Report on Solutions for Part 1 and Part 2

OPERATING SYSTEMS, DV1697/DV1698

Felix Beniamin Cenusa, Yamen Chams

21-11-2024

---

1. Introduction

This lab is all about understanding how processes and threads work, including how they communicate and stay in sync. It also dives into how memory systems, especially virtual memory and page replacement, affect performance.

The compressed file submission (source code) will be submitted alongside this report.

---

2. Detailed Implementation

2.1 Part 1 – Task Solutions

- **Task 7**:
  - We added a new variable called squaredId in threadArgs
  - We squared the ID of each child that is made and then we call pthread_join after creating all the threads to wait for all of them to be done and to get what they return (the squaredID) and print the squaredID.
- **Task 9**:
  - We create x amount of threads with pthread_create again but this time the child calls the do1000Transactions, the problem is that if they all run at the same time, the data will be corrupted or unreliable, so we have to edit the data one at the time. We do this the same way that a start transaction in SQL works, we lock the data, then we edit it, then when we are done we unlock it. This ensures integrity and atomicity. We chose to do this with semaphores and we used sem_init to initialize the semaphores, sem_wait to “lock” the data and sem_post to unlock it when it's done. This still does not “lock” the do1000Transactions function from being called, so we must not call it anywhere else other than with semaphores, but since we don't and this isn't production code, the code works perfectly fine.

- **Task 11**:
    - We can guarantee that no deadlock occurs in a lot of different ways. We were thinking to make every professor / philosopher take the left fork / chopstick first and then the right then eat then think EXCEPT the last processor / philosopher which takes the right then left fork / chopstick, but in the end we made a different implementation to experiment with different options. Our implementation has the professors / philosophers wait until BOTH forks / chopsticks are free, then they lock both of them if they are both free, eat then think. This way there are no deadlocks since deadlocks only happen when x is waiting for y and y is waiting for x but in this method since this is not parallelized picking up both forks / chopsticks at the same time ensures no deadlocks happen. We were not asked to make sure starvation does not happen, for that we would need to implement more code. We removed the “take left chopstick” and thinking right after to make it have no deadlocks.
- **Task 13**
    - The parallel program took 2.9 seconds to run (matmulseq13) and the sequential one (matmulseq) took 5.1 seconds to run. This means that the parallelization sped up the process by 75% which is great.
- **Task 14**:
    - The parallel initialization and calculation program took 3 seconds to run (matmulseq14) and the sequential one (matmulseq) took 5.1 seconds to run. This means that the parallelization sped up the process by 70% which is great. We think the output of matmulseq14 is similar to matmulseq13 because the task that it parallelized (the initialization) is a linear constant task that just sets 1 to every place of the matrix which doesn't take that much time, so instead of taking time to do that, we take time to create the threads and use them which also takes some time so it evens out. Perhaps if the array size was 1000000 then it would make a difference we are not sure but in our case it gave the same result. The parallelization of the matrix multiplication however sped up the program significantly since there were a lot of steps that were parallelized which seemingly take more processing time then creating and running a thread. Similarly, if we ran x threads on a small array size like 2, the difference would not be visible with parallelization.

2.2 Part 2 – Task Solutions

- **Task 17**:
    - In this task we wrote an implementation of the FIFO algorithm where we call the file, give it how many physical pages it should have, the page size and what file it should read from (the sample file we got). We tested our code against known results that we got from the lab 1 paper and we got the same results indicating a correct implementation.
    - We implemented FIFO by just continuing a while loop as long as there are lines to read from the sample file, then for every loop we increment a number by one that is modulated by the number of physical pages. If we have 4 pages it will keep counting 1 2 3 4 1 2 3 4 1 2 3 4 and so on, this gets passed to the readFromFile function that opens the file and checks if the new number

exists already in a page, if it doesn't exist it replaces whatever index of page it got with the new number and it keeps going.
    - This algorithm is very inefficient, we get a lot of page faults.
- **Task 20**:
    - We implemented the LRU page replacement algorithm by implementing a circular linked list and when all the pages are full, the last updated page gets replaced by a new page. If a page is updated, its “timer” is reset so it is deleted from its current place and it goes to the end of the linked list. When removing the oldest page, we pop the first element.
- **Task 23**:
    - We also implemented a linked list for Optimal page replacement (Bélády’s algorithm) but we added some more functions like leastCommonInFuture. This function finds the least common element currently in the pages and it returns the one that appears latest in the future. With this value returned, we can remove it from the linked list to make space for the new page. We used linked lists in tasks 20 and 23 to avoid the issue of gaps in an array when removing an element. We get the correct results in all tasks.

---

3. Discussion and Reflections

- The tasks were interesting and it was cool to see how many threads can be ran at the same time and the sheer amount of speedup we get by doing so. It was very confusing at first however since we have never used C before only C++ so the syntax threw us off, but we got used to it after a while. We misunderstood some tasks and spent a long time doing the solution the wrong way and getting the wrong answer, but those mistakes helped us understand how threads work better. The page replacement algorithms were quite hard to begin with since we had to write the entire code, but once we had the page replacement code, changing the algorithm from FIFO to LRU to Optimal was more straightforward.
