# CS431 Operating Systems
# Programming Assignment 5 - Memory Mapping and Protection

## 1 Overview

Unix provides a capability to read and write to files using memory mapping, which is a mechanism allowing a process to access file data directly by including it inside its address space. There are two benefits to using memory mapping. First, it can significantly reduce file data movement during I/O operations since mapping eliminates the need for it to be copied into buffers, as is done by the traditional `read()` and `write()` functions. Additionally, memory mapping can be used to implement an efficient mechanism for inter-process communication in that multiple processes can map the same file and see the effects of any writes to that file. When multiple processes map the same file, its contents are shared among them.

In addition to memory mapping, we are going to look at memory protection and the mechanisms used to set access permissions on regions of memory within a process' virtual address space. Memory can have read, write, and/or execute permissions, or no permissions at all. This assignment involves using memory mapping and memory protection operations.

## 2 Assignment

In this exercise, you will need to download the file *mmapexercise.c* from WorldClass and build the executable. A Makefile is provided to facilitate compilation on different platforms. When executed, this program creates a new file, named *mymmapfile.txt*, or truncates it to zero bytes if it already exists. The file is then mapped into memory using the `mmap()` system call, which means it can be read from or written to by accessing memory, i.e., just like using an array of type `char`. The program writes 12,289 bytes to the file, `munmap()`s it, then closes it.

For this assignment you will write very little code. Your job here is more to observe the behavior of the program and then draw conclusions about the underlying operating system internals. For this assignment do the following tasks:

- First, build and run the program unmodified. When we call `mmap()`, we establish a direct mapping between addresses in memory and positions in a file. In the program code on line 124 there is a statement (commented out) that deletes the file, i.e., `unlink(FILEPATH);`. Uncomment this line, rebuild the program, and run it again. Answer the following questions:

  1. **What happens in your program when you delete a memory mapped file?**
  2. **What happens to the file?**
  3. **Why do you think that is?** Hint: check the man page for `unlink()`.

- Modify the call to `mmap()` on line 115 to replace the argument `PROT_READ | PROT_WRITE` with `PROT_NONE`. This removes all access permissions to the memory the file is mapped into. Rebuild the program and run it again. Answer the following question:

    4. **What happened when you ran the program this time?**

- In your segmentation fault handler (the function `segfault_handler()`), use the `mprotect()` system call to set read and write permissions for the memory address the program attempted to access. The function prototype is:

```
int mprotect(void *addr, size_t len, int prot);
```

  The first argument is the memory address for which you wish to set protection. In your program, you have the starting address of the memory map, i.e., the pointer variable `map`. As you write to the map, you are accessing the memory in one-byte increments starting with `map[0]`. For purposes of this assignment, set the protection for memory addresses at the offset to `map` indicated by the array index $i$. In other words, let's say `map` points to address 1000. That means `map[0]` also points to address 1000, `map[1]` points to address 1001, `map[2]` to address 1002, and so on. That means you will simply add `i * sizeof(char)` to the address represented by `map` to point to the appropriate memory address.

  The second argument to `mprotect()` is the length of the segment you wish to protect. Use 1 for this argument, so we are only trying to protect a single address.

  The last argument is the protection. In the call to `mmap()`, we removed the value `PROT_READ | PROT_WRITE`, which is a bitwise OR of read and write permissions. Use that value here.

  Remove the call to `exit(1)` in the signal handler on line 45 of the program.

  **IMPORTANT NOTE FOR MAC OS X USERS**: The above instructions apply, but instead of making this modification in the segmentation fault handler, make those changes in the bus error handler, removing the call to `exit` on line 60 rather than line 45. Your program should work properly.

- Build and run the program once again. Answer the following questions:

    5. **How many times in total did the program attempt to access the memory mapped file?** Hint: every read or write operation to/from memory is an access.

    6. **How many times was your signal handler called?**

7. The output of the signal handler shows the addresses (in decimal notation) that were being accessed which triggered SIGSEGV (or SIGBUS if you're using Mac OS X). **What is the difference in value between these addresses?**

8. Go into your terminal and run the command `getconf PAGESIZE`, which gives you the virtual memory page size for your system. **What is the value returned?**

9. Considering the answers to the previous two questions, **what can you conclude about memory protection operations on your system?**

# 3 Deliverables.

Submit the C code source file for your program with the modifications you made and the answers to the above nine questions (in a separate document) through the WorldClass assignment dropbox no later than the due date specified in the weekly assignment. **Be sure to add your name to the header block of comments in the program**.

# 4 Academic Integrity.

I recognize that there are only so many ways to write code that meets the requirements of this assignment, and you may consult with other students in this class on the general approach to a solution. Having said that, sharing code, pseudocode, or other implementation details is not permitted, and I expect all work to be your own as evidenced by:

- Program structure, control flow, variable and function names (aside from C library functions, of course), etc. should be manifestly distinct from others' work.
- Your program should be commented in a manner sufficient for me to ascertain that you understand what the code you submit does.

# 5 Some Food for Thought

The approach taken in this assignment can be used to implement demand paging in user space. Virtual memory can be implemented as a memory mapped file with segments not currently in physical memory simply having their protection bits set to `PROT_NONE` and those in memory having protection bits set to `PROT_READ | PROT_WRITE`. Accessing a segment not in physical memory generates a SIGSEGV signal from the operating system, and the signal handler can then be used to implement a page replacement algorithm. This is not graded, but something for you to think about. What else might we need to fully implement this approach to demand paging?