





Response	Percentage
Yes, the current government is responsible	85%
No, the current government is not responsible	15%

November 05, 2024


**add timeout to tests**  
FARIHA TABASSUM ISLAM authored 5 days ago

Name	Last commit	Last update
 <a href="#">tests</a>	<a href="#">add timeout to tests</a>	5 days ago
 <a href="#">xv6-public</a>	<a href="#">Add xv6 base</a>	1 week ago
 <a href="#">README.md</a>	<a href="#">Add p5 description</a>	1 week ago

📄 README.md

```
title: CS 537 Project 5
layout: default
```

## Administrivia

- **Due Date** by November 19, 2024 at 11:59 PM

- **Due Date** by November 19, 2024 at 11:59 PM
- **Questions:** We will be using Piazza for all questions.
- This project is to be done on the [lab machines](#), so you can learn more about programming in C on a typical UNIX-based platform (Linux).
- **Handing it in:**
  - Copy the whole project, including solution and tests folder, to ~cs537-1/handin/login/p5 where login is your CS login.
  - Be sure to `make clean` before handing in your solution.
  - Only one person from the group needs to submit the project.
- **Slip Days:**
  - In case you need extra time on projects, you each will have 2 slip days for the final three projects. After the due date we will make a copy of the handin directory for on time grading.
  - To use a slip days or turn in your assignment late you will submit your files with an additional file that contains only a single digit number, which is the number of days late your assignment is(e.g 1, 2, 3). Each consecutive day we will make a copy of any directories which contain one of these slipdays.txt files. This file must be present when you submit you final submission, or we won't know to grade your code.
  - We will track your slip days and late submissions from project to project and begin to deduct percentages after you have used up your slip days.
  - After using up your slip days you can get up to 90% if turned in 1 day late, 80% for 2 days late, and 70% for 3 days late, but for any single assignment we won't accept submissions after the third days without an exception. This means if you use both of your individual slip days on a single assignment you can only submit that assignment one additional day late for a total of 3 days late with a 10% deduction.
  - Any exception will need to be requested from the instructors.

- Example slipdays.txt

1

• **Collaboration:**

- The assignment may be done by **yourself or with one partner**. Copying code from anyone else is considered cheating. [Read this](#) for more info on what is OK and what is not. Please help us all have a good semester by not doing this.
- When submitting each project, you will submit a `partners.txt` file containing the cslogins of both people in the group. One cslogin per line. Do not add commas or any other additional characters.
- Only one person from the group needs to submit the project.
- Partners will receive the same grades for the project.
- Slip days will be deducted from both members of the group if used. If group members have unequal numbers of slip days, the member with the lower number of days will not be penalized.

# xv6 Memory Mapping

In this project, you're going to learn more about the memory management subsystem in xv6. In this process, you will improve some existing mechanisms in xv6 to provide a more secure and performant system. Additionally, you will implement a new functionality which allows the user to map files into memory.

To provide better security, you will change existing xv6 to set correct protection for different program sections. Moreover, you will increase the performance by using copy-on-write when a process forks. Finally, you will support memory mappings through a pair of system calls, `wmap` and `wunmap`.

## Objectives

- To implement `wmap` / `wunmap` system calls (simplified versions of `mmap` / `munmap`)
- To understand the xv6 memory layout
- To understand the relation between virtual and physical addresses
- To understand the use of the page fault handler for memory management
- To understand how the OS loads programs into memory

## Project Details

### Task 1: `wmap` System Call

`wmap` has two modes of operation. First is "**anonymous**" memory allocation which has aspects similar to `malloc`. The real power of `wmap` comes through the support of "**file-backed**" mapping. Wait - what does file-backed mapping mean? It means you create a memory representation of a file. Reading data from that memory region is the same as reading data from the file. What happens if we write to memory that is backed by a file? Will it be reflected in the file? Well, that depends on the flags you use for `wmap`. When the flag `MAP_SHARED` is used, you need to write the (perhaps modified) contents of the memory back to the file upon `wunmap`.

This is the `wmap` system call signature:

```
uint wmap(uint addr, int length, int flags, int fd);
```

- `addr` : Depending on the flags ( `MAP_FIXED` , more on that later), it could be a hint for what "virtual" address `wmap` should use for the mapping, or the "virtual" address that `wmap` MUST use for the mapping.
- `length` : The length of the mapping in bytes. *It must be greater than 0.*
- `fd` : If it's a file-backed mapping, this is the file descriptor for the file to be mapped. You can assume that `fd` belongs to a file of type `FD_INODE`. Also, you can assume that `fd` was opened in `O_RDWR` mode. In case of `MAP_ANONYMOUS` (see flags), you should ignore this argument.
- `flags` : The kind of memory mapping you are requesting for. Flags can be **ORed** together (e.g., `MAP_SHARED | MAP_ANONYMOUS`). You should define these flags as constants in the `wmap.h` header file. Use the snippet provided in the [Hints](#) section. If you look at the man page, there are many flags for various purposes. In your implementation, you only need to implement these flags:

a) `MAP_ANONYMOUS`: It's NOT a file-backed mapping. You can ignore the last argument (`fd`) if this flag is provided.

b) `MAP_SHARED`: This flag tells `wmap` that the mapping is shared. You might be wondering, what does "*shared*" mean here? It will probably make much more sense if you also know about the `MAP_PRIVATE` flag. Memory mappings are copied from the parent to the child across the `fork` system call. If the mapping is `MAP_SHARED`, then changes made by the child will be visible to the parent and vice versa. However, if the mapping is `MAP_PRIVATE`, each process will have its own copy of the mapping. Furthermore, the in-memory changes will be written to the underlying file on `wunmap` if `MAP_SHARED` is set. In case of `MAP_PRIVATE`, changes are not written back. You do **NOT** have to implement `MAP_PRIVATE` in this project, and there will be no tests for that. **MAP\_SHARED** should always be set. If it's not, return error.

c) `MAP_FIXED`: This has to do with the first argument of the `wmap` - the `addr` argument. Without `MAP_FIXED`, this address would be interpreted as a *hint* to where the mapping should be placed (The kernel makes the final decision on the virtual address to place the mapping at). If `MAP_FIXED` is set, then the mapping MUST be placed at exactly "`addr`". In this project, you only implement the case with `MAP_FIXED`. *Return error if this flag is not set.* Also, a valid `addr` will be a multiple of page size and within `0x60000000` and `0x80000000` (see [A Note on Address Allocation](#)).

Your implementation of `wmap` should do "lazy allocation". Lazy allocation means that you don't actually allocate any physical pages when `wmap` is called. You just keep track of the allocated region using some structure. Later, when the process tries to access that memory, a *page fault* is generated which is handled by the kernel. In the kernel, you can now allocate a physical page and let the user resume execution. Lazy allocation makes large memory mappings fast. To set up the page fault handler, you'll add something like this to the `trap` function in `trap.c`:

```
case T_PGFLT: // T_PGFLT = 14
    if page fault addr is part of a mapping: // lazy allocation
        // handle it
    else:
        cprintf("Segmentation Fault\n");
        // kill the process
```

You can make some simplifying assumptions to implement `wmap`:

- All mapped memory is readable/writable. If you look at the man page for `mmap`, you'll see a `prot` (protection) argument. We don't have that argument and assume `prot` to always be `PROT_READ | PROT_WRITE`.
- The maximum number of memory maps is 16.
- For file-backed mapping, you can always expect the map size to be equal to the file size in our tests.

## Task 2: `wunmap` System Call

The signature of the `wunmap()` system call looks like the following:

```
int wunmap(uint addr);
```

- `addr`: The starting address of the mapping to be removed. *It must be page aligned and the start address of some existing wmap.*

`wunmap` removes the mapping starting at `addr` from the process virtual address space. If it's a file-backed mapping with `MAP_SHARED`, it writes the memory data back to the file to ensure the file remains up-to-date. So, `wunmap` does not partially unmap any `mmap`.

## A simple example of `wmap`

Here, we walk you through a simple end-to-end example of a call to `wmap`. You may want to take a look at the helper functions referenced in this example - you'll most likely need to use most of them in your implementation. For simplicity, assume that we don't support lazy allocation and just allocate all the physical pages up front. Later, we will explain what needs to change to support lazy allocation. Suppose that we make the following call to `wmap`:

```
uint address = wmap(0x60000000, 8192, MAP_FIXED | MAP_SHARED | MAP_ANONYMOUS, -1);
```

This is the simplest combination of flags - we're requesting a shared (`MAP_SHARED`) mapping having the length of two pages and starting at virtual address `0x60000000` (`MAP_FIXED`). Let's assume no memory mappings exist at the virtual address range `0x60000000` to `0x60002000` (8192 = `0x2000`). So we can use that virtual address mapping to place the new memory map. You should keep track of these mappings in your code so you know if a range in virtual address space is free.

So far we only talked about virtual address - there needs to be some corresponding physical addresses that we can *map* these virtual addresses to. Physical addresses are managed at the granularity of a page, which is 4096 bytes long on x86. At this point you need to allocate some physical addresses to be mapped. Good news! Physical memory is already managed in the xv6 kernel and you'll just need to call existing functions in the kernel to get free physical addresses. Bad news! you can't request a large chunk of physical memory in a single call. In other words, you can't simply tell the kernel "Give me a physical region of memory that's 8192 bytes long so I can map my virtual addresses to". As it was said before, it works at the granularity of a page (4096 bytes). So if you need 8192 bytes of physical memory, you need to call the kernel helper function twice, each time it gives you the starting address of a physical memory region that is 4096 bytes long. Needless to say that the physical addresses returned by the kernel are not necessarily contiguous in the physical address space.

Now let's continue with our example. We need two physical pages because the virtual address range we're mapping is two pages long. The kernel function `kalloc()` (defined in `kalloc.c`) can be used to get a physical page. So we call

```
char *mem = kalloc()
```

and `mem` is the starting physical address of a free page we can use.



Note that we haven't actually *mapped* anything yet. We have just allocated virtual and physical addresses. To do the mapping, we need to place an entry in the page table. The `mappages` function does exactly that. In this case, the call to `mappages` would be something like the following:

```
mappages(page_directory, 0x60000000, 4096, V2P(mem), PTE_W | PTE_U);
```

The first argument is the page table of the process to place the mapping for. Each process has a page table as defined in `struct proc` in `proc.h`. The last argument is the protection bits for this page. `PTE_U` means the page is user-accessible and `PTE_W` means it's writable. There's no flag for readable, because pages are always readable at least. For this project, you can always pass `PTE_W | PTE_U` as the last argument. You should always apply the `V2P` (converts a virtual to physical address in the kernel) macro to the address that you get from `kalloc`. This is because of the way xv6 manages physical memory. The address returned by `kalloc` is not an actual physical address, it's the virtual address that the kernel uses to access each physical page. Yes, all physical pages are also mapped in the kernel. To get more details on this, refer to the second chapter in the xv6 manual on page tables. So, after the call to `mappages`, did we successfully map 8192 bytes of memory at 0x60000000? No! look at the call again - it maps only a single page. If we could get a large chunk of physical memory at once, we could have just called `mappages` once with the specified length, and get done with the mapping. Since life is not perfect, we have to map one page at a time, each time requesting a new physical page from `kalloc`. So we may complete our mapping by doing a second call:

```
mem = kalloc();
mappages(page_directory, 0x60001000, 4096, V2P(mem), PTE_W | PTE_U);
```

Notice how we advanced the virtual address by one page (0x60001000 = 0x60000000 + 0x1000). Now you can quickly build up on this example and do the mapping in a loop, getting as many pages as necessary from `kalloc`.

Now let's see how to remove a mapping. Suppose the user accesses the pages we allocated at 0x60000000 for a while, and does a call to `wunmap`:

```
wunmap(0x60000000);
```

First we need to find any meta data that we keep in the kernel for the mmap starting at 0x60000000, and remove that from the data structure (e.g. a linked list). Next, the page table must be modified so that the user can no longer access those pages. `walkpgdir` function can be used for that purpose. A typical call to `walkpgdir` may look like this:

```
pte_t *pte = walkpgdir(page_directory, 0x60000000, 0);
```

It returns the page table entry that maps 0x60000000. We'll eventually need to do `*pte = 0`, which will cause any future reference to that virtual address to fail. Before that, we need to free the physical page we received from `kalloc`. Each `pte` stores a set of flags (e.g. R/W) and a physical page address, called *Page Frame Number* or *PFN* for short. Using a simple mask operation, you can extract the address part of a `pte`. Look at the macros defined in `mmu.h`. The final piece of the code will look like this:

```
physical_address = PTE_ADDR(*pte);
kfree(P2V(physical_address));
```

We need to apply `P2V` because `kfree` (and `kalloc`, as explained before) only work with kernel virtual addresses. Only one page has been freed so far. You'll need to do the exact same calls, but this time passing a virtual address of 0x60001000 to free the second page:

```
pte_t *pte = walkpgdir(page_directory, 0x60001000, 0);
physical_address = PTE_ADDR(*pte);
kfree(P2V(physical_address));
```

So far we have assumed no lazy allocation is done. If lazy allocation is supported, all we need to do is to allocate physical pages "lazily". This is achieved by breaking the end-to-end process into two phases. In the first phase, you just "remember" the mappings for the process when `wmap` is called. To do that, you can use whatever data structure you wish. Virtual address and length of the mapping are the most important information to keep track of. The second phase is triggered on a page fault. You refer to your data structure to decide if the access should have been allowed. If so, you allocate a physical page and map **ONLY** the page that caused the page fault (i.e. which page does the page fault address fall into?). Therefore, the calls to `kalloc` and `mappages` should happen in the page fault handler, not during the `wmap` call.

### Task 3: Poor Protection in XV6 - A Simple Motivating Example

Compared to a real world operating system, xv6 uses a very simple memory layout. You are going to solve some of the issues in xv6 memory management. Consider the following C program:

```
#include <stdio.h>
char *str = "You can't change a character!";
int main() {
    str[1] = '0';
    printf("%s\n", str);
}
```

```
    return 0;
}
```

Compile and run this program on a Linux machine. What do you see when you run the program? Perhaps you are wondering why the program fails with a segmentation fault while we haven't gone out of bounds for the `str` array. You will soon know why this happens.

Now compile and run the same program as a user program on xv6 (you will have to change the header and the `printf` statement). What happened? Why didn't you get a segmentation fault? There must be something that the Linux machine is doing but xv6 is not, and it leads to this behavior. Let's see what the reason is.

## ELF

Run the `file` command on your executable file on a Linux machine: `file ./main`. What do you see as output? This is a part of output from an Ubuntu machine: `./main: ELF 64-bit LSB shared object ...`. Have you ever wondered what ELF is? Why can't we just take the machine code for a program and generate a file out of it? It turns out that you need to tell the OS where to load your program, where to load the program data, and many other reasons that require you to provide some metadata along with your program. That's the purpose of ELF, to help OS load/link your program and start executing.

We are not going to discuss all the ELF details here, there are many resources online you can check if you are interested. In ELF, we have some program sections, with each section having some specific information of some type. Simply put, the OS just looks at ELF program headers and decides what to do with that part. For example, one program header might contain the code for your program, then it tells the OS to load some part of the ELF file into a specified virtual address in memory.

It's best if we look at an example. Run `readelf -l ./main` on the same executable. The output will be a table showing program headers (Not all columns are shown for brevity):

Type	VirtAddr	FileSiz	MemSiz	Flags
PHDR	0x0000000000000040	0x02d8	0x02d8	R
INTERP	0x0000000000000318	0x001c	0x001c	R
LOAD	0x0000000000000000	0x0618	0x0618	R
LOAD	0x00000000000001000	0x0235	0x0235	R E
LOAD	0x00000000000002000	0x01a8	0x01a8	R
LOAD	0x00000000000003db8	0x0298	0x1268	RW

Look at the 4th row. The type is `LOAD`, meaning that the OS should load this into memory. The `VirtAddr` specifies which virtual address should it be loaded at. File and Mem Size fields specify the size of this header in file and memory. Flags tell which permissions should be set on memory pages containing these segment.

Now let's get back to our problem with the tiny C program. Why do you think in xv6 you can modify the second character, but in a real machine you cannot? That `str` is actually a pointer to an array that's stored in the read-only data section of the program. When the OS reads the ELF file, it knows it needs to mark that section as read-only, and hence, upon your first write, you will get a memory violation error. But xv6 doesn't respect the permissions set in the ELF file, and it just marks all memory as `RW` (read-write permissions). In xv6, you can actually overwrite your own code! Try it!

## Fix the Problem

There's a small modification you need make in the Makefile. Run `readelf -h` on one of the user programs in xv6. You'll see code and data are merged into a single read-write segment. To allow code and data to be put into different segments with different permissions, you need to find the target that's used for compiling user programs. It should look like the following:

```
_%: %.o $(ULIB)
    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
    $(OBJDUMP) -S $@ > $*.asm
    $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym
```

In the first line, get rid of the `-N` linker flag. Recompile and check the ELF segments again, you should see two segments. Once you've made this change, we can proceed with our fix.

To solve the problem, we need to find the place where we want to load an executable. What system call is it? Right, it's the `exec` system call. Look at `exec.c:exec` (format is `file:function`). You see a for loop in that function: `for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph))`. This is basically going through all the ELF headers. Inside the for loop, there is a function call: `if(loadvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz, ph.flags) < 0)`. This call is to load that program header into memory. What's missing in the input list of `loadvm`? Check the ELF program header table above, you see that flags are actually ignored here, which is exactly what we are looking for. You need to modify the `loadvm`

function so that it takes ELF flags as input and sets the correct permissions on memory pages. You do not need to consider the executable permissions, because not all 32-bit systems have an explicit exec permission bit in the page table entry. So, you only need to set correct read/write permissions.

After this fix, run your program in xv6 again. This time you should see a trap and panic (which is xv6's segmentation fault). Congratulations! You just solved a very dangerous memory issue in xv6. You deserve a cookie.

## Task 4: Copy on Write

When a process forks, xv6 duplicates all the memory pages from that process. This is not efficient, because if both child and parent processes only read a page, there is no need to duplicate that. To implement that, you will modify the fork system call to do copy on write when it is creating the child process. This works by using the same set of physical pages for the child process. But this can only go so far as no one writes to that shared memory. Therefore, there needs to be a mechanism to notify the OS if a write is issued to those pages. You might have noticed that this is very similar to lazy allocation. So, no surprise that the same trick is used here as well: we use the page fault handler to resolve writes to these pages. On fork, you just mark everything as read-only, in **both** child and parent page tables. If those processes try to write to those pages, your page fault handler is invoked, and there you can duplicate the page.

Implementing copy-on-write has a few challenges in xv6:

1. You might have thought the page fault handler code is very simple, you just duplicate the page and make both copies writable. But, what if your child also forks? Now you have three processes sharing the page. If you make the page writable on the first write, it results in incorrect behavior. This is solved by holding reference counts. For each physical page, you keep a reference count, which represents the number of references to that page. For instance, in our example of nested fork, the reference count would be 3. By incrementing and decrementing reference counts, you know exactly what to do for each page.
2. Keeping the reference count itself turns out to be an issue. Probably you are thinking about a large array of `int`s, which can be indexed by page frame number (pfn). Well, that's not totally off, but think about whether this array should be dynamically allocated. How about something like `ref_cnts = malloc(MAX_PFN * sizeof(int))`? Unfortunately, we don't have the luxury of `malloc` in the kernel. xv6 can only allocate one page at a time. So a linked list of pages? Maybe, but complex to manage. The almost better alternative is to use a static array. But static arrays have got their issues too. If you read the xv6 manual, it says until the first page table is set up, you cannot use memory above 4MB. So, you need to be careful so that your kernel in-memory image size doesn't grow larger than that. You can use 1 byte for each reference count, and we guarantee that maximum number of processes sharing a page wouldn't be more than 256 that can fit in 1 byte. This way, you can reference count 4GB of memory (max on 32-bit machines) using only 1MB of space.
3. During fork, you duplicate pages and mark them as read-only. This is all good and well, but how do you know later on if a write to a page should be allowed or not? In other words, what if the page was initially read-only? If so, you should not make it writable, instead you should kill the violating process with a segmentation fault. Again, some state needs to be kept to know if a page was originally writable. You can think of arrays again, but this time you can do something cooler! Let's take a look at the format for a PTE on a x86-32 machine:

Bit(s)	31-12	11-9	8	7	6	5	4	3	2	1	0
Field	Page Frame Address	Available	Global	Page Sz	Dirty	Accessed	Cache Dis	Write -Thru	User/Super	Read/Write	Presen

What are bits 9-11 for? Yes, available for OS use. There you go! Use those bits to know if something was writable at the first place. You only need one of the three bits. To do that, look at `PTE_*` constants inside `mmu.h`. Add your own constants to work with PTEs.

4. Another complication arises due to the TLB cache. Ensuring that we read the most up-to-date information is the problem with any caching system. If an entry is cached in TLB, and you change the corresponding PTE in memory, that TLB entry is not automatically invalidated. So, upon future accesses, TLB may have stale information about that entry. This would become an issue in some cases. For instance, assume a page is initially read-write and you mark it as read-only. In TLB, that page is still read-write (although it's read-only according to the in-memory PTE). So, if that page is shared between two processes, changes from one of them could be observed by the other (which obviously shouldn't be the case). The solution for this is to flush the TLB once you're changing a PTE. One way to flush the TLB is to reload the CR3 register. Upon a write to CR3, the hardware will make sure that changes to the in memory page table will take effect. CR3 is the register that holds the physical address of the page directory in x86. There's a helper called `lcr3` to load this register. Look at examples of how it's used inside `vm.c`. Reloading the CR3 simply means load the current page directory again.

## Task 5: va2pa System Call

```
uint va2pa(uint va);
```

- `va`: The virtual address to translate Add a new system call `va2pa` to translate a virtual address according to the page table for the calling process. Return the corresponding physical address (**NOT** the physical page! also consider the offset). If the virtual address doesn't translate to a valid physical address, return -1.

## Task 6: wmapinfo System Call

```
int getwmapinfo(struct wmapinfo *wminfo);
```



- `wminfo` : A pointer to `struct wmapinfo` that will be filled by the system call.

Add a new system call `getwmapinfo` to retrieve information about the process address space by populating `struct wmapinfo`. This system call should calculate the current number of memory maps (mmaps) in the process's address space and store the result in `total_mmaps`. It should also populate `addr[MAX_WMMAP_INFO]` and `length[MAX_WMMAP_INFO]` with the address and length of each wmap. You can expect that the number of mmaps in the current process will not exceed `MAX_UPAGE_INFO`. The `n_loaded_pages[MAX_WMMAP_INFO]` should store how many pages have been physically allocated for each wmap (corresponding index of `addr` and `length` arrays). This field should reflect lazy allocation.

## Return Values

`SUCCESS` and `FAILED` are defined in `wmap.h` to be 0 and -1, respectively (see [Hints](#)).

`wmap` : return the starting virtual address of the memory on success, and `FAILED` on failure. *That virtual address must be a multiple of page size.*

`wunmap`, `wmapinfo` : return `SUCCESS` to indicate success, and `FAILED` for failure.

`va2pa` : return the physical address on success, -1 on failure.

## Task 7: Modify `fork()` System Call

In addition to the changes for doing copy-on-write, you also need to make sure that memory mappings are inherited by the child. To do this, you'll need to modify the `fork()` system call to copy the mappings from the parent process to the child across `fork()`.

You can make some simplifying assumptions here:

- Because of lazy allocation, there might be some pages that are not yet reflected in the page table of the parent when calling `fork` (i.e. those pages have not been touched yet). *We'll make sure to touch all the allocated pages in the parent before calling `fork`, so you need not worry about this case.*
- Moreover, you can assume that a child process is not going to unmap a mapping created by the parent.
- Also, the parent process will always exit after the child process.

## Modify `exit()` System Call

You should modify the `exit()` so that it removes all the mappings from the current process address space. This is to prevent memory leaks when a user program forgets to call `wunmap` before exiting.

## A Note on Address Allocation

In xv6, higher memory addresses are always used to map the kernel. Specifically, there's a constant defined as `KERNBASE` and has the value of `0x80000000`. This means that only "virtual" addresses between 0 and `KERNBASE` are used for user space processes. Lower addresses in this range are used to map text, data and stack sections. `wmap` will use higher addresses in the user space, which are inside the heap section. For this project, you're **REQUIRED** to only use addresses between `0x60000000` and `0x80000000` to serve `wmap` requests. If a mapping doesn't fall within this range, return error.

### Hints

You need to create a file named `wmap.h` with the following contents:

```
// Flags for wmap
#define MAP_SHARED 0x0002
#define MAP_ANONYMOUS 0x0004
#define MAP_FIXED 0x0008

// When any system call fails, returns -1
#define FAILED -1
#define SUCCESS 0

// for `getwmapinfo`
#define MAX_WMMAP_INFO 16
struct wmapinfo {
    int total_mmaps;           // Total number of wmap regions
    int addr[MAX_WMMAP_INFO];  // Starting address of mapping
    int length[MAX_WMMAP_INFO]; // Size of mapping
    int n_loaded_pages[MAX_WMMAP_INFO]; // Number of pages physically loaded into memory
};
```

You are strongly advised to start small for this project. Make sure you provide the support for a basic allocation, then add more complex functionality. At each step, make sure you have not broken your code that was previously working -- if so, stop and take time to see how the introduced changes caused the problem. The best way to achieve this is to use a version control system like `git`. This way, you can later refer to previous versions of your code if you break something.

Following these steps should help you with your implementation:

1. First of all, make sure you understand how xv6 does memory management. Refer to the [xv6 manual](#). The second chapter called 'page tables' gives a good insight of the memory layout in xv6. Furthermore, it references some related source files. Looking at those sources should help you understand how mapping happens. You'll need to use those routines while implementing `wmap`. We strongly advise you take time and understand this chapter of the manual. You should be able to explain how xv6 manages the free physical pages (why should you use `V2P` and `P2V` macros and when?). You will appreciate the time you spent on this step later.
2. Try to implement a basic `wmap`. Do not worry about file-backed mapping at the moment, just try `MAP_ANONYMOUS` | `MAP_FIXED` | `MAP_SHARED`. It should just check if that particular region asked by the user is available or not. If it is, you should map the pages in that range. The goal of this step is to make you familiar with xv6 memory helpers (e.g. `mappages`, `kalloc`). You should get comfortable in working with PTEs (`pte_t`).
3. Implement `wunmap`. For now, just remove the mappings.
4. Implement `wmapinfo` and `va2pa`. As mentioned earlier, most of the tests depend on these two syscalls to work.
5. Support file-backed mapping. You'll need to change the `wmap` so that it's able to use the provided fd to find the file. You'll also need to revisit `wunmap` to write the changes made to the mapped memory back to disk when you're removing the mapping. You can assume that the offset is always 0.
6. Try the basic fix to load ELF headers with correct permissions into memory. Make sure you don't break your `wmap` functionality.
7. Go for `fork()` and `exit()`. Copy mappings from parent to child in the `fork()` system call. Also, make sure you remove all mappings in `exit()`. Finally, make sure `fork()` does copy on write.