Jay Offerdahl
EECS 678
Lab 9 Report

Lab 9 Report: Memory Mapped I/O

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. **This question is worth 20 points.**

This happens to be the case because of a couple of factors. First, with a smaller buffer size, smaller amounts of data can be copied within one iteration. This only means that more read/write calls would be used. Even if the overhead for the read() and write() system calls is relatively low, with a small buffer of say, 5 bytes, a 1 MB file would still require 200,000 read/write combinations to copy the file. The reason the two methods take similar time when using a larger buffer on the read_write version is because less read/write combinations are used. If I wanted to copy a 1 MB file with a buffer of 20,000 bytes, instead of 5 at a time, it would only take 50 read/write combinations, which is much less time consuming. As stated in the slides, the read/write commands no longer bottleneck the operation with larger buffers, because disk I/O takes the bottleneck's spot.

Considering the number of times the data is copied, using the read_write method, the data is essentially copied twice, once when reading into the buffer, and once when writing to memory from the buffer. This is highly inefficient when compared to the memmap implementation, which copies the data only once by directly mapping from the kernel-space buffers to the user-level process' virtual address space. Again, when using a smaller buffer, less data is copied at a time, and since the read_write solution copies the data twice, this will take longer to perform.

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 here is that the buffer will read in until there's nothing left to read in the input file. If the buffer size is 5, and the input file is only 10003 bytes, an extra two bytes will be read in, causing the sizes of the files to differ. This is more noticeable when using a larger buffer, because the file size differences will be larger. If I use a buffer of 100,000,000, or 100 MB, the file size becomes much larger, (97,657 KB) on the cycle machines. The reason the file sizes match with smaller buffers is because the increment is simply smaller, so there's less chance to read in a large amount of data that's not relevant to the file.

This mistake can be corrected by possibly checking if the end of the file has been read. According to the man-page on read() (http://man7.org/linux/man-pages/man2/read.2.html), "On files that support seeking, the read operation commences at the file offset, and the file offset is incremented bhyy the

number of bytes read. If the file offset is at or past the end of file, no bytes are read, and read() returns zero.