Assignment 4

CS330: Operating System

November 25, 2020

1 Introduction

This assignment aims to consolidate Memory Management concepts of the CS330 course. You will be implementing memory-related system calls in gemOS.

2 Background

This section revisits some concepts you know already, while introducing some other concepts that will help you in implementing this assignment.

VM Area

A VM Area is a contiguous region in the virtual address space of a process. Each area is associated with a start address, end address, and some protection information. mmap() system call is used to allocate VM Areas.

Paging, TLBs, and Efficiency

You are familiar with the concept of paging. Most operating systems implement pages of 4KB and the mappings between virtual pages and physical frames are stored in the page tables. Further, to speed up the process of virtual-to-physical translations, some of these mappings are stored in the TLB, a high speed, limited-capacity cache. Hence, each entry in the TLB essentially provides information about 4KB memory. A TLB miss results in a page table walk, which is a costly affair. For applications with a large working set (ex. databases), a TLB miss consumes significant system effort. Hence, to make things more efficient, Huge Pages are used. Huge Pages are typically of 2MB size, thus each TLB entry now maps to larger area of the address space, leading to fewer translations. A part of physical memory is reserved to allocate frames to huge pages.

Huge Pages: Addressing, and Page Tables

Now, if the size of the page is changed, there would obviously be changes in the way the pages are addressed. Figure 1 depicts differences in the addressing schemes for 4KB and 2MB pages. The addressing scheme for a 4KB page is covered in the course slides. For 2MB pages, the third

level page table points to the physical frame, instead of the fourth level page table. Further, in a 4KB system, the last 12 bits give the offset in the physical frame, while in a 2MB system, the offset is given by the last 21 bits.

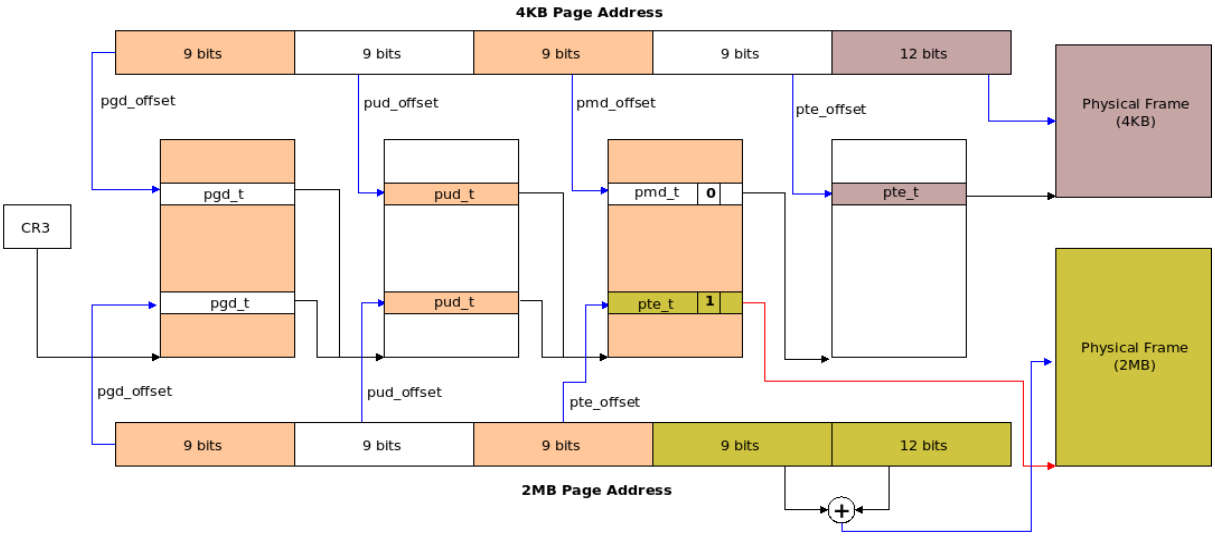

Figure 1: Address Translation for 4KB and 2MB pages

In the third level of the page table, the bit position 7 (Page Size (PS)) indicates if the address contained points to the next level of page table or it points to a page. If PS=0, the entry points to the base of a 4th level page table. If PS=1, the entry points to a 2MB page. The structure of the 3rd and 4th level page tables are shown in Figure 2 and Figure 3 respectively. For this assignment, you can ignore bit positions 3-11 and 51-63. You only need to modify bit positions 0, 1, 2, 7 (only for huge pages) and 12-50.

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
(M–1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1, determines the protection key of the page (see Section 4.6.2); ignored otherwise
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Figure 2: PTE Entry for 4KB page (4th level page table)

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 2-MByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 2-MByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 2-MByte page referenced by this entry (see Section 4.8)
7 (PS)	Page size; must be 1 (otherwise, this entry references a page table; see Table 4-18)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
12 (PAT)	Indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2)
20:13	Reserved (must be 0)
(M-1):21	Physical address of the 2-MByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1, determines the protection key of the page (see Section 4.6.2); ignored otherwise
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Figure 3: PTE Entry for 2MB page (3rd level page table)

With that background, let's now jump to the tasks of the assignment. The APIs to help you in implementation are described in section 5. If you wish to read more about the topics discussed, you can refer to the following resources:

- `https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html`
- `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`
- `https://www.kernel.org/doc/gorman/pdf/understand.pdf`
- `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-pdf` (PTE: Page 2913)

3 Task 1: Mappings - Show them their place! [50 marks]

In this task, you will be implementing the following system calls:

- void *mmap(void *addr, size_t length, int prot, int flag)
- int munmap(void *addr, size t length)

3.1 void *mmap(void *addr, size_t length, int prot, int flag)

3.1.1 Details

mmap() creates a new mapping (i.e vm_area) in the virtual address space of the calling process. A `vm_area` is represented by the structure `vm_area`, which holds information about the start and end address of the mapping, the protections(read/write) and the type(normal/huge page). The `type` of a `vm_area` on creation is *always* `NORMAL_PAGE_MAPPING`; it may later be changed to `HUGE_PAGE_MAPPING` as described in 3.3

Argument Description

`addr`: `addr` specifies the starting address of the new mapping. If `addr` is `NULL`, you have to choose a address which is page aligned in the virtual address space to create the mapping. If `addr` is not `NULL`, it should be considered as a hint about where to place the mapping. If requested mappings are free, you can either create a new mapping or merge with the existing mapping based on the scenarios explained here.

`length`: The `length` argument specifies the length of the mapping. It must be greater than 0 and need not be a multiple of page size. Length is in bytes.

`prot`: The `prot` argument describes the desired memory protection of the mapping. It can be bitwise OR of one or more of the following flags:

- `PROT_READ` - The protection of the mapping `vm_area` is set to `READ_ONLY`. The physical pages which map to this `vm_area` are also set to `READ_ONLY`. Any attempt to write to this page results in `SIGSEGV`.
- `PROT_WRITE` - The protection of the mapping `vm_area` is set to `WRITE_ONLY`. When it comes to physical pages, `PROT_WRITE` implicitly provides read access to them. Hence the physical pages which map to this `vm_area` will have `READ_WRITE` access.

`flag`: This argument takes the values `MAP_FIXED` or 0.

- `MAP_FIXED` - Don't interpret `addr` as a hint: place the mapping exactly at the address that is passed as an argument to the mmap() function. The `addr` should be a multiple of page size. If the specified address is already mapped with some `vm_area`, it cannot be used and mmap() will fail in that case. If `addr` is NULL, throw an error.

Return Value

On success, mmap() returns a pointer to the mapped area. On error, returns `-1`.

Creating and Merging VMAs

- Always create new mapping `vm_area` starting from `MMAP_AREA_START` (start) to `MMAP_AREA_END` (`end`). You shouldn't create new mapping in intermediate addresses unless hint address is specified. Refer Figure 4.

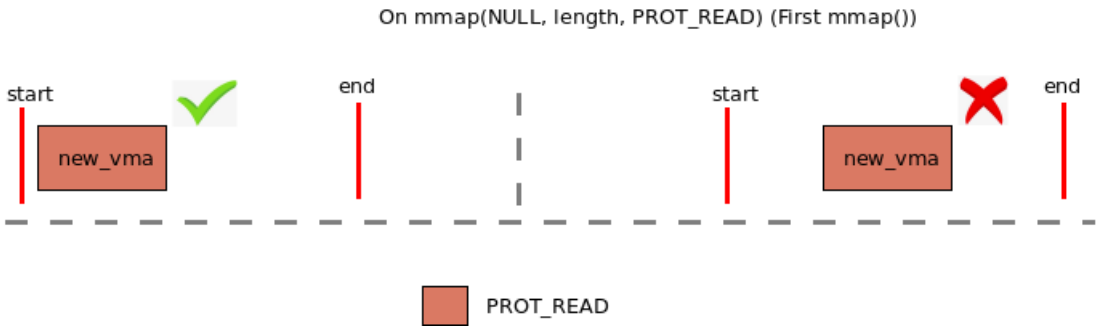

Figure 4: Creating new mapping

- If the hint address provided is not 4K aligned, start the scan from the next 4KB address. Example: mmap(10K, 1K) –> VMA starts from 12K, ends at 16K (assuming space is available between 12k-16k).
- The subsequent mapping (new mapping) should be created in next contiguous free address in the virtual address space. Refer Figure 5.

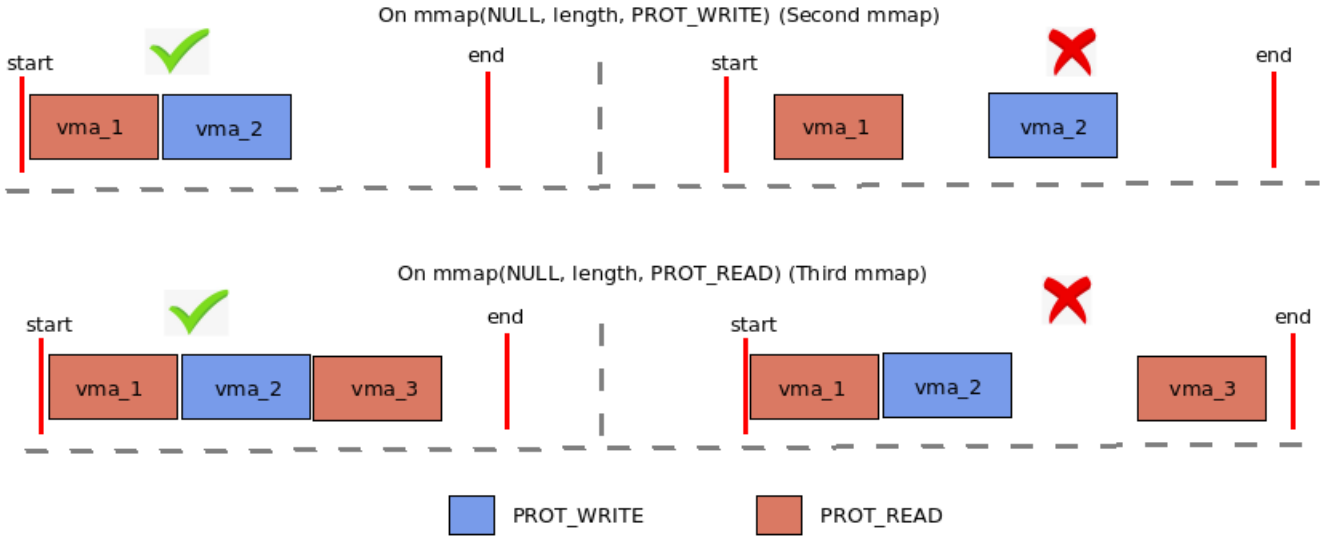

Figure 5: Subsequent mappings

- When the new mapping follows the end of an existing mapping and has same protection flags and type as the the existing one, new mapping should be merged with the existing mapping. Refer Figure 6.

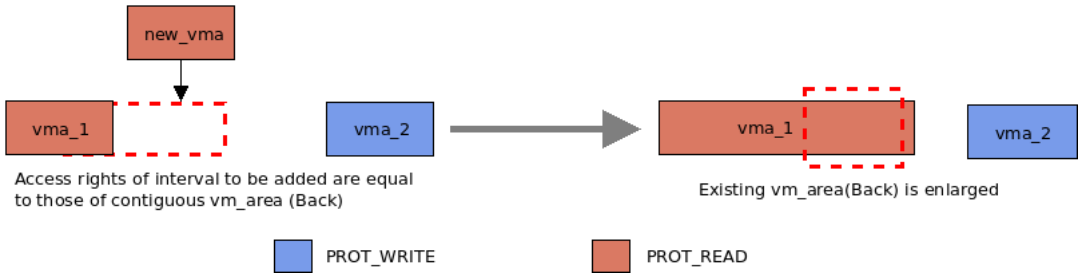

Figure 6: Merging at the end of existing mapping

- When the new mapping's end is followed by the start of an existing mapping and has same protection flags and type as the existing one, new mapping should be merged with the existing mapping. Refer Figure 7.

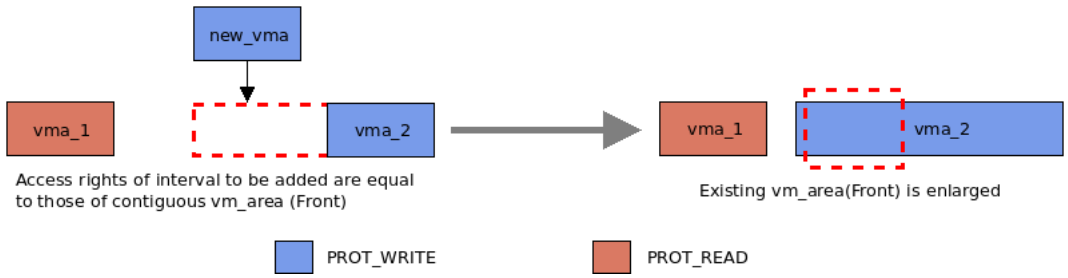

Figure 7: Merging at the start of existing mapping

- When the new mapping cannot be merged with any of the existing mappings, a new mapping should be created. Merging may not be possible due to different protections or different mapping types. Refer Figure 8 and Figure 9.

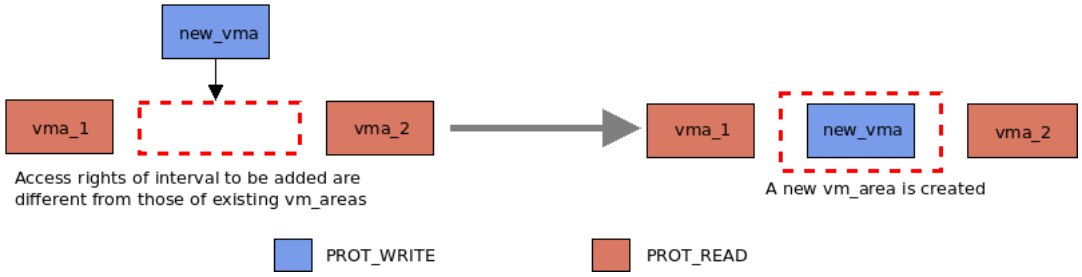

Figure 8: Merging of VMAs not possible (due to different protections)

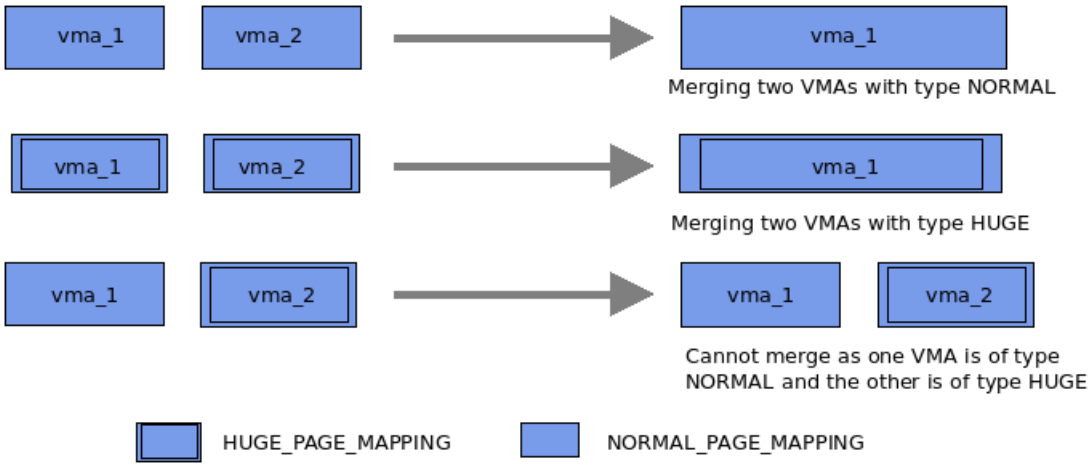

Figure 9: Merging of VMAs not possible (due to different mapping type)

- If there are two adjacent VMAs with PROT_READ|PROT_WRITE(vma1), and PROT_WRITE(vma2) do not merge them.
- While merging, always choose the first immediate available free space in the virtual address space. Refer Figure 10.

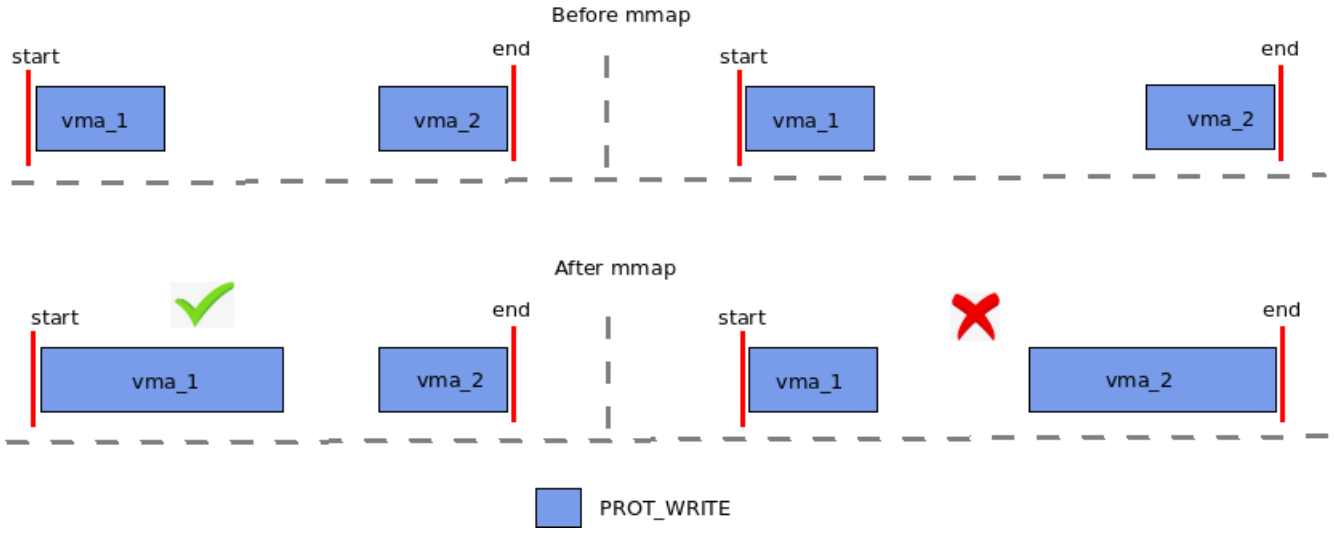

Figure 10: Always choose the first immediate free slot available

- However, give precedence to creating a new mapping at the first free slot over merging with an existing mapping (unless it is required by the hint address). Refer Figure 11.

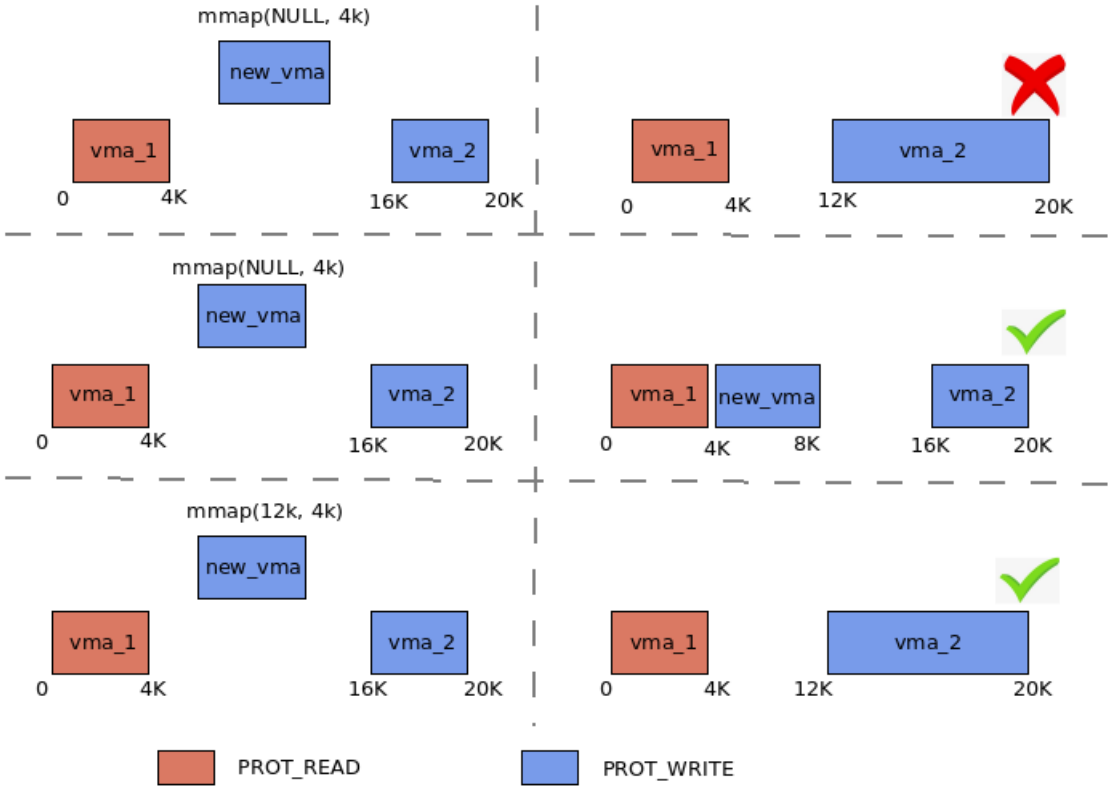

Figure 11: Give precedence to free slot over merging

- If another mapping already exists for the provided hint address and `flag = 0`, pick a new address that need not depend on the hint address. If `flag = MAP_FIXED`, and the address is already mapped, mmap() fails returning -1. You might have to create new mappings or merge with existing mapping as mentioned above. The new address which was picked is returned as the result of the mmap call.
- Space allocated must be of size multiple of 4KB.

3.1.2 Implementation

To implement `mmap()` system call, you are required to provide the implementation for the template function `vm_area_map` in the file `[src/mmap.c]` which takes the current context, address, length, protection and flags as arguments. You are supposed to maintain all the `vm_area` mappings in the singly linked list data structures. The head of singly linked list is inside the current context (`current->vm_area`). Based on the request and the hint address, you might create (add a node to the singly linked list), shrink, expand and merge the `vm_areas` in the singly linked list. To create and delete use `alloc_vm_area()` and `dealloc_vm_area()` respectively.

Your list of VMAs should ***always*** have a dummy node with:

`start_address = MMAP_AREA_START`

`size = 4KB`

`mapping_type = NORMAL_PAGE_MAPPING`

`access_flags = 0x4`

3.1.3 Page Fault Handling

We have provided a template function `vm_area_pagefault()` which takes the current context, address (faulted address) and error code. For a valid access, a physical frame is mapped and the function should return `1` (not 0, as mentioned in the code template). For an invalid access the function should return -1. The `vm_area_pagefault()` function will be called in case of page fault. You have to map physical page, set access flags and update page table entries of the faulted address. The following page faults have to be handled:

- All the allocations have to be done lazily. i.e. after mapping, the first access to a mapped address MUST result in a page fault. Subsequent accesses must NOT result in a page fault.
- A page fault arising due to a read/write access to an unmapped page.
- A page fault due to a write access to a page marked as READ_ONLY.
- Mapping/Unmapping here mean virtual-to-physical mapping, i.e. the faulting address does belong to a VMA, just not present in the physical memory (valid access). If the referenced address does not belong to any VMA (invalid access), simply return -1.

The error codes for the above faults have been explained in section 5.

3.2 int munmap(void *addr, size t length)

3.2.1 Details

The munmap() system call deletes the mappings for the specified address range. After the deletion, `vm_area` can either be shrunk or split into two `vm_area`, as shown in Figure 12. Any access to addresses within the deleted `vm_area` results in invalid memory references.

All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate `SIGSEGV` . It is not an error if the indicated range does not contain any mapped pages. An unmap will also result in freeing of the allocated frames in physical memory (if any), and appropriate changes to the page table and TLB.

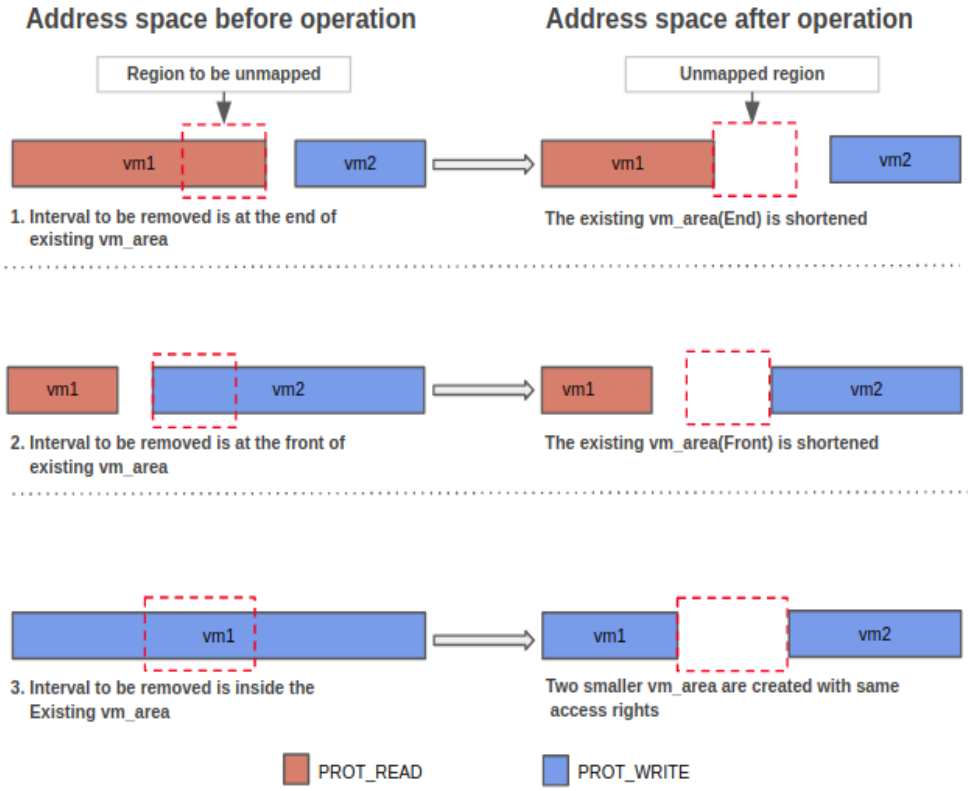

Figure 12: Unmapping of the VM area

`Argument Description`

`addr`: The `addr` argument specifies the start address of the region to be unmapped. It must be page aligned.

`length`: The `length` argument specifies the length of the mapping. It must be greater than 0 and need not be a multiple of page size. Length is in bytes.

`Return Value`

On success, munmap() returns 0. On failure, it returns -1.

Unmapping Scenarios

- Unmapping across VMAs is allowed, even if they have different protections (protections are anyway irrelevant while unmapping). However, do take care to free all the physical frames allotted to the pages being unmapped (if any). Any read/write to the unmapped region must result in a invalid memory reference. Refer Figure 13.

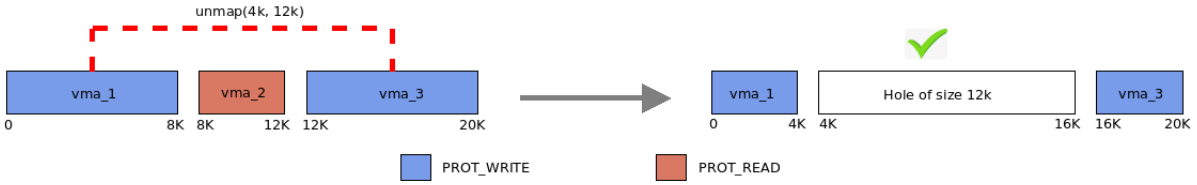

Figure 13: Unmapping across VMAs: Allowed

- Unmapping even a single byte from a page, unmaps the whole page. Further reference to that page results in a fault, refer Figure 14.

```c
int *p1 = mmap(NULL, 8192, PROT_READ | PROT_WRITE
if(p1 == MAP_FAILED)
{
        printf("Mapping Failed\n");
        return 1;
}
else
{
        p1[0] = 100;
        p1[1] = 200;
        munmap(p1, 4);
        printf("p1[1] = %d\n", p1[1]);
}
return 0;
```

```
cs330@cs330:~/Desktop$ gcc -o mmap mmap.c
cs330@cs330:~/Desktop$ ./mmap
Segmentation fault (core dumped)
cs330@cs330:~/Desktop$
```

Figure 14: Unmapping at page granularity (i)

- Keep in mind that the unmap happens only for a page, and not the whole VMA; reference to other pages in the VMA is valid, refer Figure 15.

```c
int *p1 = mmap(NULL, 8192, PROT_READ | PROT_WRITE
if(p1 == MAP_FAILED)
{
        printf("Mapping Failed\n");
        return 1;
}
else
{
        p1[0] = 100;
        p1[1024] = 200;
        munmap(p1, 4);
        printf("p1[1024] = %d\n", p1[1024]);
}
return 0;
```

```
cs330@cs330:~/Desktop$ gcc -o mmap mmap.c
cs330@cs330:~/Desktop$ ./mmap
p1[1024] = 200
cs330@cs330:~/Desktop$
```

Figure 15: Unmapping at page granularity (ii)

- To reiterate the point of unmapping at page granularity, if the address passed belongs to a VMA having mapping type as HUGE_PAGE_MAPPING, the entire huge page gets unmapped (and de-allocated from physical memory if need be). Refer Figure 16.

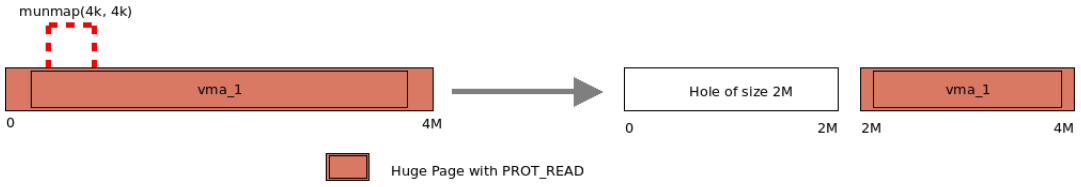

Figure 16: Unmapping at page granularity (iii)

- Unmapping across VMAs with holes in between/at start/at end is allowed. Refer Figure 17.

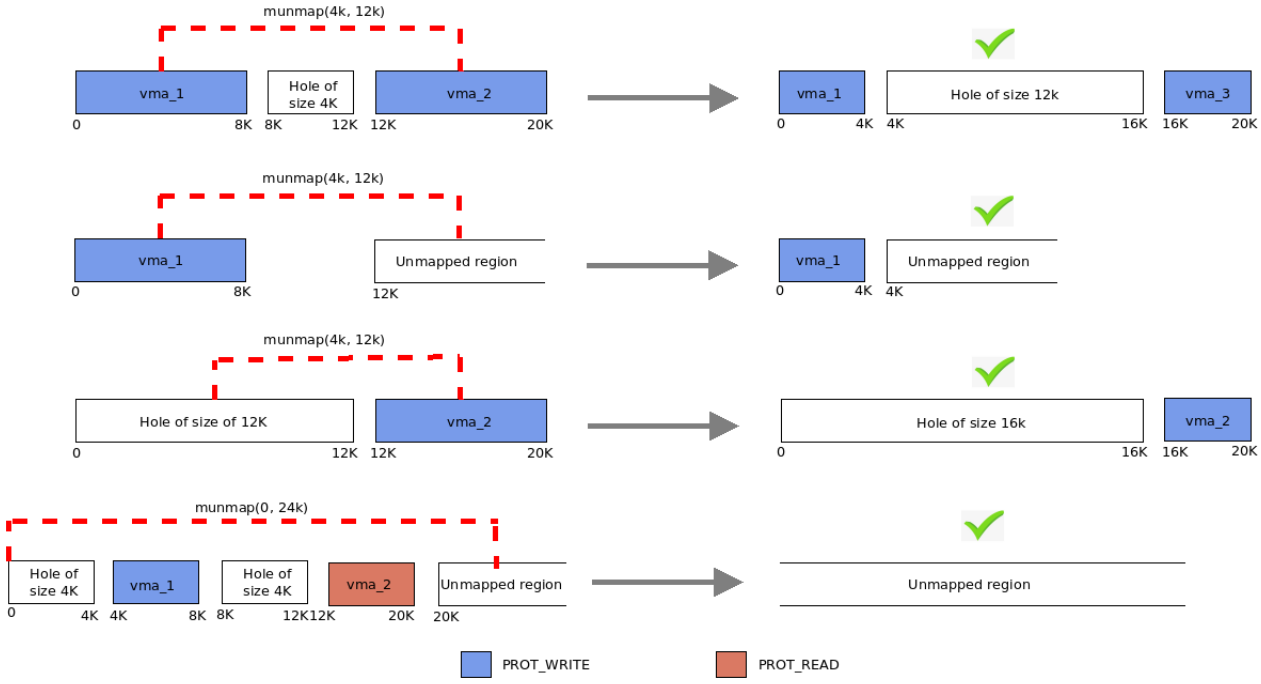

Figure 17: Unmapping with holes: Allowed

- Unmapping the same area multiple times does not give an error. Example:
  `unmap(4K, 4K); /* success */`
  `unmap(4K, 4K); /* success */`

3.2.2 Implementation

To implement `munmap` system call, you are required to provide the implementation for the template function `vm_area_unmap` in the file `[src/mmap.c]` which takes the current context, address, length as arguments. Based on the address and length, you might delete and shrink the `vm_area` mappings in the singly linked list (refer to the munmap specification). On unmapping a `vm_area` mapping, if a `vm_area` has physical pages mapped to it then you have to free the physical frames and update the page tables accordingly.

3.3 Testing and Assumptions

For Task 1, the testing procedure and assumptions are as follows:

- We will be validating the state of `vm_area`(nodes in the single linked list) such as number of `vm_area` before and after the syscall operations, the protection and access rights of the `vm_area`.
- We will be evaluating number of page faults and access rights of physical pages.
- There can be at most 128 `vm_area` at any point of time.
- The address ranges for all the above system call will be between the address ranges `MMAP_AREA_START & MMAP_AREA_END`.
- The test cases shared with you assume that the dummy node is always present. Always make sure your output matches the expected output in the test cases.
- You can assume that this dummy node will never be included in the address range specified by mmap(), munmap(), make_hugepage() or break_hugepage()

4 Task 2: HugePages - Make 'em & Break 'em [50 marks]

In this task, you will be implementing the following system calls:

- long make_hugepage(void *addr, u32 length, u32 prot, u32 force_prot)
- int break_hugepage(void *addr, u32 length)

4.1 long make_hugepage(void *addr, u32 length, u32 prot, u32 force_prot)

4.1.1 Details

The `make_huge_page()` system call merges 4KB pages starting from `addr` up to `addr+length`[1] into 2MB huge page(s) and allocates new frame(s) in the reserved area, frees the old frames and makes the necessary changes in the page tables and the TLB. Also, you must change the `mapping_type` in the structure `vm_area` to `HUGE_PAGE_MAPPING`.

Argument Description

addr: The starting address of the area that has to be converted into a huge page. `addr` cannot be NULL.

length: Specifies the length of the area to be converted into a huge page. Must be greater than 0. Length is in bytes.

prot: Same as read/write protection mentioned in mmap().

force_prot: This takes values 0 or 1. If 0, the operation fails if the read/write permissions of any of the `vm_area` in the range is different from `prot`. If 1, ignore the permissions of the `vm_areas` in the range and set the permission of the new huge page `vm_areas` thus created to `prot`.

Return Values

On success return the start address of the mapping. Else return a suitable error code.

-ENOMAPPING: Some area between `addr` to `addr+length` is not mapped (i.e. VMA does not exists)

-EDIFFPROT: When protections of at least one of the pages to be collapsed is different from `prot` and `force_prot` is 0.

-EINVAL: If `addr` or `length` are invalid.

-EVMAOCCUPIED: If a VMA of HUGE_PAGE_MAPPING already exists in the given range.

-1: Any other error

[1] Assuming addr is 2MB aligned. If not, refer to Figure 18

Handling VMAs and Page Frames

Consider `make_hugepage()` as two part process:

- Part 1 - Handling Virtual Memory:
  - First and foremost, for a given range i.e. between `addr` to `addr+length`, there must NOT be any unmapped region. If any, it results in -ENOMAPPING. Remember, `make_hugepage()` DOES NOT create new VMAs (new mappings in an unmapped region), it just manipulates the existing ones.
  - If a VMA of HUGE_PAGE_MAPPING already exists in the given range, the operation fails with the error, -EVMAOCCUPIED. For example:
    `make_hugepage(2M, 2M);` /*Success. created a huge page VMA between 2M-4M*/
    `make_hugepage(0, 8M);` /*Failure*/
  - If `force_prot` is 1, ignore the protection of `vm_areas` in the given range, and proceed to next step. The protection of newly created huge page `vm_areas` will be set to `prot` in this case. However, if `force_prot` is 0, and the VMAs in the range have different protections than `prot`, the operation fails and results in -EDIFFPROT.
  - Now, you need to merge the existing VMAs for the given range and change their mapping type to HUGE_PAGE_MAPPING.
  - A huge page (and the VMAs backing them) must be 2MB aligned and the size must be a multiple of 2MB. However, the onus of ensuring these constraints is not on the user (who will be using the `make_hugepage()` system call), but on the developer writing it (you!).
  - If the `addr` given is 2MB aligned, example:`make_hugepage(2MB, 2MB)`, well and good. You can start converting the VMA (mapping type, protection, etc) from 2MB and stop at 4MB.
  - However, if the user gives a value which is not 2MB aligned, example:
    `make_hugepage(4KB, 6MB)`, start your huge page VMA from next possible 2MB boundary. In this case, it will be at 2MB. Next, go up till `addr+len`, (4KB+6MB), making as many 2MB regions as possible. In this case, two huge pages will be created. Refer to Figure 18 and Table 1 for more clarity. (For the examples in the table/figure, we have ignored flags/protections/other conflicts that could arise).

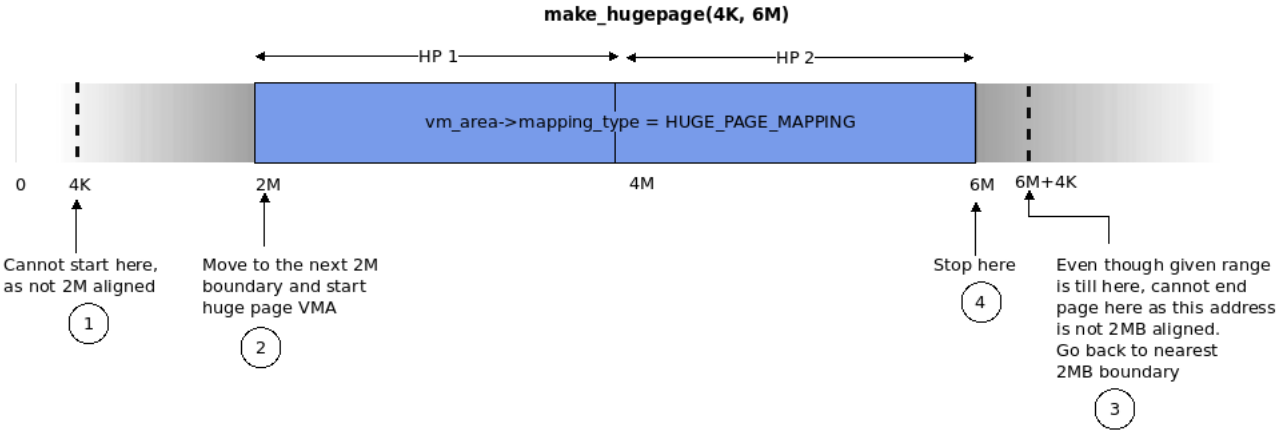

Figure 18: Making Huge Pages

User Input mhp(addr, length)	Start addr of VMA	End addr of VMA	No. of HugePages	Reason
mhp(0, 2M)	0	2M	1	addr,length 2M aligned
mhp(0, 5M)	0	4M	2	use *l1<length* where *l1* is 2M aligned
mhp(4K, 2M)	–	–	0	addr not 2M aligned. Next 2M aligned addr gives page of size 0.
mhp(4K, 4M)	2M	4M	1	next possible 2M boundary after given addr is 2M. use *l1<length* where *l1* is 2M aligned
mhp(4K, 8M)	2M	8M	3	same as above
mhp(4K, 9M)	2M	8M	3	same as above
mhp(1M, 2M)	–	–	0	addr not 2M aligned. Next 2M aligned addr gives page of size 0.
mhp(1M, 5M)	2M	6M	2	next possible 2M boundary after given addr is 2M. The length can go max. uptil (1M+5M=6M). Since 6M is 2M aligned, that's the end addr.

Table 1: make_hugepage() Cases

- Make sure that the VMA thus obtained for huge page (`mapping_type=HUGE_PAGE_MAPPING`) is a contiguous chunk of the maximum length possible. It must not have any holes of normal pages in between. Refer Figure 19 (However, this kind of hole is possible, if later a munmap()/break_hugepage() is done. Check here for example).

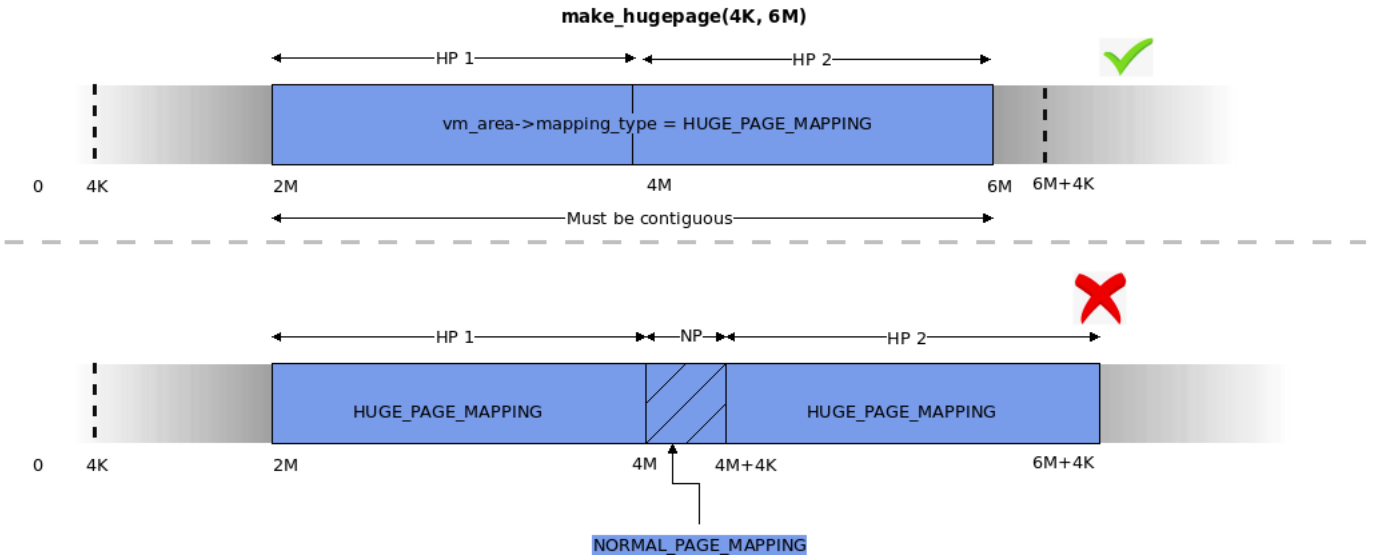

Figure 19: Correct way to make Huge Pages

- Needless to say, if there exists VMA(s) of `mapping_type=`HUGE_PAGE_MAPPING and `protection=prot` adjacent to the newly created VMA (i.e. the one created during make_hugepage()), merge them (just like you did in mmap()).
- But do keep in mind that `force_prot` must only be used while merging NORMAL_PAGE_MAPPING VMAs to convert them to a HUGE_PAGE_MAPPING VMA. It must NOT be used to merge two VMAs of type HUGE_PAGE_MAPPING. Illustrated in Figure 20.

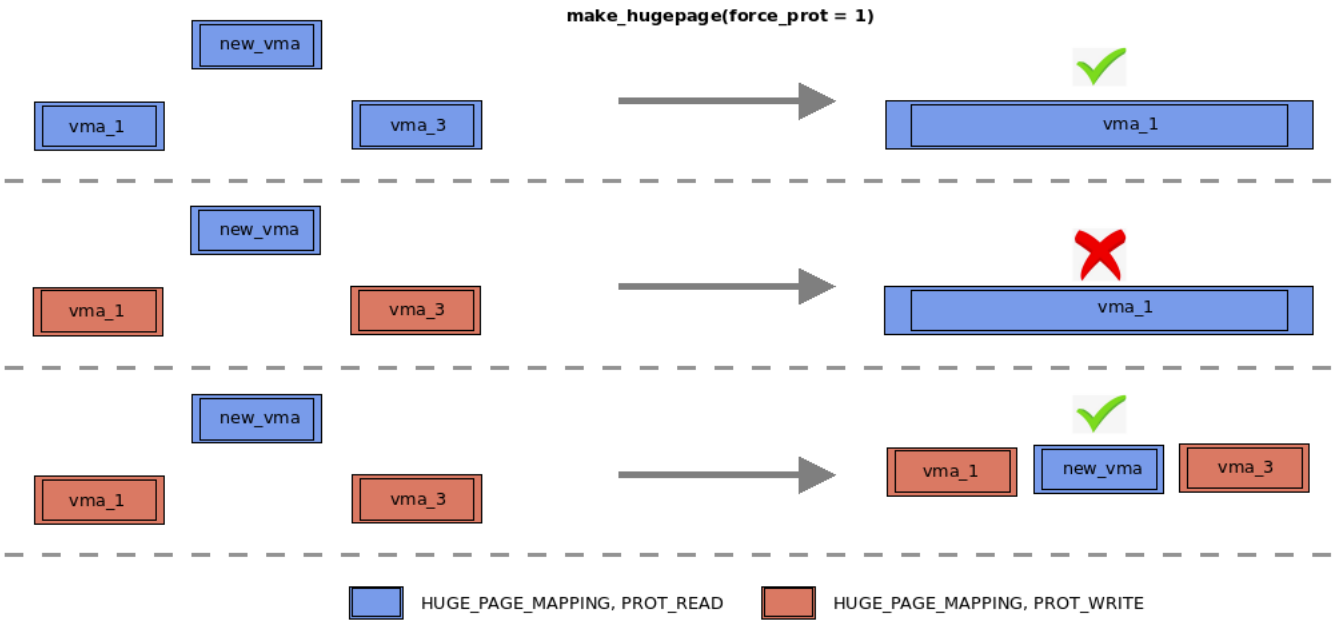

Figure 20: To merge or not to merge

- Part 2 - Handling Physical Memory:
  Now that we have fixed virtual memory, let's talk about physical memory.

- Case 1: Suppose from the given range, none of the pages (i.e the original, 4KB page) were resident in the physical memory (due to lazy allocation, or perhaps due to swapping). In this case, after manipulating the VMA, you don't have to do anything. (Recall, we are following lazy allocation). When a reference to an address in the given range will be made, a page fault will occur. You have to handle the page fault as usual, with only two changes. First, the physical frame will be allocated in the reserved area of the physical memory (and of course, will be of size 2MB). Second, the third level page table will point to the frame, instead of the fourth level. Also remember to appropriately set the bit position 7 of the PMD entry.
- Case 2: If at least one of the pages in the given range were resident in the physical memory, i.e. occupying a physical frame, then do the following. Allocate frame(s) in the reserved area, copy contents copy contents of the old physical frame(s) (of size 4KB) to the new physical frame (of size 2MB). While copying, make sure you do that at the right offset for each page, i.e. pages and their content have to to be copied according to their address and offset. Free the old frames. The read/write protection of the new frame is set to `prot`. Update the page table to reflect a huge page as explained here, and flush the TLB. Figure 21 shows the working of a `make_hugepage()` in the range 0 to 2MB.

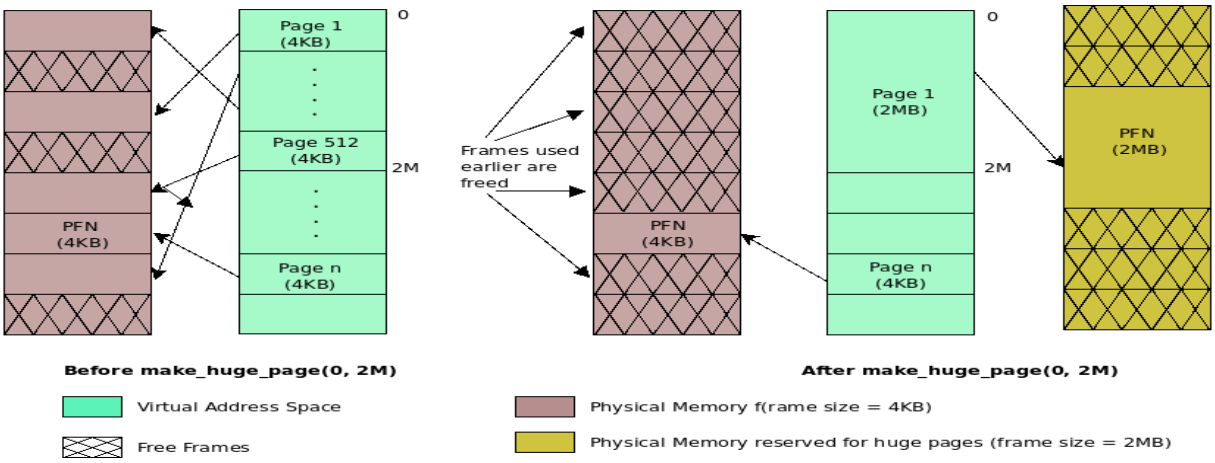

Figure 21: Physical Memory manipulation for Huge Pages

4.1.2 Implementation

To implement `make_hugepage()` system call, you are required to provide implementation for the template function `vm_area_make_hugepage` in the file `[src/mmap.c]` which takes the current context, address, length, protection and force_prot as arguments. You will have to make changes in the page tables and TLB. The APIs for this are mentioned in the section 5.

4.1.3 Page Fault Handling

The first reference to a huge page arising out of Case 1 results in a page fault. Other page faults generated for huge pages will be similar to that of normal pages. However, the page fault handler for huge pages will be different from that of the normal pages in that it will be dealing with only three levels of page tables instead of four.

4.2 int break_hugepage(void *addr, u32 length)

4.2.1 Details

This system call basically does the reverse of make_hugepage(). It disintegrates huge pages present in range, starting from `addr` upto `addr+length`, into normal, 4KB pages. If there is no huge page inside the specified range, then do nothing and returns 0 (success).

- Part A - Physical Memory:
  - For each huge page, allocate the required number of 4KB frames in the physical memory (i.e. 512 frames).
  - Copy contents of the huge page frame to the newly acquired frames (of size 4KB).
  - Make the necessary changes in page table and TLB so that instead of one huge frame, we now have 512 smaller frames corresponding to the virtual addresses covered by the huge frame being broken.
  - If the page is not in physical memory, skip this part and handle only part B. It will be allocated in the reserved area on its first reference (lazy allocation).
- Part B - VM Areas:
  - After breaking all the huge pages of a VMA having mapping type `HUGE_PAGE_MAPPING`, change the `mapping_type` of this VMA to `NORMAL_PAGE_MAPPING`.
  - After changing the mapping type of the VMA, merging of VMAs may be possible. Perform merging if possible. Some examples have been explained below. You have to handle ALL possible scenarios as explained in `mmap()`, `munmap()`, and `make_hugepage()`.

Argument Description

`addr`: The starting address of the area to be broken down into normal pages. `addr` cannot be NULL and must be 2MB aligned.

`length`: Specifies the length of the area to be broken down into normal page. Must be greater than 0 and a multiple of 2MB. Length is in bytes.

Return Values

0: On success

-EINVAL: If `addr` or `length` are not 2MB aligned, or if `addr` points to a non huge page area.

-1: Any other error

Breaking Scenarios

- Given, 1 VMA with start address of 0MB, end address of 6MB, with read permissions and `HUGE_PAGE_MAPPING` mapping. If break_huge_page() is called with start addr = 0 and length = 6MB, then break each huge page within this VMA and change mapping type of this VMA to `NORMAL_PAGE_MAPPING`

- A given range may have a combination of VMAs with 4KB mapping and VMAs with 2MB mapping. `break_hugepage()` should find VMA's with 2MB mappings within the range and break each huge page within them. After breaking, merging of VMAs is possible. Example: VMA1(0 to 2MB, Read, hugepage mapping), VMA2(2MB to 4MB, Read, 4KB page mapping). After breaking 0 to 2MB, we will have single VMA (0 to 4MB, Read, 4KB page mapping).
- A given range may have a hole. Example:
  1. `make_hugepage(0, 6M);` /* success; start addr:0, end addr: 6M, #Pages: 3*/
  2. `munmap(2M, 4096);` /* success */
  OR
  2. `break_hugepage(2M, 2M);` /* success */
  3. `break_hugepage(0, 6M);` /* this should break two huge pages: (0-2M) and (4M-6M) */
- Some portion of a VMA may be inside the range and some portion may be outside the range. In this case, if the VMA is of mapping type `HUGE_PAGE_MAPPING`, split that VMA into 2 VMAs. One outside the range and one inside the range. VMA outside the range will have mapping type `HUGE_PAGE_MAPPING` and no breaking operation will happen on it. Whereas, the one inside will undergo breaking and its mapping type will be changed to `NORMAL_PAGE_MAPPING`.
  Eg: Let, there be VM (start addr = 0 , end addr = 4MB, read, huge pages) and
  VM (start addr = 4MB , end addr = 8MB, write, huge pages)
  If `break_huge_page` is called with start address of 2MB and length 4MB, then we will have following mappings after breaking.
  VMA (start addr = 0 , end addr = 2MB, read, huge pages)
  VMA (start addr = 2MB , end addr = 4MB, read, normal pages)
  VMA (start addr = 4MB , end addr = 6MB, write, normal pages)
  VMA (start addr = 6MB , end addr = 8MB, write, huge pages)

4.2.2 Implementation

To implement the `break_hugepage()` system call, you are required to provide implementation for the template function `vm_area_break_hugepage()` in the file `[src/mmap.c]`.

4.3 Testing and Assumptions

For Task 2, the testing procedure and assumptions are as follows:

- We will be checking the number of 4KB and 2MB physical frames before and after the system calls.
- We will be validating the page faults and TLB misses.
- The status of the structure, `vm_area`, will also be checked before and after the system calls.
- Assume that physical memory will never be insufficient.

5 Utilities

In order to make things easier, we have given some template functions, structures and a few utility functions that facilitate object creation and deletion. Let's have look at those functions.

The process control block (PCB) is implemented using a structure named `exec_context` defined in `src/include/context.h`. One of the important members of `exec_context` for this assignment is the structure vm_area.

`struct vm_area`

`vm_start` - The starting address (virtual address) of the `vm_area` mappings.
`vm_end` - The ending address (virtual address) of the `vm_area` mappings.
`access_flags` - The protection or access flags of the current `vm_area` mappings.
`mapping_type` - Indicates if the `vm_area` is backing a normal page(4KB) or huge page(2MB).
`vm_next` - The pointer to the next `vm_area` mappings.

`MMAP_AREA_START & MMAP_AREA_END`

These are constants defined in the file `[src/include/mmap.h]` which is used to specify the overall start and end limit of the mmap space. All the mappings (`vm_area`) which are created using the mmap syscalls should reside within this limit. if the hint address is not within limit, then mmap syscalls should return `EINVAL` or (-1).

`struct vm_area* alloc_vm_area()`

This function is used to create a new `vm_area` and returns a pointer to the created `vm_area`. This function does not initialise members of the `vm_area` returned. You should only use this function to create `vm_areas` for this assignment.

`struct vm_area *create_vm_area(u64 start_addr, u64 end_addr, u32 flags, u32 mapping_type)`

This function is defined in `mmap.h`. This function is used to create a new `vm_area` with its fields initialized with the values passed as arguments to this function.

`void dealloc_vm_area(struct vm_area *vm)`

This function is used to delete or deallocate the `vm_area` which is passed as an argument. You should only use this function to delete or deallocate `vm_areas` for this assignment.

`void *os_hugepage_alloc()`

This function is used to allocate a hugepage. It returns OS virtual address to the beginning of the huge page. This address can be used to copy contents to the huge page using functions like `memcpy`.

`void os_hugepage_free(void *page_addr)`

This function is used to free an allocated huge page. The argument passed is the address of an already allocated huge page.

`u64 get_hugepage_pfn(void *page_addr)`

This function returns the frame number of the huge page, given its OS virtual address.

`int memcpy(char *dest, char *src, u32 size)`

This function copies size bytes from src address to dest address.

`void *osmap(u64 pfn)`

Given a page frame number, it returns a virtual address corresponding the passed page frame number.
Refer install_page_table() inside entry.c for example usage.

`u32 os_pfn_alloc(u32 region)`

This function allocates a new frame in the specified region.
To allocate a frame that is to be used to store page table content, use region = OS_PT_REG.
To allocate a frame that is to be used to store normal data content, use region = USER_REG.
Refer install_page_table() inside entry.c for example usage.

`u32 os_pfn_free(u32 region, u64 pfn)`

This function frees a page frame given the page frame number and the region to which it belongs to.

`int pmap(int details)`

You can use the pmap method to check the `vm_area` details. If `details` is zero, pmap will print the count of `vm_areas`, number of huge pages and pagefaults for the address ranges between `MMAP_AREA_START` and `MMAP_AREA_END`. If `details` is 1. Then pmap will dump the entire `vm_area` details.

`Page-Fault Error Code`

The error codes generated in case of a page fault are shown in Figure 22, which is taken from course slides. Some of them are enlisted below:
`0x4` - User-mode read access to an unmapped page
`0x6` - User-mode write access to an unmapped page
`0x7` - User mode write access to read-only page
For this assignment, you need to only modify P, W, U bits. Also, check note.

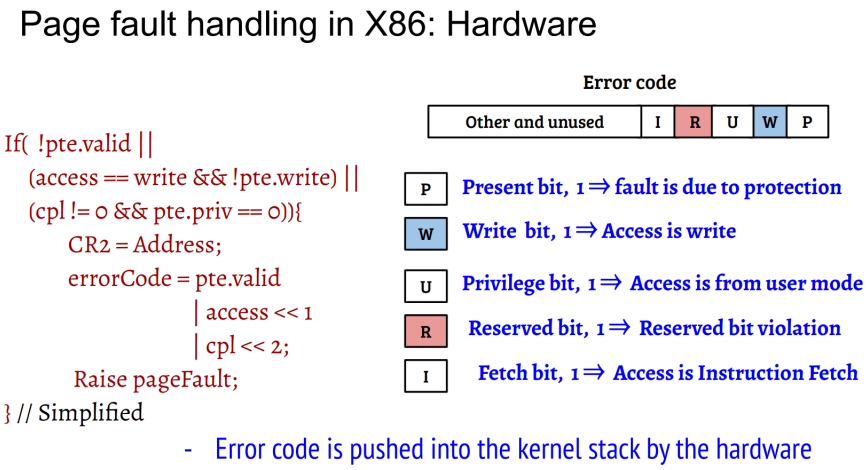

Figure 22: Page-Fault Error Codes

6 Submission guidelines

- The assignment is to be done individually. You have to submit only one file (mmap.c). Put this file in a directory named with your roll number. Create a zip archive of the directory and upload in hello.iitk.
- *Don't modify any other files.* We will not consider any file other than (mmap.c) for evaluation.
- *You should remove all printf/printk debug statements from submission files.* We will taking diff of your output and expected output for evaluation.

In case of any issues, reach out to us at the earliest. All the best!