COMP 340: Operating Systems
Spring 2024

Project 2: Dining Philosophers Experiment

Assignment: In this exercise, you will evaluate different implementations of the dining philosophers' problem. The two implementations are:

- N philosophers with N chopsticks using semaphores/condition variables and mutex locks
- N philosophers with N+1 chopsticks using semaphores/condition variables and mutex locks

Task 1: N philosophers with N chopsticks using semaphores/condition variables and mutex locks [50 points]

Start by implementing the dining philosophers' problem in C. For reference, read the pseudocode in Figure 5.18 on page 228 in the textbook. The program should start by creating N threads, each representing a philosopher. The parent thread waits for the philosopher threads to finish executing.

The philosopher starts by thinking for a random amount of time. Thus, he starts in the thinking state. Once done thinking, the philosopher changes his state to hungry and attempts to pick up the chopsticks. When successful, he changes his state to eating and eats for a random amount of time. Once done eating, he returns his chopsticks and goes back to the thinking state.

Each philosopher thread should go through the thinking-hungry-eating cycle five times. Once all the philosopher threads are done executing, the parent thread should exit.

For Task 1, implement the synchronization between the philosopher threads using semaphores or conditional variables and mutex locks.

As part of this assignment, a couple sample random number lists are attached. Your program should read a file, the name of the file should be inputted as a command line argument. The numbers stored in the file need to be parsed and saved into an array. Then, whenever a random number is needed, the program should read the next value in the array.

Task 2: N philosophers with N+1 chopsticks using semaphores/condition variables and mutex locks [25 points]

Modify the code you implemented in Task 1 so that the philosophers share an additional chopstick in the middle of the table. In this task, a philosopher can eat whenever the chopsticks on the left and on the right are available, or whenever either the chopstick on the left or the chopstick on the right is available and the middle chopstick is available.

The implementation of the additional chopstick is left up to you. You can use either semaphores or mutex locks or condition variables to synchronize access to the additional chopstick.

Task 3: Analysis [25 points]

In this task, compare the two different implementations using the wait time as a fairness metric. A lower wait time implies that the implementation was fairer. A longer wait time implies that the implementation was less fair. We define the wait time as the time elapsed between:

- The time at which the philosopher changed his state to hungry
  and
- The time at which the philosopher started eating

Follow the following steps:

1- Modify the code you implemented in Tasks 1-2 so that it measures the amount of time between when the philosopher changes to hungry state and when he starts eating. You can use any library of your choosing to measure time. Below is a suggestion to calculate the wait time in milliseconds:

```
#include <sys/time.h>

…

struct timeval  before, after;

double waitTime;

gettimeofday(&before, NULL);

…

gettimeofday(&after, NULL);

waitTime =
(double)(after.tv_usec-before.tv_usec)/1000 +

            (double)(after.tv_sec-
before.tv_sec)*1000;
```

2- Set the number of philosophers to 5. Run each task 5 times and collect the average wait time and the maximum wait time for each task.

3- Set the number of philosophers to 7. Run each task 5 times and collect the average wait time and the maximum wait time for each task.

4- Set the number of philosophers to 10. Run each task 5 times and collect the average wait time and the maximum wait time for each task.

5- Report the numbers you collected in steps 2, 3 and 4 in a table and write a couple of paragraphs to analyze the results of this experiment. In your analysis, answer the following:

a. What effect did having an additional chopstick have on the wait time in each implementation?
b. What did you learn from this experiment?

To help you manage the complexity of this assignment, you are provided with a header file dp.h that contains the following:

- `NUMBER:` the number of philosophers
- `MAX_SLEEP_TIME:` maximum amount of time the philosopher spends thinking or eating.
- `enum {THINKING, HUNGRY, EATING} state[NUMBER]:` each philosopher goes between thinking state, hungry state and eating state. A hungry philosopher is seeking to acquire chopsticks in order to eat.
- `int thread_id[NUMBER]:` array containing the ID numbers of the philosophers.
- `void *philosopher(void *param):` this is a function that is called whenever the philosopher thread is created. It is used to simulate the lifetime of a philosopher. The passed parameter should be cast as an integer to represent the ID number of the philosopher.
- `void pickup_chopsticks(int number):` this is the function that the philosopher whose ID is number uses to pick up his chopsticks.
- `void return_chopsticks(int number):` this is the function that the philosopher whose ID is number uses to return his chopsticks.
- `int get_next_number():` this is the function that is used to get the next random number from the array
- `pthread_mutex_t mutex_rand`: mutex lock to use in order to protect the order of random numbers
- MAX_LENGTH: the maximum number of random numbers
- `int rand_position`: the position of the next random number
- `int rand_numbers[MAX_LENGTH]:` the array holding the list of random numbers
- `sem_t sem_vars[NUMBER]:` semaphore variables to use for synchronization
- `pthread_mutex_t mutex_lock:` mutex lock to use for protecting critical sections

For tasks 2 and 3, you need to make changes to the header file to accommodate the extra chopstick and the analysis.

Rubric: To receive credit, submit a `.zip` file via mygcc containing the following items by the beginning of class on due date listed above:

1. **Source code:** Submit all of your code, including any non-standard headers or source files on which your code depends.

   Your code must compile, link, and execute in the Linux environment using the class VMs: no compile equals no credit.

   Modularity is required in the design of your program. Use good software design and engineering techniques, such as procedural abstraction, object-oriented programming, and separation of interfaces/implementations (as appropriate), to increase the extensibility, maintainability, and readability of your code.

   In addition, you should include a comments section at the beginning of each of your files that provides information about the file and its intended purpose. For example,

```
/*
  Author:  He (David) Zhang
  Course:  COMP 340, Operating Systems
  Date:    1 November 2021
  Description:   This file implements the
                 functionality required for
                 Program 1, Task 1.
  Compile with:  gcc -o task1 task1.c
  Run with:      ./task1

*/
```

   Your comments must include the commands necessary to compile the code and execute the program, as illustrated above.

   The following criteria will be used to grade your submission:

   - Does the code compile?
   - Does the code function according to the problem specification?
   - Is there an appropriate comments section at the beginning of each file (similar to the one shown above)?
   - Is the code readable and well-formatted? Is it well-documented and clear?

   The points available for each task are distributed according to the following weights:

Correctness:	95%

Each minor error:	-10%
Each major error:	-40%
Comments:	5%

A program that does not compile or link will not be graded.

2. **Analysis file:** submit a word or OneNote document that contains the analysis you wrote in Task 3.

Extensions will not be granted for technology-related issues. Leave yourself enough time to complete the assignment, submit the assignment using mygcc, and contact the instructor if you run into problems.

Project Policies:

- Assignments must be submitted electronically via my.gcc. Be sure to upload your files correctly the first time.
- This project is a group project. Every student needs to work as a part of a 2-people team. Students are expected to keep the same teams as project 1.
- Each team must submit a report and each student needs to submit an independent source code.
- 20% of the grade will be weighed with the peer evaluation. Students are expected to turn in the peer evaluation form posted on mygcc at the end of the semester.