Project 2: Interprocess Communication and Shared Memory

This assignment will give you some practice working with the POSIX shared memory API. Your task is to simply square an MxM matrix. For example, squaring matrix A will result in the following:

$$A=\begin{bmatrix} -4 & -1 & 2 & 1 \\ 6 & 0 & -9 & 10 \\ -3 & -10 & 6 & 0 \\ -10 & 6 & 7 & 9 \end{bmatrix} \qquad A^2=\begin{bmatrix} -6 & -10 & 20 & -5 \\ -97 & 144 & 28 & 96 \\ -66 & -57 & 120 & -103 \\ -35 & -6 & 31 & 131 \end{bmatrix}$$

Another example:

$$A=\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad A^2=\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

The interesting part of this assignment is that each value in the resulting matrix will be calculated in a separate process. This means that if you have a 3x3 matrix, you will create 9 processes that will execute concurrently, one process for each of the 9 values. The parent process (the initial executable) will create the child processes. Each child process will exec a new executable program. The default name of the child executable on the disk will be **`child-matrix`**. This program will perform the actual arithmetic of multiplying a row by a column.

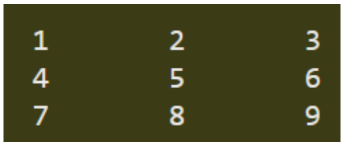
There are 5 parameters in the `args` array (plus a terminator) that will be passed to exec'd program. Examples using *execv* were shown in class and are present in the notes for you to review. You can use either *execv* or *execl*. The easiest way to format the command line arguments is to use *sprintf*. (Remember, only NUL-terminated strings can be passed to a program. You can't pass integers. They must be converted to strings.)

The parameters that will be passed to the child program are:

Arguments (NUL-terminated strings)	Notes
0 – The name of the program (to exec) on disk	This is the convention for `argv[0]`. This is supplied to the parent process on its command line.
1 – The id of the shared memory	This was returned from `shmget` in the parent process. This is how the child will be able to access the shared memory setup by the parent.
2 – The child number	This is a number from 0 to MxM – 1.This is so each child knows where in the matrix to put its result. See the diagrams on the web page.
3 –Row from the original matrix to work with	This is an integer between 0 and M – 1. The child is simply multiplying one row by one column.
4 – Column from the original matrix to work with	This is an integer between 0 and M – 1. The child is simply multiplying one row by one column.
5 – NULL	This will terminate the argument array.

The parent must setup the shared memory for the interprocess communication between the parent and child. There are 3 "pieces" of information that will be written to and read from the shared memory. The first 4 bytes of the memory are used to store the width of the array (integer). Since it's a square matrix, the width and height are both the same and we only need to specify one dimension. The next several bytes are the actual values (integers) of the matrix to square.

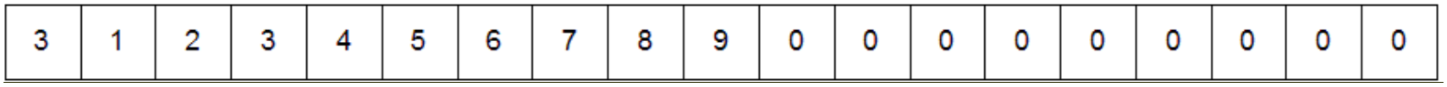
Let's suppose that the matrix is a 3x3 matrix:



```
1        2        3
4        5        6
7        8        9
```

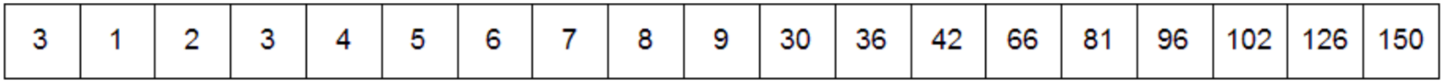
Immediately following the first 4 bytes will be 36 bytes `(9 * sizeof(int))`, representing each of the 9 integers in the matrix. Row 1 will be written first, then row 2, and finally, row 3. Immediately following the elements of the input matrix are another 36 bytes which will be used by the child processes to save the results of their calculations.

This is what the shared memory will look like after the parent fills it with the width and input matrix:



3	1	2	3	4	5	6	7	8	9	0	0	0	0	0	0	0	0	0

And after each child process has written its result to the shared memory:



3	1	2	3	4	5	6	7	8	9	30	36	42	66	81	96	102	126	150

AFTER all of the children have finished, the parent will then access the shared memory to read the values of the resulting matrix and print the matrix to the screen. **This means that the parent will do nothing while the children are performing the calculations.** The parent MUST simply block (call a wait function) while the children are busy. This goes along with the *wait* and *waitpid* functions we saw in class.

Common Pitfalls

IPC can be tricky if you've never done it before. One thing to look out for is error checking ALL of your system calls. You should be ensuring that all of your system calls are wrapped in some kind of error handling that at least checks for what the error is. You can `#include <errno.h>` to facilitate this.

A large area of consternation is that the call to **shmget** may fail, which can happen if you did not hit the corresponding **shmctl** call to release the shared memory from your previous run. To solve this problem, do the following in your WSL/Linux shell (the left side is just my linux terminal username, the right side is what you must type):

```
ehall:~/projects#     ipcs
```

The **ipcs** command will list the shared memory segments that are still active. You should be able to pick out the one you created by looking at the key, which should match the key you gave it in your program (123 corresponds to 7B in hex).

```
ehall:~/projects#     ipcrm –m shmid
```

You can then use the **ipcrm –m** command to release the shared memory, replacing **shmid** with the corresponding **shmid** from the Shared Memory Segments table.

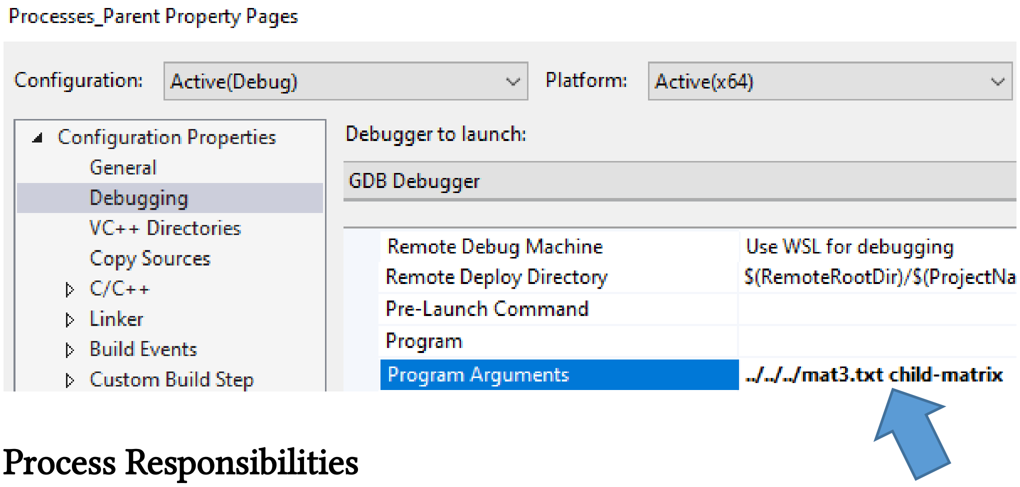
Running the Parent Process

To run the parent process, you will need to supply it with its own command line arguments, like so:

```
    ./parent-matrix input.txt child-matrix
```

Where the first parameter is the name of the text file that has the matrix in it, and the second parameter is the name of the child program to execute. **DO NOT ASSUME THAT THE NAME OF THE CHILD PROCESS EXECUTABLE WILL ALWAYS BE child-matrix**. It is totally fine if the parameter passed to the parent program is "child-matrix", but you MUST use the name provided on the command line. Hard-coding this value in your parent process will cause your program to fail when you test it on Moodle.

To facilitate this in Visual Studio, open the properties of the parent-matrix project and go to the Debugging section of Configuration Properties. You can edit the command line arguments like so, **remember to replace them with your own**:



Processes_Parent Property Pages

Configuration: Active(Debug)

Platform: Active(x64)

Debugger to launch:

GDB Debugger

Remote Debug Machine	Use WSL for debugging
Remote Deploy Directory	$(RemoteRootDir)/$(ProjectNa
Pre-Launch Command	
Program	
Program Arguments	../../../mat3.txt child-matrix

Process Responsibilities

The parent process is responsible for:

- Reading the name of the input matrix (argv[1]) and program to exec (argv[2]) from the command line arguments.
- Reading the input matrix from a file into it's non-shared memory. (Code is provided.)
- Printing the input matrix. (Code is provided.)
- Creating a shared memory area with **key of 123**.
- Writing the width of the matrix and all matrix values to the shared memory area.
- Forking all of the child processes, one for each element in the matrix.
- Waiting for all of the child processes to finish. The parent should be doing **nothing** while waiting for the children.
    - **If the parent is doing busy waiting (polling), you will receive a zero because this will defeat the entire purpose of running multiple processes in parallel. The parent should just wait (block) until a child finishes. You can wait for the children in order, or wait on any child. The choice is yours, but you must wait (block) indefinitely until a child exits.**
- Printing the resulting squared matrix **AFTER** all of the children have finished.
- Cleaning up any memory or resources that the parent needed, including the shared memory.

The child (after forking, before exec'ing) is responsible for:

- Determining which row number and column number to pass to the exec'd program.
- Setting up all of the parameters to pass to the exec'd program.
- Exec'ing (using *execl* or *execv*) the new child process.
- If the exec fails, the child must clean up any resources and print an error message. (Any message is fine.)

Once the child process successfully execs the executable program, the child's code has now been replaced with a new process. This new process is responsible for:

- Reading all of the command line parameters that were passed to it.
- Reading the width from the shared memory.
- Reading the correct row and column of the matrix from the shared memory.
- Multiplying the row and column together to get a single integer value.
- Writing the integer value to the proper location in the shared memory.
- Releasing any memory or resources needed, including the shared memory.

As in most systems programming, there is not a lot of code, but it can seem a little complex because most of you have never done this before. Most of the bugs that students have are related to passing parameters to the child process. So, when you start coding, the absolute first and only thing you should do in the child code is to simply

PRINT OUT THE ARGUMENTS! Many of you will find that you are not getting what you thought you were because you are using command line arguments incorrectly. This can save you hours of frustration.

Compiling/Running your Code

For the gcc compiler:

```
gcc -Werror -Wall -Wextra -ansi -pedantic -g -O -Wno-unused-result parent-matrix.c -o parent-matrix
gcc -Werror -Wall -Wextra -ansi -pedantic -g -O -Wno-unused-result child-matrix.c -o child-matrix
```

For the clang compiler:

```
clang -Werror -Wall -Wextra -ansi -pedantic -g -O -Wno-unused-result parent-matrix.c -o parent-matrix
clang -Werror -Wall -Wextra -ansi -pedantic -g -O -Wno-unused-result child-matrix.c -o child-matrix
```

Remember: you are building TWO DIFFERENT EXECUTABLE PROGRAMS! One is for the parent process, the other will be what the child process becomes after the exec call. If your compilation succeeded, you should have two different binaries, one for the parent process and one for the child process.

Memory Leaks (Valgrind)

Your implementation should not have memory leaks. To ensure you don't have any, you can run valgrind with the line below. Be aware that this could take a minute or two. Also of note: the bolded flag below is needed so Valgrind can trace through the forked child processes as well.

```
valgrind -q --leak-check=full --show-reachable=yes --trace-children=yes --tool=memcheck
./parent-matrix mat5.txt ./child-matrix
```

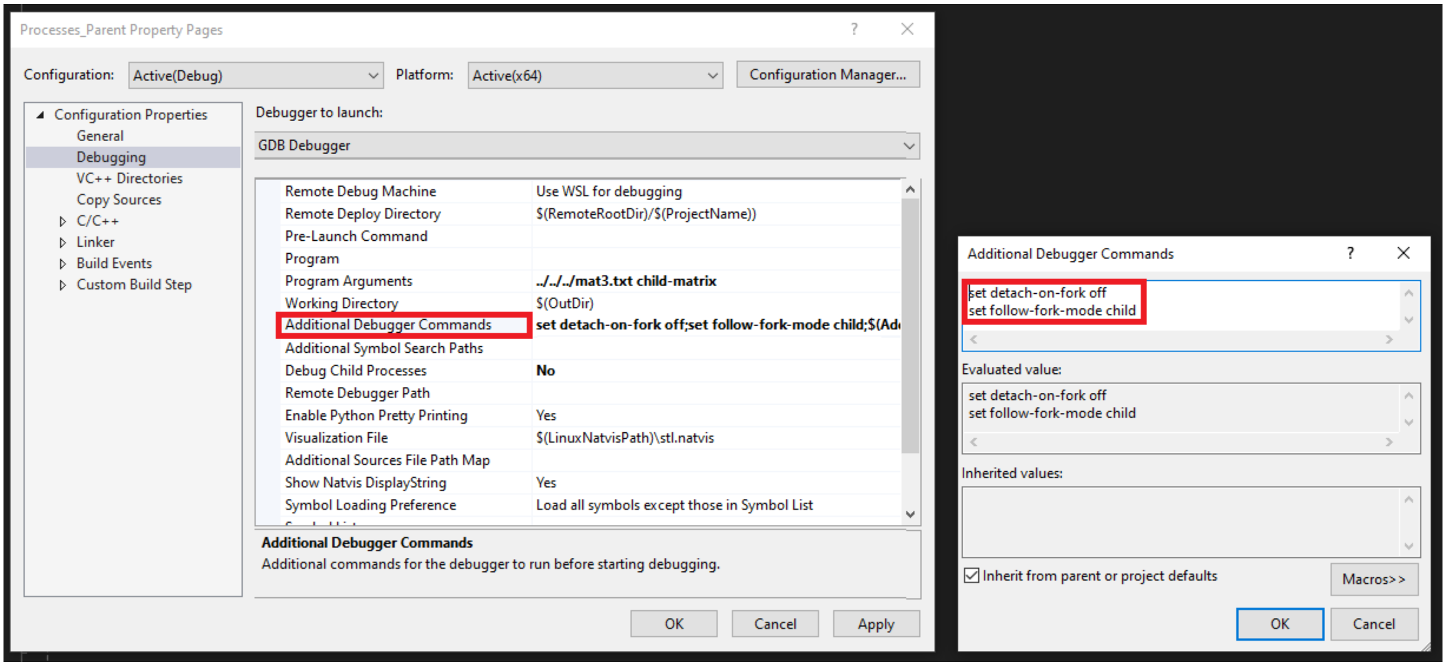
Debugging

Whenever you are debugging a program, the debugger (a separate program) is attaching itself to the running process you are trying to debug. But things get more complicated when you are debugging parent/child processes. You may find that any breakpoints that would normally hit for code running in a child process won't be hit; this is because **gdb** is not aware that it needs to attach itself to a child process when it is forked from the parent.

To alleviate this issue, use the following gdb commands while gdb is running, BEFORE the parent process calls fork:

> `set follow-fork-mode child`
> (This command allows gdb to attach to a child process once it is created)
>
> `set detach-on-fork off`
> (This command allows gdb to switch between the parent/child processes during debugging, but this must be done manually using the `inferior` command when stopped at breakpoints. If you'd like to dive into this, look at the examples in the VisualGDB.com page for detach-on-fork)

If you're using a Linux distro or the WSL terminal directly (WITHOUT VS/Code), then you can use gdb directly to debug your program. However, if you're using gdb through either Visual Studio or VS Code, debugging child processes can be a pain. In Visual Studio, you can run the above commands when the debugger starts by filling in the following property on the parent-matrix Visual Studio project (a similar solution exists for VS Code, in the launch.json file. Please ask the TAs about this solution if you're using VS Code!):



Configuration: Active(Debug)

Platform: Active(x64)

Debugger to launch: GDB Debugger

Remote Debug Machine	Use WSL for debugging
Remote Deploy Directory	$(RemoteRootDir)/$(ProjectName))
Pre-Launch Command	
Program	
Program Arguments	**../../../mat3.txt child-matrix**
Working Directory	$(OutDir)
Additional Debugger Commands	**set detach-on-fork off;set follow-fork-mode child;$(Ad**
Additional Symbol Search Paths	
Debug Child Processes	**No**
Remote Debugger Path	
Enable Python Pretty Printing	Yes
Visualization File	$(LinuxNatvisPath)\stl.natvis
Additional Sources File Path Map	
Show Natvis DisplayString	Yes
Symbol Loading Preference	Load all symbols except those in Symbol List

Additional Debugger Commands
Additional commands for the debugger to run before starting debugging.

Additional Debugger Commands:

```
set detach-on-fork off
set follow-fork-mode child
```

Evaluated value:

```
set detach-on-fork off
set follow-fork-mode child
```

Inherited values:

☑ Inherit from parent or project defaults

Testing

There are matrix files included that can be used to test your work. You can also use the provided **gen_matrix.c** file to create a program that outputs random matrix files to use as tests (note that you need to build this program yourself first). To run the matrix program to generate a test file, call it like so:

```
./gen_matrix 5 -20 20 > mat5.txt
```

The numbers above can be changed to create a different matrix. The run above will create a 5x5 matrix with values ranging from -20 to +20. The program output then redirects into a text file called **mat5.txt**.

Project Files

File	Description
parent-matrix.c	The parent process file
child-matrix.c	The child process file
tests (directory)	Test input files and corresponding output files, if you want more use gen_matrix.c
gen_matrix.c	OPTIONAL: build this into an exe that creates test matrices, you must redirect the output into a file

Deliverables/Submission

You must submit two files to the appropriate Moodle VPL assignment:

- **parent-matrix.c**
- **child-matrix.c**

When submitting to Moodle, there will be an option to evaluate your submission. Press this button to see the results of the tests (the same tests included via the driver file). If your program passes all of the tests, you should see a 100/100 grade. YOU DO NOT NEED TO DO ANYTHING AFTER YOU SEE THIS GRADE. Your submission has been accepted and your grade was recorded on Moodle. If you do not pass all of the tests, VPL will show you the diff between the expected output and your program's output. You may submit an unlimited number of times before the due date.