Feature Description

The system call *mmap* maps a 'file' into memory. Thanks to the file descriptor layer, most resources can be abstractly represented as a 'file', including actual files, devices or any other memory objects that a process can read or write to. *mmap* returns a pointer to the mapped region in a process's virtual memory but does not initially consume physical memory. Instead, it only consumes a portion of the caller process's virtual address space when called, and when this virtual memory is accessed or written to, the CPU generates a page fault as this virtual memory is not backed by physical memory. The kernel then attempts to resolve the page fault by mapping some physical memory around the faulting address, such as a physical page from the kernel's page cache or a page-sized block of physical memory. It then reads a page-sized block of a file into the allocated memory, allowing the process to read, or write changes to, the data of a mapping's underlying 'file'. *mmap* thus provides copy-on-write access to a file's data. This is how *mmap* implements demand paging, as the file is only read from the disk when it is accessed or written to, and initially, no physical memory is consumed at all.

At the most basic level, *mmap* mappings can be shared or private and file-backed or anonymous. A private mapping makes a mapping's changes invisible to other processes mapping the same file and the changes are never written back to the file. Contrarily, a shared mapping makes a mapping's changes visible to other processes mapping the same file, and all changes are guaranteed to be written back to the underlying file. A file-backed mapping maps a file in multiples of the size of a page. If a mapping only partially consumes a physical page after a page fault, the unused portion of the partially-used page is zeroed, and this portion would not be written back to the file in the case the mapping is shared. A private or shared mapping can also be anonymous, which means the mapping is not backed by a file and thus it is similar to requesting a portion of memory from the operating system. This is why anonymous mappings are used to allocate memory for the stack and the heap in some operating systems and can be utilized in the implementation of *malloc*. When multiple processes have mappings to the same file, their individual virtual addresses will point to the same physical address, so writes from one process become visible to another without the need to write back to the 'file'. This reduces memory overhead and eliminates inconsistencies. The kernel periodically synchronizes 'dirty' or modified pages of a shared file-backed mapping back to the file, which also can be done by a call to *msync*, although this will not be discussed further in this report. 'dirty' or modified pages to a shared file-backed mapping are also written back to the file if the mapping is unmapped in *munmap* or if the process terminates. It is also the case that the system call *fork* copies *mmap* mappings to a child process, and the attributes of the parent's mappings are preserved.

As seen in Figure 1, *mmap* attempts to create a mapping in a process's address space of 'length' bytes at offset 'offset' in the file specified by the file descriptor 'fd', preferably at the virtual address 'addr'. If 'addr' is NULL, then *mmap* chooses the page-aligned address when creating a new mapping. The type of mapping created is specified using the 'flags' parameter, which determines whether the mapping is either anonymous or file-backed and either shared or private. The 'prot' parameter describes the memory protection used for the mapping, which must not contradict the open mode of the file specified by the file descriptor 'fd' if the mapping is file-backed. The void pointer returned by *mmap* points to the mapped region in the virtual address space of a process, which is -1 upon failure.

Figure 1

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

Source: (see Reference 1)

mmap has a counterpart system call *munmap* which allows a process to remove mappings from a process's address space so any references to the mapped region are invalid. Any changes to a shared file-backed mapping are written back to the file in this system call. If the mapping was private, changes are not written back and are simply discarded. As seen in Figure 2, *munmap* removes the mapping starting at address 'addr' with length 'length' bytes. If the mapping starting at 'addr' does not exist, then *munmap* returns 0 to represent success. Similarly, if the mapping is successfully found and removed, then *munmap* returns 0. An example of failure would be if *munmap* were called with invalid parameters, e.g., an invalid virtual address or length. In this case, it returns -1.

Figure 2

```
int munmap(void *addr, size_t length);
```

Source: (see Reference 1)

mmap is a faster alternative to reading a file into heap memory, and reduces system call and memory overhead greatly in certain use cases. It does however come with its fair share of disadvantages. *mmap* maps files in multiples of the size of a page, leaving slack space which is wasted. *mmap*ping large files can also be less optimal as a greater a number of page faults is required, which is expensive.

Design Considerations

The *mmap* described in this report shall be minimalist to favour a simpler implementation, without sacrificing too many benefits. First, the *mmap* system call shall produce a 'mapping structure', which will be stored for use in some page fault handling function called in *usertrap*. In the handling function, a page fault should allocate a single page using *kalloc* and map it around the page-aligned faulting address for a mapping or, for shared mappings, the function should seek a physical page that is allocated for another process mapping the same file. Either way, a page table entry should be created for a mapping by a call to *mappages* using the faulting address. If the mapping is file-backed, the file should be read into this page by a call to *readi* using the correct page-aligned offset, and *usertrap* should resume to return to the user-space process. The *munmap* system call shall also be implemented, which will seek a mapping with the specified parameters and remove said mapping from the mapping structure storage (and potentially the process's page table if it has already faulted), removing any references to shared physical pages or file structures. It is at this point, *munmap* can also write all allocated pages for the mapping back to the file with *writei*, which eliminates the need to mark page table entries as dirty when they are modified. If a shared physical page is only referenced by one mapping, it can be freed when the mapping is unmapped. Of course, *fork* should be modified to copy mappings to the new child process and *exit* should unmap all of a process's mappings the same way *munmap* unmaps mappings. To follow, a number of specific design considerations will be discussed.

The first design consideration to be made is where the *mmap* mappings will reside in a process's address space. The 'traditional' organization of a process's address space (see Appendix 1) places the stack above the heap, which extends downwards towards the heap, which itself grows upwards. A region dedicated to *mmap* mappings is typically placed in between the heap and the stack and grows upwards towards the stack to "accommodate new mappings" (lwn.net. (June 30, 2004)). The address space of a process in Xv6 is a little different to the 'traditional' process address space as mentioned before (see Appendix 2). The Xv6 address space places a statically sized stack of size PGSIZE below the heap, and the heap itself expands from KERNBASE to PHYSTOP. This means there exists some unoccupied address space just above the heap and beneath the trapframe. This space is relatively small but could be used for *mmap* mappings and would mean the base address for mappings could be PHYSTOP, and the mappings would expand up to TRAPFRAME.

The following consideration to be made is the data structure used to store the mapping structures. As mappings are not actually mapped into a process's page table until they are accessed or written to, a composite data structure of independent mapping structures is required to store the information necessary for the mappings so they can be handled during a page fault in *usertrap*. When a process calls *mmap*, a new instance of the mapping structure must be created and stored amongst others in an appropriate data structure. An example of an appropriate dynamic data structure would be a linked list which would be extended per *mmap* call and contracted per *munmap* call. However, as most structures in Xv6 are statically-sized, a statically-sized table of mappings would suffice. This table could exist as a per-process table or a global table accessible by all processes. The former would be more appropriate to ensure mappings are encapsulated by a process and can be copied to child processes through calls to *fork*. An appropriate number of mappings per process would be 8 mappings.

The final consideration to be made is whether processes mapping the same file should share the same physical page after a page fault in *usertrap*. During a page fault, the mapping structure that maps the faulting address should be identified and its virtual address and permissions should be used with *mappages* to map some physical memory for the page fault. A very lazy *mmap* would map separate physical pages during page faults like this for all mappings which is simple to implement. In this lazy approach, shared mappings would still write modified pages back to the file when unmapped but, without sharing physical pages, changes made by one mapping would not be visible to another process mapping the same file. The inconsistencies between mappings would cause issues, as when *writei* is used to write to the file in *exit* or *munmap*, it would overwrite a previous write to the same file. Thus, it is sensible for processes mapping the same file to share physical pages, so changes are consistent.

Implementation Details

First, the two new system calls must be created so *mmap* has the 6 arguments as detailed in Figure 1 and *munmap* has the 2 arguments as detailed in Figure 2. These system calls must retrieve these arguments or return -1 if this is unsuccessful. Then they must finally return calls to an internal *mmap* and *munmap* respectively which take the same arguments. Pre-processor macros may be declared so it is easier to produce a desired mapping. In a new header file, the options for *mmap*'s 'prot' parameter should be defined, including PROT_NONE, which describes a mapping where

pages are inaccessible; PROT_READ, which describes a mapping where pages are readable; PROT_WRITE, which describes a mapping where pages are writable; and PROT_EXEC which describes a mapping where pages are executable. *mmap*'s 'prot' parameter takes the bitwise OR of PROT_READ, PROT_WRITE or PROT_EXEC or instead, 'prot' can be just PROT_NONE. *mmap*'s 'flags' parameter can be either be MAP_PRIVATE, which describes a private mapping or MAP_SHARED, which describes a shared mapping. If the caller process requires an anonymous mapping, they can use either MAP_PRIVATE or MAP_SHARED in conjunction with MAP_ANONYMOUS using the bitwise OR operator; otherwise, the mapping is assumed to be file-backed.

Next, a shared physical page structure can be defined (see Appendix 4). This structure will contain the id of a process, an index of the mapping in a process's mapping table, a pointer to a file structure, a reference counter, a spinlock, a pointer to an actual physical page and the pointer to the next shared physical page structure to form a linked list. The head of the global linked list should be kept in an appropriate place accessible to *usertrap*. Next, the mapping structure can be defined (see Appendix 3). This structure shall contain the the mapping's start address, the length of the mapping, the permissions of the mapping, the type of mapping, a pointer to a file structure, and an offset. An array of this mapping structure with a maximum capacity of 8 mappings can be placed in the 'proc' structure, so each process has a table of mappings.

The internal *mmap* should first perform validation to ensure the proposed mapping is not invalid. If any validation fails, *mmap* should return -1 to indicate failure. Then if 'flags' does not include MAP_ANONYMOUS, the mapping can be assumed to be file backed, and the file descriptor 'fd' can be used to retrieve a pointer to a file structure out of the current process's open file table 'ofile'. *filedup* should be used to increment this file's reference counter. The 'offset' and 'length' parameters must be rounded to be made a multiple of the size of a page, PGSIZE. It is also *mmap*'s responsibility to find a suitable start address for the mapping. If 'addr' is NULL, it should be set to the base address PHYSTOP to iterate over the process's address space until a valid region is found for the given length. If 'addr' is not NULL, 'addr' is rounded to the closest page and *mmap* do the same as described before. In both cases, the end of the *mmap* region (or TRAPFRAME) is reached and an appropriate location is not found, then *mmap* fails. A new mapping structure can then be created, and the start address, length, permissions, type, pointer to a file structure, and offset can be placed into it. *mmap* should then find an empty slot in the process's mapping table by iterating over the table and finding an entry with the start address attribute equal to -1. Then it should insert the created mapping into that index, so the mapping is stored. Finally, *mmap* returns the start address in the mapping structure for the caller process to use.

Once the caller process reads or writes a virtual address in a region mapped by *mmap*, Xv6 naturally induces a page fault as the virtual address is not mapped in the process's page table. Before outputting the page fault information and killing the process in *usertrap* as usual, an *mmap* handling function should intervene to attempt to resolve the page fault. If the faulting address, as returned by *r_stval*, is not in the mapping region from PHYSTOP to TRAPFRAME, then the page fault should be handled as usual. Otherwise, this new function must find the mapping responsible for the page fault by rounding the page fault address to the nearest page and iterating over the process's mapping table to find the mapping that surrounds the faulting address. If a mapping is not found, the function should fail so the page fault is handled as usual. If a mapping is found and it is shared, then there should be an attempt made to find a shared physical page structure that maps the same file as the mapping structure. The spinlock should be used when accessing each shared physical page structure as they may be accessed concurrently. If a shared physical page structure is found that maps the same file as the shared mapping, and the process id and index attributes are both -1, then this structure is appropriate, so the reference counter can be incremented and the physical address within the structure can be used to map the shared physical page into the process's page table using *mappages* with the page-aligned fault address and permissions specified in the mapping structure. *mappages* should mark the new page table entry with PTE_U to allow user access. If a shared physical page structure is not found, or the mapping is private, then a new linked list node must be created. A physical page should be mapped around the faulting address, which is allocated using *kalloc*. The physical page should be zeroed and then mapped into the process's page table using *mappages* with the page-aligned fault address and the permissions specified in the mapping structure. The shared physical page structure can then be created with the file structure taken from the mapping structure, the reference counter set to 1, a newly initialized spinlock and a pointer to the allocated physical page. Also if the mapping created was private, then the index attribute can be set to the mapping structure's index in the process's mapping table and the process id attribute set to the process's id to ensure the page is only available to the private mapping and cannot be copied by other mappings. If the mapping is shared, both of these attributes can be set to -1 instead. This new node can then be placed at the end of the linked list of shared physical page structures. If the mapping is anonymous, then the handling function should return successfully so the process is not killed, and *usertrapret* is called at the end of *usertrap* to return to the user space process. If it is a file-backed mapping, a page-sized block of the file can be read into the page using *readi* with the correct offset. Again, the handling function should then return successfully so *usertrap* returns to the user-space process.

fork should be modified so that it iterates over the parent process's mapping table and copies each mapping over to the new child process's mapping table. Each mapping copied should involve the use of *filedup* to increment the mapping's file structure's reference counter.

exit should be extended so that it unmaps mappings from the terminating process's address space. It must iterate over the mapping table of the terminating process to find slots in the mapping table that are used. If an index of the table contains a valid mapping, then *exit* must iterate over the global linked list of shared physical page structures to remove references to a physical page for the mapping at said index. If the mapping is shared, *exit* should find a shared physical page structure mapping the same file for the mapping at said index as well as ensuring the process id and index attributes of the structure are -1. If it is a private mapping, *exit* should find a shared physical page structure by comparing the index and process id attributes of a node with the mapping structure's index into the mapping table and the current process's id. If the mapping is shared, *writei* can be used for each node associated with the mapping structure to write pages back to the file using the offset 'off' parameter, and *fileclose* may be used to decrement the mapping's file's reference counter. Then it must evaluate whether the node at an iteration has a reference counter value greater than 1. If it is greater than 1, the reference counter is decremented and *uvmunmap* is called with the 'free' parameter set to 0 so the corresponding page table entry is unmapped from the process's page table without freeing the physical page. If it is equal to 1, then *uvmunmap* can be called with the 'free' parameter set to 1 to unmap the page table entry and free the physical page. In the latter case, the last reference to the physical page has been removed so the node can be removed from the linked list and destroyed.

The internal *munmap* check if the address 'addr' is in the *mmap* region of the process's address space and if 'length' is greater than 0. If these conditions are not met, then *munmap* should return -1 as the parameters are invalid. If these conditions are met, *munmap* should ensure the 'addr' parameter and the 'length' parameter are rounded to the nearest page, and *munmap* should iterate over the mapping table to find the mapping with a start address and length closest to the page rounded parameters. If a mapping is not found, it can return 0 to indicate success. If a mapping is found, then *munmap* should then do what *exit* does for every mapping: remove the mapping and any references to shared physical page structures (but only for the individual mapping). It can then return 0 for success.

Evaluation of the Feature

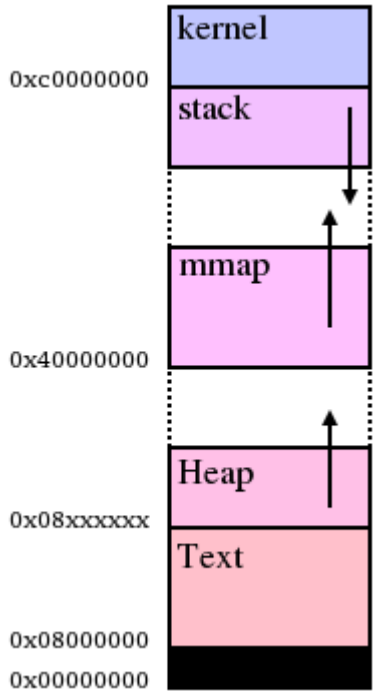
This implementation of *mmap* is sufficient and has a number of evaluable design and implementation decisions that can be discussed. For example, the placement of the *mmap* region in address space was sensible as it requires no reorganization of the process address space layout. This makes implementation simpler as it eliminates the need to change functions like *sbrk* to account for a change in the heap. There are very few disadvantages of placing mappings between PHYSTOP and TRAPFRAME, except from potentially limited space if a process wanted to map multiple large files. Furthermore, the advantage of keeping a table of mappings per process is there is no requirement for a spinlock to control concurrent access to mappings, as only the process itself has access to the table of mappings. However, even though a table of mappings complies with Xv6's statically-sized data structure 'convention', it does not leave room for much flexibility. Some processes may want to map a small number of large files (maybe 1 or 2 mappings at most) and others may want to map a large number of small files (potentially more than 8 mappings). This is inconvenient, and thus an enhancement to this implementation could be to provide each process with a dynamic linked list of mapping structures rather than a table, which would be expanded per call to *mmap* and contracted per call to *munmap.* Physical pages allocated for faulting mappings are taken from the kernel's physical page allocator, which makes for a simpler implementation. A potential enhancement to this would be to use physical memory instead of allocating a new physical page, and then reading the file into the memory in page-sized blocks during a page fault. Another limitation to the current implementation is that shared mappings write pages back to the file regardless of whether they have been modified or not. The specification of *mmap* mentions that the kernel is actually meant to mark pages with a 'dirty' flag once they have been modified. If this dirty flag is set, then the mapping writes the modified pages back to the mapping's underlying file. To do this in Xv6, a new macro could be added called PTE_D, which would be used to mark page table entries as dirty when the pages are modified. Then *munmap* and *exit* would only write back pages with the dirty flag set. The final suggested enhancement could be to add the *msync* function described in the feature description, which would write back pages with the dirty flag set to a mapping's underlying file without the need to *munmap* the mapping.

References

- Reference 1: man7.org. (n.d.). *mmap(2) - Linux manual page*. [online] Available at: https://man7.org/linux/man-pages/man2/mmap.2.html [Accessed 15 Dec. 2020].
- Reference 2: lwn.net. (June 30, 2004). *Reorganizing the address space [LWN.net]*. [online] Available at: https://lwn.net/Articles/91829/ [Accessed 21 Dec. 2020].
- Reference 3: GitHub. 2020. *Mit-Pdos/Xv6-Riscv-Book*. [online] Available at: https://github.com/mit-pdos/xv6-riscv-book [Accessed 22 December 2020].

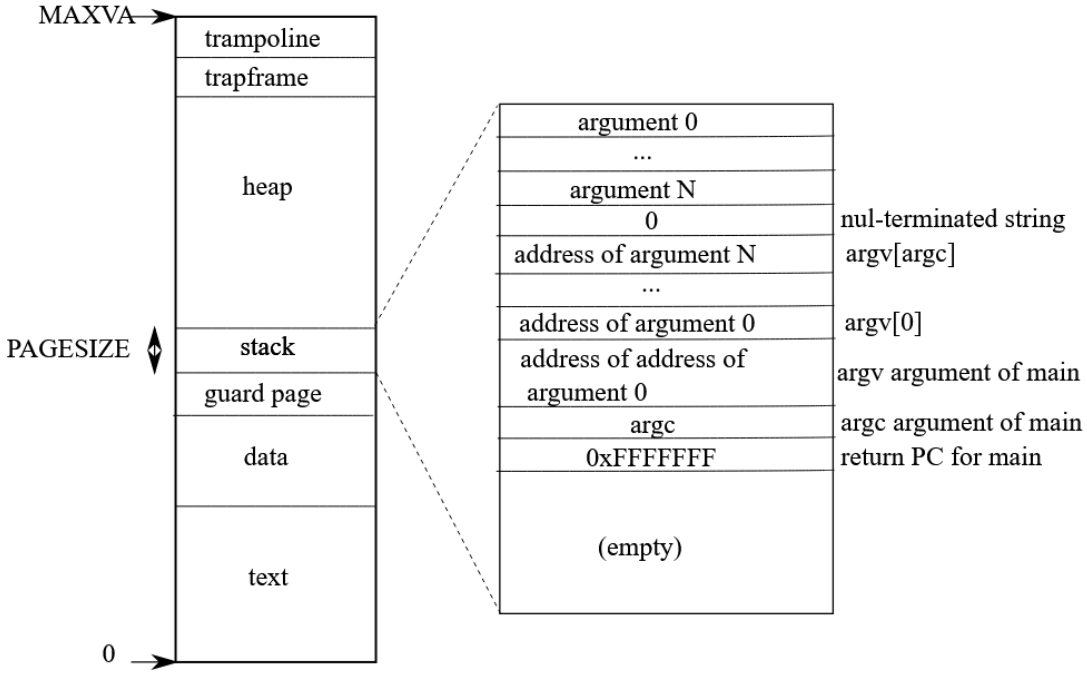
Appendices

1. Traditional Process Address Space



Source: (see Reference 2)

2. Xv6 Process Address Space



Source: (see Reference 3)

3. mapping structure

```
struct mmap_mapping
{
    uint64          start_address;
    uint            length;
    int             type;
    int             permissions;
    struct file*    file;
    uint            offset;
};
```

4. shared physical page structure

```
struct mmap_physical_page
{
    int                 process_id,
                        mapping_index;
    struct file*        file;
    struct spinlock*    lock;
    void*               page;

    struct mmap_physical_page* next;
};
```