

Practical time: THURSDAY 5:00pm

OPERATING SYSTEMS 200

OS200 Assignment Report

CONNOR BEARDSMORE - 15504319

5/1/2016

**Source Code**

The software solution to this assignment including all source code is included in Appendix A. The code is written in the C language and follows the C99 standard. Also attached in Appendix B is the readme file associated with the source code, explaining how to run and compile the program.

**Mutual Exclusion**

Mutual exclusion was achieved in different ways for the multiprocessing and the multithreaded solutions. The multiprocessing solution utilized three POSIX semaphores. The first represented a standard mutex lock, while the others represented full and empty flags. The following code segment indicates the basic locking sequence utilized by both the producer and consumer processes:

**Producer:**

wait(empty);

wait(mutex);

// critical section

signal(mutex);

signal(full);

**Consumer:**

wait(full);

wait(mutex);

// critical section

signal(mutex);

signal(empty);

The multithreading solution also employed the use of three locks. The first was a regular POSIX mutex lock, while the others were condition variables representing the "full" and "empty" conditions. These condition variables deny "hold and wait" by forcing a thread to give up its mutex lock while waiting for the condition. As a result, deadlock is not possible in this situation. The following code segment indicates the basic locking sequence utilized by both the producer and consumer threads:

**Producer:**

lock(mutex)

cond\_wait(full) // implicitly unlock mutex

// critical section

cond\_signal(empty)

unlock(mutex)

**Consumer:**

lock(mutex)

cond\_wait(empty) // implicitly unlock mutex

// critical section

cond\_signal(full)

unlock(mutex)

**Shared Memory**

Threads share the address space of the process they are created within and thus, no shared memory is required for threads. For the threading solution, variables were simply declared globally, so each thread could access without additional function import overheads. In the multiprocessing solution, POSIX shared memory was used via the three functions below:

*shm\_open()*

*ftruncate()*

*mmap()*

Unlike threads, child processes do not inherit their parents address space on creation. As a result, shared memory is required for both the parent and the child to access the same data concurrently. All three matrices were created in a separate shared memory block, as was the subtotal struct and the struct containing the semaphore variables. The product matrix differed from the other shared memory segments in that only one process ever writes to one specific row. Hence, no synchronization was required to read and write to the product matrix.

**Zombie Processes**

The creation of zombie processes was an issue in the multiprocessing solution. The parent was required to perform work as the consumer and thus, could not wait on its child. To avoid the prevention of zombie processes, the signal() function is called prior to forking with both the SIGCHLD and SIG\_IGN flags. Any child process that terminates after explicitly setting these flags will be immediately removed from the system and thus no zombie processes are ever formed.

The downside of this solution is that the exit status of the child process is ignored. Due to the scale of this program, it was not an issue. An alternative solution could have been to implement double forking. However, this dramatically slows down the program as double the child processes are essentially forked.

**Errors and Bugs**

I am currently unaware as to any error conditions that would cause either the multithreading or multiprocessing solution to fail. There are limits however to the maximum number of processes and threads that can physically be handled. The maximum limit of processes my program can handle is approximately 800 and the maximum limit of threads is approximately 300. This limitation is due to the available system resources of the laboratory computers. There have also been instances of the shm\_open() function failing to create shared memory segments on the ssh access to the lab computers. This however is a fault of the servers themselves as my program has no influence over this.

While there is primitive error checking throughout my program, more technical error conditions such as certain file input formats can result in unspecified behaviour.

**Testing**

Both solutions have been thoroughly tested on the laboratory computers in 314.219 and on *ssh* access to these computers via "*saeshell".* Functions were developed to print the entire contents of the matrices to enable debugging and to ensure that all files were read properly into the matrices. The subsequent table lists the types of all test files the programs were tested on prior to submission:

|  |  |  |  |
| --- | --- | --- | --- |
| File Name | Dimensions | Contents | Expected Total |
| allOnes.txt | 100 x 100 | Every element = 1 | 1,000,000 |
| exampleA  exampleB | 3 x 2  2 x 4 | Example from specifications | 402 |
| medium A  medium B | 10 x 10  10 x 10 | Randomly generated numbers, 1 - 10 | 27,132 |
| largeA  largeB | 30 x 10  10 x 30 | Randomly generated numbers, 1 - 10 | 243,546 |

In addition to the basic test files, two *bash* scripts were utilized to test a large number of possible number ranges and to ensure that no deadlock was ever reached. The scripts run both solutions on a 100 by 100 matrix filled with values of all 1. Every possible M,N and K values for 1 to 100 is run for a total of 1 million test runs. Copies of these test scripts are included in Appendix A in addition to the source code. The multithreading solution was run with *valgrind* to ensure that no memory leaks occurred during normal running or error situations.

**Sample Input and Outputs**

The section below indicates the input and output from the running of the several test files mentioned previously. For simplicity, only the last three lines of output has been shown.

*Part A*

$ make

gcc -c pmms.c -Wall -Wextra -std=c99 -lrt -pthread -D \_XOPEN\_SOURCE=500

gcc -c fileIO.c -Wall -Wextra -std=c99 -lrt -pthread -D \_XOPEN\_SOURCE=500

gcc -c pmms.o fileIO.o -o pmms -Wall -Wextra -std=c99 -lrt -pthread -D \_XOPEN\_SOURCE=500

$ ./pmms

Usage: ./pmms [MatrixA File] [MatrixB File] [M] [N] [K]

Please see README for detailed steps on how to run!

$ ./pmms allOnes.txt allOnes.txt 0 0 0

ERROR - matrix dimensions must be positive values

$ ./pmms allOnes.txt allOnes.txt 200 200 200

ERROR - not enough matrix values

$ ./pmms allOnes.txt allOnes.txt 100 100 100

Subtotal produced by process with ID 8120: 10000

Subtotal produced by process with ID 8121: 10000

Total: 1000000

$ ./pmms exampleA.txt exampleB.txt 3 2 4

Subtotal produced by process with ID 8623: 134

Subtotal produced by process with ID 8625: 206

Total: 402

$ ./pmms mediumA.txt mediumB.txt 10 10 10

Subtotal produced by process with ID 8694: 4671

Subtotal produced by process with ID 8689: 2829

Total: 27132

$ ./pmms largeA.txt largeB.txt 30 10 30

Subtotal produced by process with ID 8798: 8523

Subtotal produced by process with ID 8799: 8604

Total: 243546

*Part B*

$ make

gcc -c pmms.c -Wall -Wextra -std=c99 -lrt -pthread -D \_XOPEN\_SOURCE=500

gcc -c fileIO.c -Wall -Wextra -std=c99 -lrt -pthread -D \_XOPEN\_SOURCE=500

gcc -c pmms.o fileIO.o -o pmms -Wall -Wextra -std=c99 -lrt -pthread -D \_XOPEN\_SOURCE=500

$ ./pmms

Usage: ./pmms [MatrixA File] [MatrixB File] [M] [N] [K]

Please see README for detailed steps on how to run!

$ ./pmms allOnes.txt allOnes.txt 0 0 0

ERROR - matrix dimensions must be positive values

$ ./pmms allOnes.txt allOnes.txt 200 200 200

ERROR - not enough matrix values

$ ./pmms allOnes.txt allOnes.txt 100 100 100

Subtotal produced by thread with ID 139854480430848: 10000

Subtotal produced by thread with ID 139853857285888: 10000

Total: 1000000

$ ./pmms exampleA.txt exampleB.txt 3 2 4

Subtotal produced by thread with ID 140059911808768: 134

Subtotal produced by thread with ID 140059903416064: 206

Total: 402

$ ./pmms mediumA.txt mediumB.txt 10 10 10

Subtotal produced by thread with ID 140101363431168: 4671

Subtotal produced by thread with ID 140101463746304: 2829

Total: 27132

$ ./pmms largeA.txt largeB.txt 30 10 30

Subtotal produced by thread with ID 140287909222144: 8523

Subtotal produced by thread with ID 140287900829440: 8604

Total: 243546

**References**

Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 9th ed. Reading, MA: Addison-Wesley, 1994.

Soh, Sie Teng. "Process Synchronization." Class lecture, Operating Systems from Curtin University, Perth, Australia, April 1 2016.

Soh, Sie Teng. "Process and Threads." Class lecture, Operating Systems from Curtin University, Perth, Australia, April 1 2016.