Contents

[Introduction 2](#_Toc35549523)

[Algorithm and Design 2](#_Toc35549524)

[Issues 5](#_Toc35549525)

[Compilation and Execution 7](#_Toc35549526)

[Sample Output 8](#_Toc35549527)

[I. Not passing any parameters into the program, as in ./vagner.exe 8](#_Toc35549528)

[II. Passing -help into the program, as in ./vagner.exe -help: 10](#_Toc35549529)

[III. Passing -printInput into the program as in ./vagner.exe -printInput 11](#_Toc35549530)

[Conclusion 12](#_Toc35549531)

Vagner Machado

Professor John Svadlenka

CSC 340

18 Mar. 2020

Project 2 – Multithreaded Programming

# Introduction

Aiming to strengthen the understanding of multithread programming using the POSIX standard, students were instructed to design and code an application using the C language that uses multiple threads to solve a computation problem. In short, the application must produce 10,000 random integers, find the minimum and maximum values using a single thread, and after that, use a two-layer multithread approach to find the minimum value. Each layer must output its lowest values and finally, the lowest value overall. This computational problem can be a fairly simple task to accomplish using a single thread approach, however, using multithreaded programming requires extended and careful planning.

# Algorithm and Design

In order to successfully design an algorithm for a multithreaded program, one must pay close attention to the given requirements. The requirements state the main process must create five threads and each of those threads must create twenty more threads. In this document, the former five threads are referred as parent threads and the sets of twenty threads are called children threads. Each of the five parent threads is responsible for processing 2,000 of the randomly generated 10,000 integers by delegating a hundred of those integers to each of its twenty children threads. In turn, the children threads are responsible for finding the lowest value in its subset of a hundred numbers and for writing that value into a global array. Each parent thread then finds the lowest value amongst the ones produced by its children and writes the result to another global array. Lastly, the main process goes through the five lowest values and determines the lowest value overall. Using this detailed explanation, in conjunction with the comprehensive explanation given in class, I designed the following algorithm to solve the stated problem:

1. Include headers needed for the program such as pthreads.h, stdio.h and time.h.
2. Define constants to be used by input and threads in different layers such as 10,000, 2,000 …
3. Declare function stubs for the application and determine their parameters.
4. Declare and initialize three global arrays to be used throughout the program:
   1. randomArray: stores the 10,000 random integers used as input.
   2. topHundred: stores the lowest hundred numbers found by the children threads.
   3. topFive: used to save the lowest five values generated vy the parent threads
5. Declare and initialize a FILE pointer to which output is to be printed.
6. Initialize a random integer generator and produce 10,000 integers.
7. Using a single thread, find lowest and highest value in the input and print to file.
8. Declare an array of five thread ids and an array of five thread attributes.
9. Loop five times to create five threads with id, attribute, splitter function and upper range parameter.
10. Wait for all five threads to join, print to file the values in topHundred and topFive arrays.
11. Scan the the topFive array, find and print to file the lowest value.

Note that apart from required #include, #define and function stubs in steps 1 through 3, steps 4 through 11 are located in the main function. The key jobs accomplished in the main function are to create five threads that call function splitter, to print the lowest hundred and lowest five values to file and lastly, find the lowest value out of lowest five and print it to file. Let’s now explore the algorithm for the splitter function called by the threads created in the main function.

1. Cast the received parameter as char \* and use it to calculate the lower index of input that the parent thread is responsible for managing and delegating.
2. Use the lower index to calculate to which index in topFive the parent should write the lowest child value into.
3. Declare an array of twenty thread ids and an array of twenty thread attributes.
4. Loop twenty times to create twenty threads with id, attribute, runner function and range parameter.
5. Wait for all twenty threads to join.
6. Scan its range of topHundred array, find and write to topFive the lowest value.

These are all the steps in the algorithm for the function splitter. Its objective is to create twenty more threads for each of the calls. Instead of passing a comma separated sequence of parameters to each thread, I opted to pass the upper index to each thread. That value can be used to calculate the proper lower index and index to write the result to. As described in step 4, each of the twenty threads created calls the function runner, which will finally scan for the lowest value in its hundred integer subset. The function runner accomplishes this task by implementing the following algorithm:

1. Cast the received parameter as char \* and use it to calculate the lower index of input that child thread is responsible for searching for lowest value.
2. Use the lower index to calculate to which index in topHundred the child thread should write the lowest value in its range.
3. Find the lowest value in its range and write to calculated index in topHundred array.

The runner function has the simple task of finding the lowest value in a 100-integer section out of the 10,000 generated random integers. Each child writes the lowest value to topHundred array which is subsequently used by the parent thread to find the lowest value amongst its children. The parent thread, in turn, writes its lowest value to the topFive array, that is later used by the main process to find the overall lowest value.

While the algorithm for a single thread approach to solve the problem can be roughly summarized as simply, *“iterate through input linearly and find lowest value,*” the two layered multithread algorithm is considerably more complex and requires careful planning. Unfortunately, the simplicity of the single thread approach can be inefficient and may not use a multicore CPU to its fullest capabilities. This can be seen, for example, when a single thread program delays the overall execution during I/O bursts because the CPU remains idle until I/O is finalized. When solving complex tasks, programmers must code applications using a multithread approach to improve overall organization, task delegation, throughput, and execution efficiency by using the full power of a multicore CPU.

# Issues

Many people think they are multitaskers, when in reality they accomplish one task at a time, switching from task to task, from reading emails to taking notes, from thinking to writing this very sentence. Although I did not encounter a major issue when coding this project, a couple of areas required further planning. More specifically, in addition to developing a multithreaded way of thinking, researching the void \* cast for the pthread\_create calls and deciding which values to pass into it needed close attention.

The multithreaded way of thinking came to life when deciding the location for the fprintf statements for the topHundred and topFive arrays. Initially, I had placed them inside the functions called by pthread\_create, which caused the output to be printed out of order. My first reaction was that I had a bug in the code, when in reality, I was not thinking in a multithreaded approach when placing the fprint statements. However, upon analyzing the code, I realized that the disordered output was caused by the scheduler. Taking that into account, I decided the best place to write the print statements was after all the parent threads had joined. This problem required a simple solution but it underscores how important it is to develop one’s multithreaded thinking skills to solve these types of computational problems.

The second area that needed attention was during the use of void \* parameter, which was a concept that I had not come across while coding in C or C++. I had never dealt with a void pointer, and upon trying to use the pointer’s values directly, the program would not work. For example, for a function f that accepts void \* voidParam as a parameter that is intended to be an array of integers, one must point the parameter to an array of integers in order to use its values. Hence, after entering int \* arrayParam = voidParam, one is able to access the values pointed to by voidParam. In my case, I did not know this ‘repointing’ was needed or existed at all, which lead me to look for bugs in parts of my code that were working properly. This was somewhat time consuming but after some light researching, I found the true bug in the parameter passing and after pointing the parameter to a character array, the program worked as intended.

Lastly, I wanted to minimize the number of parameters passed to each function called by pthread\_create. I made this choice because all needed parameters (indexes that indicate range bounds and place to write results to) would have to be casted to string in order to be joined as a comma-separated parameter just to be split and casted back to integer in the receiving function. This seemed like unnecessary work, so I decided to pass just one parameter, an upper bound, and to use arithmetic in the receiving function to determine the lower bound and place to write the result to. Hence, for the parent threads, each pthread\_create call received values from 2000 to 10000, each incremented by 2000. In this case, lowerBound = upperBound – 2000 and writeTo = lowerBound / 2000. The children threads also received a single upper bound parameter, each incremented by 100, starting at 100 up to 10000. Like in the parent threads, the childrens’ lowerBound = upperBound – 100 and writeTo = lowerBound / 100. This arithmetic allowed for the minimization of parameters passed which subsequently reduced the need to cast and uncast values. Despite not facing a major issue when coding the program, multithreaded thinking, parameter value, and parameter casting were areas that required some research and careful planning.

# Compilation and Execution

Compiling and running this application is very straight forward. Despite the project’s specification not requiring arguments, I decided that as a good practice, to include the -help optional parameter to provide users with some information about the application. While verifying the output, I found myself wishing I could see the all the random integers generated and thought other users might want the same. Hence, I also included the -printInput optional parameter that prints all 10000 numbers well formatted to the output file. Here are the application usage instructions:

Compiles the source code in file vagner.c into object file and ignores warnings with flag -w: **gcc -c -w vagner.c**

Links the object file vagner.o to executable vagnerApp.exe: **gcc -o vagnerApp.exe vagner.o**

Runs the application and produces only required output: **./vagnerApp.exe**

Runs the application and prints all random integers in addition to required output: **./vagnerApp.exe -printInput**

Prints useful information about the application to console and file:  
**./vagnerApp.exe -help**

Using the compilation and optional parameters during the execution, the application produces the set of outputs shown in the next section.

# Sample Output

In this section, the output produced by the three different executions of the application are shown. When running the program, the user can run it without any parameter to print only the required output; run the program with parameter -printOutput to print the 10000 randomly generated integers and required output to file; run the application with parameter -help to print to console and file some useful information about the application. Below are the three outputs generated by the application using the options, compilation, and execution steps previously described.

### **Not passing any parameters into the program, as in ./vagner.exe**

\*\*\* Vagner Machado's Project #2 for CSC 340 \*\*\*

\*\*\* Values found using a single thread \*\*\*

The minimum value is -1073471402.

The maximum value is 1073734540.

\*\*\* These are the twenty lowest values for parent 1 \*\*\*

-1071568520 -1022860859 -1072368707 -1071134245 -1025740391

-958147840 -1067393412 -1062229499 -1073332912 -1054324298

-1051821803 -1064338202 -1025730169 -1065580480 -1070230830

-1072357353 -1061746926 -1064648930 -1073381681 -1070269842

\*\*\* These are the twenty lowest values for parent 2 \*\*\*

-1054152886 -1040039493 -1052453016 -1044070157 -1064927839

-1060029051 -1061210514 -1043800499 -1052152768 -1052201813

-1040710386 -1062642422 -1058958302 -1065197325 -1046912396

-1064697224 -1062026852 -1066611697 -1072658363 -1066046740

\*\*\* These are the twenty lowest values for parent 3 \*\*\*

-1033074921 -1031774032 -1050030766 -1028387162 -1063796805

-1052174027 -1059512548 -1066293600 -1072167408 -1031309861

-1070197262 -1058676718 -1051331757 -1030453722 -1056686980

-1023878922 -1066089089 -1039941617 -1050794961 -1073471402

\*\*\* These are the twenty lowest values for parent 4 \*\*\*

-1069284333 -1070973988 -1040664954 -1042291296 -1017719079

-1013846052 -1070941170 -1060726412 -1055464119 -1072507713

-1062728769 -1043387154 -1046448191 -1050183080 -1033452718

-1070632955 -1072379050 -1057604109 -1062924819 -1068238963

\*\*\* These are the twenty lowest values for parent 5 \*\*\*

-1059665786 -1066634753 -1073464956 -1062417487 -1040272004

-1057421146 -1061369937 -1065220562 -1050793974 -1048106917

-1059740068 -1066823384 -1063447932 -1049995690 -1041135815

-1031456253 -1072231312 -1072875211 -1049762937 -1056019591

\*\*\* These are the five lowest values \*\*\*

-1073381681 -1072658363 -1073471402 -1072507713 -1073464956

\*\*\* The lowest value using multithreading is -1073471402 \*\*\*

### **Passing -help into the program, as in ./vagner.exe -help:**

\*\*\* Here is some helpful information \*\*\*

\*\*\* STEPS IN THE PROGRAM \*\*\*

0. The program generates 10,000 random integers.

1. The program finds the lowest and maximum values using a single thread and prints the values.

2. Given the parameter -printInput as parameter, the program prints the 10,000 random integers to file.

3. The main process creates 5 threads and each is responsible for processing 2,000 of those random integers.

4. Each of the 5 threads creates 20 child threads. Each of 20 child threads finds the lowest value in 100 random integers.

5. This will find the 100 lowest values, 20 in each parent thread.

6. Each parent thread finds the lowest of its 20 integers, so there are 5 lowest values left.

7. The lowest 20 values for each child thread are printed.

8. The lowest 5 values for the parent threads are printed.

9. The lowest value out of lowest 5 is printed.

10. The main process scans the 5 lowest values to find the lowest value.

11. All output is printed to file vagner-output.txt.

NOTE: User can pass parameter -printInput to see all generated integers printed to file.

\*\*\* COMPILATION INSTRUCTIONS \*\*\*

gcc -c vagner.c // compiles the file to object file.

gcc -o app.exe vagner.o // links object file to executable.

OPTION 1: ./app.exe // runs the program without printing the random integers.

OPTION 2: ./app.exe -printInput // runs the program printing the random input integers.

OPTION 3: ./app.exe -help // runs the program and prints this message to console and vagner-output.txt file.

### **Passing -printInput into program as in ./vagner.exe -printInput**

\*\*\* Vagner Machado's Project #2 for CSC 340 \*\*\*

\*\* All randomly generated numbers \*\*

134120788 -550051310 -280295952 398249826 1059275012 -1057202059 400831076 231584613 -82203802 -671776944

// random integers between first and last two lines were omitted   
 // from this document print out, but are displayed in output file.

-694545503 218639736 938032010 -55167664 -70411270   
 244970942 710919512 -25850501 741615962 513108334

\*\*\* Values found using a single thread \*\*\*

The minimum value is -1073291411.

The maximum value is 1073739446.

\*\*\* These are the twenty lowest values for parent 1 \*\*\*

-1072023480 -1065184028 -1061088044 -1072134603 -1002748126

-1064182357 -1066994353 -1072808122 -1055717260 -1071042998

-1048885830 -1027304139 -1052602228 -1066733320 -1062757296

-1060171925 -1053340379 -1068577199 -1067290845 -1050344697

\*\*\* These are the twenty lowest values for parent 2 \*\*\*

-953454951 -1014378526 -1071847332 -1052049698 -1072741528

-1070270549 -1066904723 -1029557146 -1005123032 -1071929083

-997096671 -1042729670 -1035622277 -1026515613 -1065597807

-1042425672 -1067336086 -1061052393 -1062212905 -1049734467

\*\*\* These are the twenty lowest values for parent 3 \*\*\*

-1018015339 -1005522393 -1054039753 -1073053598 -1061071046

-1073148508 -1057081259 -1065908905 -1034764921 -1035990192

-1072030030 -1067164557 -1031006639 -1022947122 -1062518819

-1036782414 -1039675556 -1037877014 -1071898522 -1072270301

\*\*\* These are the twenty lowest values for parent 4 \*\*\*

-1033596433 -1048705115 -1061677632 -1070455888 -1041359432

-1070214550 -984164625 -1030257730 -1073291411 -1047561366

-1064816677 -1065199576 -1069026352 -1040644575 -1070476426

-1072306257 -1069705486 -1007916160 -1066676250 -1073193601

\*\*\* These are the twenty lowest values for parent 5 \*\*\*

-1044484980 -1054198594 -1068607817 -1058803505 -1056653678

-1032109271 -1057541527 -1060907348 -1039939901 -1066507755

-1048170015 -1040910136 -1054784591 -1044175481 -1028433141

-1072186396 -1043308196 -1072987812 -1011095284 -976804640

\*\*\* These are the five lowest values \*\*\*

-1072808122 -1072741528 -1073148508 -1073291411 -1072987812

\*\*\* The lowest value using multithreading is -1073291411 \*\*\*

# Conclusion

Deviating from the usual thinking of programs as a linear execution, students were required to design and implement a multithreaded program to find the minimum value in a range of random integers. While I did not encounter any major problems with planning and implementing the application, a careful and detail-oriented approach was needed to plan for parameter passing and to produce output in the desired sequence. Programming using the POSIX multithreaded standard introduced me to a powerful new approach to problem solving that uses the power of a multicore CPU to its fullest capability. This improvement happens, for example, when a multicore CPU is able to run tasks even when other tasks might be blocking, which improve throughput and ultimately, overall efficiency.