CS 3013 Operating Systems — WPI, A Term 2021
Craig E. Wills — Project 2 (20 pts)
Assigned: Tuesday, September 7, 2021 — Due: Friday, September 17, 2021

Project Description

This project allows you to explore file I/O performance and multiprocessing. You will first 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. You will next extend the memory-mapped portion of your project to parallelize the processing of memory amongst multiple processes.

For the project, you should write a program *proj2* 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 characters 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 character. Your program takes a file name and a search character as command-line arguments and determines the number of instances for the given character in the file. You will need to do a byte-by-byte comparison between the file source and your search character. **Note that command arguments are strings (arrays of characters) so you'll need to access the first character of the argument to get the search character.**

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

```
% ./proj2 proj2 s
File size: 12969 bytes.
Occurrences of the character 's': 12

% ./proj2 proj2.cpp " "
File size: 2894 bytes.
Occurrences of the character ' ': 257
```

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 character 's' 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 257 occurrences of the space character. Note that quotes are needed on the command line to include a space as part of an argument.

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:

```
% ./proj2 srcfile searchchar [size|mmap]
```

where `srcfile` is the file in which to search for occurrences of `searchchar`. 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 character 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 C/C++ programs of `mmapexample` 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 character. 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 5 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 processes (you do not yet know about threads!). You may want to use the function *atoi()* to convert the numeric string to the corresponding integer. Your program should allow no more than 16 parallel processes. Thus a command line such as

```
% ./proj2 proj2.cpp z p4
```

should cause four child processes to be created where each process searches a chunk that is one-fourth of the file contents. You should divide the work amongst the processes in such a manner to provide the best performance. Once a process is complete it should print out its own results. Remember that processes do not normally share memory and for this

assignment we are not going to try and do so. You will have a result output line for each of your child processes where the total file size and number of character occurrences should match the results as if done by a single process.

You want to make your parallelized program as fast as possible so each process should only process its portion of the file contents. All approaches should be contained within the same executable with the command line controlling which approach is used.

Performance Analysis

For the remaining two 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.

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 statistic of interest for this project is the total response (wall-clock) time. A sample invocation of your *proj2* program on itself using *doit* with the largest read size would be the following where *proj2* prints its output and then *doit* prints the resource usage statistics for the program.

```
% ./doit proj2 proj2.cpp z 8192
File size: 2894 bytes.
Occurrences of the character 'z': 6
< resource usage statistics for proj2 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 processes). 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.

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 wall-clock time is on the y-axis. The graph should have one line for the results of each configuration.

After comparing the performance of memory-mapped (single process) versus different read sizes, you should perform another set of tests for different numbers of processes. You should test with 1, 2, 4, 8 and 16 processes and have a performance line in your graph for each of these levels of parallelization. A question is if additional processes 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 InstructAssist to submit your project using the assignment name "proj2". You should submit the source code for your program, a script showing sample executions and a copy of your report (in `pdf` format) if you do the last portion of the project.