Alejandro Martinez
Programming 3
Program #6 - Processes & Shared Memory
Pseudocode

Before Pseudocode

Before the pseudocode, I like to dismantle completely what is being asked. Every item that Mr. Feild is asking will be shown. This first part is exactly copied from Professor Feild assignment, but it is done this way to make sure everything I do is done to the standards asked. If you would like to jump directly to the pseudocode start with the title ***Display program purpose.***

Input:

Accept: a number of arguments from the command-line – between 1 and 7 (inclusive) – each a unique integer between 0-9 (inclusive). (COMPLETED BY inputValidationFork.c)

Process:

- Request shared memory for these integers. (COMPLETED BY parentFunctions.c)
- Attach the shared memory to its own address space and fill the space (using an array) with the integers from the command-line. (COMPLETED BY parentFunctions.c)
- The parent will display the initial state of the shared memory and then spawn a number of child processes equal to the number of command-line arguments, one after the other, and then wait for each and every child to complete its tasks. (COMPLETED BY parentFunctions.c)
- The waiting cannot begin until every child is spawned. (COMPLETED BY parentFunctions.c)
- Upon completion of each child, the parent will acknowledge said completion and store the respective PID & exit code. (COMPLETED BY parentFunctions.c)
- Upon completion of all children, the parent will display each PID and its respective exit code, display the final state of the shared memory, detach the shared memory, remove the shared memory, and then terminate. (COMPLETED BY parentFunctions.c)
- Each child process will receive a unique ID when spawned – an integer between 1 and the number of command-line arguments. (COMPLETED BY parentFunctions.c)
- Each child will display the initial state of the entire shared memory and their own private memory - e.g. their unique ID. (COMPLETED BY childFunctions.c)
- Next each child will locate the array element (by index) value associated with their unique ID, double that value, and then multiply that value by their unique ID, storing the result back into the same array element. (COMPLETED BY childFunctions.c)
- Each child will then display the current state of the entire shared memory, provide their exit PID & status/code, and exit. (COMPLETED BY childFunctions.c)

Output:

- Each step (function?) that a process (parent or child) takes (starting, validating, requesting/removing shared memory, attaching/detaching shared memory, forking/completing child processes, exiting, etc.) will output a trace statement in order to show which statements are currently being executed. (COMPLETED)
- The parent will prefix each trace statement with 'Parent:' and each child will prefix their trace statements with a unique set of tabs and/or spaces, and then 'Child ID:' where ID is their unique ID, such that the child output stands out from the parent and other child output. (COMPLETED)
- Consider buffered versus un-buffered requirements in the output. A sample output is provide below, but this sample is NOT complete - only instructive. (COMPLETED)

Requirements:

- Functions should focus on a single task. (COMPLETED)
- Document compilation/execution, header file, and command-line requirements in your program header. (COMPLETED)
- Compiler warning should be resolved (COMPLETED – NO COMPILER WARNINGS)
- Each step in the attach/remove process for shared memory must be its own separate function: asking for a shared memory, attaching the shared memory, detaching the shared memory, and removing the shared memory. (COMPLETED)
- Main() should be high-level tasks only. (COMPLETED)
- Appropriate error handling must be included for each step of the shared memory process and the spawning of processes. (COMPLETED BY EACH FUNCTION)
- Minimum of three source code files – one for main(), one for the child process functions, and one for the parent process functions. (COMPLETED)
- Use of ONE proper user-defined header file is required (COMPLETED)
- Use of a Makefile is required. Include a copy of your Makefile with your submissions (Canvas and in-class), and include in your compile/execute instructions. Provide a "make build" target or allow "make" to compile your programs. Be sure you 'clean up' when finished. Do not have your Makefile execute, only compile. (COMPLETED)

Submission:

- Your source code file name(s) should be meaningful and documented in the first line in your programs. (COMPLETED)
- Include "Main" in your source code file name for the source code file(s) that includes a main(). (COMPLETED)
- Algorithm (pseudocode) should be submitted in a separate text file and included with the Canvas posting and class submission. (COMPLETED)

NOTE: EXPLANATION HOW THE PROGRAM USES BUFFER AND UNBUFFERED OUTPUT

Unbuffered output causes its output to appear immediately to the screen. Unbuffered outputs are usually used to show errors. The necessity to use this type of output stream is because, one needs to know exactly what each child happens to be doing every time. For this, you are going to see in some functions the use of a char buffer that sprintf() adds things to that buffer, and write that prints what was add to that buffer.

Buffered output is an I/O performance-enhancement technique. Buffered outputs cause its output to be held in a buffer (that is, an area in memory) until the buffer is filled or until the buffer is flushed. The used of this type of output will be when using printf().

The technical information explained in the two paragraphs above is from the book DD – C How to program 8th Edition from page # 817.

DETAILED PSEUDOCODE

Display program purpose:

This program allows us to work with shared memory space. The program launches children processes by the technique of fork. All these children run independently but they all have accessed to a shared memory space that is requested at the beginning of the program if the input from the command line happens to be validated. These children will change the shared memory by doubling the index in memory and then multiplying by the child id. The parent will wait until the process of all children us complete. Whenever, the process is complete the parent will prompt gain the updated memory. As the children manipulate the memory, they will state a status message in the standard output.

Before these processes are launch, the input must be validated in the command line. According to the specifications valid input is to type in the command line up to seven inputs after the name of the file that runs. That means that the user can type more than one and less than seven inputs. Also, these inputs must be single digits from zero to nine, and there must not be any duplicates. This means that the integers cannot be below zero or above nine.

The same number of children created will equal to the same amount of command inputs. These children will change the numbers in the memory share, and they will display everything they do in the standard output. After all children finish, the parent will terminate as well, displaying the final status of the memory shared.

This program will strategically be divided in four source files; forkingMain.c, childFunctions.c, parentFunctions.c, and inputValidationFork.c. There is going to be a header file that will share all the constants multiprocessLibrary.h.

The file forkingMain.c will be very general and will just hold the execution steps.

The file inputValidationFork.c will validate all input from the command line.

The file parentFunctions.c will request for memory, attach memory, populate memory, create children processes, wait for children to finish, detach the memory, and remove the memory.

The file childFunctions.c will change the shared memory for each process. All the status of each children will be displayed at each time.

A file dismantling the whole assignment will be after the pseudo code plus an explanation of why is so important to understand buffered and unbuffered output for this specific assignment.

Functions used to get to Answer:

To understand each function keep reading the pseudocode or go to each source file that contains the function.

```
void displayPurpose();
void explainRequiredInput();
int duplicateSearch(int input[], int arraySize);
int convertString(char *string);
int check_range(int number, int minValue, int maxValue);
int transformInput(char *commandInput[],int convertedInput[], int length);
int validateInput(char *commandInput[],int convertedInput[],int arraySize);
int requestMemory();
int* attachMemory(int shm_id);
void populateMemory(char* argv[], int* shm_ptr, int childrens);
void multiforking (int childrens, int* ptr_memory);
void childManipulation(int* shm_ptr, int counter, int childrens);
void printMemory(int* shm_ptr, int childrens);
void childCalculation(int *shm_ptr, int childIDNumber);
void printChildMemoryChange(int* shm_ptr, int childrens, int childNumber);
void wait_for_child(int childrens, int pidsNum[], int childStatus[]);
void display_child_info(int childrens, int* pidsNum, int *childStatus);
void parentDisplaysFinalMemory(int* shm_ptr, int childrens);
void detachFromSharedMemory (int* shm_ptr);
void removeSharedMemory (int shm_id);
void parentFinishes();
```

Source File: forkingMain.c

The main file will call all the necessary functions to get to the desire outcome. All of these functions are actually coded in the other files and for this reason will be explained below. However, a very general way of how the main should be:

- Display purpose of program
- Validate the input

- Request for memory
- Attach the memory
- Populate the memory
- Fork all required multiprocessors
- Wait for the children
- Display the child information
- Detach from the shared memory
- Remove the shared memory
- Display final message from parent

Source File: inputValidationFork.c - The purpose of the source file is to validate the input from the command line that the user will enter. According to the specification, a valid input can be from one input to seven inputs. Then to be clear input could be from one to seven, inclusive or [1-7]. This means that the user can enter any number of inputs within that range.
Also, according to the specifications, the command input should be integers from zero to nine. That means that are below zero or above nine are not allowed. This also means that no strings are allowed or empty input. Lastly, expanding from the previous requirement, all numbers input it should be unique. This means that no duplicates are allowed.

The set of functions required are as follows:

void displayPurpose()

- Print the purpose of this program to the user.

void explainRequiredInput()

- The purpose of this function is to advice the user if they happened to input anything that is not in the specifications in the command line

int inputValidation(char *input[],int convertedInput[],int length)

- This function will act as the brain of the program in regards of input validation. This function will call three functions to check if the input that' was inserted in the command line happens to be the correct input.
- Call function rangeCheck to make sure that the arguments that were put in the command line are within the requirements
- Call function convertInput to make sure that the input that was put can be converted to an integer digit.
- Call function findDuplicate to make sure that the partially approve array has no duplicates.

int rangeCheck(int number, int minimum, int maximum)

- If number is less than the required minimum and more than the required maximum return INVALID or (-1)

int findDuplicate(int input[], int length)

- This function takes two parameters, an integer array containing the converted input, and the length of that array.
- There is one outer loop that iterates through the entire array saving each element temporarily.
- The inner loop compares that temporarily element with the rest of elements
- There is a counter that counts if a number is duplicated; if this is the case then a return of invalid is returned, otherwise a return value of valid is returned.

int convertInput(char *input[],int convertedInput[], int length)

- This function takes three parameters; an array of strings coming from the command input. a new array that will store the validated input, and length that is argc.
- This function loops through all the inputs of the command line
- If the input of the command line is invalid return invalid or (-1)
- If not add that element into the array
- If that element that was just added does not happened to be in the invalid range then return.

int convertString(char *string)

- While the string passed does not get to the end of string or null character, iterate through that string
- Use the function isdigit(c) that checks if the passed character is a decimal digit character. The function isdigit(c) is from the <stdlib.h> library
- According to the book DD - C How to 8th Edition the function isdigit(c) returns a true value if c is a digit and 0 (false) otherwise.
- If the character passed is a digit, then we convert it into an integer with the line of code number = number * (base of ten) + (c - ASCII number)
- Advance through all characters of that string.
- If the character is not a digit then return INVALID CHARACTER
- Return converted character

Source File: parentFunctions.c This source file as the name entails, will have all the functions that the parent will execute in the main. The parent will request memory, it will attach the memory, populate the memory, print the memory when it is necessary to print it (recall that this function will also be used by the child), then the most important function of the child will be to fork the multiprocessors that will manipulate the shared memory. Please recall that all those functions that will manipulate the shared memory will be in the source file childFunctions.c. After all manipulations are done within the shared memory the parent will detached from the shared memory and remove.

The set of functions required are as follows:

int requestMemory()

- Request memory with shmget
- If request is not successful exit
- Otherwise print that the parent received shared memory

int* attachMemory(int shm_id)

- Attach memory with shmat
- If shmat is not successful, exit the program.
- Else state that parent was able to attached the shared memory

void detachFromSharedMemory (int* shm_ptr)

- Detach memory with shm_ptr
- State that the parent detaches memory

void removeSharedMemort (int shm_id)

- Remove the shar ememory with shmctl
- Print that the parent has removed the memory

void parentFinishes()

- Print that the parent has finished

int populateMemory(char *validInput[], int* shm_ptr, int childrens)

- Initialize counter and validateNum to 0. Do a counter that starts at 0 and the constant INITIAL_MEMORY_START (1).
- We start the counter at INITIAL_MEMORY_START or (1) because we want to start at index one for the first children and move one with the index matching the number of the children to facilitate the calculations.
- Then the loop iterates through the share memory storing the input numbers.
- Note that the input is accepted as a string that is changed to an int with the function atoi.
-

void printMemory(int* shm_ptr, int childrens)

- A char array was created to act as buffer. This is a very beneficial tool to use with the write function. As we have many children working at the same time, we need to print the memory shared as soon as this function is called.

- Function receives a pointer to the shared memory and the number of all children.
- Then a loop iterates through all elements of the shared function each element appends it a char array that is inside the loop using the sprintf function.
- Use the function strcat to concatenate both strings.
- According to the book; DD - C How to Program 8th Edition, page 349 strcat is defined as follows:
  Appends string s2 to array s1. The first character of s2 overwrites the terminating null character of s1. The value of s1 is returned.
- The reasoning behind using strcat was that as we are having several processes, we want to make sure that the whole memory is stored in the buffer before it actually is sent to the standard output to be printed. This was not the initial thought as the write function was inside the loop, but doing several tries it was noticed that the output in the screen would not shown correctly as we have different processes running at the same time.

void multiforking (int childrens, int*shm_ptr)

- Loop through the share memory from INITIAL_MEMORY_START as (1) until the total number of children.
- Fork every time you loop; when you fork a child process is created.
- If the child process is successfully created then the forkNumber will be equal to 0. If the child process fails then it returns a value that is less than 0, and the execution the program will print that the child was not successfully created.
- In this if statement we will get the pid of the child and will also call the function childManipulation.
- We have to exit after the childManipulation is called. This will prevent the system from creating grandchildren

void wait_for_child(int childrens, int pidsNum[], int childStatus[])

- Create an array of characters that is going to be used
- Loop through the number of child processes we have calling the function wait.
- The wait function takes the address of the child in the memory address and returns its pid.
- As the function loops through all the child process the parent waits until all the process are completed.

parentDisplaysFinalMemory(int* shm_ptr, int childrens)

- Create char buffer
- Input initial display final shared memory
- Call printMemory function

Source File: childFunctions.c works directly with all the other source files but most importantly with the parentFunctions.c. The reason why is because in the parentFunctions.c child processes are launched. The functions of those child processes are shown here. Every child will update the shared memory according to their unique ID given from the parent. As stated below each children will double their required index number in the shared memory and then it will multiply it by its own unique ID. Everything that the children does is sent to the standard output.

The set of functions required are as follows:

childManipulation(int* shm_ptr, int childNumber, int childrens)

- Create an array of characters that is going to be used as a buffer.
- Print the child id number.
- Print the memory before that children changes it
- Call the function, childCalculations()
- Print the alter shared memory, stating what child alter it.

void printChildMemoryChange(int* shm_ptr, int childrens, int childNumber)

- int: the file descriptor to write the output (standard input, standard output, standard error).
- A pointer to a buffer, which will be written to the file.
- The number of bytes to write.

void childCalculation(int *shm_ptr, int childIDNumber)

- Create an array of characters that is going to be used as a buffer.
- Multiply the number in shared memory by TWO or (2).
- Multiply that new number by its child id number again (index)
- Put the statement shown as the buffer in order to print

display_child_info(int childrens, int pidsNum[], int childStatus[])

- Loop through all children, and print their pidsNum and exit status

---

Source File: multiprocessLibrary.h

All constants needed from all the other files will be stated here.