Assignment 3

CS330: Operating Systems

Total Marks: 100

Submission Deadline: 11.55PM, 12th November 2023

Introduction

The objective of this assignment is to expose you to OS memory management concepts covered in the lectures. You will be implementing memory-related system calls in **gemOS**.

The gemOS source can be found in the *gemOS/src* directory. Structure of the *gemOS/src* directory is as following:

- `gemOS/src/user/` contains the user space components of the gemOS.
  - `init.c` is the userspace program that will run on gemOS. This program will interact with gemOS using system calls.
  - `lib.c` is library that contains the implementation of common functions such as `printf()`. It is equivalent to the C library (`libc`) in Linux. (**Not to be modified**)
  - `ulib.h` is a header file containing declaration of various functions and system calls. It is equivalent to the user space header files in Linux. (**Not to be modified**)
- *gemOS/src/include/* contains the header files related to the kernel[1] space implementation of the gemOS. (**Not to be modified**)
- *gemOS/src/*.c, *.o* files contain the implementation of the kernel space of the gemOS. (**Modify only the specified files**)
- *gemOS/src/Makefile* is used to build the gemOS kernel binary `gemOS.kernel`. (**Not to be modified**).

Please refer to the piazza post `https://piazza.com/class/lknrcb2tsv579r/post/252`. It contains instructions regarding the setup required to complete this assignment.

Background

This section revisits some known along with additional concepts required to finish 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. The `mmap` system call is used to allocate VM Areas.

[1] OS and kernel are used interchangeably in this document

Paging, TLBs, and Efficiency

You are familiar with the concept of multi-level 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. Figure 1 depicts the addressing scheme for 4KB pages. Figure 2 depicts the contents of a Page Table Entry in 4 level paging scheme.

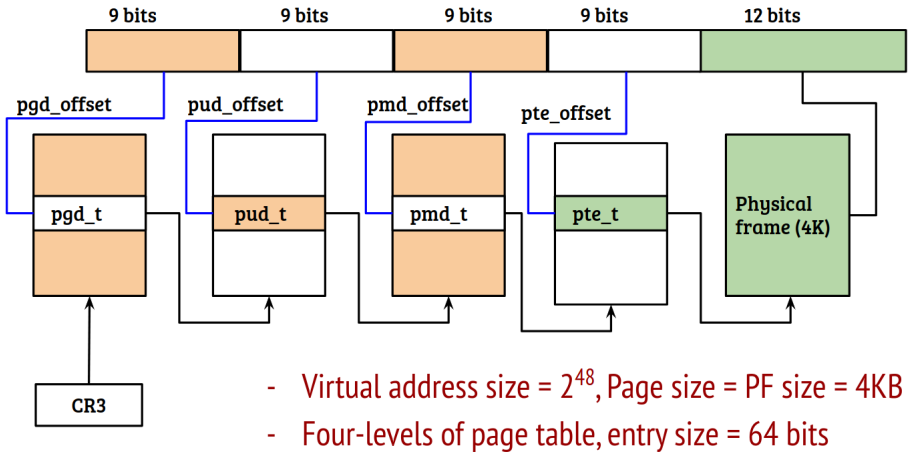

Figure 1: Address Translation for 4KB pages

Bit Position	Contents
0 (P)	Present Bit. Value = 1 implies this entry points to a 4KB page else page is not present
3 (R/W)	Read/Write Bit. Value = 1 implies read and write are allowed for the 4KB page referenced by this entry, else only read is allowed
4 (U/S)	User/Supervisor Bit. Value = 0 implies user mode access is not allowed for the 4KB page referenced by this entry
12 onwards	Physical address of the 4KB page referenced by this entry

Figure 2: Structure of PTE at each level of the page table (only relevant fields are shown)

Copy-on-Write

As you know, the `fork` system call creates a new process, called as *child process* where all resources used by the *parent process* are duplicated. Considering the size of the memory state of a process, this mechanism is highly inefficient as it requires replication of memory content along with the meta-data and translation information. Specifically, if the *child process* immediately calls `exec`, the duplication of the memory state becomes wasteful.

To address this shortcomings, many OSes use a technique known as `Copy-on-Write`, or COW. COW delays, or altogether avoids duplication of *memory content* as long as possible by maintaining the read-only mapping to pages from both the parent and child processes. When either of the processes (i.e., parent or child) performs a write operation, a *COW fault* is generated and a copy of that page is created in the address space of the process that performed the write operation. Subsequent accesses to that

page from the same process do not raise any fault. The raising of CoW fault (which is essentially a page fault) and fixing of the fault through copy operations is performed by the OS in a user transparent manner.

1 Memory mapping support in gemOS [30 marks]

In this task, you will be implementing the following system calls. Note that, the semantics of these system calls, while are similar to Linux/Unix POSIX calls, they are simplified.

- void *mmap(void *addr, int length, int prot, int flag)
- int munmap(void *addr, int length)
- int mprotect(void *addr, int length, int prot);

1.1 void *mmap(void *addr, int length, int prot, int flag)

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 below. You can assume that, `addr` will always be page aligned.
length: The `length` argument specifies the length of the mapping. It is assumed to be a positive integer and need not be a multiple of page size. Length is in bytes and assume that, the length will not be more than two megabytes.
prot: The `prot` argument describes the desired memory protection of the mapping. It can be one 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 pages with read only permission results in segmentation fault.
- `PROT_READ | PROT_WRITE` - The protection of the mapping `vm_area` is set to both read and write. Physical pages corresponding this `vm_area`, will have both read and write access.

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

- `MAP_FIXED` - Don't interpret `addr` as a hint, instead place the mapping exactly at the address that is passed as an argument to the mmap() function. 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, return an error (see below).

System call handler: To implement the `mmap()` system call, you are required to provide implementation for the template function `long vm_area_map(struct exec_context *current, u64 addr, int length, int prot, int flags)` (present in *gemOS/src/v2p.c*). Note that this system call handler is passed one extra argument i.e., the `exec_context` of the current process.
Description: `mmap()` creates a mapping in the virtual address space of the calling process. `vm_area` in `struct exec_context` is a sorted linked list of vm areas (referred as VMAs) corresponding to a process. Based on the arguments passed to the `mmap()`, you have to scan the linked list of VMAs, to find the appropriate position in the linked list where a new `vm_area` node can be added or an existing `vm_area` node can be expanded.

Note that your list of VMAs should ***always*** have a dummy node (first node in the list of VMAs) with following configuration:

- `start_address = MMAP_AREA_START`
- `size = 4KB`
- `access_flags = 0x0`

Scenarios for manipulating the VMAs when an `mmap()` system call is made are as following:

- Always use lowest available address (in the range `MMAP_AREA_START` (start) to `MMAP_AREA_END (end)`) that can satisfy the `mmap()` request unless hint address is provided i.e., you should not create a new mapping in intermediate addresses unless hint address is specified. Refer Figure 3 and Figure 4.

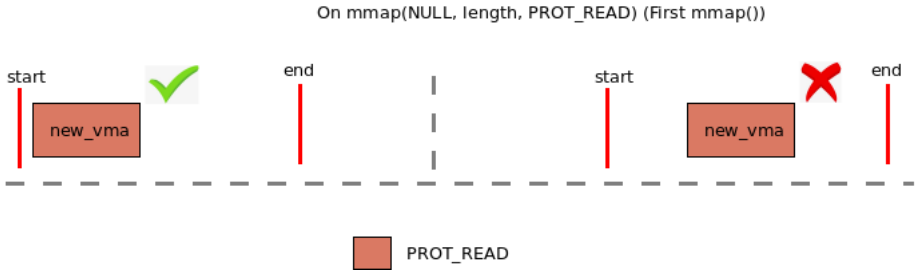

Figure 3: Creating new mapping

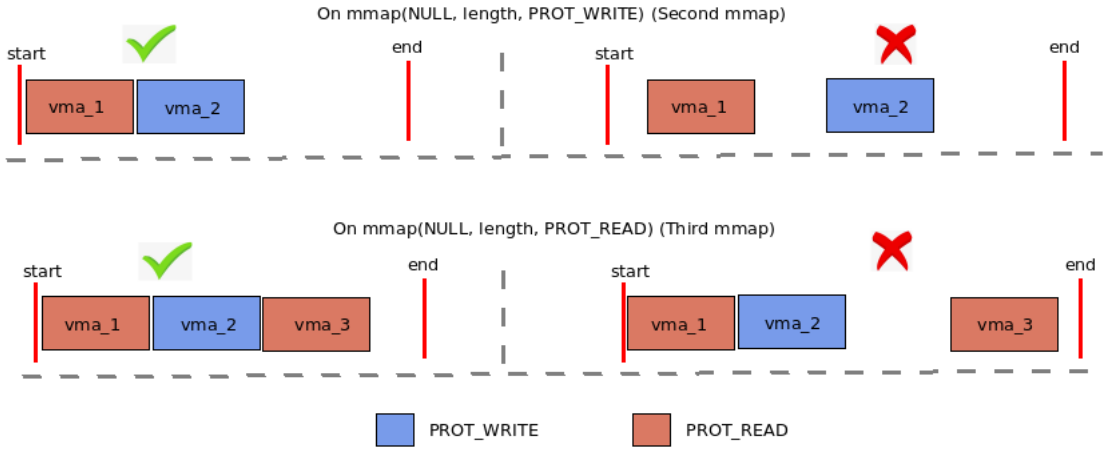

Figure 4: Subsequent mappings (assume `PROT_WRITE` means `PROT_READ|PROT_WRITE`)

- When the new mapping follows the end of an existing mapping and has same protection flags as the the existing one, new mapping should be merged with the existing mapping as shown in Figure 5.

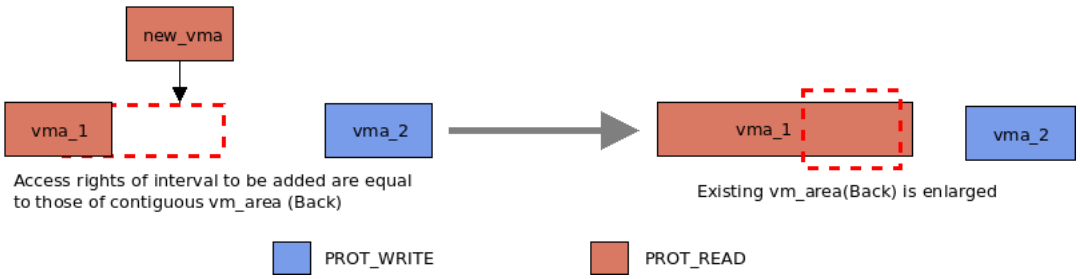

Figure 5: Merging at the end of existing mapping (assume `PROT_WRITE` means `PROT_READ|PROT_WRITE`)

- When the end of the new mapping is followed by the start of an existing mapping and has same protection as the existing one, new mapping should be merged with the existing mapping as shown in Figure 6.

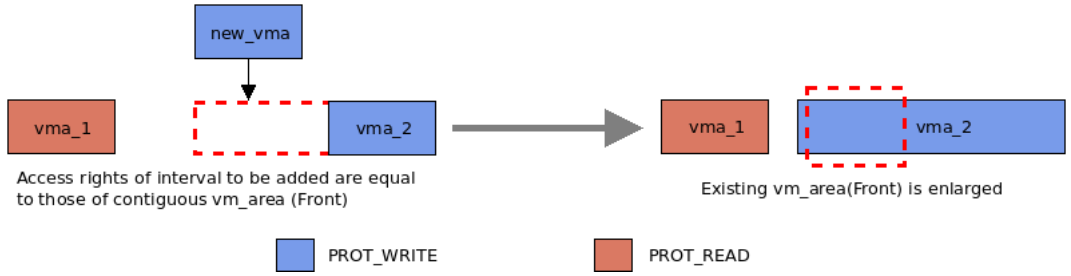

Figure 6: Merging at the start of existing mapping (assume `PROT_WRITE` means `PROT_READ|PROT_WRITE`)

- 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 absence of adjacent VMAs. Refer Figure 7.

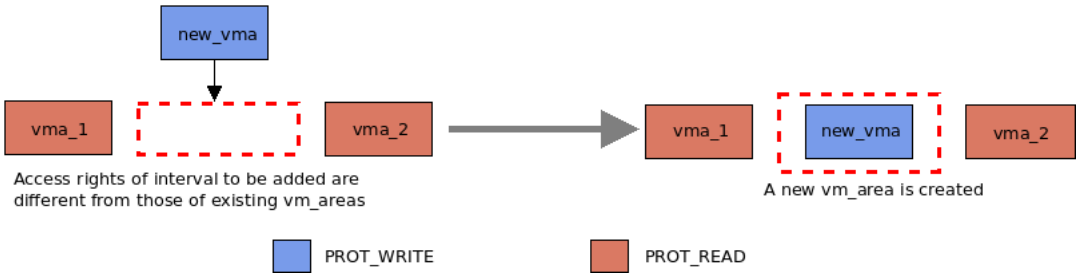

Figure 7: Merging of VMAs not possible (due to different protections) (assume `PROT_WRITE` means `PROT_READ|PROT_WRITE`)

- Note that two VMA's can be merged if and only if they have the same protections. For example, two adjacent VMAs should not be merged if one vma has protection `PROT_READ|PROT_WRITE` and adjacent vma has protection `PROT_READ`. When hint address is provided and no vma exists in the requested range, merging should be done if possible as shown in Figure 8.

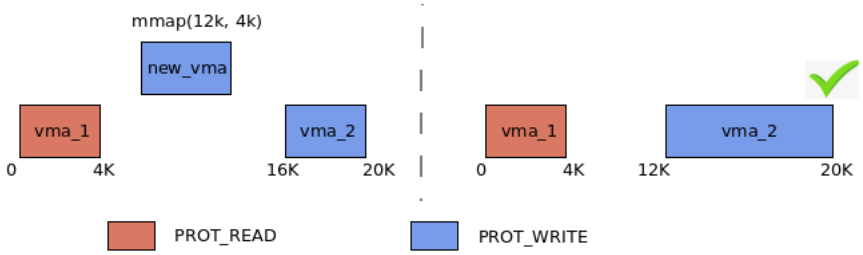

Figure 8: Address of new vma being decided by the hint address (new_vma got merged with vma_2). (assume `PROT_WRITE` means `PROT_READ|PROT_WRITE`)

- If another mapping already exists for the provided hint address and `flag = 0`, pick a new address (lowest address that can satisfy the request i.e., start scanning for free addresses from `MMAP_AREA_START`) that need not depend on the hint address.
- If hint address is provided and `flag = MAP_FIXED` such that the requested memory region is entirely or partially mapped, `mmap()` should fail and return -1.
- Note that the size of the VMAs must be a multiple of 4KB. Example: On `mmap(hint addr = 12K, length = 1K)`, you have to create VMA of size 4KB (from address 12K to 16K assuming this address region is unmapped).

- Note that the end address of your VMA should be equal to `start addr of vma + length of vma`. For example: A vma with start address of 4096 and length of 4096 bytes will have end address as 4096 + 4096 = 8192. Range of bytes corresponding this VMA will be [`addr`, `addr+len-1`] i.e. 4096 bytes to 8191 bytes.

Return Value: On success, `mmap()` should return a pointer to the start of the mapped area. If newly created VMA gets merged with an existing VMA, in that case, return the start address of the new VMA before merging. In case of error, return `-1`.

1.2 int munmap(void *addr, int length)

addr: The `addr` argument specifies the start address of the memory mapped region to be unmapped. You can assume that the `addr` passed will be page aligned.
length: The `length` argument specifies the length of the mapping (starting from `addr`) to be unmapped. It will be greater than 0 and need not be a multiple of page size. Length is in bytes.
System call handler: To implement `munmap()` system call, you are required to provide implementation for the template function `long vm_area_unmap(struct exec_context *current, u64 addr, int length)` (present in *gemOS/src/v2p.c*). Note that this system call handler is passed one extra argument (the `exec_context` of the current process).
Description: The munmap() system call deletes the mappings for the specified address range. Based on the arguments passed to the `munmap()`, you have to iterate through the list of VMAs of the current process and modify the mappings. After the modification of the mappings, `vm_areas` can be freed, shrunk or split into two `vm_areas`, as shown in Figure 9.

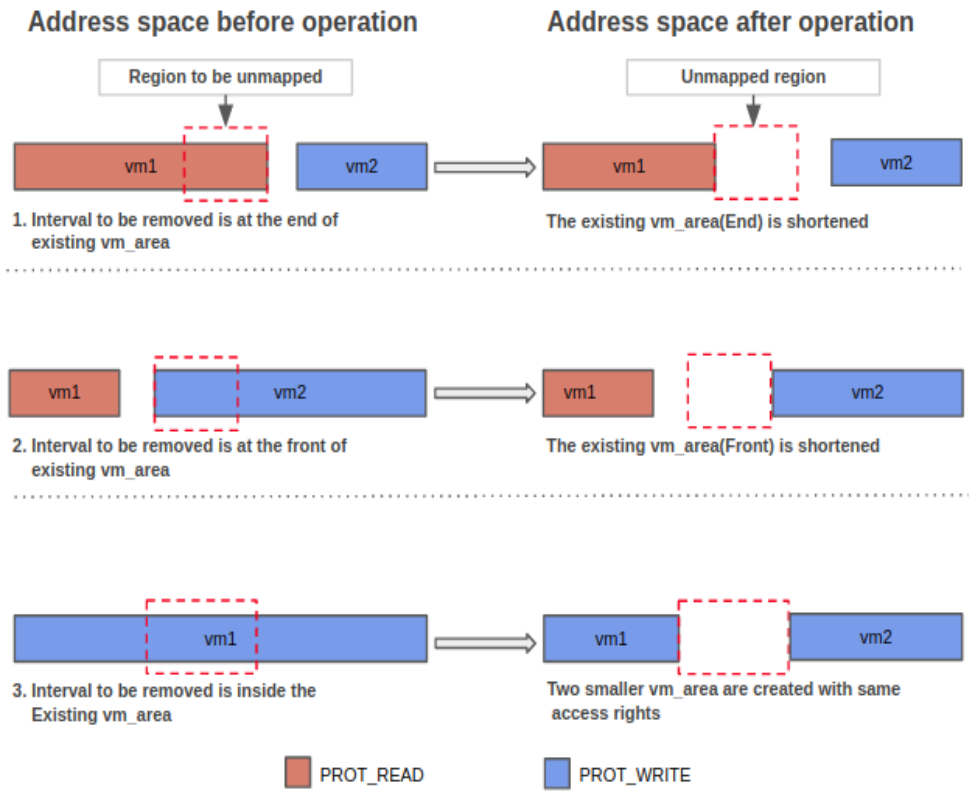

Figure 9: Unmapping of the VM area (assume `PROT_WRITE` means `PROT_READ|PROT_WRITE`)

Scenarios for manipulating VMAs when an `munmap()` system call is made are as following:

- `addr` passed to `munmap()` need not point to the start of a VMA.
- Unmapping even a single byte from a page, unmaps the whole page. Refer Figure 10.

```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 10: Unmapping at page granularity (case 1)

- `munmap()` can occur on a portion of a VMA. Refer Figure 11

```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 11: Unmapping at page granularity (case 2)

- Unmapping across VMAs is allowed, even if they have different protections (protections are irrelevant while unmapping). Refer Figure 12

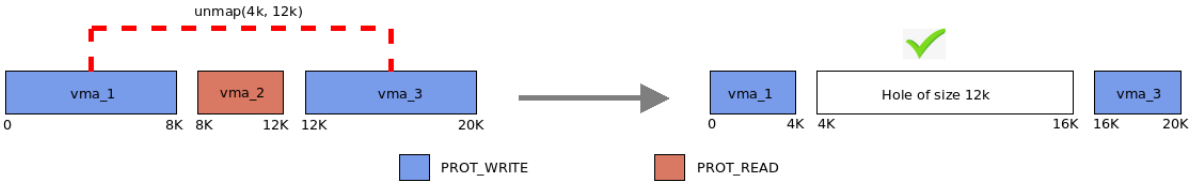

Figure 12: Unmapping across VMAs (assume `PROT_WRITE` means `PROT_READ|PROT_WRITE`)

- Unmapping across VMAs with holes in between is allowed as shown in Figure 13.

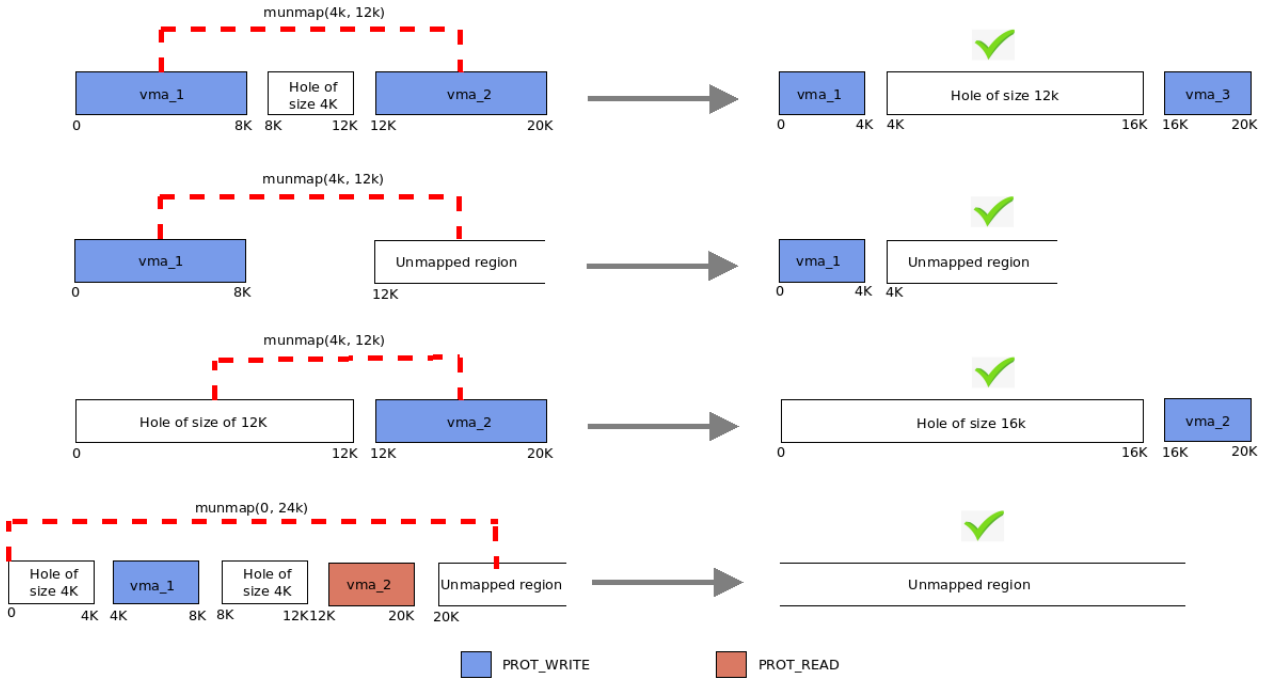

Figure 13: Unmapping with holes (assume `PROT_WRITE` means `PROT_READ|PROT_WRITE`)

- Note that it is not an error if the region to be unmapped doesn't contain any VMAs. An implication of this is that the unmapping of the same area multiple times should not raise an error. Example:
  `munmap(addr = 4K, length = 4K); /* success */`
  `munmap(addr = 4K, length = 4K); /* success */`

Return Value On success, return 0. On failure, return -1.

1.3 int mprotect(void *addr, int length, int prot)

addr: The `addr` argument specifies the start address of the memory mapped region whose protection needs to be changed. You can assume that it will be page aligned.
length: The `length` argument specifies the length of the mapping (starting from `addr`) whose protection flags needs to be changed. It will 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 one 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 pages with read only permission results in `SIGSEGV`.
- `PROT_READ | PROT_WRITE` - The protection of the mapping `vm_area` is set to both read and write. Physical pages corresponding this `vm_area`, will have both read and write access.

System call handler: To implement `mprotect()` system call, you are required to provide implementation for the template function `long vm_area_mprotect(struct exec_context *current, u64 addr, int length, int prot)` (present in *gemOS/src/v2p.c*). Note that this system call handler is passed one extra argument (the `exec_context` of the current process).
Description: The `mprotect()` system call changes the access protections for the calling process's memory pages containing any part of the address range in the interval [`addr`, `addr+len-1`]. You can assume that `mprotect()` will be used to change the access protection of VMAs only. Based on the arguments passed to the `mprotect()`, you have to iterate through the list of VMAs of the current process and modify the access permissions. `mprotect` might create (an existing VMA might be partitioned into multiple VMA's), expand (after change in protections, a VMA might get merged with adjacent VMAs)

or shrink (an existing VMA might be partitioned into multiple VMAs) the vm area mapping. Refer Figure 14. Changing the protection of even a single byte of a page changes the protection of the entire page. It is not an error if the region whose protections need to be changed doesn't contain any VMAs.

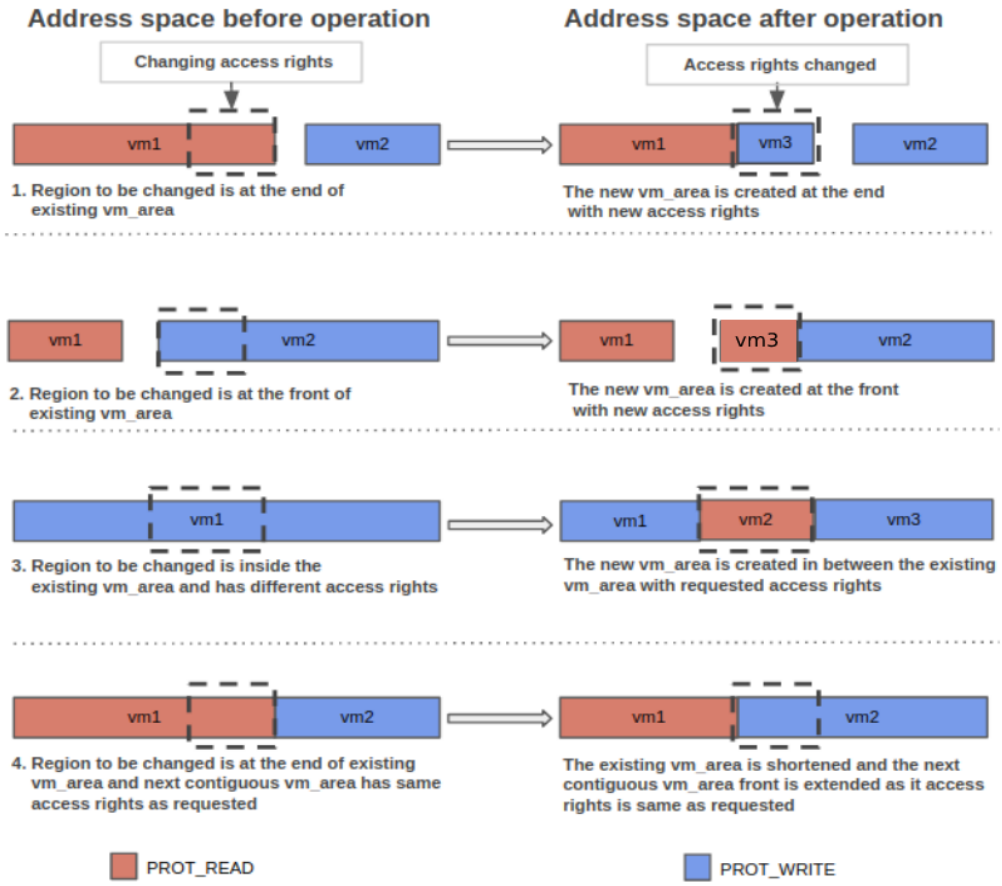

Figure 14: Impact of mprotect on VMA's (assume `PROT_WRITE` means `PROT_READ|PROT_WRITE`)

Return Value On success, return 0. On failure, return -1.

Notes

- In this part, you *are not required* to perform allocation of physical pages corresponding the VMAs, page fault handling/page table manipulation etc. In this part, you are expected to perform only address space management through VMA manipulation.
- The address range for all the above system calls will always be between `MMAP_AREA_START` - `MMAP_AREA_END`.
- You can assume that the dummy node (inserted at the head of the VMA's list) will never be included in the address range specified by `mmap()`, `munmap()`, `mprotect()`.
- You can assume that `fork()`/`vfork()`/`cfork()`/`clone()` will not be called in the test-cases for this part of the assignment.
- You are free to create your own helper functions in *gemOS/src/v2p.c*
- You have to check the validity of arguments (such as `prot`, `flag`, `length`) passed to `mmap()`, `munmap()`, `mprotect()`.
- You can assume that, maximum number of VMAs at any point of time will be 128 including the dummy vm area.

Testing

The user space program code is available in *gemOS/src/user/init.c*. You need to write your test cases in `init.c` to validate your implementation. The sample test cases in *gemOS/src/user/part1/* can be copied into `init.c` to make use of them.

2 Page Table Manipulations [30 marks]

This part of the assignment, when implemented correctly, will enable lazy allocation support in gemOS. As discussed in class, lazy allocation delays the physical memory allocation corresponding to a virtual address till the virtual address gets accessed (i.e., read or write operation is performed on that address) for the first time. In this task, you will be modifying your implementation of the system calls in Part-1, to enable memory access to the allocated VMAs using page table manipulations. Specifically, you will provide implementation/modifications for the following:

- long vm_area_pagefault(struct exec_context *current, u64 addr, int error_code)
- int munmap(void *addr, int length)
- int mprotect(void *addr, int length, int prot);

2.1 long vm_area_pagefault(struct exec_context *current, u64 addr, int error_code)

current: `exec_context` of the process whose execution resulted in the page fault
address: Accessed virtual address that resulted in the page fault
error code: Provides the information about the fault such as whether the fault occurred in user-space execution or kernel mode execution, type of access i.e, read or write etc. The error codes for various faults are explained in Section 4.
Description: We have provided a template function `vm_area_pagefault()` (present in *gemOS/src/v2p.c*) which gets called when a page fault occurs for an address in `[MMAP_AREA_START, MMAP_AREA_END]`. When this page fault handler is called, you have to iterate through the list of VMAs of the process (whose `exec_context` is passed as an argument to this function) to the validity of the address. The validity checks if the faulting address belongs to some VMA and if the access matches with the protection flags of the VMA. The following scenarios should be handled by the page fault handler.

- Page fault can arise due to read (error code = 0x4) or write (error code = 0x6) access to an address such that a VMA exists corresponding to the accessed address and no physical page is mapped. Note that write access on an address such that VMA corresponding the address has only read permission should be flagged as an invalid access. All other combinations of accesses and VMA protection flags are considered as valid accesses. For example, read access on an address such that VMA corresponding the address has only write permission is a valid access. For a valid access, you need to do the following,
  - Allocate a new physical page frame (PFN), set access flags and update the page table entry(s) of the faulting address. Note that `pgd` member of a process's `exec_context` can be used to find the virtual address of the PGD level of the page table using `osmap(ctx->pgd)`.
  - You have to structure the PTE entry(s) (Figure 2) at different levels depending on the access flags of the virtual address that is being mapped.
- A page fault can arise due to a read/write access such that no VMA exists corresponding the accessed address. Such an access should be flagged as an invalid access.
- A page fault arising due to a write access to a page marked as `READ_ONLY` (error code = 0x7). In this case, if VMA corresponding the page fault address does not have write permission, flag this access as invalid. If VMA corresponding the page fault address has write permission, it implies that this access is a copy-on-write access (can occur only if you implement the next part). To handle the CoW fault, call the `handle_cow_fault` passing the context, fault address and the access flag of the VMA. You will be implementing this function in the next part of the assignment i.e., if are not required to implement `handle_cow_fault` in this part.

Return Value After fixing fault for a *valid access*, return 1. For an invalid access return -1.

2.2 int munmap(void *addr, int length)

Modification: On unmapping a VMA that has some physical memory allocated to it, you should free the PFNs and update the virtual-to-physical translation accordingly. Subsequent accesses to the addresses within the unmapped memory range should result in invalid memory error (`SIGSEGV`). Note that it is not an error if there are no physical pages allocated corresponding the memory range to be unmapped.

2.3 int mprotect(void *addr, int length, int prot)

Modification: When protections of a virtual address region is changed using `mprotect`, you should update the virtual to physical translation to enforce the change in access permissions. Note that it is not an error if there are no physical pages allocated corresponding the memory range whose protections got updated.

Notes

- The `pgd` of `exec_context` contains the PFN of the PGD level in the page table for the process.
- You can assume that `fork()`/`vfork()`/`cfork()`/`clone()` will not be called in the test-cases for this part of the assignment.
- You are free to create your own helper functions in *gemOS/src/v2p.c*

Testing

The user space program code is available in *gemOS/src/user/init.c*. You need to write your test cases in `init.c` to validate your implementation. The sample test cases in *gemOS/src/user/part2/* can be copied into `init.c` to make use of them.

3 Copy-on-Write fork [40 marks]

In this task, you will be implementing the following system call and function:

- pid_t cfork()
- static long handle_cow_fault(struct exec_context *current, u64 vaddr, int access_flags)

3.1 pid_t cfork()

System call handler: To implement `cfork()` system call, you are required to provide implementation for the template function `long do_cfork()` (present in *gemOS/src/v2p.c*).
Description: `cfork()`is a variant of the `fork()` system call which implements a copy-on-write policy for the address space of a process. The features of `cfork()` are as follows:

- Like `fork()`, when a process (called parent process) calls `cfork()`, a new process, called child process, is created.
- The implementation of `cfork` does not copy the memory content of the parent. However, the address space state should be copied.
- The virtual to physical mapping should be changed such that when either of the processes performs a write on any of the pages, a copy of that page is created (in the writer's address space) before proceeding with the write. Refer Figures 15, 16, 17 for a working example.
- When CoW sharing breaks (i.e., a page fault occurs), you have to duplicate frames and update the virtual to physical translation.

```
1. int main()
2.  {
3.      int pid;
4.      char * mm1 = mmap(NULL, 4096, PROT_READ|PROT_WRITE, 0);
5.      if(mm1 < 0)
6.      {
7.         printf("Map failed \n");
8.         return 1;
9.      }
10.      mm1[0] = 'A';
11.      pid = cfork();
12.      if(pid){
13.        mm1[0] = 'B';
14.        printf("mm1[0] inside parent:%c\n",mm1[0]);
15.      }
16.     else{
17.          printf("mm1[0] inside child:%c\n",mm1[0]);
18.     }
19.   }
```

Figure 15: Example: `cfork()`

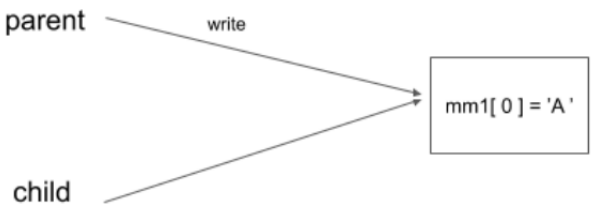

Figure 16: After parent calls cfork()

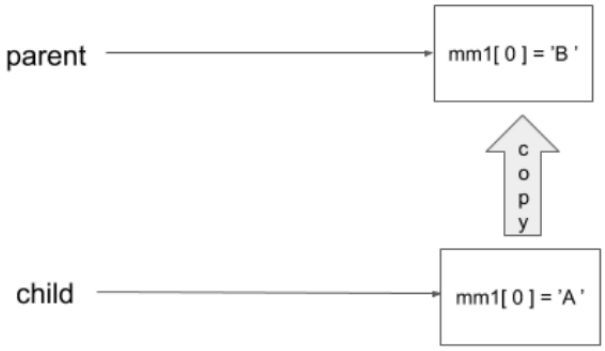

Figure 17: After parent writes to an address in shared page

In the `do_cfork()` function you have to perform the following operations:

- Copy all the members of the parent process's `exec_context` to the child process's `exec_context`. For example, copy the contents of the `files` array (array of file descriptors) from the parent process to the child proces.
- Set the `ppid` (pid of the parent process) in the child process's `exec_context`

- Build a new page table for the child process. Note that gemOS uses 4-level page table.
  - In this page table, you have to create page table entries only for the userspace part of the address space. Specifically, you have to create the page table entries for the present pages in the following memory segments of the child process—`MM_SEG_CODE`, `MM_SEG_RODATA`, `MM_SEG_DATA`, `MM_SEG_STACK` and for the VMAs of the child process. You can access the range of each memory segment using `mms[]` array in the `exec_context` of the child process. For example, `current->mms[MM_SEG_CODE]->start` and `current->mms[MM_SEG_CODE]->next_free` gives the range of address space covered by the code segment of the current process.
  - Page table entries in the PTE level/last level of the child process's page table should point to the same frames that are being pointed by the page table entries in the PTE level/last level of the parent process's page table. While updating the translation information, in both parent and child processes, the access permission in the PTE should be updated to restrict write accesses. You have to increase the reference count of the shared frames to indicate the number of page table entries pointing to the same frame.
  - Modify the `pgd` member of the child `exec_context` to update it with the PFN value of the PGD level used for the page table of the child process.

Return Values: On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created. Note that, the template function already has some crucial calls which ensures correct execution of child process (if created correctly). You should not change them.

3.2 long handle_cow_fault(struct exec_context *current, u64 vaddr, int access_flags)

current: `exec_context` of the process that received the CoW fault
vaddr: Virtual address whose access resulted in the CoW fault
access_flags: Access flags of the VMA to which the faulting address belongs to.
Description: `handle_cow_fault` function is called from `vm_area_pagefault()` function (handles the faults in the memory regions created using mmap() system call) and from other functions that handle the page faults occurring in the memory segments such as stack segment, code segment etc. So, your implementation of `handle_cow_fault` should be able to handle CoW faults occurring for any userspace address. In this function, depending on the address and flags, you have to update the virtual-to-physical translation information and adjust the reference count of PFNs.

Note the following changes to the implementation of `munmap` and `mprotect`.

- With CoW mapping, you may not free any mapped page on a `unmap` system call. You are required to use the PFN reference count to ensure correctness.
- With CoW mapping, while changing the access flags of the VMA and the translation entries, you must consider shared nature of PFN. For example, you can not blindly update the page table to allow write to a CoW mapping.

Return value: Return 1 if fault has been fixed, return -1 otherwise.

Testing

The user space program code is available in *gemOS/src/user/init.c*. You need to write your test cases in `init.c` to validate your implementation. The sample test cases in *gemOS/src/user/part3/* can be copied into `init.c` to make use of them.

4 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.
vm_next - The pointer to the next `vm_area` mappings.

MMAP_AREA_START & MMAP_AREA_END

These are constants defined in the file *gemOS/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 -1.

void *os_alloc(u32 size)

Allocates a memory region of `size` bytes. Note that you can not use this function to allocate regions of size greater than 2048 bytes. This function returns 0 in case of error.
Example usage: `struct vm_area *vm = os_alloc(sizeof(struct vm_area));`

void os_free(void *ptr_to_free, u32 size)

Use `os_free` function to free the memory allocated using `os_alloc`.
Example usage: `os_free(vm, sizeof(struct vm_area));`

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.

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.

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.

s8 get_pfn(u32 pfn)

Increments the reference count of a frame by one.

s8 put_pfn(u32 pfn)

Decrements the reference count of a frame by one.

s8 get_pfn_refcount(u32 pfn)

Get reference count of a frame.

struct exec_context* get_current_ctx(void)

This function returns the `exec_context` of the current running process.

Page-Fault Error Code

The error codes generated in case of a page fault are shown in Figure 18, which is taken from course slides. Example interpretation of some of the error codes is as following:
`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 worry about P, W, U bits.

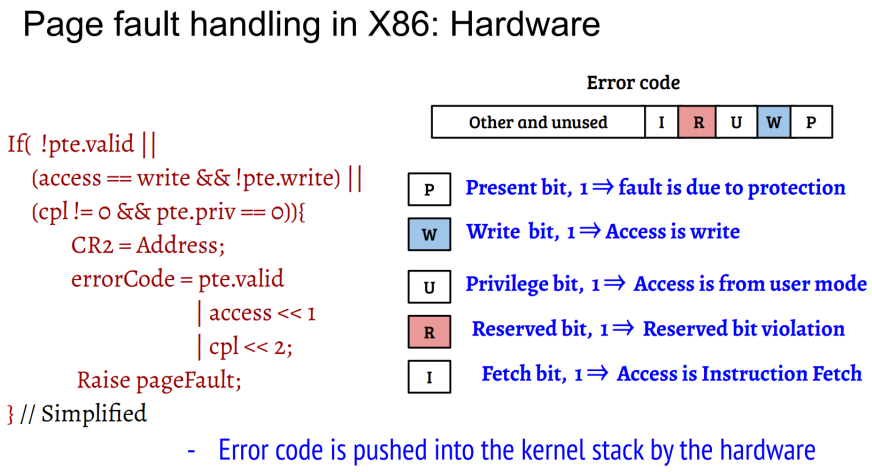

Figure 18: Page-Fault Error Codes

5 Submission

- Do not have any additional logging/printing in the submitted code.
- You have to include a file named '`declaration`' in your submission. In the '`declaration`' file, you have add the following statement:

  "`I have read the CSE department's anti-cheating policy available at https://www.cse.iitk.ac.in/pages/AntiCheatingPolicy.html. I understand that plagiarism is a severe offense. I have solved this assignment myself without indulging in any plagiarism. If my submission is found to be plagiarized from the internet, fellow students, etc., then strict action can be taken against me. <Your Name and Roll No>`"
- In the '`declaration`' file, you also have to mention the resources, such as websites, open source content you referred to while solving this assignment.
- You have to submit zip file named `your_roll_number.zip` (for example: `1211405.zip`) containing **only** the following files in specified folder format:

```
your_roll_number.zip
        |
        |----- your_roll_number
                    |
                    |---- v2p.c
                    |---- declaration
```

- If your submission is not as per the above instructions, a penalty of 20 marks will be applied on the marks obtained in this assignment.
- **Note:** No code changes will be allowed after the assignment submission period has ended. So, test your implementation thoroughly with the provided test cases as well as your custom test cases.