Assignment 3

CS330: Operating Systems

1 Introduction

As part of this assignment, you will be implementing system calls in a teaching OS (gemOS), same as in Assignment-2. We will be using a minimal OS called gemOS to implement these system calls and provide system call APIs to the user space. The gemOS source can be found in the `src` directory. This source provides the OS source code and the user space process (i.e., `init` process) code (in `src/user` directory).

gemOS is a teaching operating system. Typically OS boots on a hardware. To set up the gemOS, you can refer to the document (`Documentation/gemOS_Setup.pdf`)

The Syscalls Specifications

The specifications of the systems calls that you need to implement are explained below.

- `void *mmap(void *addr, size_t length, int prot, int flags)`
- `int munmap(void *addr, size_t length)`
- `int mprotect(void *addr, size_t length, int prot)`
- `int cfork(void)`
- `int vfork(void)`

`mmap(void *addr, size_t length, int prot, int flags)`

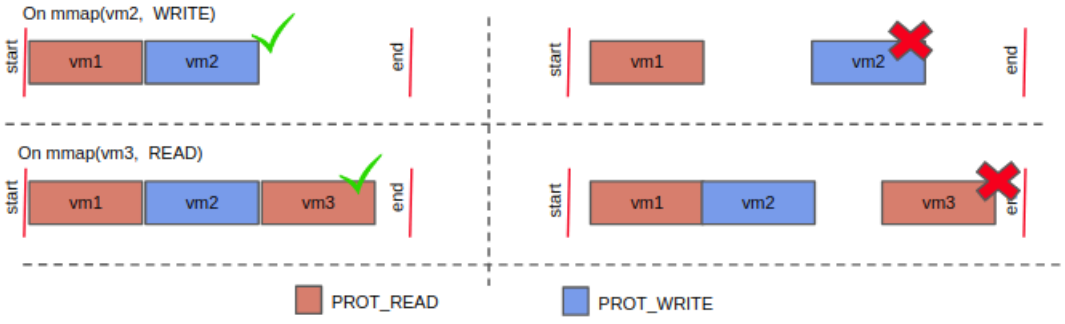
Creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in `addr` which is page aligned. The `length` argument specifies the length of the mapping (which must be greater than 0 and may not be in the multiple of page size).

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

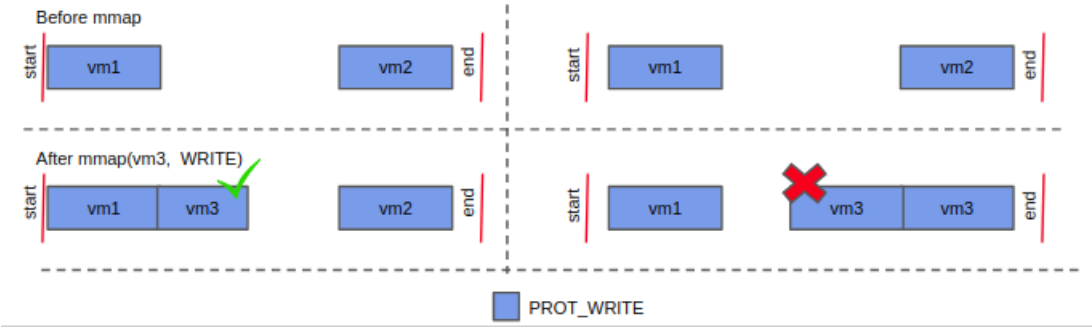
`Without hint address`

- 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.

- The subsequent mapping (new mapping) should be created in next contiguous free address in the virtual address space.



- While merging, always choose the first immediate available free space in the virtual address space



`With hint address`

- 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. (Refer to case 1 in the following diagram)
- When the new mapping's end is followed by the start of an existing mapping and has same protection flags as the existing one, new mapping should be merged with the existing mapping. (Refer to the case 2 in the following diagram)
- When the new mapping cannot be merged with any of the existing mappings, a new mapping should be created (Refer to the case 3 in the following diagram)

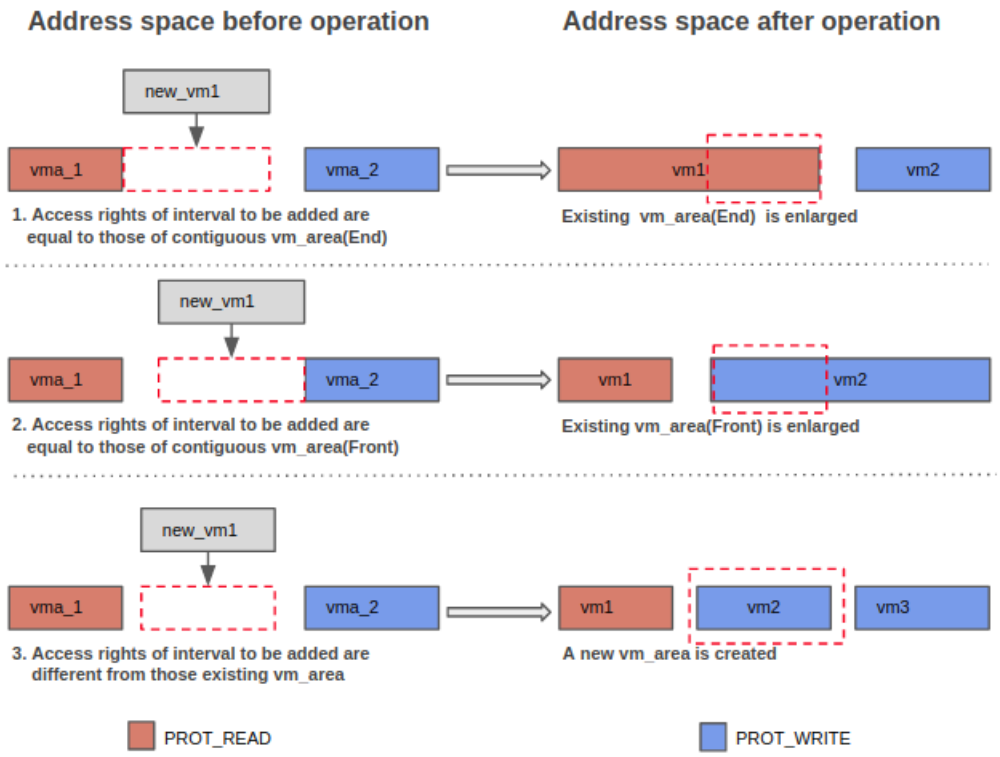


Figure 1: `mmap` with hint addres

If another mapping already exists for the provided hint address, pick a new address that may or may not depend on the hint address. You might have to create a new mappings or merge with existing mapping as mentioned in the above figure. The new address which was picked is returned as the result of the `mmap` call.

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`. If any process tries to write on this physical page it will result 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.

The usage of `flags` argument is explained below.

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

- `MAP_POPULATE` - Map the physical pages to the mappings(`vm_area`) at the time of creation. If a mapping (`vm_area`) is created without using `MAP_POPULATE` flag. Then created mapping will not have any physical pages mapped with it. The physical pages are mapped on demand (Lazy allocation) whenever they are accessed. The access will in turn result in a page fault and then physical pages are mapped (huge overhead).

RETURN VALUE

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

`munmap(void *addr, size_t length)`

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` (refer the below figure). Any access to addresses within the deleted `vm_area` result in invalid memory references.

The address `addr` must be a multiple of the page size (but `length` need not be). Assume that address passed as an argument is always page aligned. 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.

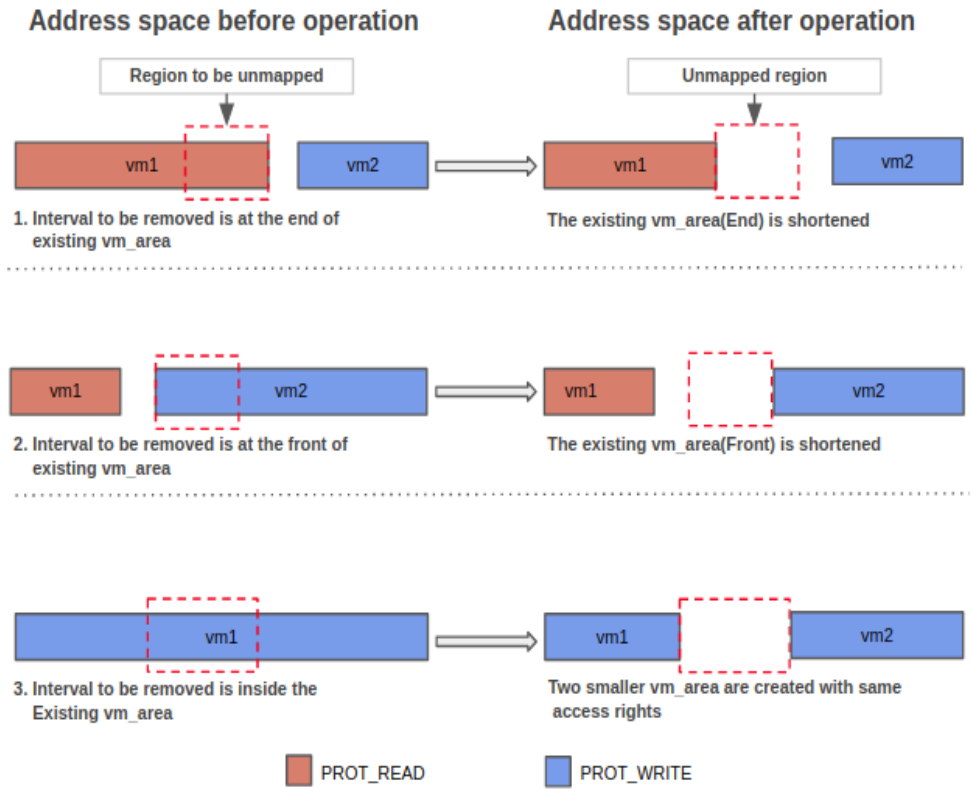


Figure 2: Unmapping of the VM area

`RETURN VALUE`

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

`mprotect(void *addr, size_t length, int prot)`

`mprotect` 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). Assume that `addr` provided as argument is always page aligned. `mprotect` might create, expand or shrink the `vm_area` mapping(refer to the below figure).

Figure 3: Unmapping of the VM area

If the calling process tries to access memory in a manner that violates the protections, then the kernel generates a SIGSEGV signal for the process. As we are updating the access rights of certain regions in the `vm_area`, the access rights should also be updated accordingly in the underlying physical pages as well if they exist.

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`. If any process tries to write on this physical page, it will result 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 physical pages. Hence the physical pages which map to this `vm_area` will have `READ_WRITE` access.

RETURN VALUE

On success, `mprotect()` returns 0. If the provided `addr` does belong to the existing `vm_area` mapping. then `mprotect` will fail and returns either (-1) or (`EINVAL`)

cfork(void)

`cfork()` creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

The child and the parent process run in separate memory spaces. At the time of `cfork()` both memory spaces have the same content. After `cfork()`, a write performed by parent or child results in copying of the page and thus child and parent stops sharing the page.

The child and parent has separate user and kernel stack and memory areas shared by them till copy-on-write happens are data segment, mmaped regions. The child and parent continue sharing code region since it is read only regions.

The child process is an exact duplicate of the parent process except for the following points:

- The child has its own unique process ID.
- The child's parent process ID is the same as the parent's process ID.

Example

```
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.   }
```

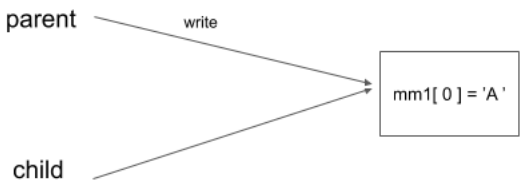


Figure 4: After parent called cfork()

After parent performs `cfork`, the parent and child continues to share the memory regions as shown in figure 4.

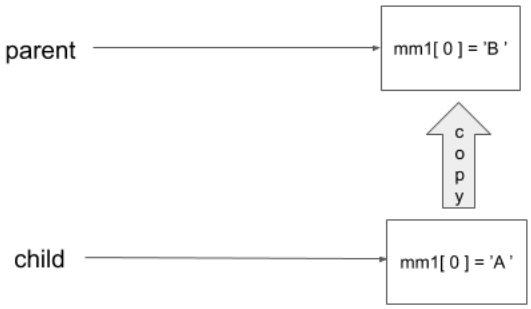


Figure 5: After parent writes

After `cfork`, parent writes to the mapped area as shown in line number 13, which results in copy-on-write and child and parent stops sharing the memory area as shown in figure 5.

```
mm1[0] inside child:A
mm1[0] inside parent:B
```

In the output shown above, as you can see, the child still sees the unmodified value where as the parent prints the new value.

RETURN VALUE

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, and errno is set appropriately.

vfork(void)

`vfork()`, just like `cfork()`, creates a child process of the calling process. `vfork()` create new processes without copying the page tables of the parent process.

`vfork()` differs from `cfork()` in that the calling thread is suspended until the child terminates (by calling exit(), or abnormally, after delivery of a fatal signal). Until that point, the child shares all memory with its parent, including the stack.

The child and parent has separate kernel stack and memory areas shared by them till child exits are user stack, data segment, mmaped regions, and code.

Example

```
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 = vfork();
12.   if(pid)
13.   {
14.      printf("mm1[0] inside parent:%c\n",mm1[0]);
15.   }
16.   else
17.   {
18.      mm1[0] = 'B';
19.      printf("mm1[0] inside child:%c\n",mm1[0]);
20.      exit(0);
21.    }
22.}
```

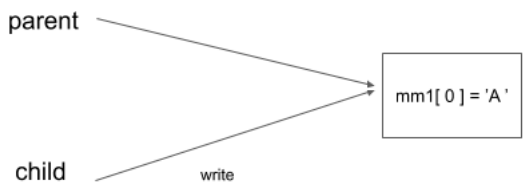


Figure 6: After parent called vfork()

After parent performs `vfork`, the parent and child continues to share the memory regions as shown in figure 6. The parent is in waiting state after `vfork`, and child is ready for scheduling. The parent continues in waiting state until child exits.

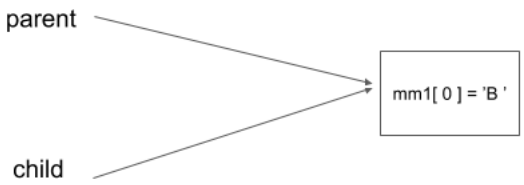


Figure 7: child writes and exits

After `vfork`, child writes to the mapped area and exits by calling `exit(0)`.

```
mm1[0] inside child:B
mm1[0] inside parent:B
```

In the output shown above, as you can see, the parent prints the value which is written by the child.

RETURN VALUE

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, and errno is set appropriately.

Background and Utilities

We have already provided a detailed specification of system calls that you need to implement. In order to make things easier, we have given some template functions, structures and a few utility functions that facilitate object creation and deletion. Lets 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 structure.

`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.

`struct vm_area* alloc_vm_area():`

This function is used to create a new `vm_area` mapping and returns a pointer to the created `vm_area`. We won't be filling or setting any values to members of the `vm_area`. You should only use this function to create `vm_area` in the entire assignment.

`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_area` in the entire assignment.

`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).

`pmap(int details):`

You can use the pmap methods to check the `vm_area` details. If `details` is zero. Then pmap will print the count of `vm_area` 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.

`struct pfn_info (in page.h)`

This object represents the physical memory page. The `pfn_info` for each page is maintained in `list_pfn_info` object, which is indexed by the `pfn` of physical memroy page.

`pfn_info` object has `refcount` which is used to maintain count of number of sharers for a physical memory page. This count helps to identify whether to copy a page at the event of a write after `cfork()`. The page needs to be copied when number of sharers is greater than one.

`get_pfn_info(u32 index) (in page.c)`

You can make use of this function to get `pfn_info` object corresponding to a `pfn` of physical memory page.

`increment_pfn_info_refcount(struct pfn_info * p)(in page.c)`

You can use this method to increment refcount of a physical memroy page at the time of `cfork()`.

`decrement_pfn_info_refcount(struct pfn_info * p)(in page.c)`

You can use this method to decrement refcount of a physical memroy page at the time of copy-on-write after a `cfork()`.

`get_pfn_info_refcount(struct pfn_info *p)(in page.c)`

This method gives current refcount of a physical memroy page.

`format of Page Table Entry`

The format of PTE entry which maps to 4KB pages in intel x86 architecture is as shown in figure 8. Those who want to know more, can refer Intel Software Manual (page 2831).

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 8: PTE Entry

For this assignment, you can ignore bit positions 3-11 and 51-63. You only need to modify bit positions 0, 1, 2 and 12-50.

Page-Fault Error Code

The error codes in intel x86 architecture incase of a page fault is as shown in figure9, which is taken from course slides. User-mode read access to an unmapped page results in error code 0x4, same way user-mode write access to an unmapped page results in error code 0x6. The error code 0x7 indicates user mode write access to read-only page which has page table mapping. Those who want to know more, can refer Intel Software Manual (page 2836). For this assignment, you need to only modify P, W, U bits.

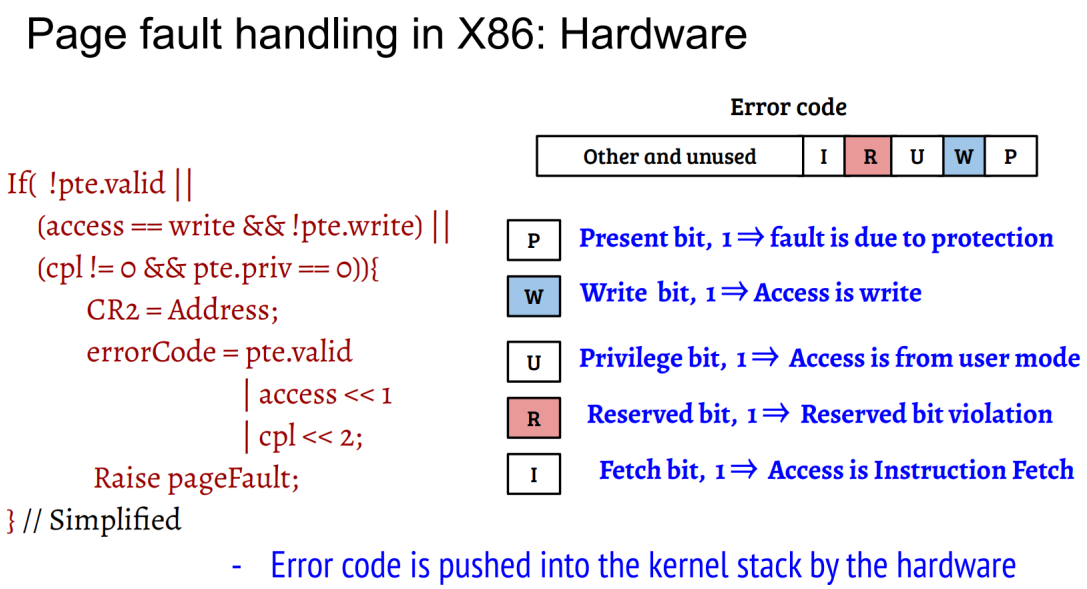


Figure 9: Page-Fault Error Code

Task-1: Virtual memory area operations (30 Marks)

mmap():

To implement mmap system call, you are required to provide 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 be creating (adding a node in singly linked list), shrinking, expanding and merging the vm_area in the singly linked list (refer the specification mmap). To create and delete use alloc_vm_area and dealloc_vm_area respectively.

munmap():

To implement munmap system call, you are required to provide 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 be deleting and shrinking the (vm_area) mappings in the singly linked list (refer the munmap specification).

mprotect():

To implement mprotect system call, you are required to provide implementation for the template function vm_area_mprotect in the file [src/mmap.c]) which takes the current context, address, length and protection as arguments.

If the provided address is not part of any vm_area. Then mprotect will fail the request and return EINVAL or (-1). Based on the request, You might be creating ,shrinking, expanding and merging the vm_area in the singly linked list (refer the specification mprotect).

Validation Procedure:

- In this part of the assignment we won't be validating anything regarding physical pages(mapping physical page, updating page table, page fault handling)
- We will only 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.
- There can be at most 128 vm_area at any point of time. If not then return EINVAL or -1.
- The address ranges for all the above system call should be between the address ranges MMAP_AREA_START & MMAP_AREA_END. If not then return EINVAL or -1.
- We will be checking the vm_area protection or access rights.
- Assume all the address provided to the syscalls are page-aligned.

Task-2: Assigning physical pages (30 Marks)

We have provided a template function `vm_area_pagefault` which takes the current context, address (faulted address) and error code. This function will be called whenever there is read page fault (`error_code` will 4) and write page fault (`error_code will be 6`) in mmap addresses ranging between `MMAP_AREA_START` and `MMAP_AREA_END`.

For valid access, map a physical page and function should return 1. For invalid access (Write fault on the address which maps to read only `vm_area`) function should return -1.

`mmap()`

If a `vm_area` mapping is created without providing a `MAP_POPULATE` flags. Then all the subsequent access to the `vm_area` mapping will result in page fault (Lazy allocation).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.

If a `vm_area` mapping is created with `MAP_POPULATE` flags. Then you have to map physical page, set access flags and update page table entries at the time of `vm_area` creation, Any access to the `vm_area` mapping is created with `MAP_POPULATE` flags should not result in page fault.

`munmap()`

On unmapping a `vm_area` mapping, if a `vm_area` has physical pages mapped to it then you have to unmap the physcial pages and update the page tables accordingly.

`mprotect()`

On changing the protection or access rights of `vm_area` mappings, if a `vm_area` has physical pages mapped to it. Then you have to update the access rights as the physical page as well.

`Validation Procedure:`

- We will be validating state of `vm_area`, number of page faults, access right of physical pages and `vm_area`.

Task-3: Smart Process Creation (40 Marks)

cfork()

cfork(copy on write fork) creates a new process by duplicating the calling process. The new process is called child process and the calling process is called parent. The child process and parent process run in separate memory spaces. The child and parent share physical memory pages till either of them modifies it. The page is mapped as read-only to both parent and child initially and at the event of a write, copy of the page is created and PTE entry is updated to point to new page.

As part of this task, you need to implement cfork_copy_mm(in cfork.c). The functionality of cfork_copy_mm is to setup page table for the child. The page table entries for the child point to the same physical pages of the parent on copy-on-write basis. You need to mark pages are readonly (so that write to these pages can be identified to trigger copy-on-write) by clearing R/W bit in PTE entry of child and parent page table (0 value at R/W bit indicates writes may not be allowed to the 4-KByte page referenced by this entry). You can use install_ptable(in context.c) to setup page table for the child. The child will share all memory segments and vm areas of parent on copy-on-write basis except for user stack area, child has a separate user stack from that of parent.

You also need to implement handle_cow_fault(in cfork.c) which will be called from do_page_fault incase of a copy-on-write fault (error_code will be 7). In handle_cow_fault, you should ensure that the faulting address(cr2) is in a valid range and write is allowed as per access_flags of faulting segment/vm area.

cfork_copy_mm will be called from do_cfork(in entry.c). The do_cfork(in entry.c) gets a new exec_context and copies current execution context. The do_cfork finally calls setup_child_context to create os stack and make child ready for scheduling.

Please note that there is no ASID support in the system, so the TLB entries should be handled accordingly.

Validation Procedure:

- We will check number of copy-on-write page faults.
- We will check number of pages before and after a write followed by cfork().

vfork()

vfork is same as cfork. vfork differs from cfork in that the calling process is suspended until the child terminates by calling exit(). Until that point, the child shares all memory with its parent, including the stack. The child must not return from the current function and should call exit() in user code. The child process should take care not to modify the memory in unintended ways, since such changes will be seen by the parent process once the child terminates.

As part of this task, you need to implement vfork_copy_mm (in cfork.c) which is called from do_vfork (in entry.c). The functionality of vfork_copy_mm is to make the child use parent's page table.

The `child` will share all memory segments and vm areas of `parent` including the user stack. You need to make sure that, the child does not overwrite user stack entries of parent to ensure correct execution of parent after child exits. You can overwrite the `entry_rsp` and `rbp` in `user_regs` which is present in `exec_context`(in context.h) of the process to manipulate user stack position, since these values are loaded to hardware registers by `schedule`(in schedule.c) at the time of process schedule.

The `parent` state should be changed to WAITING before scheduling `child` by calling `schedule(new_ctx)` in `do_vfork`.

When the `child` exits, change state of `parent` to READY in `vfork_exit_handle` (in cfork.c), which is called from `do_exit` (in entry.c).

`Validation Procedure:`

- We will check sharing of page table between parent and child.
- We will check that parent is waiting till child exits.

Submission guidelines

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

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