CS 3013 Operating Systems — WPI, A Term 2016
Craig E. Wills — Project 4 (25 pts)
Assigned: Friday, September 30, 2016 — Due: Thursday, October 13, 2016 (5:00pm)

Important: This project is to be done by each individual student. This is NOT a group project. The project may be done on the Linux systems of the CCC or on your virtual machine. It will be graded on the CCC Linux systems.

Project Description

The purpose of this project is twofold. First, you will compare the performance of standard file I/O using the *read()* system call for input with memory-mapped I/O where the *mmap()* system call allows the contents of a file to be mapped to memory. Access to the file is then controlled by the virtual memory manager of the operating system. In both cases, you will need to use the *open()* and *close()* system calls for opening and closing the file for I/O.

Second, you will extend the memory-mapped portion of your project to parallelize the processing of memory amongst multiple threads. For this portion of the project, ideally you will run your program where you have multiple-core machines and a controlled environment for testing.

For the project, you should write a program *proj4* that has some similarity to the *strings* command available in Linux. The Linux command prints all strings of printable characters with length four (the default) or more and is noteworthy because it works on any type of file—whether the file contains all text or not. You should try executing the *strings* command on a few files to get a sense of the type of strings that are embedded in various types of files.

For your project, you do not need to print the strings of characters, but instead you will be searching for specific instances of a given string. Your program takes a file name and a search string as command-line arguments and determines the number of instances for the given string in the file.

Please note that the strings of characters that you find will not be NULL-terminated strings as used in C/C++ so using a routine such as *strcmp()* will not work. Rather you will need to do a byte-by-byte comparison between the file source and your search string.

The following shows sample output from two executions of your program:

```
% ./proj4 proj4 start
File size: 12969 bytes.
Occurrences of the string "start": 12

% ./proj4 proj.C z
File size: 2894 bytes.
Occurrences of the string "z": 6
```

In the first execution, the program is run on its own executable file. In this case there are many non-printable characters in the file with 12 occurrences of the string “start” specified on the command line. In the second execution, the program is run on a source file where all bytes in the file are text. In this case, there are 6 occurrences of the string “z”. This example illustrates that the string may indeed be a single character in which case your program will return the count of occurrences of this character in the file.

In order to simplify your search, you may assume that the first character of your search string will not be repeated in the search string itself so that if an ongoing search fails, you can start over at the beginning of your search. For example, if the search string is “baby” and the file contents are “hellobababy” then a straightforward implementation would not find the search string. For purposes of this project a straightforward implementation is fine. If you want to create a solution that works for all search strings that is good, but not required and no additional credit will be given for its implementation.

Project Implementation

Now that we have described the functionality of your program, which will require all bytes of the input file to be read, the remainder of the project focuses on how your program reads the input. The default behavior of the program should be to read bytes from the file in chunks of 1024 bytes using the *read()* system call. However your program should have an optional third argument that controls the chunk size for reading or to tell the program to use memory-mapped file I/O. In the latter case your program should map the entire contents of the file to memory. The syntax of your program:

```
% ./proj4 srcfile searchstring [size|mmap]
```

where `srcfile` is the file in which to search for occurrences of `searchstring`. If the optional third argument is an integer then it is the `size` of bytes to use on each loop when reading the file using the *read()* system call. Your program should enforce a chunk size limit of no more than 8192 (8K) bytes. Your program should traverse the buffer of bytes read on each iteration and keep track of occurrences of the given search string as described above.

If an optional third argument is the literal string “mmap” then your program should *not* use the *read()* system call, but rather use the *mmap()* system call to map the contents of `srcfile` to memory. You should look at the man pages for *mmap()* and *munmap()* as well as the sample program `mmapexample.C` for help in using these system calls. Once your program has mapped the file to memory then it should iterate through all bytes in memory to determine occurrences of the search string. You should verify that the file I/O and memory mapped options of your program show the same output for the same file as a minimal test of correctness.

Parallelization

Correct implementation of the project functionality using both file and memory-mapped I/O is worth 13 of the points for the project. For 8 additional points on the project, you need to extend your program to allow the memory-mapped portion of the project to be parallelized. For this portion you need to allow an additional command line option of the form "pn" where n is the number of parallel threads. Your program should allow no more than 16 parallel threads. Thus a command line such as

```
% ./proj4 proj.C z p4
```

should cause four threads to be created where each thread processes a chunk that is one-fourth of the file contents. You should divide the work amongst the threads in such a manner to provide the best performance. Once each thread is completed you need to combine the results from each thread. You need to give careful thought how to combine results as the search string may start in the chunk of memory processed by one thread and end in the chunk processed by another thread. The search string can even extend across the entire chunk for a thread. You want to make your parallelized program as fast as possible so minimize or eliminate cases where threads must access shared resources. The output of the parallelized approach should be identical to the output for the other approaches. All approaches should be contained within the same executable with the command line controlling which approach is used.

Performance Analysis

For the remaining four points of the project you need to perform an analysis to see which type of I/O works better for different size files and how much performance improvement you observe for parallelization.

If you have access, you should try to test using a multi-core environment with minimal concurrent activity.

For this portion of the project, you should reuse the first part of the *doit* project, which allows you to collect system usage statistics. The two usage statistics of interest for this project are major page faults and the total response (wall-clock) time. A sample invocation of your *proj4* program on itself using *doit* with the largest read size would be the following where *proj4* prints its output and then *doit* prints the resource usage statistics for the program.

```
% ./doit proj4 proj.C z 8192
File size: 2894 bytes.
Occurrences of the string "z": 6
< resource usage statistics for proj4 process >
```

You should initially test your program running under five configurations for input files of different sizes. The five configurations are standard file I/O with read sizes of 1, 1K, 4K, and 8K bytes as well as with memory mapped I/O (no parallel threads). You should determine performance statistics for each of these configurations on a variety of file sizes. You might look at large files created for kernel compilation in finding a range of file sizes to test. Note:

not all operating system versions (particularly if running within a virtual machine) report major page faults. If this is the case then simply indicate as much and only report timing results.

Once you have executed your program with different configurations on a range of files, you should plot your results on two graphs where the file size is on the x-axis and the system statistic of interest (major page faults or wall-clock time) is on the y-axis. Each graph should have one line for the results of each configuration.

After comparing the performance of memory-mapped (single threaded) versus different read sizes, you should perform another set of tests for different numbers of threads. You should test with 1, 2, 4, 8 and 16 threads and have a performance line in your graph for each of these levels of parallelization. A question is if additional threads provide better performance even though the machines have a smaller number of cores.

You should include these graphs as well as a writeup on their significance in a short (1-2 pages of text) report to be submitted along with your source code. Be sure to describe the testing environment that you used. You should indicate which configurations clearly perform better or worse than others on a given performance metric and whether there is clearly a "best practice" technique to use.

Submission of Project

Use Web turnin to submit your project using the assignment name "proj4". You should submit the source code for your program, a script showing sample executions and a copy of your report if you do the last portion of the project.