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.

The read/write solution is "slow" because it must read the contents of the source file located in a kernel-space buffer, write them to a user-space buffer and then write this buffer into the destination file's kernel-space buffer. This process must be repeated for each n-bytes of the file. A bigger buffer will be faster for this method because it each copy will copy more bytes and the program make less transfers overall. It has to read the file, copy this into a buffer, copy this into the destination buffer, then it has to go back to read the file again. It takes time to go between these steps so larger buffers will decrease the number of times it has to do the loop, which in turn creates less overhead.

The mmap method skips the user-space buffer entirely and only has to deal with kernel-space buffers. The buffers are allocated in single segments (making this similar in concept to doing the read/write solution with a buffer the size of the file but still managing with-out the user-space buffer). Since we allocate a destination buffer exactly the size of the file, we only have to copy the entire file once (whether we do the copy in a loop or using memcpy). The read/write method forces you to copy the entire file twice.

Mmap is faster because our bottleneck is just reading in the file from memory and we only have to write the entire file once.

Read/write still has to read in the file, and then it had to pass it though a middle buffer before writing to the destination. This still needs the system call overhead, but it adds more overhead through its internal copying.

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 read/write code can only allocate and copy data in buffer-sized chunks. This is a problem since the file size may not be exactly divisible by the inputted buffer size. This could be corrected by checking how much memory needs to be allocated for the file and modifying the last buffer memory allocation so it only allocates enough for the remainder of the file instead of the full buffer allocation. As it is now, the buffer always allocates the specified number of bytes for the copy even if there aren't that many bytes left in the input file.

Mmap allocates exactly enough space for the file, no more and no less. So it doesn't end up with awkward extra allocated space like the read/write method.