CS2002 W11-SP

Tutor: Ian Gent 210018092

5th April 2023

Contents

Overview

For this practical, we had to implement a multi-threaded pipeline framework in C.

We were provided with some examples that should run with our pipeline, and some header files, and were required to implement the functions in these header files while also extending the framework to support more features if necessary.

In this practical, I have implemented two versions of the pipeline framework, one using unix pipes for thread communication, and one using my own thread safe queue implementation. Both of my pipeline frameworks run with example files and run very quickly. I have tested both of these implementations proving that they achieve parallelism, make use of multiple cores, and are thread safe. Both of these pipelines support an arbitrary number of stages, execute each pipeline stage as a separate thread, and permit the stages to communicate with each other using the methods above.

My submission fulfills all the requirements laid out in the specification, and beyond.

In this report, I begin by discussing the high-level design and functionality of the pipeline framework with an emphasis on the design decisions I made.

In the Implementation section, I consider each part in greater detail, discussing how the unique features of C have influenced my implementation.

The report ends with some evidence of testing to prove the correctness of my approach.

Design

There was not a lot to do in terms of design as the practical is more about the implementation of pre-designed functions that were given to us. However, there were still a few design decisions that had to be made in order to implement the pipeline framework.

General design

To get a general idea of how a pipeline framework should work, I looked at the example described in [1] which shows how a Sieve pipeline would work using unix pipes on a theoretical level. I also took inspiration from golang's routine and channel model of concurrency as summarized in [2].

In terms of general design there was not a lot to consider as the main unimplemented pipeline functions were already given to us. One design decision I did have to make was whether to use a dynamic array or linked list for the implementation of the pipeline stages. I decided to use a linked list to store the stages the number of stages is not known at compile time so using a dynamic array would require using `realloc` to extend the size of the array constantly. This would be inefficient as `realloc` would have to copy the entire array every time it is called. Using a linked list allows the stages to be added to the end of the list in constant time as I keep track of the head and tail of the list. However, this does come with two downsides but these are negligible in the context of this program. Firstly, a linked less uses more memory than an array as each node in the list has to store a pointer to the next node. Secondly, the stages are not stored in contiguous memory using a linked list which means the compiler cannot optimize as much as when using an array. However, the differences in memory usage and speed are very minimal and are not worth the extra complexity of using a dynamic array.

Unix pipes vs custom queue

The first design decision I had to make was whether to use unix pipes or a custom queue for the communication between the threads. I decided to do both to compare the differences between them but my main version of the program uses a custom queue. My main version uses the custom queue as it makes better use of the fact that threads share memory and doesn't need to use the unix pipe system as a middleman for communication. As shown in my speed comparison, the queue implementation is much faster than the unix pipe implementation - the custom queue completes the same tasks in about half the time. I also felt that using a custom queue would be more interesting and would allow me to learn more about thread safety, concurrency and memory management.

Another issue with using unix pipes is that the system will eventually run out of file descriptors. The number of file descriptors is limited by the system so if the program is doing a complex pipeline that requires a lot of threads, and thus file

descriptors, then the program will eventually crash. This is not an issue with the custom queue implementation.

Thread Pool and backoff

Another interesting design decision I had to make was how to handle the creation of threads, specifically there are two things that would crash the pipeline framework when creating lots of threads: the system running out of memory for threads or hitting the maximum number of threads allowed by the system. To solve this, I use 3 features: a thread pool, a backoff system and thread detaching which allow the program to create threads many threads for complex pipelines without crashing. These features are described in more detail in the Implementation section.

Implementation

General Implementation

The first thing I had to implement were the `Pipeline` and `PipelineStage` structs which are used to store the stages and the pipeline. The `PipelineStage` struct stores the function to be performed, the input and output channels, the next stage in the pipeline, and the thread id. The `Pipeline` struct stores the head and tail of the linked list of stages.

The majority of the actual functions like the `new_Pipeline` and `Pipeline_add` functions were quite simple to implement with these structures setup, the main thing I had to remember was that these function allocate memory using `malloc` which means they have to be freed after the program is finished using them.

The most difficult function to implement was the `Pipeline_execute` function. One complication when implementing this function was the fact that posix start thread routine only takes one argument but this could be solved using my `PipelineStage` struct as it can be passed as the one argument needed for the start thread routine but stores all the information needed for the thread to run. I also had to make sure that my communication channels were linked properly, for example the output channel of one stage had to be the input channel of the next stage. Making sure that the communication channels were linked did have a knock on effect which helped when implementing the thread pool and back off system. My original implementation without these systems would loop through the stages and create threads for each of them then loop again to join them all. This caused issues as if the system ran out of memory to create threads inside of the first loop then the program would crash and join (or detach) were never called to free the memory. However, by using the knowledge that each stage in the pipeline is implicitly linked together, I changed my implementation so that after a thread had completed it's stage it would detach and I only had to call `pthread_join` on the last stage in the pipeline to make sure the whole pipeline

had finished before main exited.

The `Pipeline_free` function was also quite simple to implement as it just had to loop through the stages and free the memory allocated for each stage as well as the pipeline itself. I have included a terminal output in my Testing section showing that there are no memory leaks in my program.

Implementing the Pipeline_send and Pipeline_receive functions was also quite simple for both the unix pipe and custom queue implementations as they both just had to use their respective write(enqueue) and read(dequeue) functions. For more detail on each versions implementation of these functions please see the Unix pipe version and Custom queue version sections.

Unix pipe version

Creating the unix pipe version of the communication channels was easy as I just had to use the `pipe` function to create a pipe and then use the `read` and `write` functions to read and write to the pipe. The only issue I had with this implementation was that the pipe function returns two file descriptors, one for reading and one for writing. I had to make sure that the read and write file descriptors were used in the Pipeline_receive and Pipeline_send functions respectively. I also had to make sure that the file descriptors were closed after the program was finished using them.

As mentioned in the Design section, the unix pipe implementation has the issue of the system eventually running out of file descriptors which is not an issue with the custom queue implementation.

Custom queue version

The custom queue implementation was more difficult to implement as I had to make sure that the queue I implemented was thread safe. I did this by using a mutex lock and a condition variable to make sure that only one thread could access the queue at a time, avoiding race conditions. I also had to make sure that the dequeue function would block if the queue was empty using the condition variable. I had to do a lot more research to get this version to be properly thread safe than I did for the unix pipe version as the unix pipes are already thread safe.

Thread Pool and backoff

As I briefly mentioned in my Design section, I encountered a problem when executing complex pipelines, mainly when trying to get a large amount of prime numbers using the `SievePipeline` program. The system would eventually run out of memory to create threads with as I was initially not detaching or joining threads in the first loop of the Pipeline_execute function which was generating the threads.

As I mentioned in General Implementation section, I solved this problem by using the knowledge that each stage in the pipeline is implicitly linked together. This allowed me to change my implementation so that after a thread had completed it's stage it would detach and I only had to call `pthread_join` on the last stage in the pipeline to make sure the whole pipeline had finished before main exited. This solved the problem of the system running out of memory to create threads for the example programs but there were still two theoretical issues that could occur: the program could hit the limit on threads allowed by the system or a small amount of threads could still use up all the system memory as the "detach method" above only helps if threads have time to complete before crashing.

To solve these issues I implemented a thread pool and a backoff system. The thread pool simply uses a global semaphore to limit the number fo threads that can be created at one time. Each time a thread is created the semaphore is incremented and if the limit is reached the program will wait until a thread has completed (and decremented the semaphore) before creating a new thread. This solves the theoretical issue where a pipeline could create too many threads that take up a small amount of memory which would hit the system thread limit before running out of memory. I also created a backoff system in the event that a thread fails to be created. The most likely scenario for a thread not being created with the two previous systems in place is that a small number fo threads are using up a large amount of memory. This could happen if the pipeline is very complex and the threads are taking a long time to complete thus the thread pool limit isn't hit and no threads detach to free up memory. The backoff system simply waits for a short period of time before trying to create a new thread again in the hopes that one will complete and detach freeing up memory. It initially waits a small amount of time but ramps up the wait time if the thread still fails to be created. If the threads fails to be created after 5 attempts at waiting the maximum backoff time then the program will exit with an error message.

Testing

I have created my own set of unit tests which can be found in the Tests directory. The operation of the tests is quite basic simply using the assert.h library to create the unit tests. If the program can reach the print statements then none of the assert statements have caused a program crash and thus the tests must have passed.

Please see the README file for more information on how to make and run the tests.

The tests have been split into separate files pertaining to which function/requirements are being tested. More information on each specific test can be found within the test files as all the tests have accompanying comments explaining what is being tested and why. These tests cover a range of scenarios usually involving testing the normal use of a function as well as the edge and

extreme cases associated with that function.

I have also used `valgrind` to ensure there are no memory leaks in my programs and I have included a speed comparison of both versions of my program.

Please note all tests and comparisons were completed on the lab machines, results may vary on other machines.

Unit tests

A terminal output from running the `./RunAll.sh` script is included below showing that all the unit tests pass. Please note that teh warnings generated come from the example files and are not a result of my code.

```
./RunAll.sh
clang -g -Wall -Wextra -c ../tests/RunTests.c
clang -g -Wall -Wextra -c ../tests/TestExampleFiles.c
clang -g -Wall -Wextra -c ../tests/TestSendAndReceive.c
clang -g -Wall -Wextra -c ../tests/TestAdd.c
clang -g -Wall -Wextra -c Pipeline.c
clang -g -Wall -Wextra -c Queue.c
clang -g -Wall -Wextra -pthread RunTests.o TestExampleFiles.o TestSendAndReceive.o TestAdd.o
Starting tests...
Starting example file tests...
SumSquaresPipeline.c:19:32: warning: unused parameter 'input' [-Wunused-parameter]
static void generateInts(void* input, void* output) {
                               ^

SumSquaresPipeline.c:38:48: warning: unused parameter 'output' [-Wunused-parameter]
static void sumIntsAndPrint(void* input, void* output) {
                                               ^

2 warnings generated.
SievePipeline.c:20:32: warning: unused parameter 'input' [-Wunused-parameter]
static void generateInts(void* input, void* output) {
                               ^

1 warning generated.
WordSumPipeline.c:19:33: warning: unused parameter 'input' [-Wunused-parameter]
static void generateWords(void* input, void* output) {
                                ^

WordSumPipeline.c:35:49: warning: unused parameter 'output' [-Wunused-parameter]
static void sumWordsAndPrint(void* input, void* output) {
                                                ^

2 warnings generated.
testSumSquaresPipeline passed
testSumSquaresPipelineLarge passed
testSievePipeline passed
testSievePipelineLarge passed
testWordSumPipeline passed
```

```
testWordSumPipelineLarge passed
All example file tests passed
Starting send and receive tests...
test_pipeline_send_receive() passed
test_pipeline_receive_blocking() passed
All send and receive tests passed
Starting add tests...
test_pipeline_add passed
All add tests passed
All tests passed for custom queue version of the pipeline framework
All tests passed
rm -f SumSquaresPipeline SievePipeline WordSumPipeline RunTests RunTestsPipe *.o
clang -g -Wall -Wextra -c ../tests/RunTestsPipe.c
clang -g -Wall -Wextra -c ../tests/TestExampleFilesPipe.c
clang -g -Wall -Wextra -c ../tests/TestSendAndReceivePipe.c
clang -g -Wall -Wextra -c ../tests/TestAddPipe.c
clang -g -Wall -Wextra -c ./Pipe_version/PipelinePipe.c
clang -g -Wall -Wextra -pthread RunTestsPipe.o TestExampleFilesPipe.o TestSendAndReceivePipe
Starting tests for pipe version of the pipeline framework...
SumSquaresPipelinePipe.c:19:32: warning: unused parameter 'input' [-Wunused-parameter]
static void generateInts(void* input, void* output) {
                               ^
SumSquaresPipelinePipe.c:38:48: warning: unused parameter 'output' [-Wunused-parameter]
static void sumIntsAndPrint(void* input, void* output) {
                                                ^
2 warnings generated.
SievePipelinePipe.c:20:32: warning: unused parameter 'input' [-Wunused-parameter]
static void generateInts(void* input, void* output) {
                               ^
1 warning generated.
WordSumPipelinePipe.c:19:33: warning: unused parameter 'input' [-Wunused-parameter]
static void generateWords(void* input, void* output) {
                                ^
WordSumPipelinePipe.c:35:49: warning: unused parameter 'output' [-Wunused-parameter]
static void sumWordsAndPrint(void* input, void* output) {
                                                ^
2 warnings generated.
testSumSquaresPipelinePipe passed
testSumSquaresPipelineLargePipe passed
testSievePipelinePipe passed
testSievePipelineLargePipe passed
All example file tests passed for pipe version of the pipeline framework
test_pipeline_receive_blockingPipe() passed
All send and receive tests passed for pipe version of the pipeline framework
test_pipeline_addPipe passed
All add tests passed for pipe version of the pipeline framework
```

```
All tests passed for unix pipe version of the pipeline framework
All tests passed
rm -f SumSquaresPipeline SievePipeline WordSumPipeline RunTests RunTestsPipe *.o
```

Memory leaks

I also used `valgrind` to ensure that my two versions of the program were free of memory leaks. The output from running `valgrind` on the custom queue version is shown below:

```
./SievePipeline
==46139== Memcheck, a memory error detector
==46139== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==46139== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==46139== Command: ./SievePipeline
==46139==
5 15
Setting up pipeline to sieve out the first 5 prime numbers up to 15
generateInts: thread 0x5677640
sieveInts: thread 0x5e78640: prime = 2
sieveInts: thread 0x6679640: prime = 3
sieveInts: thread 0x6e7a640: prime = 5
sieveInts: thread 0x767b640: prime = 7
sieveInts: thread 0x7e7c640: prime = 11
==46139==
==46139== HEAP SUMMARY:
==46139==     in use at exit: 0 bytes in 0 blocks
==46139==   total heap usage: 91 allocs, 91 frees, 5,540 bytes allocated
==46139==
==46139== All heap blocks were freed -- no leaks are possible
==46139==
==46139== For lists of detected and suppressed errors, rerun with: -s
==46139== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The output from running `valgrind` on the unix pipe version is shown below:

```
valgrind ./SievePipelinePipe
==46163== Memcheck, a memory error detector
==46163== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==46163== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==46163== Command: ./SievePipelinePipe
==46163==
5 15
Setting up pipeline to sieve out the first 5 prime numbers up to 15
generateInts: thread 0x5677640
sieveInts: thread 0x5e78640: prime = 2
sieveInts: thread 0x6679640: prime = 3
```

```
sieveInts: thread 0x6e7a640: prime = 5
sieveInts: thread 0x767b640: prime = 7
sieveInts: thread 0x7e7c640: prime = 11
==46163==
==46163== HEAP SUMMARY:
==46163==     in use at exit: 0 bytes in 0 blocks
==46163==   total heap usage: 21 allocs, 21 frees, 3,984 bytes allocated
==46163==
==46163== All heap blocks were freed -- no leaks are possible
==46163==
==46163== For lists of detected and suppressed errors, rerun with: -s
==46163== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As you can see from the above outputs, there are no memory leaks in either of my implementations.

Speed Comparison

I have created a simple demonstration to show that the custom queue implementation is faster than the unix pipe implementation. I use the `SievePipeline` program to generate 10,000 prime numbers and time how long it takes the two implementations to complete. The output from running the custom queue version is shown below:

```
echo 10000 120000 | time ./SievePipeline
...
sieveInts: thread 0x7f993affd640: prime = 104717
sieveInts: thread 0x7f99d77fe640: prime = 104723
sieveInts: thread 0x7f992affd640: prime = 104729
9.79user 3.80system 0:01.20elapsed 1132%CPU (0avgtext+0avgdata 69348maxresident)k
0inputs+0outputs (0major+14120minor)pagefaults 0swaps
```

The output from running the unix pipe version is shown below:

```
echo 10000 120000 | time ./SievePipelinePipe
...
sieveInts: thread 0x7f2e6ddde640: prime = 104717
sieveInts: thread 0x7f2e6d5dd640: prime = 104723
sieveInts: thread 0x7f2e6cddc640: prime = 104729
7.51user 25.82system 0:02.83elapsed 1178%CPU (0avgtext+0avgdata 16680maxresident)k
0inputs+0outputs (0major+8027minor)pagefaults 0swaps
```

As you can see from the above outputs, the custom queue implementation is significantly faster than the unix pipe implementation.

Evaluation

My submission successfully implements all of the practical specifications. This includes implementations of all the functions laid out in the initial header file as well as two seprate implementations, one that uses unix pipes for communication and another that uses a custom thread safe queue. I have also implemented safety features like a thread pool so that the program does not crash from not being able to create more threads. I have tested my submission extensively using a wide variety of tests which proves my software is solid and reliable. My submission is very well documented with comments explaining every function and in-line comments throughout my code when they are required to explain more complex lines of code. I have also included a README in various formats with instructions on how to make, run and clean my program files.

Conclusion

In this practical, I have designed and implemented a multi-threaded pipeline framework in C. I have implemented two versions of the pipeline framework, one using unix pipes for thread communication, and one using my own thread safe queue implementation. Both of my pipeline frameworks run with the example files provided and run very quickly. I have tested both of these implementations proving that they are solid and reliable.

Future work on this pipeline could include options for setting the thread pool maximum size, the ability to set the number of threads to use for each stage, and set how many times the backoff function should be allowed to do the maximum backoff before it gives up and returns an error.

Performance could also be improved through using pre made libraries for the thread pool and queue implementations but this would go against the specifications of the practical.

References

[1] M. Douglas McIlroy, "Coroutine prime number sieve," *Dartmouth College*. Available: https://www.cs.dartmouth.edu/~doug/sieve/sieve.pdf

[2] Y. Fernando, "Concurrency in golang, goroutines, and channels explained," *Medium*. Level Up Coding, Aug. 2022. Available: https://levelup.gitconnected.com/concurrency-in-golang-goroutines-and-channels-explained-55ddb5e1881