CS 3841 – Shared Memory Matrix Addition

1 Introduction

The purpose of this lab is to design, code, and test programs to add two matrices together using both a single process and multi-process program. It is intended to familiarize yourself with the system calls needed to create processes and share memory on Linux.

The first program will perform matrix addition with a single process storing the result in a matrix allocated on the heap (via malloc). In the second program, child processes will be created with fork() to perform the addition for each row in the matrix storing their result in a matrix created in shared memory.

The matrices will be read from a file. The format for the file is given later.

2 References

You will be using the fork() system call to create child processes to compute a portion of matrix addition. The children will then write their result to the correct position in a matrix stored in shared memory. You will also be timing your code's executing using the clock_gettime() system call. Before you start, it might be helpful to read the 'man' pages for these system calls.

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

Recall from linear algebra you can add two matrices using the following:

If A = $[a_{ij}]$ is an *m x n* matrix and B = $[b_{ij}]$ is an *m x n* matrix, the addition of A + B is an *m x n* matrix.

A + B = $[c_{ij}]$, where $c_{ij} = a_{ij} + b_{ij}$

The definition of matrix addition indicates adding the element of matrix A with its corresponding element in B (at the same position).

Matrix addition is commutative. So A + B does is always equal B + A.

For example:

$$A = \begin{bmatrix} 1 & 10 & 0 \\ 3 & -2 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 4 & 0 \\ 1 & 2 & 3 \end{bmatrix} \quad A + B = \begin{bmatrix} 1+1 & 10+4 & 0+0 \\ 3+1 & -2+2 & 6+3 \end{bmatrix} = \begin{bmatrix} 2 & 14 & 0 \\ 4 & 0 & 9 \end{bmatrix}$$

Your task is to create two programs that will perform matrix addition for two matrices given to you as two files and print the result. Each program must also record the amount of time it took to perform the matrix addition.

- The first program will perform the matrix addition with a single process.
- The second program will perform the matrix addition with multiple processes – one process for each row.

The format for each matrix file is:

- The first line will contain two numbers: the number of rows in the matrix followed by the number of columns

- The rest of the file will contain rows of numbers representing the numbers in the matrix

For the example given above the files for A and B would be

```
2 3             2 3
1 10 0          1 4 0
3 -2 6          1 2 3
```

All numbers in the file will be separated by spaces.

4 Development: Single Process

It's simplest to break down the development into pieces:

1. Read the matrices
2. Perform the matrix addition
3. Print the result

4.1 Read the matrix files

A simple way to read integers from a file is to use the fscanf function. This function allows you to do formatted reading from a file. We'll use this to read numbers. To use fscanf you first must open a FILE* with fopen (take a look at 'man fopen' for more information). Then, like printf for printing a number you use the "%d" format as a parameter to fscanf to read a number. Here is an example:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
	int value;
	FILE* input = fopen(argv[1], "r");
	fscanf(input, "%d", &value);
	fclose(input);
}
```

NOTE: the matrix file names will be given on the command line as arguments. For example: ./a.out matA matB Command line arguments are passed to main through the argc and argv values (see example above). The argc value tells the program the number of command line arguments, and argv is an array of character pointers for the argument strings. There is always 1 argument passed to your program (argv[0]) which contains the name of your program. So your matrix input files will be in argv[1] and argv[2]. If the user didn't provide enough arguments (e.g. argc != 3) then print an error and exit.

You can use this fscanf to read in all the numbers you need from the file. It takes care of all the white space for you. So to read your matrix file you can start with:

```
#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);
}
```

Obviously, you'll need to store the row and column values somewhere. Since the matrix can be any size (within reason), you'll need to store the matrix in memory on the heap allocating a space for it using malloc. Remember that malloc creates a contiguous space in memory on the heap for the given size. Think about how much memory you'll need to malloc to store each matrix. When storing the values in the malloc'ed space you'll also have to do some math to make sure the indexing is correct.

4.2 Adding the Matrices

When you store the matrices in the space created by malloc they are in one contiguous space in memory. So, to perform the matrix math, you'll also need to correctly compute the correct memory offset. Thankfully to access the memory at an index you can just reference it like an array:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
	// malloc a space
	int* space = malloc(sizeof(int) * 100);

	// Store data in the space
	for(int i = 0; i < 100; i++) {
		space[i] = i + 10;
	}

	// Read data from the space
	for(int i = 0; i < 100; i++) {
		printf("%d\n", space[i]);
	}
	return 0;
}
```

For this assignment, you'll just need to do the index calculation for the correct index based on the value's row and column. From there the calculation is just based on the matrix addition formula.

4.3 Print the Result

It's probably easiest to store the matrix for the addition result in its own malloc'ed space. Once you have the result, you'll need to perform the index math to print out the matrix elements to the screen. For our example, that would be:

```
2 14 0
4 0 9
```

Print out each element in the rows and columns separated by a space. Don't worry about making sure they line up. You will also need to print out the amount of time it took to perform the matrix addition. For that you will use the clock_gettime function to get the CLOCK_REALTIME for the system before and after you do the matrix addition. You only need to record the time for the addition, don't include the time it took to read the matrix files or to print the result.

5 Development: Multi-Process

Once you get your single process program working you'll follow the same steps for the multi process version, except that now you have use fork() to create a process for each row in the matrix element. Breaking this down into pieces:

1. Read the matrices
2. fork() child processes to perform the matrix addition (one per row in the result matrix)
3. Child processes perform the matrix addition for their given row
4. The child processes write their results to the result matrix stored in shared memory
5. The parent waits for all child process to finish
6. Parent prints the result

NOTE: Recall that heap space is not shared between parent and child processes. So the result matrix is to be created in shared memory. Do this by creating a named share memory segment using shm_open and mmap. Choose an appropriate name for the segment.

It is up to you to decide how the matrix is 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. Do NOT use a separate shared memory segment for each row.**

5.1 Read the matrix files

This is the same as the single process version

5.2 fork() child processes to perform the matrix addition

You will need to fork() a process for each row in the matrices. For our example above A + B has 2 rows so that requires 2 child processes. You'll need to fork() in a loop, make sure you keep track of the pid for each child process because the parent will need to wait for them to finish. 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.

5.3 Child processes perform the matrix addition for their given row

Once you fork() the child processes you'll need to figure out how to make sure that each child process performs the matrix addition calculation each element in their given row. The formula for the calculation is the same as the single process version.

5.4 Child processes write their computation to the shared result matrix

Allocated and mapped shared memory persists after a fork() system call so all child processes have access to a result matrix created in shared memory. The child processes can write their result directly to the shared memory space for the result matrix. The parent does not need to collect results.

5.5 The parent waits for all child process to finish

The parent needs to keep track of all the process ids (pids) for all child processes and perform a waitpid for each one. This will require a loop and call to waitpid. See the man page for waitpid for additional information.

5.6 Parent Prints the Result

Once the child processes have finished, printing the result is the same as the single process program. Make sure the parent waits for the children to finish otherwise the result matrix won't be correct.

6 Additional Development Tips

- Remember that to successfully add two matrices the dimensions must match. So an *m x n* matrix can be added to an *m x n* matrix, but an *m x n* matrix cannot be added to a *m x p* matrix unless *n == p*. 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.
    - NOTE: when using gdb it is helpful to compile your program with additional debug symbols included. These allow gdb to show you more information when running commands like backtrace (bt). To compile with additional debug symbols use the -g flag on gcc. For example:

      ```
      gcc -Wall -g -o myprog mysourcefile1.c mysourcefile2.c mysourcefile3.c
      ```

- Don't forget to free your mallocs! Remember that when a parent malloc's space on the heap, the child will get an EXACT COPY of that data. The child also inherits the responsibility to free the storage. Don't forget to free your mallocs!
- Using valgrind will be helpful in this lab to ensure you do not have any memory leaks.
- 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!
- When compiling your shared memory program you will need to include an additional compiler flag: -lrt. This tells the compiler to use the real-time c library where the wrapper function calls for the shared memory system calls. The in-class examples use this in the Makefile so you can see how to use it there as well. For example:

  ```
  gcc -Wall -g -o myprog mysourcefile1.c mysourcefile2.c mysourcefile3.c -lrt
  ```

  NOTE: the -lrt flag must be the LAST thing on the command line. If it is not, your program will not compile.

7 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.

8 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 (single process and multi process) need to print out the time to perform the matrix addition. Use this information to evaluate the usefulness of multi-process programming. Is there an advantage to using multi-processing for matrix addition? Be thorough. Also answer the following in your analysis
    - How does the runtime of the single process compare to the multiple process version? Faster? Slower? Why do you think this is the case? Think in terms of operating system overhead.
    - How could identifying patterns in the source matrices be used to speed up the computation?
    - 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. What conclusions can you draw from this?
- 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 Canvas per your instructor's instructions.