Memory/mmap Test Programs

A sophisticated programming technique is to create "test programs" which probe for the existence of certain features on a target platform, or which "prove" that something works a certain way. For example, many open-source programs use the GNU autoconf utility to automatically test for things such as the size of a `long`, or the presence of a specific library version, and create header files so that code can be compiled automatically on a variety of platforms.

In this assignment, you will be creating test programs to discover the answers to a variety of questions having to do with the virtual memory system and the `mmap` system call. Of course, you (should) already know the answers....they are probably in this unit's lecture notes, but your task is to create a program or system of programs to learn each answer through computational experimentation and **without user intervention**. To be a true test program, it must be capable of **determing the answer through conditional test, not simply printing out a foregone conclusion!** For example, the following is a valid test:

```
if (3+1==0) printf("ints are 3 bits long\n");
```

While the following is bogus:

```
/* Make sure to set this #define so the right answer is printed LOL */
#if INTSIZE==3
printf("ints are 3 bits long\n");
#endif
```

In addition to printing helpful messages for the benefit of a human observer, each test program will return a specific exit code as specified below. This would, if this assignment had any real benefit outside of this class, allow your test programs to be used in an autoconf-like scripting environment.

Problem 1 -- PROT_READ violation

If a file is mmap's with PROT_READ, but an attempt is made to write via that memory mapping, does this succeed, fail, or cause a signal? Test program will return: 0 if it succeeds (the value in memory changes), 255 if it fails (value remains the same) without causing a signal, or if a signal is received, exit value will be the signal number. To do this, you'll have to trap (handle) all possible signals.

```
$ ./mtest 1
Executing Test #1 (write to r/o mmap):
map[3]=='A'
writing a 'B'
Signal [redacted, figure it out yourself!] received
$ echo $?
{a number corresponding to the signal number is echoed here}
```

Problem 2 -- writing to MAP_SHARED

If one maps a file with MAP_SHARED and then writes to the mapped memory (you can test by writing a single byte), is that update visible when accessing the file through the traditional lseek(2)/read(2) system calls? Exit value 0 if your test shows that the file's byte changes, 1 if the byte remains the same.

Problem 3 -- writing to MAP_PRIVATE

Same question as above, except for MAP_PRIVATE.

Problem 4 -- writing beyond the edge

Create a small file which is larger than a page but smaller than two pages, i.e. not an exact multiple of the page size. mmap it MAP_SHARED with read/write access. Write to mapped memory just beyond the byte corresponding to the last byte of the file. Does the size of the file through the traditional interface (e.g. `stat(2)`) change? Exit code 0 if it changes, 1 if it does not.

Problem 5 -- writing into a hole

Again, create a small file not a multiple of the page size and write to the mapping corresponding to one byte beyond the last byte. Let's call this byte X. Now increase the size of the file by say 16 bytes by lseek'ing and writing one byte, thus creating a "hole" near the end of the file. Explore the file again and check to see if byte X is now visible in the file. Exit code 0 if it is visible, 1 if it is not.

In addition to test program #4 & 5, attach a small narrative of about a paragraph to your write-up explaining why you feel you are getting the results you are getting in tests 4 and 5.

Problem 6 -- reading beyond the edge(s)

Create a small test file less than one page long. Establish an mmap'ing of the file which has a length of two pages (8192 bytes). a) First attempt to read a byte in memory which is beyond the corresponding end of file, but still falls within the first page. If that read succeeds, print a debugging message announcing this and showing the byte value. If the read fails and causes a signal, trap the signal, print a message with the signal number, and exit with the signal number as the exit code. b) Now perform the same test on a byte in memory which falls within the second page. The exit code of your program will be 0 if and only if both reads succeed. Otherwise it will be the signal number which was received, and the debugging output will tell us where it failed.

In addition to program #6, write a short (paragraph or two) narrative explaining what is happening behind the scenes and why you are getting the results that you are getting.

NOTE: You could implement 6 separate programs, or you could combine all of these into one test program which accepts a single command-line argument that is the test number to run. Either way, your test program(s) must be entirely self-contained. Do not rely on any prior manual setup (such as creating test files). You can assume that the current working directory in which the program is run has the proper permissions. You are encouraged to have additional debugging output which helps you visualize the contents of memory and file as you go along. However, the test results must be reflected in the exit codes as described above.