Operating Systems Project2 Report

Thread-based Process Simulation and Synchronization

Course: 4320/6320 – Operating Systems (Fall 2025)

Student: Yangyang Jiang

Github: `https://github.com/techyangj/operation-system-project-2`

1 Introduction

This project extends Project 1 from CPU scheduling to **multithreading and synchronization** in C language.

The goals are:

- Turn each process from `processes.txt` into a thread and simulate its CPU burst.
- Implement one classic synchronization problem using locks.
- Print clear thread activity messages to understand concurrent behavior.

I chose the **Dining Philosophers** problem for the synchronization part and implemented both parts in a single C program.

2 Implementation

2.1 Input and Data Structures

The program reuses the Project 1 input file:

```
PID  Arrival_Time  Burst_Time  Priority
1    0             5           2
2    2             3           1
3    4             2           3
```

Each line is stored in a `Process` structure:

```
typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
} Process;
```

2.2 Step 1: Processes as Threads

The `run_process_simulation()`:

1. Reads all processes from the file.
2. Creates one `pthread_t` per process with `pthread_create()`.
3. Waits for all threads using `pthread_join()`.

This part shows how multiple logical processes can run concurrently as threads, but does not use any locks because the threads do not share data.

2.3 Step 2.1: Dining Philosophers with Mutexes

For synchronization, the program implements the Dining Philosophers problem with 5 philosophers and 5 forks.

Model. Each philosopher is a thread. Forks are represented as an array of mutexes:

```
#define NUM_PHILOSOPHERS 5
pthread_mutex_t forks_mutex[NUM_PHILOSOPHERS];
```

Philosopher i (1–5) owns forks $(i - 1)$ and $(i \bmod 5)$ in the array.

Thread behavior. Each philosopher repeats the following several times:

1. Print `[Philosopher i] Started.` (once at the beginning).
2. **Thinking**: print `Thinking...` and `usleep()`.
3. Print `Waiting for forks...`.
4. Lock two fork mutexes, print which forks are picked up:

   ```
   [Philosopher 2] Picked up fork 1 and 2
   ```

5. Print `Eating...` and `usleep()` to simulate eating.
6. Unlock the forks and print `Released forks`.

These messages match the example in the project handout and clearly show when a philosopher is waiting, acquiring locks, and releasing them.

Deadlock avoidance. If every philosopher always picked up the left fork first and then the right fork, the system could deadlock. To avoid this, the program uses a simple ordering rule:

- For forks with indices `left` and `right`, each philosopher always locks `min(left,right)` first and `max(left,right)` second.

This global ordering removes the circular-wait condition, so all philosophers eventually get both forks and finish eating.

2.4 Compilation

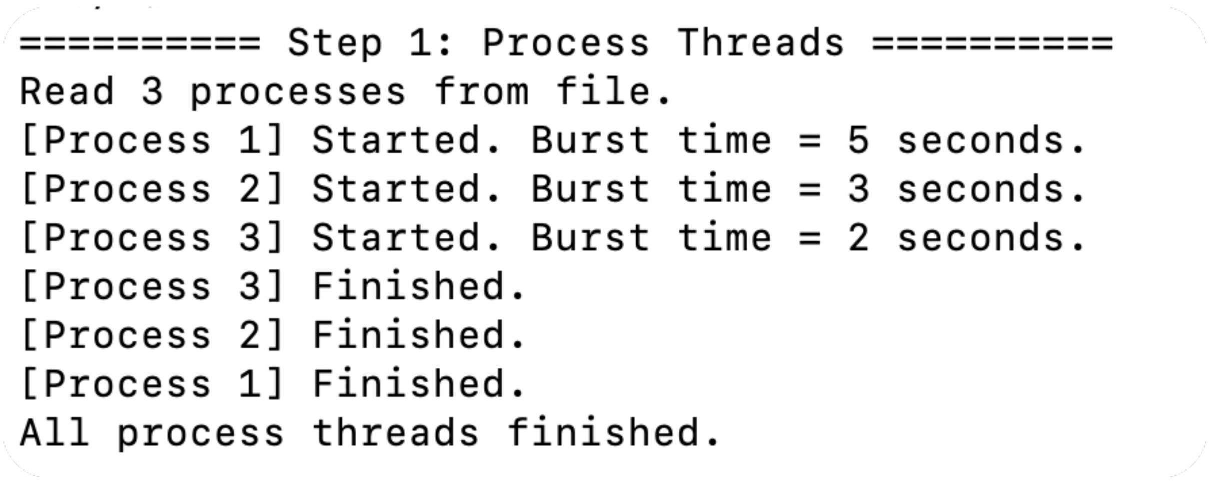
The code is compiled and run from the macOS terminal:

```
gcc os_project.c -o os_project
```

3 Results

3.1 Process Thread Simulation

A screenshot of this part is included as Figure 1.

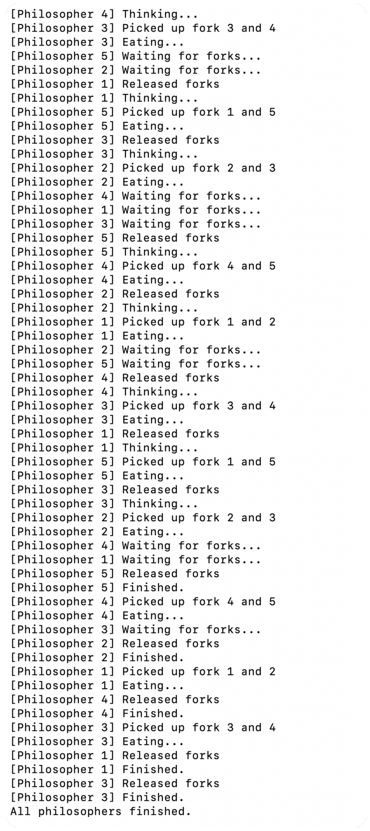


```
========== Step 1: Process Threads ==========
Read 3 processes from file.
[Process 1] Started. Burst time = 5 seconds.
[Process 2] Started. Burst time = 3 seconds.
[Process 3] Started. Burst time = 2 seconds.
[Process 3] Finished.
[Process 2] Finished.
[Process 1] Finished.
All process threads finished.
```

Figure 1: Process threads simulating CPU bursts using `sleep()`.

3.2 Dining Philosophers Activity

A screenshot of the philosopher activity is shown in Figure 2.



```
[Philosopher 4] Thinking...
[Philosopher 3] Picked up fork 3 and 4
[Philosopher 3] Eating...
[Philosopher 5] Waiting for forks...
[Philosopher 2] Waiting for forks...
[Philosopher 1] Released forks
[Philosopher 1] Thinking...
[Philosopher 5] Picked up fork 1 and 5
[Philosopher 5] Eating...
[Philosopher 3] Released forks
[Philosopher 3] Thinking...
[Philosopher 2] Picked up fork 2 and 3
[Philosopher 2] Eating...
[Philosopher 4] Waiting for forks...
[Philosopher 1] Waiting for forks...
[Philosopher 3] Waiting for forks...
[Philosopher 5] Released forks
[Philosopher 5] Thinking...
[Philosopher 4] Picked up fork 4 and 5
[Philosopher 4] Eating...
[Philosopher 2] Released forks
[Philosopher 2] Thinking...
[Philosopher 1] Picked up fork 1 and 2
[Philosopher 1] Eating...
[Philosopher 2] Waiting for forks...
[Philosopher 5] Waiting for forks...
[Philosopher 4] Released forks
[Philosopher 4] Thinking...
[Philosopher 3] Picked up fork 3 and 4
[Philosopher 3] Eating...
[Philosopher 1] Released forks
[Philosopher 1] Thinking...
[Philosopher 5] Picked up fork 1 and 5
[Philosopher 5] Eating...
[Philosopher 3] Released forks
[Philosopher 3] Thinking...
[Philosopher 2] Picked up fork 2 and 3
[Philosopher 2] Eating...
[Philosopher 4] Waiting for forks...
[Philosopher 1] Waiting for forks...
[Philosopher 5] Released forks
[Philosopher 5] Finished.
[Philosopher 4] Picked up fork 4 and 5
[Philosopher 4] Eating...
[Philosopher 3] Waiting for forks...
[Philosopher 2] Released forks
[Philosopher 2] Finished.
[Philosopher 1] Picked up fork 1 and 2
[Philosopher 1] Eating...
[Philosopher 4] Released forks
[Philosopher 4] Finished.
[Philosopher 3] Picked up fork 3 and 4
[Philosopher 3] Eating...
[Philosopher 1] Released forks
[Philosopher 1] Finished.
[Philosopher 3] Released forks
[Philosopher 3] Finished.
All philosophers finished.
```

Figure 2: Thread activity for the Dining Philosophers synchronization.

4 Conclusion

This project connects two important OS topics: process scheduling (from Project 1) and thread-level concurrency.

In Step 1, reading processes.txt and turning each entry into a thread shows a simple way to simulate process execution. Although the simulation uses only sleep(), it reveals that threads can start and finish in an interleaved order and that execution is non-deterministic.

In Step 2.1, the Dining Philosophers implementation demonstrates how to use mutexes to protect shared resources and how an ordering strategy can avoid deadlock. The printed activity lines such as “Waiting for forks”, “Picked up fork X and Y”, and “Released forks” make the behavior of the synchronization algorithm easy to observe and debug.

Overall, this project helped me practice:

- Creating and joining threads with `pthread`.
- Using mutexes to guard shared data.
- Designing a simple but effective deadlock-avoidance scheme.