CS 3841 – Multi-Process Matrix Multiply

1 Introduction

The purpose of this lab is to design, code, and test programs to multiply 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 on Linux.

2 References

You will be using the fork() system call to create child processes and passing data back to the parent through the process exit status code using waitpid(). You will also be timing your code's executing using the clock_gettime() system call. The matrices will be read from a file. The format for the file is given later. Before you start, it might be helpful to read the 'man' pages for these system calls.

- `man 2 fork`
- `man 2 waitpid`
- `man 2 clock_gettime`

3 The Exercise

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

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

AB = $[c_{ij}]$, where $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj}$

The definition of matrix multiplication indicates a row-by-column multiplication. The elements in the rows of A are multiplied by the elements in the columns of B and added together.

Matrix multiplication is NOT commutative. So AB does not always equal BA.

For example:

$$A = \begin{bmatrix} 1 & 4 \\ 2 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 4 & 0 \\ 1 & 2 & 3 \end{bmatrix} \quad AB = \begin{bmatrix} 1*1+4*1 & 1*4+4*2 & 1*0+4*3 \\ 2*1+6*1 & 2*4+6*2 & 2*0+6*3 \end{bmatrix} = \begin{bmatrix} 5 & 12 & 12 \\ 8 & 20 & 18 \end{bmatrix}$$

Your task is to create two programs that will perform matrix multiplication 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 multiply.

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

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 2
1 4
2 6
```

```
2 3
1 4 0
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 multiplication
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 Multiply 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 multiply formula.

4.3 Print the Result

It's probably easiest to store the matrix for the multiplication 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:

```
5 12 12
8 20 18
```

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 multiplication. For that you will use the clock_gettime function to get the CLOCK_REALTIME for the system before and after you do the matrix multiply. You only need to record the time for the multiplication, 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 result element. Breaking this down into pieces:

1. Read the matrices
2. fork() child processes to perform the matrix multiplication
3. Child processes perform the matrix multiplication for their given row and column
4. Parent collects the results from the child processes
5. Parent prints the result

5.1 Read the matrix files

This is the same as the single process version

5.2 fork() child processes to perform the matrix multiplication

You will need to fork() a process for each result matrix element. For our example above A*B has 6 result elements (2 rows and 3 columns) so that requires 6 processes. 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 and send the result to the parent. 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 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. The formula for the calculation is the same as the single process version.

5.4 Parent collects the results from the child processes

Recall from class, that parent and child processes don't share memory. Instead, the child processes get an exact copy of parent. This works great for the malloc'ed and read in matrices. Once the parent reads in the matrices and stores them in the malloc'ed spaces, the children will get an exact copy of the data. They can perform the calculation by accessing the same variables. However, to return values to the parent, the children can't write to a variable. To send the result value to the parent we'll use the process return value. When a child process returns from main, it can set a return value that the parent process can read using the waitpid() system call. For example:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char* argv[])
{
      pid_t pid = fork();
      if(pid < 0) {
            printf("FORK FAILURE\n");
            exit(EXIT_FAILURE);
      } else if(pid == 0) {
            printf("Child process\n");
            return 10;
      } else {
            printf("Parent process\n");
            int child_status;
            waitpid(pid, &child_status, 0);
            printf("Parent got %d from the child.\n", WEXITSTATUS(child_status));
      }
}
```

Notice that we had to do a little extra to get the actual value returned from the child. The actual value set in child_status has more information in it besides the return value. To get the actual return value we need to use the WEXITSTATUS() function.

NOTE: while using this function works great for sending values from child processes to a parent, the value is limited to an 8-bit unsigned number. So, for the purposes of this lab the result of a matrix multiplication element must be between 0 and 255 (inclusive).

5.5 Parent Prints the Result

Once the result elements are collected, printing the result is the same as the single process program.

6 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 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.

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 multiplication. Use this information to evaluate the usefulness of multi process programming. Is there an advantage to using multi-processing for matrix multiplication? Be thorough. Also answer the following in your analysis
    - We used the exit status to send data from the child process to the parent. Is there a better way? How would you implement that?
    - If a matrix contains an entire row or column of zeros the resulting multiplication element is zero. How could we optimize the number of processes needed if this happens a lot in our input data?
- 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.