Computer Architecture and Technology Area

Universidad Carlos III de Madrid

OPERATING SYSTEMS

Programming assignment 3. Multithread. Manufacturing process control.

BACHELOR'S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

Year 2016/2017

Skúli Arnarsson – 100369706

Group 89 - 100369706@alumnos.uc3m.es

Table of contents

Code Description

Factory manager

The program starts in the factory_manager. It begins by checking if an input file was given as parameter and checks its contents if It is provided. The factory_manager then opens the file and reads its contents to a buffer and closes the file. The buffer now contains the instructions. The instructions are read and validated by checking if the number of needed process_managers is larger than the maximum number of process_managers or if a process_manager gets an incorrect number of instructions. Next the factory_manager initializes a named semaphore with an initial value of 0. Then the factory_manager begins the process of creating all the needed process_managers by setting the correct value for the ID, the size its belt and the number of elements to be created for each process manager. The number of children provided by the input file are then created with fork(). The parent process then waits for all the children to finish and then goes on to free the resources by freeing the buffer, closing the semaphore and unlinking the name of the semaphore.

Process manager

The process_manager is a child of the factory_manager. It executes from the factory_manager that provides the arguments for the process_manager. First it asserts that the arguments are correctly passed to the process and then continues to open the semaphore by the name that was passed down by the factory_manager. The process then waits until the factory_manager signals to start production, that is when all process_managers have been created. Next the process_manager initializes its belt/queue and creates two threads, a “consumer” and a “producer”. The treads each run their dedicated functions. So as not to create a race condition both the producer and consumer request access to a critical section with requests to mutex and thread conditions. The Producer creates the indicated number of elements by requesting access to the mutex, checking whether the belt is full, if so wait until it is not full and put the element in the back of the belt. Then the producer signals that it is exiting the critical section and signals that the belt is not empty. The consumer also requests access to the critical section through a mutex. It consumes the indicated number of elements by checking if it has consumed the last element on the belt. Until then it checks whether the belt is empty, if so it waits until it is not and then takes an element of the belt and signals that the belt is not full. When a thread ends, it waits for the other. Here, purpose of the process manager has been fulfilled so we free the resources by destroying the mutex, condition variables and the belt.

The queue

The queue or belt, is created by each process_manager. It is in charge of putting and taking elements off the belt for the producer-consumer system. The queue is implemented as an array of the element structure. It is created by allocating memory determined by the maximum size of the belt. It puts an element on the queue by creating an element at the position behind the last element of the queue. It takes an element of the queue by storing of the element that is at the front of the queue and then moving all elements one position prior, then it returns the stored element. The queue destroys itself by freeing the allocated memory for the queue.

Test Cases

The expected output for the correct result of the test cases should be in this format (see picture of example output below), unless an error is expected.

- The factory manager creates all the process mangers and prints “[OK][factory_manager] Process_manager with id has been created.”.
- When a process manager is created, he prints “[OK][process_manager] Process_manager with id waiting to produce elements.”
- When all process managers have been created the all process managers start producing in parallel and the queue prints “[OK][queue] Introduced element with id: in belt” and “[OK][queue] Obtained element with id: in belt. “ in that order whereas an element cannot be obtained before it is introduced.
- The process manager should print “[OK][process_manager] Process_manager with id: has produced elements.” Only when he has introduced and obtained all the elements from the belt.
- The factory_manager prints “[OK][factory_manager] Process_manager with id has finished.” Only when the process_manager has indicated that he has produced all his elements.
- Finally, the factory_manager prints “[OK][factory_manager] Finishing” at the end of the output.
- The processes run in parallel so the actual output can be unpredictable but should follow this schema and the rules stated above.

```
[OK][factory_manager] Process_manager with id 5 has been created.
[OK][factory_manager] Process_manager with id 1 has been created.
[OK][factory_manager] Process_manager with id 3 has been created.
[OK][process_manager] Process_manager with id: 5 waiting to
produce 2 elements.
[OK][process_manager] Process_manager with id: 1 waiting to
produce 3 elements.
[OK][process_manager] Process_manager with id: 3 waiting to
produce 2 elements.
[OK][process_manager] Belt with id: 5 has been created with a
maximum of 5 elements.
[OK][queue] Introduced element with id: 0 in belt 5.
[OK][queue] Introduced element with id: 1 in belt 5.
[OK][queue] Obtained element with id: 0 in belt 5.
[OK][queue] Obtained element with id: 1 in belt
5[OK][process_manager] Process_manager with id: 5 has produced 2
elements.
[OK][factory_manager] Process_manager with id 5 has finished.
[OK][process_manager] Belt with id: 1 has been created with a
maximum of 2 elements.
[OK][queue] Introduced element with id: 0 in belt 1.
[OK][queue] Introduced element with id: 1 in belt 1.
[OK][queue] Obtained element with id: 0 in belt 1.
[OK][queue] Introduced element with id: 2 in belt 1.
[OK][queue] Obtained element with id: 1 in belt 1.
[OK][queue] Obtained element with id: 2 in belt 1.
[OK][process_manager] Process_manager with id: 1 has produced 3
elements.
[OK][factory_manager] Process_manager with id 1 has finished.
[OK][process_manager] Belt with id: 3 has been created with a
maximum of 5 elements.
[OK][queue] Introduced element with id: 0 in belt 3.
[OK][queue] Introduced element with id: 1 in belt 3.
[OK][queue] Obtained element with id: 0 in belt 3.
[OK][queue] Obtained element with id: 1 in belt 3.
[OK][process_manager] Process_manager with id: 3 has produced 2
elements.
[OK][factory_manager] Process_manager with id 3 has finished.

[OK][factory_manager] Finishing.
```

Test case 1:

Contents of Input file: 4 5 5

Description: Running an invalid input file with too few parameters.

Motivation: To check whether the program recognizes that the input file is missing arguments.

Expected output: [ERROR][factory_manager] Invalid file

Obtained output: [ERROR][factory_manager] Invalid file

Status of test: Test passed. The program handles the error correctly by identifying the invalid file and stopping the execution.

Test case 2:

Contents of Input file: 4 5 5 2

Description: Running an invalid input file with two spaces between numbers in input

Motivation: To check whether the program recognizes that the input file contains two spaces, it is therefore invalid, and it stops its execution.

Expected output: [ERROR][factory_manager] Invalid file

Obtained output: [ERROR][factory_manager] Invalid file

Status of test: Test passed. The program handles the error correctly by identifying the invalid file and stopping the execution.

Test case 3:

Contents of Input file: None

Description: Running the program without an input file.

Motivation: To check if the program handles the error that no arguments are given

Expected output: [ERROR][factory_manager] Invalid file

Obtained output: [ERROR][factory_manager] Invalid file

Status of test: Test passed. The program handles the error correctly by identifying that not enough arguments are provided so it stops the execution.

Test case 4:

Contents of Input file: 1 5 2 2 3 2 1

Description: Running the program where the maximum number of process_managers are fewer than the ones needed

Motivation: To check if the program detects the fact that the input file is invalid, because it requires 2 process_managers but allows only one. The program should then stop execution

Expected output: [ERROR][factory_manager] Invalid file

Obtained output: [ERROR][factory_manager] Invalid file

Status of test: Test passed. The program detects that the maximum number of project managers are fewer than those needed to process the input.

Test case 5:

Contents of Input file: 5 5 10 2

Description: Running the program with double-digit commands in the input file.

Motivation: To check if the program can handle parse the arguments for double-digit commands correctly.

Expected output: As expected from a valid file, as stated above.

Obtained output: [ERROR][factory_manager] Invalid file

Status of test: Test failed. The program gets a single character at a time and does not recognize double-digit characters as a single number.

Test case 6:

Contents of Input file: 5 5 a 2

Description: Running the program with a character instead of a number in the input file.

Motivation: To check if the program can detect the character in the input file and stop execution.

Expected output: [ERROR][factory_manager] Invalid file

Obtained output:

[OK][factory_manager] Process_manager with id 5 has been created.

[ERROR][process_manager] Arguments not valid.

[OK][factory_manager] Finishing.

Status of test: Test failed. The program does not detect that the argument is invalid until it the process manager has been already been executed.

Test case 7:

Contents of Input file: 4 5 5 2” “

Description: Running the program with a space at the end of the input. The quotation marks in the contents of the input file represent a space character.

Motivation: To check if the program can detect the space character at the end of the input file and stop execution.

Expected output: [ERROR][factory_manager] Invalid file

Obtained output: [ERROR][factory_manager] Invalid file

Status of test: Test passed, if the format of the input file is strict. It is debatable whether a space at the end of the input invalidates the file.

Test case 8:

Description: Running a valid input file initiating one process_manager.

Contents of Input file: 4 5 5 2

Motivation: To check whether the program can run the program with a valid input file and execute a process_manager that dispatches workers and the processes finish correctly.

Expected output: As expected from a valid file, as stated above.

Obtained output:

[OK][factory_manager] Process_manager with id 5 has been created.

[OK][process_manager] Process_manager with id: 5 waiting to produce 2 elements.

[OK][process_manager] Belt with id: 5 has been created with a maximum of 5 elements.

[OK][queue] Introduced element with id: 0 in belt 5.

[OK][queue] Introduced element with id: 1 in belt 5.

[OK][queue] Obtained element with id: 0 in belt 5

[OK][queue] Obtained element with id: 1 in belt 5

[OK][process_manager] Process_manager with id: 5 has produced 2 elements.

[OK][factory_manager] Finishing.

Status of test: Test Failed. The output does not contain "[OK][factory_manager] Process_manager with id has finished." Because the child process exits abnormally.

Test case 9:

Description: Running a valid input file initiating more than process_manager.

Contents of Input file: 4 5 5 2 1 2 3 3 5 2

Motivation: To check whether the program can run the program with a valid input file and execute multiple process_manager that dispatches workers and the processes finish correctly.

Expected output: As expected from a valid file, as stated above.

Obtained output:

[OK][factory_manager] Process_manager with id 5 has been created.

[OK][factory_manager] Process_manager with id 1 has been created.

[OK][factory_manager] Process_manager with id 3 has been created.

[OK][process_manager] Process_manager with id: 5 waiting to produce 2 elements.

[OK][process_manager] Belt with id: 5 has been created with a maximum of 5 elements.

[OK][process_manager] Process_manager with id: 1 waiting to produce 3 elements.

[OK][process_manager] Belt with id: 1 has been created with a maximum of 2 elements.

[OK][process_manager] Process_manager with id: 3 waiting to produce 2 elements.

[OK][process_manager] Belt with id: 3 has been created with a maximum of 5 elements.

[OK][queue] Introduced element with id: 0 in belt 3.

[OK][queue] Introduced element with id: 1 in belt 3.

[OK][queue] Obtained element with id: 0 in belt 3

[ERROR][factory_manager] Process_manager with id 3 has finished with errors.

[OK][queue] Obtained element with id: 1 in belt 3

[OK][process_manager] Process_manager with id: 3 has produced 2 elements.

[ERROR][factory_manager] Process_manager with id 3 has finished with errors .

[OK][factory_manager] Finishing.

Status of test: Test Failed. The process manager process exits abnormally after the producer-consumer functions, when the threads call to join. The return value from the process_managers is 11, that represents a segmentation fault.

Test case 10:

Description: Checking for memory errors with valgrind

Contents of Input file: 4 5 5 2 1 2 3 3 5 2

Motivation: To check whether the program allocates and frees memory correctly in order to not leak memory.

Expected output: All heap blocks were freed -- no leaks are possible. That the project contains no memory leaks

Obtained output:

Starting after output of the program, same as in test case 9 (above). The output is:

==17610==

==17610== HEAP SUMMARY:

==17610== in use at exit: 0 bytes in 0 blocks

==17610== total heap usage: 3 allocs, 3 frees, 99 bytes allocated

==17610==

==17610== All heap blocks were freed -- no leaks are possible

==17610==

==17610== For counts of detected and suppressed errors, rerun with: -v

==17610== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Conclusion

I thought that the project was very time consuming, quite hard and fun as well. I had a hard time testing my solution because the output was buffered out and I did not realize that straight away, so I had to use fflush(stdout) in order for my output to print while developing. I also had trouble opening the named semaphore in the child process, most likely because of synchronization errors that I managed to resolve. I spent hours trying to execute the process_manager from the factory_manager because I did not manage to convert the integer numbers to chars or the other way around correctly at first. I figured that out by creating an array of characters of size 10(figured that would be enough) and using snprintf to convert the integer number to a string and then creating an array of strings with those strings and executing the process_manager referencing arguments in that array of strings. That solution works well but i realize that there must be an easier way to implement this behaviour. I had trouble joining the threads, that resulted in the process_managers exiting abnormally and therefore made synchronization very hard whereas I did not manage to figure out the error yet. I had trouble getting my solution to work in Guernika Lab because my implementation worked in bash for windows on my machine. Guernika lab reads the files differently and that resulted in the fact that my implementation did not work as expected. I had to change the way I implemented the read and write POSIX commands and the functions that handle the input file. Ultimately my solution worked with mostly the same functionality as it did before.

I included error messages for error handling in my implementation, however they never appear in the output in any of the test cases. In the project description is a note that reads: "*NOTE: These messages are the only ones accepted. No other messages should appear in the screen."* So I assume that I am supposed to handle all errors, but these are the only ones that appear on the screen. So I commented out all my custom error messages but they are still there if you want to grade error handling.

Working on this Lab I learned how to synchronize processes and operations using semaphores, mutexes and thread conditions, all of which I had trouble understanding before. I am fairly certain that I implemented the creation of processes, memory allocation, threading and semaphores correctly however I am not certain if the implementation of the producer-consumer, usage of mutexes and thread conditions works without errors.

Thank you for a very interesting, challenging and well organized course with fun projects.