Memory Mapped I/O

Michael Jantz

Prasad Kulkarni

Introduction

- This lab discusses various techniques user level programmers can use to control how their process' logical address space can be manipulated to map to physical memory and elements of the file system in Linux.
- It is not directly related to the third project assignment, but it should help firm up your understanding of virtual memory in that it will use the virtual memory subsystem to do file I/O
- Our tar file this week is slightly bigger than normal. Go ahead and make the starter code for this lab:

```
bash> cd eecs678-mmio-lab/; make
```

Logical Address Space

- The loading and storing of data from disk and into memory is essential to almost all user programs
- However, memory management can be complicated as programmers have to manage concerns over concurrent access to a shared resource and memory storage limitations
- For these reasons, many operating systems abstract main memory into an extremely large, uniform array of storage, separating logical memory as viewed by each user process from physical memory
- Today, we will look at some of the system calls Linux provides to give application programmers simple and efficient access to their process' logical address spaces.

read_write

- This program copies a file using the read and write system calls we've already seen. It takes three command line arguments:
  - Source File to copy
  - Destination file to copy to
  - Buffer size per read and write
- Recall read and write use a fixed size buffer to store the data read from memory and to refer to before writing back to memory
- This program uses *malloc* to allocate memory for this buffer.
- When a process performs a *malloc(n)* the operating system increases its logical address space by *n* bytes and returns a pointer to the first byte in the (linear) address space

Running read_write

- To run read_write on the provided sample.ogg video with a buffer of size 5 bytes, type:

```
-bash-3.2$ ./read_write sample.ogg copy.ogg 5
```

- This may take some time. You should verify the generated copy.ogg is the same as the video sample.ogg. Run:

```
-bash-3.2$ totem copy.ogg
```

- sample.ogg is only a 12MB file. Copies of this size do not normally take such a long time.
- The reason the operation took so long is that we only allowed our process to allocate 5 additional bytes of memory to perform the copy. For every iteration of the read / write loop in the program, only 5 bytes was written to memory.

A Faster read_write

- We can tweak the command line argument to see how the system reacts to allowing larger buffer sizes.
- We can also use the Unix 'time' command to time how this process performs differently when used with a different sized read / write buffer. *time* reports the total (real) time needed to execute the program, the time spent executing in user mode, and the time spent executing in system mode. It's resolution is milliseconds:

```
-bash-3.2$ time ./read_write sample.ogg copy.ogg 10

real    0m4.417s
user    0m0.383s
sys     0m2.949s
```

- After a few buffer size increases, you should see the speed of the copy no longer increasing. This is because, as you increase the buffer size, the main copy loop does not have to loop as often to copy the file. Eventually, however, the overhead of the read and write system calls is no longer the copy operation's bottleneck.
- No matter how it is done, 12MB of file data will have to be read into memory, from disk, to copy the file. A single disk read is orders of magnitude slower than a system call. Data must also be written to the new file. Disk I/O eventually becomes the bottleneck.

Segmentation Faults Explained

- Now, let's switch to the *memmap* program.
- If you run the program in its current state, you will see:

```
-bash-3.2$ ./memmap sample.ogg copy.ogg
Segmentation fault
```

- Why is this happening? The program issues the instruction:

```
*dst = *src;
```

- dst and src are pointers initialized by their declaration to point to some location in memory. The space for the dst and src pointers is that for local variables – on the stack. However, the space *pointed to* by dst and src has not been allocated. In fact, they both point to NULL, or 0x0, a logical address which is designated as illegal on most systems.
- When a user process attempts to dereference a logical address (in C this is done by applying the * operator to a pointer to a logical address) that is not in its address space, the kernel sends the process a SIGSEGV signal to indicate it has performed an operation resulting in a segmentation fault.

Copying in Memory

- In this lab, we will copy the sample.ogg file using only direct memory operations. We will dereference a pointer to every byte in the source file and copy the data into space reserved for the destination file.
- To avoid segmentation faults, we could create a memory buffer area for the whole file using *malloc*, perform a small set of large or a large set of small read operations to get the data into the buffer, then write this buffer to a file.
- However, Linux provides a way of creating a direct byte-to-byte mapping of a file directly into your user process' logical address space. Copying data from file to file can then be done copying it from one region of a programs logical address space to another.
    - The OS takes care of updating file contents on disk
- This approach is called *memory mapping* and is done using the *mmap* system call.

mmap

- The *mmap* system call is described below:

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

  - *start* – The address at which to create the mapping. If the value passed in is NULL, the kernel will choose the address for you.
  - *prot* – The desired memory protection of the mapping
  - *flags* – Determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file.
  - The contents of the shared segment are initialized using *length* bytes starting at *offset* in the file referred to by the file descriptor *fd*.
- On success, this call returns a pointer to the memory mapped area. In the case of an error, it returns -1.

Preparing for the *mmap*

- There are a few preparations we need to make before mapping both the source and destination files into regions of logical memory
- Most obviously, we see that *mmap* takes a size argument. We should find the size of the input file before calling *mmap* to create a memory image of its contents
- We can use the *fstat* system call to obtain the size of an open file:

```
int fstat(int fd, struct stat *buf);
```

- This call stores several statistics about the open file corresponding to *fd* into the stat structure pointed to by *buf*. The stat structure is on the following slide.
- For our purposes, we will use the st_size field in this structure as the length of the file we wish to copy.
- For error checking purposes, you should capture the return value of this function, which is 0 on success and -1 on failure, and call *err_sys* with an appropriate error string if the *fstat* fails

struct stat

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

Allocating Destination File Space

- Less obviously, notice, that we have an open file descriptor to the destination file, but, as of yet, this file descriptor does not correspond to any file contents in blocks on the physical disk.
- If we map the destination file into memory before writing to that file, we create a situation where we have allocated physical memory that does not correspond to file contents of the destination file in allocated space on the physical disk.
- In the current state, if the program writes to this mapped memory, the operating system does not know where to store this data when the memory image is written out to the destination file
- The result is an error which is sent to our process in the form of a SIGBUS signal.

Allocating Destination File Space

- We can get around this by using file operations to force the allocation of space for the file on the file system.
- We will do this by, first, *seeking* to the last possible byte we may want to write for our copy operation, using the source file size we obtained earlier, and writing a dummy byte to this location in the file. In effect, writing the last byte requires that all previous bytes exist.
- The *lseek* system call takes a file descriptor and moves its cursor to the position specified by its arguments:

```
off_t lseek(int fd, off_t offset, int whence)
```

- *fd* is the file descriptor we wish to seek on, and *offset* is how far to seek in according to the directive *whence*. You may use the (size-of-the-source-file – 1) for *offset* and the SEEK_SET macro to simply set the cursor to an offset in the destination file corresponding to the size of the source file.
- Once its cursor has been set, you can simply write a dummy byte to the file using the *write* system call. This forces the operating system to allocate disk space for the entire file.
- Remember to check for error conditions on all your system calls
  - *lseek* and *write* both return -1 on error

Doing the *mmap*

- Now, we can go ahead and perform the *mmap* for both our source and destination files. One call to *mmap* will use the *fdin* FD to map the source file, the other will use the *fdout* FD to map the destination file. Refer to slide 9 for the type and order of *mmap* arguments
- For both *mmap* calls we can use the following arguments:
  - NULL for the start address to tell the OS to choose the address at which to create the mapping.
  - MAP_SHARED for the *flags* argument to tell the OS to carry the changes to the mapping through to the underlying file.
  - 0 for the *offset* to tell the OS we want the mapping to start at the start of the file.
- Now, the file mapped using *fdin* should be mapped with only read access privilege (PROT_READ), and the file mapped using *fdout* should be mapped with both read and write access privileges (PROT_READ | PROT_WRITE).
- On success, this call returns a pointer to the memory mapped area. As with the other system calls, *mmap* returns -1 on error.

Copying the File

- For this part of the lab, we leave it to you to determine how you wish to copy the file now that you have memory mapped both the source and destination files in your logical address space
- One possibility is to start at the pointer returned by *mmap* and copy each byte of source memory into the destination memory one at a time in a loop using pointer arithmetic
- Another is to use the *memcpy* system call. See the *memcpy* man page for information on this system call.

Testing

- When you are through, you should test your implementation again with the 'time' command.
  - Also make sure your copied video plays as well
- You should note that this implementation is faster than the *read_write* program with a small buffer, but about the same as *read_write* with a larger buffer.

Conclusion

- Theoretically, the *mmap* solution should outperform the *read / write* solution:
  - The *read / write* solution copies the contents of the source file (stored in a kernel-space buffer of memory) to a user-space buffer before copying these contents back to a kernel-space buffer corresponding to the destination file.
  - The *mmap* solution avoids this additional copy by directly mapping the kernel-space buffers allocated for the source and destination files directly to our user-level process' virtual address space.
- However, both methods require that data on the physical disk first be pulled into physical memory before it can be copied, and that the contents of the file must be present in physical memory before it can be written out to disk.
- The overhead of performing these disk read/write operations is the main performance bottleneck in many situations, including this one.