# CP386: Assignment 4 – Fall 2024

Due on November 18, 2024

Before 11:59 PM

It is a group (of two) assignment to practice the concept of multi-threading, process synchronization, and some related programming concepts.

## General Instructions:

- For this assignment, you must use C language syntax. Your code must compile using make without errors. You will be provided with a Makefile and instructions on using it.
- Test your program thoroughly with the GCC compiler in a Linux environment.
- If your code does not compile, then you will score zero. Therefore, ensure you have removed all syntax errors from your code.
- GitHub Classroom-based repository will be used to track your work and its split with your partner daily. Follow the steps described at the end of the document to create your repository.
- Gradescope platform would be used to upload the assignment file(s) for grading. The link to the Gradescope assignment is available on Myls course page. The link to the Gradescope assignment is available on Myls course page. For submission, connect the GitHub Classroom repository and select the branch master to submit on Gradescope. Ensure your file name is as suggested in the assignment; using a different name may score Zero. If working in the group, make sure only one person from the group is uploading the files to Gradescope; if both have submitted, your files will be flagged for plagiarism. Kindly upload the files with caution.
- Please note that the submitted code will be checked for plagiarism. By submitting the code file(s), you would confirm that you have not received unauthorized assistance in preparing the assignment. You also ensure that you are aware of course policies for submitted work.
- Marks will be deducted for any questions where these requirements are not met.
- Multiple attempts will be allowed, but only your last submission before the deadline will be graded. We reserve the right to take off points for not following directions.

## Question 1

Assume that a finite number of resources of a single resource type must be managed. Processes may ask for several of these resources and will return them once finished. As an example, many commercial software packages provide a given number of licenses, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such a request will be granted only when an existing license holder terminates the application and a license is returned.

Write a C program that manages a finite number of instances of an available resource and its allocation to threads. The maximum number of resources, the number of available resources, and the threads are declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
#define NUM_THREADS 5
```

The program will create the threads and each thread will call a function `void* thread_function (THREAD_NUMBER)`, which will invoke a function decrease_count() to obtain a resource and if succeeded, go to sleep for 1 second, otherwise print "Thread NUMBER could not acquire enough resources".

```
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int thread_number,int count) {
  if (available_resources < count)
    return -1;
  else {
    available_resources -= count;
    printf("The thread %d has acquired %d resources and %d more resources are
available.\n", thread_number, count, available_resources);
    return 0;
  }
}
```

After 1s sleep, the thread wants to return the resource by calling the increase_count() function:

```
/* increase available_resources by count */
int increase_count(int thread_number, int count) {
  available_resources += count
  printf("The thread %d has released %d resources and %d resources are now
available.\n", thread_number, count, available_resources);
  return 0;
}
```

The thread will exist after printing appropriate messages, and parent process will wait for the joining of all the threads and once all threads are finished. It will print a message and exit.

```
printf("All threads have finished execution. Available resources: %d\n",
available_resources);
```

Now, this is a problem of synchronization, and you have to identify the critical section where a race condition can occur and generate inconsistent results. When multiple threads execute in critical sections, their results vary according to the order they execute. Race conditions can be avoided when critical sections are treated as atomic instructions. The prevention of race conditions can be achieved through proper thread synchronization using mutex locks.

The mutex locks represent the fundamental synchronization technique used with Pthread. A mutex lock is used to protect critical sections of code that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section. Fix the race condition by using Pthread mutex locks (described in Section 7.3.1).

The following image shows the one inconsistent output of the program without mutex lock:

```
The thread 0 has acquired, 1 resources and 4 more resources are available.
The thread 2 has acquired, 1 resources and 2 more resources are available.
The thread 1 has acquired, 1 resources and 3 more resources are available.
The thread 4 has acquired, 1 resources and 0 more resources are available.
The thread 3 has acquired, 1 resources and 1 more resources are available.
The thread 0 has released, 1 resources and 1 resources are now available.
The thread 1 has released, 1 resources and 2 resources are now available.
The thread 4 has released, 1 resources and 3 resources are now available.
The thread 2 has released, 1 resources and 1 resources are now available.
The thread 3 has released, 1 resources and 4 resources are now available.
All threads have finished execution. Available resources: 4
```

The image below shows the expected output of the fixed code with mutex locks, although the order of the execution may not match, the race condition must be fixed in your code:

```
The thread 0 has acquired, 1 resources and 4 more resources are available.
The thread 2 has acquired, 1 resources and 2 more resources are available.
The thread 1 has acquired, 1 resources and 3 more resources are available.
The thread 3 has acquired, 1 resources and 1 more resources are available.
The thread 4 has acquired, 1 resources and 0 more resources are available.
The thread 1 has released, 1 resources and 2 resources are now available.
The thread 2 has released, 1 resources and 3 resources are now available.
The thread 0 has released, 1 resources and 2 resources are now available.
The thread 3 has released, 1 resources and 4 resources are now available.
The thread 4 has released, 1 resources and 5 resources are now available.
All threads have finished execution. Available resources: 5
```

Note: When submitting a source code file for this question, name it like this:

- `resource_management.c`

Question 2

In this question, a process will create multiple threads at different times. These threads may have different `start_time` but there is no lifetime. Each thread, after its creation, runs a small critical section and then terminates. All threads perform the same action/code. Most of the code, such as reading the input file, creating the threads, etc., is provided. Your task is to implement the following synchronization logic with the help of POSIX semaphores:

- Only one thread can be in its critical section at any time in this process.
- The first thread, in terms of creation time, enters first in its critical section.
- After those threads are permitted to perform their critical section based on their ID.
    - Threads are given IDs in the format `txy` where `x` and `y` are digits (0-9). Thread IDs are unique. Threads may have same or different start_time. Thread entries in the input file can be in any order.
    - The "`y`" in thread IDs thus will either be an even digit or odd digit.
    - After the first thread, the next thread that will be permitted to perform its critical section must be the one in which "`y`" is different i.e. if "`y`" was even in first thread then in the next it must be odd or vice versa.

    - For the rest of the process, you must follow the same scheme i.e. two threads with odd "`y`" or even "`y`" can not perform critical sections simultaneously.
- Since synchronization may lead to deadlock or starvation, you must make sure that your solution is deadlock free i.e. your program must terminate successfully, and all the threads must perform their critical section.
- One extended form of starvation will be that towards the end, we have all odd or all even processes left, and they are all locked. Once the process reaches to that stage, you must let them perform their critical section to avoid starvation. However, you must make sure that there are no other threads coming in future which could help avoid this situation.

Description of Question 1:

For this Question, you are provided a skeleton code in a file `sample_code_skeleton_thread_synchronization.c`. Some functions are completely implemented, and some are partially implemented. Additionally, you can write your own functions if required. Complete the program as per following details so that we can have functionality as described in the synopsis above. Write all the code in single C file:

1. The code provided reads the content of file for you and populate the threads information in a dynamic array of type `struct thread`. You may some more members to this data structure. If you want to initialize those members, then you can possibly do that during the file read.
2. The `main()` function already contains the code to create and invoke threads. However, there is no synchronization logic added to it. If required, you will add some suitable code in the while loops to perform the tasks required.
3. The `threadRun()` function also contains the code that a thread must run. However, again the synchronization logic is missing. Add the suitable code before and after the critical section.
4. You will need to create and use POSIX semaphore(s) to implement the required logic.
5. The image below shows the expected output for the sample input file provided with this assignment (the code should be correct as long as it follows the rule odd/even number of threads, but it is best to try to get the output mentioned below):

```
[1] New Thread with ID t00 is started.
[1] Thread t00 is in its critical section
[1] Thread with ID t00 is finished.
[2] New Thread with ID t03 is started.
[2] Thread t03 is in its critical section
[2] Thread with ID t03 is finished.
[3] New Thread with ID t07 is started.
[4] New Thread with ID t05 is started.
[5] New Thread with ID t02 is started.
[5] Thread t02 is in its critical section
[5] Thread with ID t02 is finished.
[5] Thread t07 is in its critical section
[5] Thread with ID t07 is finished.
[20] New Thread with ID t01 is started.
[20] Thread t05 is in its critical section
[20] Thread with ID t05 is finished.
[20] Thread t01 is in its critical section
[20] Thread with ID t01 is finished.
```

Note: When submitting source code file for this question, name it like:

- `thread_synchronization.c`

## GitHub Repository Creation Instructions:

We will use a GitHub Classroom-based repository to keep track of the assignment work. To submit a group assignment project on GitHub Classroom, follow these steps:

- Make sure that each member of the group has a GitHub account.
- Use the link https://classroom.github.com/a/h4wCNvnE to access the assignment.
- Once you have accessed the assignment, you will see a green button that says, "Accept this assignment." Click on the button to create a repository for your group. Carefully select your name.
- Next, you will be prompted to choose a team for your group. If your team already exists, select it. Otherwise, click the "Create a new team" button to create a new team for your group. Include your teammate in the new team (only one of you will be creating the team). The team's name must be your first name and your teammate's first name, e.g., “team-firstname1-firstname2”. Only one of you will create a team (named as mentioned), and the other will join it. Caution: Do not create a random name of your team.
- After creating the repository (by default, Private and keep it as is), each group member must clone it to their local machine using Git. To do this, navigate to the repository's page and click the green "Code" button. Copy the HTTPS link and use it to clone the repository. You can use GitHub Desktop on your local OS to manage the GitHub repository better.
- Work together to complete the assignment, ensuring everyone contributes and regularly pushing your work onto GitHub. Your contributions and division of the work will be graded based on the repo activities. If your GitHub account is missing from the commits/contributions list, you will get zero for the assignment.
- Keep pushing your changes daily to GitHub so I can check your progress. Once the assignment is complete, push the changes to the repository using Git.
- When you are ready to submit the assignment, navigate to the assignment page on Gradescope and click the "Submit assignment" button. This will prompt you to select the repository that you want to submit. Select the repository that your group created.
- Now, confirm that you want to submit the assignment. You will be asked to provide a comment explaining your submission. Provide relevant details and click on the "Submit assignment" button.
- In the final step, visit Gradescope and select the project assignment. It will ask you to connect your GitHub Account. Once authorized, you can select the repository you were working with and select the branch master to submit on Gradescope. Do not forget to add your assignment partner to the Gradescope Assignment.
- That's it! Your group assignment has now been submitted to GitHub Classroom. I and marker will be able to review your submission and provide feedback.