programs, and your objective is to insert the appropriate OpenMP pragmas to parallelize them. **The regions of the program that should execute in parallel are indicated through comments** in the original source code of the programs. To submit your solution, please fill in the corresponding fields in the file `task_3/part_1/answers.txt`.
To compile and run both the original and parallelized programs, execute the following command (on Linux): `gcc -fopenmp my_file.c -o my_prog`, where `my_file.c` represents the name of the file to be compiled, and `my_prog` represents the desired name for the compiled file.

3.2 Part 2: Manual Thread-Based Parallelization

In another scenario, we suppose that unfortunately our compiler does not support OpenMP and its directives. Instead, we have to manually implement parallelization of for-loop regions marked in the code.
To do that, for each program of Part 1, we define a structure called `ThreadData`, which contains the arguments passed to a thread participating in parallelizing a for loop (e.g, see the below code snippet, lines 1–7).
After that, we define an unimplemented function to be executed in a thread. For example, as shown in the code snippet below, the function `thread_multiply` (lines 9–13) will execute the portion of the work assigned to a single thread. In our case, it means executing a portion of the for loop that we want to parallelize. For the three given programs, the thread function would always have a name of the form `thread_<something>`.
Finally, we define another function (lines 15–23), which contains the code that spawns the threads, waits for the results and, if applicable, aggregates the results. This function always has a name of the form `parallel_for_<something>`.

Excerpts from task_3/part_2/array_multiplication.c

```c
typedef struct {
        float *array1;
        float *array2;
        float *result;
        int start;
        int end;
} ThreadData;

void *thread_multiply(void *arg) {
        ThreadData *data = (ThreadData *)arg;
        // implement the rest here, i.e, the for loop
        return NULL;
}

void parallel_for_multiply(float *array1, float *array2, float *result,
                           int size, int num_threads) {
        pthread_t threads[num_threads];
        ThreadData thread_data[num_threads];

        // fill in data structure for each thread
        // create the thread with respective workload
        // join the threads in the end and, if necessary, aggregate results
}
```

Your tasks: For each program of Part 1, we suggest a new implementation using threads, which can be found in the "part_2" folder. The ThreadData structure is already implemented, as well as the headers (plus few lines) of the two additional functions thread_<something> and parallel_for_<something>. **You need to implement the body of those two functions.**
The main function is already implemented and calls parallel_for_<something>. That is, if your implementation of the two functions is complete and correct, your code should run and produce the same results as in Part 1. To compile your code, you need to run the command: gcc my_program.c -o my_program -lpthread

4 Task 4 (15% of total points)

This task focuses on using promises and async/await to handle long-running functions. The task_4 folder contains multiple files, each containing the implementation of a function that currently uses callbacks and/or promises. Your objective is to transform these functions as follows:

- For program_1: Transform the code inside uploadFilesOld to use promises instead of callbacks. Refer to the example computeSum given in the lecture for help. Two test cases are given at the end of the file. Look for a TODO comment which indicates where you should write your implementation.
- For program_2: Transform the code inside authenticateUserOld to use async/await with promises instead of callbacks. Again, refer to the example given in the lecture for help. Two test cases are given at the end of the file. Look for a TODO comment which indicates where you should write your implementation.
- For program_3: Transform the code inside getWeatherOld to use promises instead of callbacks. Refer to the example computeSum given in the lecture for help. Two test cases are given at the end of the file. Look for a TODO comment which indicates where you should write your implementation.

Please refer to the comments and instructions provided within each file in the task_4 folder for more details and guidance.