Coursework 5: Independent project

Feature description

```
void *mmap(void *addr, uint64 length, int prot, int flags,int fd ,uint64
offset);
int munmap(void *addr, uint64 length);
```

For this coursework I will be describing an implementation of `mmap()` and `munmap()`. This is a way of creating a new mapping of files or devices into the virtual address space and allow us to read and write bytes from files, as well as being faster than using traditional methods which use the open(), read(), write() system calls. The *mmap* function also allows us to map in a big file quicker by only mapping in the page/s that we actually require (this depends on the type of implementation). This mapping of pages also allows for shared memory, which if implemented means different processes can use the same memory that in turn saves memory for any other processes.

The speed increase from using traditional methods comes in the form of the decrease in system calls which take up computational time as they need to switch between user mode and kernel mode to perform operations. When using *mmap()* the system calls upon the *open()* system call and then the *mmap()* system call to map in the file described by the file descriptor `fd` into the virtual address space for `length` bytes into the address `addr` with an offset of `offset` in the file. The offset argument must be a multiple of the page size of the system. The `prot` argument allows us to specify the protections of the mapping which specifies in which way the file can be used. The `flags` argument is used for specifying if the mapping can be seen by other processes and allows for shared memory which can be used for inter-process communication to be private and only available to one process. If the function is successful, then it will return a pointer to the mapped space or if it fails it returns [(void *) -1] which is an invalid address and indicates it has failed.

The `munmap()` function is used to delete the mappings created by a call of `mmap()` and means any further references to this will result in a SIGSEGV signal being sent to the process. The range of which mappings to delete is indicated by the arguments that are passed in, `void *addr, uint64 length` which respectively indicate where the mapping range starts and how many mappings there are to delete.

Design considerations

When creating this feature there are considerations to be taken which will have an effect on the final product I create. This means that there will be benefits and losses to certain features and I will have to decide what is deemed more important. One of these is the inclusion of many flags that will change how a file is mapped into memory and what can be done with it and by which process.

The first consideration of what can be done with the file will come in the form of the `int prot` argument which change the protections of the mapped page and in form it can be accessed:

PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.
PROT_NONE	Data cannot be accessed.

The protections can be used together using a bitwise OR of multiple protections, apart from PROT_NONE which can only be used by itself otherwise mmap will fail. By combining multiple protections, the process will be able to do more than one thing to the page, such as passing in both PROT_READ and PROT_WRITE would mean that the process is free to both read and write to the mapped page.

Another consideration for the implementation of mmap is the `int flags` argument which mmap uses to determine if updates to the current mapping will be visible for other processes and whether the changes will then be executed to the file from which the region is mapped from (pointed to by `int fd`). The different flags are:

MAP_PRIVATE	Data can be read.
MAP_SHARED	Any updates to the mapping can be seen by other processes which have mapped the same region and these changes will also be executed to the file.
MAP_ANON/MAP_ANONYMOUS	The fd argument is ignored and no file's contents are mapped into the mapping, instead all the contents are initialized with 0. The offset argument must be 0 since there is no file to offset.

The MAP_SHARED flag allows other processes to read the same memory which can actually allow inter-process communication through this shared memory. This is therefore a very important feature.

Another way in which my implementation of mmap can be altered is considering whether I want to implement an eager or lazy implementation of mmap. These two methods alter the way in which mmap is called and at what time they actually map the instructed file into memory. In an eager implementation of mmap, we have the kernel open the file and map it into virtual memory as instructed to the address provided (or have the kernel decide that if address is NULL) and then return the pointer to the mapping. On the other hand, in a lazy mmap implementation when mmap is called nothing actually gets mapped but we make sure to remember that a mmap request was called earlier. Later, when the user attempts to access this memory that was previously called with mmap there will be a page fault and when this occurs, we know that the user wants to access this memory, so we map in only the pages which the user wants access to.

There are benefits and drawbacks to either of these implementations. For example using eager implementation allows for the loading/mapping of the file to be done initially so later on when the data needs to be accessed there will be no performance penalty (such as page faults) which means that for tasks which require the fastest possible speeds when performing instructions on the data this may be the better option. However, a lazy implementation would mean that the mapping only happens when it is required which in turn means that if the data is never actually required it won't waste time mapping it. Also, if the mapping attempts to map a large amount when only some pages of it are required then only those pages would need to be loaded in instead of the whole amount which mmap was called for.

When I implement munmap() I must make sure that the mappings between the range provided by the arguments are deleted but also set to invalid memory addresses so if they are attempted to be accessed it will be obvious that they are incorrect.

Implementation details

In this case we will be creating an **eager** mmap() implementation.

To begin the implementation, we should understand what each function (both mmap and munmap) need to return in either a successful or unsuccessful operation. When running mmap() we shall take in the needed arguments and output a pointer to the mapped area we have created if successful or if unsuccessful we will return a pointer of [(void *) -1] and an error code so we know what happened that made this mapping unsuccessful. You could instead also use different negative numbers to indicate different error codes. On the other hand, munmap will return 0 if successful and -1 if unsuccessful (or a different negative number depending on the error code) and sets any further references within the specified range to invalid memory addresses. If a process is terminated then the region that has been previously mapped will be unmapped.

We will begin this implementation by creating a mmap.h file and inside of this we will define both the protections and flags inside here as unique integers for each type of them. This file will be included in any other files that require these flags. I am choosing to define them in the following way:

MAP_SHARED	1
MAP_PRIVATE	2
MAP_ANON	3

PROT_READ	1
PROT_WRITE	2
PROT_NONE	3
PROT_EXEC	4

To find pages in the virtual address space we can map to we can use the `struct kmem` which points to the head of a list of free pages that we can use. However, if an argument other than NULL is provided for the `void *addr` argument then we will try and find the nearest page above this address to use. If the address provided as an argument is where another mapping is located, the system will select a random mapping using the previously mentioned `kmem struct`. This will be done by the kalloc() call later.

In proc.c we will create the following functions and add them to defs.h:

```
  void *mmap(void *addr, uint64 length, int prot, int flags,int fd ,uint64
 offset);
  int munmap(void *addr, uint64 length);
```

In the mmap function we will begin by checking to make sure that all the arguments are all valid, we do this by checking if we have all the necessary arguments, such as checking that if the flags argument is set to MAP_ANON the offset is 0. Then we will use the `int fd` argument which has been passed in and check that this is also valid by checking it. Then also saving the file pointed to by the `int fd` in a struct file. If all the parameters are valid, we can proceed but otherwise we will return a negative value to indicate what error we have hit (this shows if the problem is with the file descriptor, the values of the other arguments such as length being 0 or other issues).

We then continue by creating a pde_t variable and set it to the current process' pagetable (PAGE_TABLE). We then set a pointer equal to the last pointer in the current mmap region (OLD_POINT) (this will be the processes mmap_sz variable plus the starting point of our mmap region). The OLD_POINT pointer now needs to be rounded up to the nearest page and is now the address at which we can begin to map things into it (NEW_POINT). We then use this pointer to determine at which point we have mapped all that we need to map by rounding up the sum of the file size and this pointer to the nearest page (FILE_POINT).

We then begin to run a loop while the pointer of the current page (CURRENT_POINT) is less than the FILE_POINT and every loop we do, we increase the CURRENT_POINT by the page size. In this loop we will check that mmap is not out of memory, if it isn't we use kalloc() (which uses a kmem struct as described earlier to find a page of memory) and assign the returned value to a variable called MEM_PAGE and proceed to use the mappages() function with the correct arguments [in order: PAGE_TABLE, CURRENT_POINT, the page size, the physical memory address of MEM_PAGE, any permissions for the file such as reading, writing etc.]. Then we will read in a pagesized number of bytes into the CURRENT_POINT and the loop is finished. We then keep doing this until we go past the FILE_POINT pointer. After the loop we will change the size of the processes mmap_sz to the new size. At the end we now return a pointer to the start of where we began the mmap process and this is where our mapping of the file begins. If the flag MAP_ANON was passed in originally, we do not map a file but rather just fill it with 0s. If the length is specified, then we don't map in the whole file but rather just as many bytes as is specified in `uint64 length`.

For munmap() we need to completely undo the mappings we have created using the mmap() function. Therefore, we will take in the arguments of `void *addr, uint64 length` and use them to do so. We will go to the address provided and set the value to a negative number (an invalid address) and continue doing this for `length` times.

Evaluation of the feature

The mmap feature is very useful for an operating system and can be used to provide benefits to the system and functionality to a multitude of features such as allocating memory, inter-process communication, the quick reading of files and precise control over memory protections.

A more in-depth dive into one of the features available to programmers if mmap is implemented correctly is the quick changing of bytes inside a bigger file. For example, the programmer can mmap with a file descriptor and give an offset to the function and a short length (compared to the file size) that allows for quick loading of certain bytes which then can be altered (given the protections allow it) and saved to the underlying file. This may be useful in a situation where the programmer knows that only these bytes will change (such as updating a database), and they will change frequently enough that loading the whole file each time and reading through it may not be worth it computationally speaking.

The implementation describe above has completed a basic mmap implementation in an 'eager' format. This now allows the programmer to use some of the features that come with implementing mmap. I have included protections that can be used depending on what the programmer wants to use, this will allow some data to be read only, or read and write and other variations. However, my implementation has a long way to go before some features would be useful.

One feature I would have like to include which I have not implemented is the choice between lazy and eager implementation. As I mentioned previously the lazy implementation (which I did not do) would use page faults as signals to map in specific pages only when needed, this means that no memory or extra pages would be mapped when it isn't used which would be useful for efficiency.

Another feature could be the use of flags such as MAP_SHARED which can allow for the memory to be accessed by more than one process. This could be done with a struct that contains all the processes that are using this mapping and it will only get unmapped when there are no processes using it. This allows for inter-process communication through shared memory which would open up a multitude of possibilities for processes that are running separately (for example if a computer has many cores).

Bibliography

Anon. (2020). *Lab: mmap*. Retrieved from Lab: mmap: https://pdos.csail.mit.edu/6.828/2019/labs/mmap.html

Kanich, C. (2020, November 3). *understanding mmap, the workhorse behind keeping memory access efficient in linux*. Retrieved from YouTube: https://www.youtube.com/watch?v=8hVLcyBkSXY&t=705s

Kerrisk, M. (2020, 12 21). *mmap(2) — Linux manual page*. Retrieved from mmap(2) — Linux manual page: https://man7.org/linux/man-pages/man2/mmap.2.html#EXAMPLES

Sorber, J. (2018, September 10). *Simple shared memory in C (mmap)*. Retrieved from YouTube: https://www.youtube.com/watch?v=rPV6b8BUwxM&t=60s

Tsoding. (2020, 12 28). *Why Linux Has This Syscall?!* Retrieved from YouTube: https://www.youtube.com/watch?v=sFYFuBzu9Ow&t=385s

YehudaShapira. (2017, July 16). *Xv6 Explained*. Retrieved from Github: https://github.com/YehudaShapira/xv6-explained/blob/master/xv6%20Code%20Explained.md