CIS 415 Operating Systems

Project 3 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Josh Jilot

UO ID - jjilot

951863225

Report

Introduction

In this project, I was tasked to implement account and transaction processing for the Duck Bank. Initially, this only required simple file parsing and updating the account struct for each account. However, as I was tasked to implement more and more complicated multithreading and interprocess communication, this seemingly simple project quickly became far more difficult. I was required to familiarize myself with the pthreads threading library which involved an entirely new element of coding that I had never interacted with before. While the concept of multithreading is not incredibly complicated in theory (in fact I really felt that I understood it well after the midterm), it is much more complicated to implement in your own code. I had to deal with race conditions, deadlocks, memory leaks, and strange file writing bugs that I had never encountered before. This project absolutely tested my coding abilities more than anything in this class, but it also tested my overall knowledge of multithreading, interprocess communication, and file I/O in general.

Background

In order to even begin this project I initially needed to brush up on my knowledge of threads and processes. While I had extensively learned about both during lecture and midterm studying, I quickly realized that implementing these concepts into my own code would be far more difficult than I expected. I learned about the necessary thread commands in the pthreads library first, learning how to initialize mutex objects and lock or unlock them. I used this knowledge in the second part where I locked critical sections of the transaction processing function so that each worker could only modify account values when no other worker would be doing the same. Mutexes and conditional variable locks are powerful in the world of multithreading, where multiple instances of a function call are occurring at once and all of them could access variables that impact the outcome of the program. Locking these critical sections properly ensures that data is updated as expected and the number of lines or values of accounts in the resulting output files are consistent and correct.

After part 2, I had to learn about implementing barriers, which wait for a threshold to be reached in the program's operation before pausing or releasing threads. For this instance, I needed to pause all worker threads after their creation and only release them for work when all threads were ready. This situation was perfect for a barrier and I implemented one to effectively control the beginning of the program's execution. This part was also the first section in which I needed to use interprocess communication. While I felt that I understood pipes fairly well, actually implementing one opened my eyes to exactly how they work. Opening and closing the read and write ends of the pipe depending on which process I was working with (parent or auditor child) finally made sense to me after I had to type the code myself–not just read about it in a lecture slide.

Finally, in part 4 I had to implement memory mapping as interprocess communication. I never really felt likeI understood how this worked and I'm still a little unsure about it but I understand it enough to implement it. While I still don't entirely understand where exactly the shared memory is on my machine, the experience of creating my own shared memory object, accessing the data members from two different processes, and observing the results certainly lent insight to how this method of interprocess communication works. It was satisfyingly effective to be able to store information "between" processes, accessing it from either end with ease. While pipes feel more intuitive, memory mapping feels very powerful as a method of communication between a parent and child process, and I look forward to expanding upon the differences between the two in my future programming.

Implementation

When implementing the requirements of this project, I felt that there was less room for creativity than with some other projects; if you mess up multithreading, the code simply will not give you the desired results. Each critical section must be properly protected from race conditions and deadlocks or else the program will output varying results or nothing at all. However, I still found room for some unique solutions. For part 1, the implementation was very straightforward, using a switch statement to parse transaction instructions and a for loop to assign each account struct its proper values while reusing the rest of the parsed lines from the input file as an array of transaction lines. In part 2 I needed to write account updates to the pipe and the output file at the same time. As shown below, I achieved this by storing the message in one variable and writing that variable to each destination, all within mutex locks to avoid race conditions. This effectively combined what could have been two separate functions, while my auditor process waited in a while loop to write anything in the pipe directly to an output file as soon as it was available.

```c
void* update_balance(void* arg) {
    /* run by bank thread to update each account.
       return number of times updated each account */
    FILE* f_out = fopen("Output/output.txt", "w");

    pthread_mutex_lock(&account_mutex); // lock before updating balances
    for (int i = 0; i < NUM_ACCS; i++) {
        account_arr[i].balance += (account_arr[i].reward_rate * account_arr[i].transaction_tracter);
        account_arr[i].transaction_tracter = 0;

        // set time val
        time_t now = time(NULL);
        char time_str[26];
        ctime_r(&now, time_str);
        time_str[strlen(time_str) - 1] = '\0';
        // write final balance to pipe
        char message[128];
        snprintf(message, sizeof(message),
                 "Applied interest to account %s. New Balance: $%.2f. Time of Update: %s\n",
                 account_arr[i].account_number,
                 account_arr[i].balance,
                 time_str);
        write(pipe_fd[1], message, strlen(message));

        fprintf(f_out, "%i balance:  %.2f\n\n", i, account_arr[i].balance);
    }
    pthread_mutex_unlock(&account_mutex); // unlock after updating balances

    fclose(f_out);
    return NULL;
}
```

While this implementation was useful for making my code smaller and easier to read, it was the auditor function that served to make my programming life much easier. As I needed to change how often account updates were written to the pipe in part 3, the implementation of my auditor function was perfect. Instead of having to change the entire function, I didn't have to alter a single line. Instead, I just needed to change when I would write to the pipe and the auditor would automatically write whatever I had written to the output file immediately. This allowed me to much more easily alter how the auditor output worked in part 3 and sync it with the new multithreading requirements for that part as well.

```c
void auditor_process(int read_fd) {
    /* write pipe info to ledger.txt */
    FILE *ledger = fopen("Output/ledger.txt", "w");

    char buffer[256];
    ssize_t bytes_read;
    while ((bytes_read = read(read_fd, buffer, sizeof(buffer) - 1)) > 0) {
        buffer[bytes_read] = '\0';
        fprintf(ledger, "%s", buffer);
    }

    fclose(ledger);
}
```

Performance Results and Discussion

All four parts of my project compiled and executed with no memory leaks or errors as demonstrated by the Valgrind logs in each folder. Additionally, after 10000 runs each, none of the multithreading parts of the project deadlock. I also checked that each ledger.txt file and account output file contain exactly the correct number of lines after multiple runs–which they do. Every account file and ledger file (and savings account file) has the same number of lines after each run and results in the correct end balance for each account. The only part I was confused about was the section of part2 that required the bank thread to return a value using pthread_join. The only other time something like this is mentioned is at the very beginning of the project when the update_balance function mentions returning the number of times each account is updated. I was not able to figure out what these requirements were asking, and after combing the rest of the project and not seeing another mention of this requirement, I did not implement it. I believe that this should not be penalized, as it was not mentioned in any part of the instructions other than these two brief, nondescript instances that leave its implementation and purpose clouded, however I understand if I simply misinterpreted these instructions.

Conclusion

Overall, this was a very educational project. I usually find myself using the concepts I already understand to complete projects as quickly–though maybe not as effectively–as possible. For this project, I was forced to learn many new techniques which were complicated at first, but eventually began to make more sense and give me more power over my code. I know that after this project, and class in general, I will be a much more effective programmer in any class I take. Thank you for a great term and a very informative ten weeks!