CS 3841 – Shared Memory Matrix Multiply

1 Introduction

The purpose of this lab is to design, code, and test a program to multiply two matrices together multi-process program that makes use of shared memory for exchanging information between processes. It is intended to familiarize yourself with the system calls needed to manipulate shared memory in Linux.

Last week you created a program for multi-processed matrix multiply. The matrices themselves were read in from files and stored in a malloc'ed space that, when a fork() was executed, was copied from the parent address space to the child. The child used malloc'ed matrices to compute their portion of the matrix multiplication finally returning the result in the return code.

This week you will modify this implementation to use shared memory instead of malloc. Furthermore, the result matrix will also be stored in shared memory allowing the child processes to directly write their computation results to the result matrix. No message passing, pipes, or return codes needed!

2 References

You will need to use several system calls/functions for this lab. Some of which you've used already. Here is a short summary of what you'll need. Don't forget to check out the 'man' pages for more information.

1) You will be using the fork() system call to create child processes
2) You will also be timing your code's executing using the clock_gettime() system call
3) You will have to allocate a share memory segment using shm_open()
4) You will have to initialize the segment to the correct size using ftruncate()
5) You will have to map the shared memory into the process address space using mmap()
6) You will have to release the space at the end of your program using munmap()
7) You will have to delete the shared memory segment using shm_unlink()

3 The Exercise

The matrices will be read from a file, the same format as the previous lab. Use the same formula for matrix multiplication as described in the previous lab.

Instead of using malloc'ed storage for the 'A', 'B', and result matrix you will use shared memory in two different ways.

1) Using shm_open to create a named shared memory segment. You will have to make up your own name to use.
2) Using mmap with MAP_ANONYMOUS to create an anonymous shared memory segment

As a result, you will end up creating two programs. One that uses the named share memory and the other that uses anonymous mmap().

Once you have your two programs created, you will evaluate the performance on input matrices to determine if there is a clear advantage over using one method vs another.

4 Development

As with the previous lab, break down your development into pieces:

1. Read the matrices into shared memory
2. fork() child processes to perform the matrix multiplication
3. Child processes perform the matrix multiplication for their given row and column
4. Child processes write their computation to the shared result matrix

5. Parent prints the result

You will have to repeat this method for each "flavor" of shared memory. Don't forget to implement them both!

4.1 Read the matrix files

NOTE: You will need 3 shared memory segments for this lab:

1) One for the A matrix
2) One for the B matrix
3) One for the result matrix

The format for the matrices will be the same as the previous lab. The first row contains the number of rows and the number of columns. The remaining rows contain the values for the rows and columns. Using fscanf to read the matrix (see previous lab examples) is recommended. For example:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
	int rows, columns;
	FILE* input = fopen(argv[1], "r");
	fscanf(input, "%d", &rows);
	fscanf(input, "%d", &columns);
	for(int i = 0; i < rows; i++) {
		for(int j = 0; j < columns; j++) {
			int value;
			fscanf(input, "%d", &value);
		}
	}
	fclose(input);
}
```

It is up to you to decide how the matrices are organized in the shared memory space. You can organize it as a one-dimensional array and use index calculation to find the row and column or as a two-dimensional array. However, remember that a shared memory segment is a single contiguous address range. **Use only a single shared memory segment per matrix. Do NOT use a separate shared memory segment for each row**.

4.2 fork() child processes to perform the matrix multiplication

You will need to fork() a process for each result matrix element. As with the previous lab, you'll need to fork() in a loop, make sure you keep track of the pid for each child process because you'll need to wait for them to finish before printing the result. Since you won't know how many processes you need when the program starts, you'll need to malloc a space to store the child pids. You can use a shared memory segment for this if you want, but since the parent process is the only one using this list, a shared memory segment is not needed.

4.3 Child processes perform the matrix multiplication for their given row and column

Once you fork() the child processes you'll need to figure out how to make sure that each child process performs the matrix multiplication calculation for their given row and column.

4.4 Child processes write their computation to the shared result matrix

In the previous lab, the parent received the result from the child via a return code. However, with shared memory, we don't need to do this. The child processes can write their result directly to the shared memory space for the result matrix. Not only does the parent not need to collect results, the results themselves are no longer limited to 8 bits. This greatly extends the values you can use in your matrices. Make sure you test with larger values

4.5 Parent Prints the Result

Once the result elements are collected, printing the result is the same as the previous lab. Don't forget to print the time for the matrix multiply. Make sure the parent waits for the children to finish. Since you don't need to retrieve the return code from the child, you are welcome to have the parent wait in any way that works. Just make sure the parent waits for ALL children to finish before printing the result.

5 Additional Development Tips

- Remember that to successfully multiply two matrices the inner dimensions must match. So an *m x n* matrix can be multiplied by an *n x p* matrix, but an *m x n* matrix cannot be multiplied by a *q x p* matrix unless *n == q*. If the input matrices are not the correct size, print an error message to the user and quit.
- Don't forget about using gdb to help with debugging.
- Don't forget to munmap your mmaps! Remember that when a parent mmap's address space, the children processes also get that mmap. The child must therefore munmap the address space AND close any used file descriptors when it is done. Don't forget to munmap your mmaps!
- Don't forget to close your file descriptors! When you call shm_open you get back a file descriptor. The children inherit file descriptors after a fork(). All opened file descriptors must be closed by all processes when they are done using them (the parent too).
- Don't forget to shm_unlink your named shared memory segments! Named shared memory segments are persistent (they live beyond the life of your process). Make sure you shm_unlink() your named shared memory segments. BUT make sure you don't shm_unlink before you need to. Don't remove a shared memory segment until you are absolutely sure that nobody else is using it!

6 Testing and Debugging

There are several matrix files located provided along with this specification. You can use those to help get your programs working. You are also required to create some of your own and include those in your submission. Don't forget to test large matrices.

7 Deliverables

You will need to include all your source files, test case matrix files, and any other resources you used to complete lab. Please don't just google search for a solution, but if you do use google for any help, include a description and URL of what you used to help you.

You should ensure that this program compiles without warning (-Wall and -Wextra) prior to submitting. Using gnu90 standard is allowed (gcc default). If your implementation requires C99 or gnu99, please make that clear in the comments.

Also include a text, doc, or pdf file with a report that provides the following:

- Introduction – describe the lab in your own words
- Design – describe your thought processes in creating your solution
- Build Instructions – what must be executed (gcc, make, etc.) to build your two programs
- Analysis – Your two programs need to print out the time to perform the matrix multiplication. Use this information to evaluate the usefulness of the two types of shared memory. Is there an advantage to using anonymous shared memory vs a named shared memory segment? Be thorough. Also answer the following in your analysis
    - How does the runtime compare to using the return code (as you did in lab3), anonymous shared memory, and named shared memory?
    - How does shared memory affect the overall heap needed by the process? If you run your program using valgrind, it will print out a summary of the total heap used.

    - How does the return code, anonymous shared memory, and named shared memory compare in terms of ease of use? Is one easier to implement and/or debug than the others?
    - From your evaluation, when would you recommend using return code vs anonymous shared memory vs named shared memory? Would you always use one over the others? Can you think of a situation where one would work better?
- Conclusion – What specifically was challenging about this lab? What did you like about it? What could we do to improve it for others?

Prepare a zip file with all submitted files and upload the file to Blackboard per your instructor's instructions.