Virtual Memory Manager

Adela Yang, Son Ngo, Venecia Xu

1 Introduction

1.1 Problem Statement

The purpose of the project is to understand address spaces and virtual memory management. To do so, an external pager is implemented. The pager is a process that handles virtual memory requests for application processes. The pager handles address space creation, read and write faults, address space destruction, and simple argument passing between spaces. The pager manages a fixed range of the address space (arena) of each application that uses it. The pager is single threaded, handling each request to completion before processing the next request. Valid pages in the arena will be stored in simulated physical memory or in simulated disk. The pager manages these two resources on behalf of all the applications using the pager.

2 Methodology

2.1 Extra Bits: Mod, Ref & ZeroFilled

- *Modify Bit* (modBit): This bit indicates if a virtual page has been recently modified since the last time it was accessed. This bit gets updated in accordance with the *write_flag* from a page fault. That being said, whenever this bit is set to 0, we need to set the *write_enable* bit to 0, so that the next time the application is trying to write to the page, we can update the modBit. Furthermore, the pager can use this bit to defer extra work, such as writing to disk, because if the page has not been modified since it was last brought to memory, then there is no need to write the page data onto disk since it has already been there. This is very efficient since disk writing requires a lot of time.
- *Reference Bit* (refBit): This bit indicates if a virtual page has been recently referred (or accessed). The most important use of this bit is for the second-chance clock algorithm. Pages with refBit of 1 will be set to 0 in the algorithm for a second-chance. And pages with refBit of 0 will be evicted right away.
- *Ever-modified Bit* (zeroFilledBit): This bit indicates if a virtual page has ever been modified since creation. The use of this bit distinguishes from the modBit is that once this bit has been set to 1, its value will not be changed. When a page is about to be brought into memory, we need to decide whether to read from the disk block assigned to the page or not. If the page has never been modified, then we never have written the page to disk (when evicted) in the first place. Therefore, no disk reading would be needed. However, if the page has the zeroFilledBit of 1, then the data of the page must sit on its assigned disk block, thus a disk reading has to be performed. By doing this, we can optimize the functionality of the pager, especially in *vm_fault* by deferring unnecessary work.

2.2 Structs

There are three structs that the pager uses in order to keep track of process' and pages' information.

- *process* contains a pointer to the process' page table and the next lowest valid address in the process' arena. This struct is very important in helping the pager update as well as access the most current information regarding the process, especially in *vm_switch* and *vm_destroy*.

- *node* contains all the bits and originating information of a virtual page. These include a modBit, a refBit, the pid that the virtual page belongs to, the page number itself, and a pointer to the page's corresponding entry in the process' page table. These information are very crucial in the implementation of the second-chance clock algorithm as well as the effectiveness of the pager in delaying extra work (disk writing/reading, zero filling a physical page, etc.) and de-allocating virtual pages on destruction. Specifically, the pointer to the entry on the page table plays an important role in responding to a page fault. We can directly change the information that is stored inside the entry which will be accessed constantly by the infrastructure in order to respond to the application's fault. The pid information is needed when the virtual page is being destroyed from the *ClockQueue*, we need to make sure that we are only deleting the pages of the currently finished process.

- *vpageinfo* contains binding information of a virtual page: the disk block that it is assigned, whether it's resident or not and the zeroFilledBit. These information can technically be embedded in with the *node* struct; however, we decide to separate these information because these information are mostly static through a virtual page's life time (except for the resident boolean). Furthermore, we need to access these information a lot in *vm_fault*, and if we combine these with *node* struct, then we will have to iterate through the *ClockQueue* many times which can worsen overall performance. Spacewise, separating these information will not create a significant extra space overhead since the size of this struct is relatively small compare to the *node* struct.

2.3 Pointers

Pointers play an indispensable role in utilizing the performance of external pager. Since the page_table_t object takes up quite a bit of memory because it stores a lot of information about the virtual pages, it would be a bad idea to copy a *page_table_t* object. Moreover, each process will have its own page table, which can be significant in terms of memory allocation. Therefore, for every process, we will need a pointer to its own page table, which will be stored in the *process* struct. Additionally, we also have a pointer that points to the current process' map of all virtual pages. This pointer comes in very handy because the map structure is particularly large as it holds all virtual pages from a process, and we will need to obtain information about the virtual pages constantly throughout the pager. Even though pointers are very convenient in the implementation of pager, they come with great caution. Because pointers are also contained inside these structs, so de-allocation of pointers to those structs might create memory leaking if not handled carefully.

2.4 Major Data Structures

- *Clock queue* (ClockQueue) is composed of a queue of *node* structs that keeps track of the physical pages. When a page faults and there is no free physical memory, it needs to decide which pages to evict. The second-chance clock algorithm keeps a strict set of rules to help determine which nodes should be evicted – for example, pages that have not been accessed are instantly evicted. However, whether it is necessary to call *disk_write* depends on whether the evicted page has been modified. We choose to use queue here because the second-chance clock algorithm can access the top of the queue and push back the non-evicted nodes efficiently.

- *Free physical memory and disk block queues* (FreePhysMem and FreeDiskBlocks) hold the amount of free physical memory and free disk blocks, respectively. They are both queues of unsigned ints. The blocks are sequentially numbered upon initialization, and the maximum free spaces of each are designated in *vm_init*. When *vm_extend* is called, we check whether there are still available disk blocks using a helper method that pops off the front of the availability queue. When a fault occurs, we check available physical memory using a similar method. If there is no memory, the clock algorithm is activated to evict pages. Again the use of queue here is mostly for the fact that we can access the first item of the queue efficiently, which is what desired from the functionality of the pager.

- *Current process map* (CurrMapP) is a pointer that always points to the map of virtual pages for the current process. The key to the map is an unsigned int that is the virtual page number, and the value is *vpageinfo* struct. Everytime *vm_switch* is called, the pointer updates. We refer to CurrMapP for two important pieces of information – referencing the zero fill bit, and keeping track of residency.

- *All pages map* (AllPagesMap) uses pids as keys, and the values are current process maps. In *vm_switch*, the current map is switched to the correct one indicated by the AllPagesMap. Since the map holds all non-resident virtual pages, it can be used to free all the blocks under that condition. Since it is a map of all the current process maps, it can also be used to keep track of residency.
- *Process map* (ProcessMap) makes sure there is enough space in the arena and is fairly straightforward as a map. Process ids are the keys, and *process* structs are the values. In *vm_switch* it keeps the information for switching page table registers. The map also keeps track of and updates the lowest valid address for each process.

The use of map here is very crucial to the efficiency of the pager. Since we can access elements in map by key in constant time, looking up in the map by virtual page number will be very fast. Thus, the pager can spend more time on implementing the functionality needed in order to respond to calls from the infrastructure.

2.5 Functions

There are seven main functions in the pager, and the user can *indirectly* access these functions through the infrastructure.

- *vm_init* initializes the internal data structures and variables when the pager starts. These data structures include: a queue for free physical memory and a queue for free disk blocks. The pointer that points to *page_table_t* struct is also initialized with exception handler. Two arguments passed to *vm_init* are also stored into two global variables for possible future use (not particularly used in this project). It's a good practice to have these variables handy, especially for reproductive code. This initialization is needed for *vm_create* and subsequent requests from processes.
- *vm_create* creates process-specific data structures that are needed in order to keep track of a process' pages. Firstly, a pointer that points to a *page_table_t* struct is initialized and stored into the *process* struct. The lowest valid address in the process' arena is also initialized and stored into this struct. This struct is then put into a map of all processes with the key being their IDs. Then, since the process is new, all of its pages' bits (read & write) in the page table should be set to 0 so that the infrastructure can fault if the user has not requested any pages before any operations.
- *vm_switch* updates three global variables that can indicate the state of the program (i.e which process both the infrastructure and the pager should be dealing with): the *CurrentPid* - a key to all maps that handle process-related data structures, the *page_table_base_register* that will be used by the infrastructure and the pointer to the map that contains all pages of currently running process. This function will make sure that these global variables will always be updated to the currently running process since these variables are very crucial in the implementation of other functions in the pager, as well as the functionality of the infrastructure.
- *vm_fault* occurs when the current process has a fault at a virtual address. This is a response to a read and/or write fault by the application. The infrastructure will detect the fault and hands it off to the pager for fault handling. The pager determines which accesses in the arena will generate faults by setting the *read_enable* and *write_enable* fields in the page table. This setting behavior is important because the pager will have extra information (bits) about the virtual pages, and these information need to be constantly updated through fault, so that not only the process can continue to proceed from the fault, but the pager can make wise decision in delaying extra work in the future. By using these bits, the pager can observe different states of a virtual page since creation to destruction, thus decide how to handle the page fault.

  - *Non-Resident Pages*: If a fault occurs on a virtual page that is not resident, the function needs to find a physical page to associate with the faulted virtual page.

    If there are free physical pages, then the function will find the next free physical page from the *FreePhysMem* queue and associate it with the faulted page. Afterwards, the bits associated with this virtual page needs to be updated correctly. If the *write_flag* is true, then the all the bits should be updated to 1. Otherwise, all except the *read_enable* and *mod* bit should be 1.

    If there are no free physical page, then the second-chance clock algorithm will be activated. All pages that are currently in memory are kept in the *ClockQueue*, so the pager will examine each page one by one starting from the head of the queue. By checking the extra bits set by the pager, the function can decided which page to be evicted. Once the decision has been made, if the mod bit of evicted page is 0 then the function will not perform any write out of the page. Otherwise, the evicted page is written to disk. Then, the physical memory data of the physical page that is associated with the evicted virtual page should be filled with 0s so that the faulted virtual page when it enters memory should see a zero-filled page. One extra question to ask is if the faulted virtual page is on disk by using the zeroFilledBit. If it is, we need to read the data from disk. Otherwise, it would not be ideal to read from disk since disk operations take a lot of time.

  - *Resident Pages*: If a fault occurs on a resident page, then the function will simply update the *read_enable* and *write_enable* bits according to the request from the application users. The *ref* of this faulted virtual page will be updated to 1 regardless, and the *mod* bit will be 1 if the *write_flag* is true.

  This function will return 0 on success and -1 otherwise. The infrastructure will check these return values and finish the fault handling process.

- *vm_destroy* deallocates all resources held by the current process and extra related data that are managed by the pager. The crucial part of this function is to remove all active pages (*node* struct) from the current processes from the *ClockQueue*. Additionally, entries in *PhysMemMap* also need to be erased. As the process of freeing memory and space, the function also puts back free disk blocks and free physical pages to their respective queues. First we do this while we iterate through *PhysMemMap* and remove the key with the same page number from pages map in *AllPagesMap*. Then, we need to free the disk blocks from the virtual pages that are not active by iterating through the virtual pages map in *AllPagesMap*. Since we have already deleted some of the elements in this map while we are iterating through *PhysMemMap*, we reduce the time to iterate through this map. At the end, we need delete the page table object that is associated with this process and set *page_table_base_register* to NULL.

- *vm_extend* is called when there is a request by the current process make another virtual page in its arena valid. We first try to associate a free disk block with the requested virtual page, and thus will be unique to the page until destruction. This disk block "reservation" will ensure we will never make errors in writing and reading from the wrong disk blocks. In other words, we will disallow disk block sharing. Then, we need to make sure we still have enough address space in the arena. After the availability test has passed, we simply need to update the variable that indicates the lowest valid virtual address, which will be returned to the user. Because a virtual page has been declared valid at this point, we need to initialize a *vpageinfo* struct and stored it into *AllPagesMap* for future references.

- *vm_syslog* is a request by the current process to log a message that is stored in the process' arena starting at address "message" and is of length "len". Firstly, the function does a sanity check on the arguments and make sure that all addresses are valid within the process' arena. Then, we simply go from the beginning address to the ending address and construct a string that subsequently adds each data at each address. There are two things that need to be taken into consideration in this function. The first one is that an address can belong to a page that is not resident, thus the function (the pager) will need to fault at that page actively as this will not be detected by the infrastructure. The second thing is that two pages can belong to two different pages, so we need to make sure we compute the corresponding virtual pages at each address so that we can check the residency of those pages correctly. The procedure can be optimized by computing the ending address of a page at the beginning so we do not need to retranslate from addresses to pages at every address. However, we decided not to implement that algorithm because of time constraint and the complexity of the code, while the performance improvement from this change is not significant.

There are also other helper functions that help to improve not only the appearance of the code but also the comprehensibility given their descriptive names. One crucial helper function is *updateInfo*, which takes in a lot of arguments. Even though the function itself is really complicated with bits updating, it makes the *vm_fault* function very comprehensible to other programmers as it encapsulates the main functionality of the pager when handling page faults.

3 Data and Results

3.1 Test Cases

Test cases are created to check the functionality of the three main user functions, *vm_syslog*, *vm_extend*, and *vm_yield*. The test cases were designed to test different transition paths that a page can take through a pager's state machine. There are test cases that causes a page to take each path.

The following is a list of all our test cases and its testing description:

Tests	**Description**
test1	Checks the functionality of syslog
test2	Syslog at an invalid address, check updates of state $(1, 0, 1, 0, 0)^*$, syslog before extend
test3	Illegal write to an invalid page
test4	Illegal read to an invalid page
test5	Trying to extend more than available disk blocks
test6	Check syslog, check updates of state $(1, 0, 1, 0, 0)^*$
test7	Multiple paged message

* refers to the $(read, write, reference, modified, zero-filled)$ bits

3.2 Results

The following are tests that are selected because their collective output exposed all of the instructor's buggy pagers.

```
vm_create (18188)
returning to (18188) with pages:
vm_extend (18188)
vm_extend returned 0x60000000
returning to (18188) with pages:
vm_fault (18188) 0x60002001 write
vm_fault returned -1
returning to (18188) with pages:
vm_destroy (18188)


libpager: Segmentation violation: write to address 0x60002001
```

Figure 1: Output given by test3
Passed: A S

Test3 performs an illegal read to an invalid page by trying to read page element 8193. The infrastructure first detects a fault and let the pager handle the fault. The pager should detect that this is an illegal address and return -1.

```
vm_create (18285)
returning to (18285) with pages:
vm_extend (18285)
vm_extend returned 0x60000000
returning to (18285) with pages:
vm_fault (18285) 0x60000000 write
vm_fault returned 0
returning to (18285) with pages:
rw vpage 0x30000 ppage 0x0

vm_extend (18285)
vm_extend returned 0x0
returning to (18285) with pages:
rw vpage 0x30000 ppage 0x0
vm_destroy (18285)


not enough arena size for 1023
test 5
```

Figure 2: Partial output given by test5
Passed: A K M

Test5 extends more than the number of 1available disk blocks. There are at most 1024 disk blocks so it returns null.

```
vm_create (18735)
returning to (18735) with pages:
vm_extend (18735)
vm_extend returned 0x60000000
returning to (18735) with pages:

vm_fault (18735) 0x60000000 write
vm_fault returned 0
returning to (18735) with pages:
rw vpage 0x30000 ppage 0x0


vm_fault (18467) 0x60010001 write
disk_write block 0x6 ppage 0x1
vm_fault returned 0
returning to (18467) with pages:
rw vpage 0x30007 ppage 0x0
rw vpage 0x30008 ppage 0x1
vm_syslog (18467) 0x6000c000 5
syslog
vm_syslog returned 0
returning to (18467) with pages:
r vpage 0x30006 ppage 0x3
rw vpage 0x30007 ppage 0x0
rw vpage 0x30008 ppage 0x1
vm_destroy (18467)


c2(2): c
test 10
f2(2): c
test 12
test 13
```

Figure 3: Partial Output given by test6
Passed: A C D E F G H J K L O P Q T

Test6 tests the optimization of the algorithm by testing to see if page *b* is written out only once. There are arbitrary *vm_yield* statements sprinkled throughout the program. It also makes sure that *e* is never written out, and thus never read from the disk when brought to memory. It also checks the updates of state (1,0,1,0,0), when evicted by the clock algorithm and the functionality of the zero-filled bit.

```
vm_create (19067)
returning to (19067) with pages:
vm_extend (19067)
vm_extend returned 0x60000000
returning to (19067) with pages:
vm_fault (19067) 0x60000000 write
vm_fault returned 0
returning to (19067) with pages:
rw vpage 0x30000 ppage 0x0

vm_syslog (19067) 0x60000000 24576
syslog  denneciaosmeep
vm_syslog returned 0
returning to (19067) with pages:
rw vpage 0x30000 ppage 0x0
rw vpage 0x30001 ppage 0x1
rw vpage 0x30002 ppage 0x2

vm_destroy (19067)


switched
a is renewed
succeeds
failure
succeeds
test 7
```

Figure 4: Partial Output given by test7
Passed: A B C D E F G H I J K N O P Q R

Test7 looks at a multi-paged message by calling *vm_syslog* with length much greater than a page's size.

There are three other tests whose output is not listed above. Test1 checks the functionality of syslog. The pages should be 0 for unmodified addresses. Syslog should return -1 when the length is 0 or -1. Test2 tests syslog at an invalid address. It also checks the updates of state (1,0,1,0,0) when evicted by the clock algorithm and the functionality of the zero-filled bit. Then it checks what happens when syslog is called before extend as well as what happens when there is an invalid message. In both cases, syslog returns -1. Test4 issues an illegal read to an invalid page of 8193 and thus creates a segmentation violation.

4 Conclusion

In a normal computer system, the CPU's memory management unit (MMU) and exception mechanism perform several important tasks. The MMU is invoked on every virtual memory access. This project was important in the understanding how applications on real hardware communicate with real operating systems. The applications issue load and store instructions and these are translated or faulted by the infrastructure. The infrastructure transfers control on faults and system calls to the pager, which receives control via function calls. Our virtual memory manager mimics the MMU and by implementing this project, it helped us gain a better understanding of page faults and translations work.