JOS mmap()

Memory-Mapped File I/O Library

Christopher M. Jones
jonescm@mit.edu

Eugene Y. Sun
suny@mit.edu

Clark D. Minor
clarkm@mit.edu

I. Introduction

For our final project, we implemented a POSIX-compliant mmap library for JOS, an exokernel-style operating system. The library allows the user to map a memory object, such as a file or part of a file, to a region in virtual memory. During the implementation, we focused on the following features: lazy mapping of pages from the filesystem to the physical memory, modification of page fault and exception handling in the kernel, and the completion of read and write capabilities for the file system via mmap.

This report explores the motivation behind mmap's development and its use cases, provides an overview of the library's functionality and expected behaviors, details the implementation process, and offers suggestions for future improvements to the library's functionality.

II. Motivation

While it may not be immediately apparent why mmap is useful or when it is appropriate to use, a good way to approach this question is to look at mmap from considerations of both performance and usability.

In terms of performance, mmap saves the amount of seeking and reading in typical file I/O operations and does random access/write to files extremely well; by using shared paging in-between processes it saves a lot of memory when multiple processes want to access the same file.

Our modifications also provide for the easy implementation of custom page fault handlers in user space. Custom handlers can be applied to specific regions in memory, which allows programs to handle page faults in different ways depending on their address. This is a powerful concept, and something that can even be helpful to other libraries outside of mmap.

III. Description

Our mmap library follows the POSIX standard, and provides the user with the following functions:

```
void *mmap(void *addr, size_t len,
   int prot, int flags, int fd, off_t off);
```

The `mmap()` function maps the region in file with descriptor *fd*, starting at offset *off* and spanning across the length of *len*, to the virtual address at *addr*, with memory protection permission *prot* (for example, `PTE_W` for writes) and mapping-mode *flags*.

This implementation provides two modes of mapping: `MAP_SHARED` and `MAP_PRIVATE`, passed in through the *flags* argument. If called with `MAP_SHARED`, the mmapped regions are shared between processes and the file system environment; if called with `MAP_PRIVATE`, the mmapped regions are private to the calling process, doing copy-on-write whenever changes occur.

```
void *munmap(void *addr, size_t len);
```

The `munmap()` function unmaps the mmapped regions with length *len* at *addr*, and restores appropriate page fault handlers. The function supports unmapping of an entire mmapped region, partial regions, and multiple mmapped regions with a single function call.

```
int *msync(void *addr, size_t length,
   int flags);
```

The `msync()` function syncs the mmapped regions with the length *length* starting at address *addr*; it flushes the current content of the mmapped region to the file system to make sure that it gets updated.

IV. Implementation

Our implementation of mmap spans three areas: the file system environment, the kernel, and a new user library addition.

A. File System

The file system was changed to include write access to files, following labs and code from last year. We also added a new file system call, `block_request()`, which allows an environment to request that an individual block from an open file to be mapped into the caller's environment. The request also includes permissions that determine if the block should be mapped as read-only or read/write, and if it should be mapped as private (`PTE_COW`) or shared (`PTE_SHARE`).

If the mapping is private, the file system reinstalls the requested file block in its own virtual address space as copy-on-write, so that any future requests for writes trigger a page fault that copies the contents of the block to a new physical address. In this way, changes to the file system are not reflected in the calling environment; and likewise, changes to the calling environment's mapping are not carried through to the underlying file system.

The file system's flush request was also modified so that it allows the passing of a range in the file as well as a `force` boolean. If `force` is true, the file system will flush every block in the given range whether or not its dirty bit is set. This is needed because changes to an mmapped region change the

physical data, but do not mark virtual pages in the file system as used. This new form of flush is used by msync to sync changes in an mmapped region to disk.

We implemented the user library functions mostly inside `<lib/mmap.c>`, which contains the definitions of `mmap()`, `munmap()`, `msync()`, as well as the custom pagefault handlers for shared and private mapping.

B. Kernel

The main change to the kernel was the addition of storage in `Env` structs for multiple handlers. There is now one global handler, which acts as a default, and multiple region handlers that are associated with an address range. When a page fault occurs, the appropriate region handler is called if it exists; otherwise the global handler is called The global handler is useful for programs that fork with mmapped regions; fork can handle the non-mapped addresses and it isn't necessary to store the $k+1$ regions separately for an environment with k region handlers. New system calls have been added to set the global handler and region handlers.

In order to support multiple handlers, the semantics of the page fault upcall needed to change. Originally it accessed a global variable to find the handler to evoke; this has been changed to a function argument. Making this change required changing the assembly code to pop an argument off the stack, and changing the kernel trap handling to push the appropriate function pointer onto the stack before switching to the upcall.

In addition, a new system call has been created to scan an environment's address space for a contiguous region of free pages. This is needed so an environment doesn't have to know ahead of time where to mmap a file, and ensures there is enough space to map a contiguous region. If a page is allocated, and not marked as reserved (see below), it is counted as part of a candidate region. When a large enough region is found, the pages are marked as reserved, and a pointer to the first page of the region is returned.

We needed a way to mark address pages as in use but not allocated, so that access to those pages would cause a page fault that could then lazily request file blocks. Therefore, we needed to distinguish between free pages that are reserved and free pages that are unallocated. This is done by using some of the 31 unused bits in an unallocated pager's page table entry. For now, the first three bits after `PTE_P` are used to mark the free page as "reserved", but there is room for other types of free pages, as well as extra data that could be stored.

C. User Library

Finally, a new user-level library has been created to house the actual mmap functions.

Each mmapped region has an associated set of metadata that contains the `fileid` that is mapped, the `offset` into the file where the region lies, and the `minaddr` and `maxaddr` which signify where the region is in virtual memory.

This metadata lies in an array of structs that fit onto a single page in the programs address space. When a page fault occurs, the faulting address is looked up in these metadata structs, and the block to request is computed.

Because the kernel now has the ability to register multiple page fault handlers, a new fault handler can be associated with each mmapped region. The handler can be one of two, dealing with either `MAP_SHARED` or `MAP_PRIVATE` regions. Depending on the handler, the type of fault, and whether the block containing the faulting address has been loaded in yet, the handlers decide which block to request from the file system and what permissions to install the faulting page with.

To implement munmap, the mmapped regions contained in the passed address range need to be modified or removed. There are a few cases:

1) The region lies within the munmap range. In this case, the metadata for the region is removed from the table (the end address is set to 0).
2) The munmap range lies within the region. In this case, the region must be split into two—one for before the munmapped area and one after. As long as there is space in the metadata array, this is mainly bookkeeping.
3) The region and munmap range overlap. In this case, the boundaries for the region need to be adjusted.

In all cases, page fault handlers need to be removed for all the address within the munmapped region. This is done by calling the region handler system call with a null pointer, which is taken to mean that all handlers for that address range should be removed.

Unit testing consists of multiple user programs, each testing various combinations of mapping permissions, file system reads and writes, and mmap reads and writes. Test cases can be run with the `make run-%` command.

V. Future Development

One interesting modification that we thought of at the last minute is to store additional mmap metadata in the reserved (but not present) page table entries. For example, one could store a page's index in the mmap metadata struct in the remaining 31-bits, which would free the environment from the responsibility of tracking all of the page fault to memory region mappings.