# EECS 678 - Lab 11 Report

### Author: Jace Kline 2881618

## Question 1

The time required to copy the file using read_write varies with the size of the buffer specified. Smaller buffer sizes take longer. The time required for memmap varies much less regardless of how you perform the copy. Discuss why this is, and in your discussion consider the influence of: (1) the overhead of the read() and write() system calls and (2) the number of times the data is copied under the two methods.

The overhead of using the read() and write() system calls is quite significant. The main factor that plays into this overhead is the mode switch that occurs when the OS kernel must take control away from the user-level process to execute the desired action, and then return control to the user-level process. As a part of this mode switch, the registers of the user-level program must be first saved into memory to allow register use by the kernel. The kernel also checks a number of security conditions to ensure that the user process and its return buffer are valid. Then, after the system call executes, the mode switch must take place "in reverse": the data retrieved by the kernel must be copied into user-space buffers, and the user context must be restored. The time required for this sequence of steps (system call) is orders of magnitude slower than the time required for a simple user-level procedure call. The mmap solution does not have the overhead of system calls (mode switching), and therefore the time required is lower than that of the read/write solution.

In both cases (read/write or mmap), the kernel must copy the source file's contents (on disk) into a kernel-level buffer on a chunk-by-chunk basis and, similarly, write the data from the kernel buffer that corresponds to the destination file to the actual destination itself (on disk). In the read/write solution, each iteration of the read/write loop also requires the specified buffer-sized block of data to be copied into the user-level buffer upon return of the system call before this data is again copied back into the kernel space's destination buffer corresponding to the destination file. Contrarily, this extra step of copying the data from kernel space to user space is bypassed during in the mmap solution, and the address of the source file's buffer will be mapped directly to the output file's corresponding buffer without the data being copied into the user process space. Hence, the data will be copied 4 times total in the read/write solution, and only 3 times total in the mmap solution. This fact also contributes to the time reduction observed by the mmap solution, as compared to the read/write solution.

## Question 2

When you use the read_write command as supplied, the size of the copied file varies with the size of the buffer specified. When you use the memmap command implemented the size of the source and destination files will match exactly. This is because there is a mistake in the read_write code. What is the mistake, and how can it be corrected?

The mistake in the read_write code is that the buffer size (in bytes) might not evenly divide the size of the source file. If the buffer size argument number does not divide the source file size, then the last read() call will not completely fill the input buffer, and then the write() call (using the same buffer) will write the entire buffer size to the file, even though the extra bytes in the buffer are not needed. The solution to this is to calculate what the remainder buffer size should be, and then perform a special read and write sequence for the remainder data in a smaller sized buffer. My solution to this problem is provided below. I have also included the file 'read_write_new.c' with the full working solution.

### Source Code for solution

```
// Get info on the source file to extract # of bytes (file size)
if(fstat(fdin, &statbuf) == -1) {
    perror("Could not read stats for input file.");
    exit(errno);
}

int srcbytes = (int)statbuf.st_size; // # of bytes in source file
int rembufsz = srcbytes % bufsz; // Size of the remainder buffer to read to
int iters = srcbytes / bufsz; // How many times we can iterate (read) with the input buffer size
char *src_rem = malloc(rembufsz);

// Copy the portion with specified buffer size
for (int i = 0; i < iters; i++) {
    if((read (fdin, src, bufsz)) > 0) {
        write (fdout, src, bufsz);
    }
}

// Copy the remainder (to make file sizes of source and destination match)
if((read (fdin, src_rem, rembufsz)) > 0) {
    write (fdout, src_rem, rembufsz);
}
```

## Evidence of working solution

```
> ./read_write sample.ogg copy.ogg 10
> stat sample.ogg | grep Size
  Size: 12595362        Blocks: 24608      IO Block: 4096   regular file
> stat copy.ogg | grep Size
  Size: 12595370        Blocks: 24608      IO Block: 4096   regular file

> ./read_write_new sample.ogg copy.ogg 10
> stat sample.ogg | grep Size
  Size: 12595362        Blocks: 24608      IO Block: 4096   regular file
> stat copy.ogg | grep Size
  Size: 12595362        Blocks: 24608      IO Block: 4096   regular file
```