
LARGE ARRAYS

OVERVIEW

This assignment will study the use and misuse of large arrays and the effects on virtual memory from a practical and theoretical perspective. You will write a single program to compare the time required to read and write elements of large arrays in both row- and column-major order. You will also perform a page-fault analysis on a theoretical problem involving memory access of a two-dimensional array.

THE PROGRAM

In a C program, `matrix.c`, create a two-dimensional matrix where each element is a character (`char`). Your matrix must have 20480 rows and the size of each row must be exactly one page in size or 4096 bytes large. You will notice that the size of the array is large, too large to be placed on the stack. Therefore, it cannot be declared locally inside of any function. It may also be too large to be dynamically created with a call to `malloc()`, so it will not fit on the heap either. The only remaining option is to make it global. You may want to experiment with trying to place the array on the stack or the heap to see what happens.

IMPLEMENTATION SUGGESTIONS

Look back at your previous programming projects for hints on using timers in C.

RUNTIME EXPERIMENT

Conduct several run-time experiments where you access the elements of the array by row and column to either read or write to the array. Your experiment should include four different run-time measurements from the following operations:

1. Write Row – write a new value, other than zero, to each element in row-major order (row-by-row).
2. Write Column – write a new value, other than zero, to each element in column-major order (column-by-column).
3. Read Row – read the value of each element in row-major order.
4. Read Column – read the value of each element in column-major order.

Write any value into the array but avoid using a random generator to generate a value to store in the array. Read and perform an operation on the read value to avoid that the compiler removes redundant read operations in its optimization stage. And if you execute the write operation first, you can be sure that the read operation will be executed and not be eliminated by the compiler during program optimization.

You can perform the four experiments in a single program execution provided that you start and stop the timer for each of the corresponding operations separately. The advantage of performing all four experiments in a single program execution is that you can initialize the matrix during the write operations before performing any read operations on the initialized matrix. Do not print the values of the elements as any I/O will distort the timing results. Repeat each operation 10 times and compute the average execution time for the operation. For example, for each operation you may want to do something like this:

```
Start timer
Repeat 10 times
    Access all fields in the array in desired order, choosing only
    one of the operation shown above (1 - 4).
Stop timer
Display average time result
```

Illustrate the average run times for each of the four different operations in a single bar chart. Use the results from the experiment to write up a short report with the following information:

1. A short description of how you conducted your experiment and on what machine.
2. A table and bar chart with four different average runtime results from accessing the matrix in memory, as well as their corresponding standard deviations (use Excel to compute the standard deviation).
3. Your interpretation of the results from your table/chart and any implications you may draw from the experiment.
4. If any, problems you had with implementation and testing.

Explore additional compiler options to ensure that the compiler is not optimizing your code, thereby eliminating the bottlenecks you want to test. Explore the man page of `gcc` on the SSH server to find the options for disabling compiler optimization.

PAGE-FAULT ANALYSIS PROBLEM

For this problem consider the two-dimensional array and the column and row read operations exactly as specified above. Two processes *ReadRow* and *ReadColumn* implement the corresponding read row and read column memory access operations. Compute the total number of page faults for the two processes. Assume the following for the computation:

1. Each process is given 10 frames of virtual memory by the system.
2. The two-dimensional array is a global data element.
3. 2 of the 10 frames are used for code and stack together.
4. An LRU page replacement algorithm is applied to evict pages from memory.

Hint: Draw a diagram that shows the layout of the process in memory – it is the same for *ReadRow* and *ReadColumn*.

DELIVERABLES

Your project submission should follow the instructions below. Any submissions that do not follow the stated requirements will not be graded.

1. Follow the submission requirements of your instructor as published on *eLearning* under the Content area.
2. You should have at a minimum the following files for this assignment:
 - a. `matrix.c`
 - b. `runtimeExperiment.pdf`
 - c. `pageFaultProblem.pdf`
 - d. README (describes any significant problems you had)
 - e. Makefile

Submit a *README* file that describes any significant problems you had. Keep in mind that documentation of source code is an essential part of programming. If you do not include comments in your source code, points will be deducted. If you do not refactor code appropriately, points will be deducted. In addition to the source code and the *README* file (if needed), submit

- a report *runtimeExperiment.pdf* that describes the run-time experiment, the results, and implications. T
- a document *pageFaultProblem.pdf* that describes the solution of the page-fault analysis problem with illustrations on memory layout and steps on how you arrived at the solution.

As always, if you do not include comments in your source code and refactor your code adequately and address memory errors, points will be deducted.

TESTING & EVALUATION

Your program will be evaluated on the Department's public Linux servers according to the steps shown below. Notice that warnings and errors are not permitted and will make grading quick!

1. Program compilation with Makefile. The options `-g` and `-Wall` must be enabled in the Makefile. See the sample Makefile that I uploaded in *eLearning*.
 - If errors occur during compilation, there will be a substantial deduction. The instructor will not fix your code to get it to compile.
 - If warnings occur during compilation, there will be a deduction. The instructor will test your code.
2. Perform an evaluation run. At a minimum, the test run address the following questions.
 - Are read and write operations by row major performed and elapsed times computed?
 - Are read and write operations by column major performed and elapsed times computed?

DUE DATE

The project is due as indicated by the Dropbox for project 3 in *eLearning*. Upload your complete solution to the Dropbox. I will not accept submissions emailed to the grader or me. Upload ahead of time, as last minute uploads may fail.

GRADING

This project is worth 100 points total. The rubric used for grading is included below. Keep in mind that there will be deductions if your code does not compile, has memory leaks, or is otherwise, poorly documented or organized. The points will be given based on the following criteria:

Submission	Perfect	Deficient		
eLearning	5 points individual files have been uploaded	0 points files are missing		
Compilation	Perfect	Good	Attempted	Deficient
Makefile	5 points make file works; includes clean rule	3 points missing clean rule	2 points missing rules; doesn't compile	0 points make file is missing

			project	
compilation	10 points no errors	7 points some warnings	3 points some errors	0 points many errors
Documentation & Style	Perfect	Good	Attempted	Deficient
documentation & program structure	5 points follows documentation and code structure guidelines	3 points follows mostly documentation and code structure guidelines; minor deviations	2 points some documentation and/or code structure lacks consistency	0 points missing or insufficient documentation and/or code structure is poor; review sample code and guidelines
Access Time Measurement	Perfect	Good	Attempted	Deficient
read/write rows and columns	20 points correct, completed	14 points minor errors	6 points incomplete	0 points missing or does not compile
computes elapsed time using clock_gettime	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
Chart, Analysis, and Conclusions	Perfect	Good	Attempted	Deficient
experiment bar chart	5 points correct, completed	4 points minor errors	1 points incomplete	0 points missing
experiment description, analysis & conclusions	20 points correct, completed	14 points minor errors	6 points incomplete	0 points missing
page fault analysis problem	20 points correct, completed	14 points minor errors	6 points incomplete	0 points missing

I will evaluate your solution as attempted or insufficient if your code does not compile. This means, if you submit your solution according to my instructions, document and structure your code properly, provide a makefile, submit the requested reports and chart but the submit code does not compile or crashes immediately you can expect at most 70 out of 100 points. So be sure your code compiles and executes properly.