CS 3013 Operating Systems — WPI, A Term 2020
Craig E. Wills — Project 4 (15 pts)
Assigned: Tuesday, October 6, 2020 — Due: Wednesday, October 14, 2020 (5:00pm)

Project Description

This project compares 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.

For the project, you should write a program *proj4* that takes a file name as command-line argument and computes the percentage of printable characters in the file. To do so you should use two Linux routines: *isprint()*, which determines if a byte value is a printable character; and *isspace()*, which determines if a byte is a space, newline, tab, etc. Check the man pages of these routines for details and the needed include file. For this assignment, if either of these routines return TRUE for a byte value then we will consider it to be "printable."

The only output from the program should be one line with the number of printable characters in the file, the total number of bytes in the file and a percentage printed as an *integer* between 0 and 100 such as:

```
% ./proj4 testfile
183 printable characters out of 217 bytes, 84%
```

The default behavior of the program should be to read bytes from the file in chunks of 1024 bytes using the *read()* system call. You will need to do a byte-by-byte check looking for printable characters. However your program should have an optional second 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 [size|mmap]
```

where `srcfile` is the file on which to determine the percentage of printable characters. If the optional second 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 the number of bytes and printable characters.

If an optional second 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` or `mmapexample.C`) for help in using these system calls. Once your program has mapped the file then it should iterate through all bytes in memory to count printable characters. 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.

Performance Analysis

The program functionally portion of the project is worth 13 of the points for the project. For the remaining two points you need to perform an analysis to see which type of I/O works better for different size files. 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 proj4.C 8192
517 printable characters out of 517 bytes, 100%
< resource usage statistics for proj4 process >
```

At the minimum, you must 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. You should determine performance statistics for each of these configurations on a variety of file sizes. You can look of subdirectories of "`/`" such as `/boot`, which might contain larger files, as you look for 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.

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. 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 "proj4". You should submit the source code for your program, a script showing sample executions and a pdf version of your report if you do the last portion of the project.