CS 502 Operating Systems WPI, Fall 2003
Craig E. Wills Project 3 (20 pts)
Assigned: Monday, October 20, 2003 Due: Monday, November 10, 2003

Introduction

In this project you will be writing a program to compute the checksum of a file. Checksums are used to characterize a string of data in a relatively few number of bytes. The better the checksum algorithm, the more likely it is that unique strings of data map to unique checksums. One use of checksums is to ensure that transmitted data is correctly received. While checksums are computed in the project, they are not the focus of the project and hence we will use a simple, computationally inexpensive, checksum where all bytes in a file are XOR'ed together to calculate a one-byte checksum. Your program should compute and print the checksum as an "unsigned" value.

Project

Given this introduction, the primary purpose of this project is to 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.

For the project, you should write a program *xcheck* that takes a file name as command-line argument and computes an XOR checksum on the contents of the file. The only output from the program should be the checksum itself. 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 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:

```
xcheck srcfile [size|mmap]
```

where `srcfile` is the file on which to compute a checksum. 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 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 a "running" checksum.

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` 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 compute the XOR checksum. You should verify that the file I/O and memory mapped options of your program compute the same checksum for the same file as a minimal test of correctness.

Performance Analysis

Once you have your program functionally working for both types of I/O then 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 usage statistics of interest for this project are page faults, both major and minor, as well as CPU time, both user and system. You may want to modify your *doit* program to print time values at the microsecond level for finer detail. A sample invocation of your *xcheck* program on itself using *doit* with the largest read size would be the following where *xcheck* prints the checksum (your checksum will likely be different) and then *doit* prints the resource usage statistics for the program.

```
% doit xcheck xcheck 8192
The XOR checksum is 204
< resource usage statistics for xcheck 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. As an aid in finding a range of file sizes, the directory `/var/log/` on the CCC machines has some large files such as `lastlog` or `wlog`. You should look for other files with a range of sizes.

Once you have executed your program with different configurations on a range of files, you should plot your results on a series of graphs where the file size is on the x-axis and the system statistic of interest (e.g. major page faults or system 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 in hard copy format in class on the due date. 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 `/cs/bin/turnin` to turn in your project using the assignment name “proj3”. You should submit the source code for your *xcheck* program. A hard-copy report on performance results using *xcheck* should be turned in at class time.