Adding mmap for files in JOS

Discussion and Reflection

Jay Kamat

Nick Liu

Ojan Thornycroft

Paras Jain

Introduction

A computer (based on the Von Neumann model) may be useless if it doesn't have I/O (be that a display, keyboard or hard drive). It is important, therefore, for an operating system to allow and manage access to shared I/O. Programs utilize this for all kinds of operations, from console input to serial communication to reading/writing files on disk.

However, typical mechanisms for device and filesystem I/O can be slow. In JOS, we have implemented a "microkernel" implementation where requests to the filesystem are sent through the standard inter-process communication system call library. To read just one byte off disk requires scheduling an IPC send, yielding control of the process, saving context, restoring context for the IPC callee, responding with the byte (from the user-space filesystem server), saving context yet again, restoring caller context and resuming execution. This is quite a bit of overhead just to read one byte! This mechanism can allow reading up to a page of memory at a time.

Moreover, this standard interface is not easy to use! One needs to manually manage calls to the filesystem library which requires learning the API and manually moving around memory. This imposes more cognitive load on the developer and adds unnecessary abstraction.

Instead, it would be better if a user could just pretend that the filesystem was just a bunch of bytes in memory. Then, the user could just access a page on disk just as

easily as they access a page in memory. This would help in removing some of the unnecessary abstractions in accessing devices like the hard drive! It would also allow the kernel developer to optimize performance of file system access - for example, if a user access one byte off disk, the system might as well read a whole page off disk since the primary cost of a disk read is in fixed costs (e.g. context switching in the OS or seek time for the hard disk). A user could transparently access memory while the system could dynamically “cache” pages off disk as the user accesses them.

This imagined filesystem library actually exists; it is called mmap. This interface allows C programs to map devices or files into memory, so they can be easily accessed by programs, with no additional syntax or a specific API. A user who wants to access a particular byte in memory can just access that byte just like they might access a byte in an array.

Background

Using the standard API implemented in JOS, we aimed to implement mmap. Before implementing mmap in JOS, however, we had to learn about what mmap did in Linux and then work to understand how it was designed.

The mmap function creates virtual pages that map to files, so we can access parts of the file via memory addresses. Mmap uses demand paging, which is essentially the idea of copying the page into physical memory on demand (during a page fault), rather than all at once when making the mmap call. In order to reclaim pages, after we

are done with a memory mapped file, we must free the memory map with the munmap function.

An mmap function is essential for JOS as it lets arbitrary C programs interact with devices and files as if they were memory. Rather than sending system calls whenever a user level program wants to read anything from a disk, mmap allows access to data on disk without issuing an expensive interrupt.

An additional feature of mmap is it's ability to create shared and private maps. With it's shared map feature, multiple processes can share the same map, which allows for an easy way to share data between programs. The private map feature allows a program to create a map which will leave the original file intact, while only making copies when the said program tries to make a modification to this file. These two features allow multiple programs to access the same file quickly, concurrently, and safely.

Problem

I/O is a critical piece of the OS interface. This system is necessary to ensure that programs can interact with system resources effectively. Traditionally, access to disk resources occurred through a file-system layer using commands like open, read and write to modify on-disk resources. This approach has key limitations, however. For example, this approach is quite expensive due to the microkernel implementation of the filesystem in JOS - the process of reading or writing bytes on disk required several

expensive context switches to execute. This also has the limitation of limiting the theoretical bandwidth that reads and writes can occur at due to the limitations on the IPC interface. We have implemented a rudimentary IPC interface for the system but to increase performance of the filesystem, we will need to improve the IPC layer.

Performance limitations are very important to the OS as the first interface for data is through the filesystem. User-mode applications often read and write input to and from file descriptors in the shell interface.

Moreover, existing interfaces impose API constraints on the user. Users must learn about a new interface and need to become accustomed to the intricacies of reading and writing files. Writing repetitive code to access the filesystem can get repetitive and is extra burden on users. In addition, more code through a more complicated interface means more bugs can occur! Users could forget to use small flags, for example, that may compromise system security.

In combination, it is clear that both performance and programmer justifications motivate the creation of a new interface for files. For devices, the justification is clear - current interfaces are difficult to use and leave much to be desired as users often have to worry about specific details of the communication protocol to interface with a device or need to worry about the specific implementation nuances of the device.

Approach

The calling process uses mmap() to instantiate the mapping. Flags can be set for options like shared or writable mappings. In this project, we focus only on mapping files into memory for reading by the calling process (as compared to anonymous mappings).

In order to implement mmap, we need to implement an mmap function and add it to the system call library in order to map devices or files into memory. In addition munmap is needed in order to free up the memory after it is done being used. Also, flags need to be used to determine whether a mapping is shared or private and different actions need to be taken for each scenario.

This project depends on a filesystem and therefore progress was stalled until the completion of the filesystem lab (lab 5). Features were then implemented step-by-step until the completion of the project: 1) trap.c was modified to capture the page fault, 2) a check was made to see if the VA had flags set for mmap, 3) then the VA was rounded down and a filesystem IPC call was made to get the relevant page(s) off disk, 4) the FS call switched control to the filesystem which fetched the file and returned the IPC request, 5) where then the page was copied and then 6) control was returned back to the faulting process.

This iterative development approach allowed debugging smaller portions of the code before proceeding onto other features. This also reduced the number of bugs by reusing the existing filesystem server, at only a minor performance cost (of using IPC).

Results

This version of mmap was implemented by first allocating enough pages to allow for mapping the desired file. We saved these pages by setting the PTE_P bit to 0 (to mark them as unused), and setting the PTE_SAV bit to 1. PTE_SAV is the same bit as PTE_W, and has meaning only when the PTE_P bit is unset.

A few modifications were made to prevent the memory manager from giving away memory with the PTE_SAV bit set. When a process requests memory from this mapped region, it triggers a page fault, which gets routed into the mmap page fault handler. This handler checks to see if the page fault was part of a mapping, and if not, it destroys the user environment. If it was part of a mapping, the page is properly acquired, and the file's data is copied into memory before the handler returns. This way, we don't need to read an entire file into memory, only PGSIZE blocks, as we use them.

Once munmap is called, the file is written back to disk, if the file was written with the write permission bit set. Right now, there is no support for shared pages or COW pages, but this version of mmap still has the primary advantage of avoiding loading all files into memory.

Example program

```
// First, open file 'lorem' and get the file id.
if ((r_open = open("/lorem", O_RDONLY)) < 0)
    panic("mmap(): opening file failed, ERROR CODE: %d \n", r_open);

// map
mmaped_addr = mmap(NULL, length, 0, MAP_PRIVATE, r_open, (off_t) 0);

// Read from second page first to test dynamic loading
cprintf("=> Read from mmapped region:\n%30s\n", (char*) mmaped_addr);

// unmap
munmap(mmaped_addr, length);
```

Sample output

The following example shows a basic mmap and a read from to test its use.

```
[jay@laythe cs3210]$ make run-testmmap1-nox
Test mmaping
READING FILE INTO 807000
=> Read from mmapped region:
Lorem ipsum dolor sit amet, consectetur……
No runnable environments in the system!
```

The next example shows how we don't need to load all of the file into memory, unless we are reading the file, as shown by the debug messages. These debug messages are printed out when we read from the file in the page fault handler.

```
[jay@laythe cs3210]$ make run-testmmap2-nox
Test mmaping
READING FILE INTO 80b000
=> Read from mmapped (post) region:
se cillum dolore eu<lots of lorem>
READING FILE INTO 807000
READING FILE INTO 808000
READING FILE INTO 809000
READING FILE INTO 80a000
=> Read from mmapped region:
Lorem ipsum dolor sit amet, consectetur<lots of loremx4>
```

This example shows how we don’t need to load all of the file into memory, unless we are reading the file, as shown by the debug messages. These debug messages are printed out when we read from the file in the page fault handler.

Conclusion

By then end of the project, we were able to successfully implement the primary features of mmap. This means that users can create private memory mappings to files using mmap and update files through those mappings using munmap. These features effectively solve the problem of slow and clunky I/O APIs by allowing users to interface with files through familiar memory mappings. Given more time, implementing copy on write and shared mappings provide users with even more speed and versatility when managing I/O. Overall, our implementation gives the core power of the mmap library while leaving room for further improvements in the future. Our approach of reusing the userland filesystem makes our mmap implementation potentially more secure than approaches used in Linux as it limits the potential harm an errant mmap call can cause.