CS 452 (Spring 23): Operating Systems

Phase 5 - Virtual Memory

due at 5pm, **Wed** 3 May 2023

1 Phase 5 Overview

In this Phase, you will implement a simple virtual memory system. **Unlike a real OS,** where the VM covers the **entire** space of addressable memory, this VM system will only cover a small fraction of the address space of each of your processes. It is important to understand that **most of your memory will continue to work as in the previous phases** - only pages in the "virtual memory region" will be affected by Phase 5.

Additionally, we have vastly simplified the system for you by making two very unrealistic assumptions:

- The virtual memory region in every process is at the same virtual address, and exactly the same size.
- Every process, at the moment is created, has its complete virtual memory filled, with every page mapping to "anonymous" memory. The user can never change this - they cannot map new pages, or unmap old pages.

Finally, Phase 5 has made one other critical **change from the norm:** `fork1()` does **not** duplicate pages from the parent process to the child! In a traditional UNIX `fork()` syscall, the virtual memory of the child is identical (including identical contents) to that of the parent - at least, when the fork is performed. But in this project, a new process always gets a new, pristine virtual memory - all filled with zeroes. (I told you, in Phase 1, that the `fork1()` call didn't work very much like the UNIX `fork()`; this is another example.)

1.1 Your Tasks

To complete this project, you must:

- Add code in `phase5_init()` to initialize USLOSS's VM subsystem
- Define various tables for storing data about processes, virtual pages, physical pages, and swap file pages
- Define an interrupt handler for page faults
- Create some "pager daemons" - the processes that will help you to find and initialize pages for use by other processes
- Fill empty pages with zeroes, when a process touches a virtual page for the first time
- Add code to keep track of `REF` and `DIRTY` bits for each frame

- Using the `REF` bit, choose which physical page to discard, when there is memory pressure
- Write pages to swap disk if necessary
- Read pages back if necessary
- Keep track of `vmStats`

1.2 `malloc()`. `free()`, and Related Functions

In Phase 5, you are allowed to dynamically allocate and free memory. But rememeber: if you try to dynamically allocate your tables, you are likely to have more bugs, and more crashes!

1.3 Limits (`CONSTANTS` to Make Your Life Easier)

In a real system, we would use dynamic memory allocation for many of our tables. But in Phase 5, to make things simpler, I am going to give you **max** sizes for each array. I encourage you to declare constant-size arrays in global memory, like this:

```
MyVirtPageMetadata vp_metadata[VM_MAX_VIRT_PAGES];
```

but if you want to dynamically allocate these arrays, this is legal.

You will be provided with the following constants:

- `MAXPAGERS` - the maximum number of "pager daemons" that you will be asked to create
- `VM_MAX_VIRT_PAGES` - the largest possible size for the "virtual memory region," in pages
- `VM_MAX_PHYS_PAGES` - the largest possible size for the physical memory, in pages.
- `VM_MAX_SWAP_PAGES` - the largest possible swap disk, in pages. (Note that all pages are 4K in this project.)

(spec continues on the next page)

2 Task: Initialization

You need to write code in `phase5_init()` which starts up USLOSS's MMU (that is, its virtual memory subsystem). I encourage you to read the USLOSS documentation about `USLOSS_MmuInit()`, but here are some key hints:

- We will be running in the MMU in "page table mode:" `USLOSS_MMU_MODE_PAGETABLE`
- Because we are in page table mode, the "number of maps" (that is, the number of page table entries) must be exactly the same as the "number of pages" (that is, the number of virtual memory pages visible to each process)

How many virtual and physical pages should you allocate? Each testcase will tell you. Each testcase will define the variables `vm_num_virtPages` and `vm_num_physPages`; you can simply pass those variables to `MmuInit()`.

You are not required to call `MmuDone()` (to shut down the VM subsystem) in this project.

2.1 Virtual and Physical Memory Regions

After you initialize the VM subsystem, you should call `USLOSS_MmuGetConfig()`. Use it to find out where USLOSS placed the virtual memory region, and also the physical memory region.

You must share the location of the virtual memory region with the testcase; place the pointer in a variable named `vmRegion`. You should not share the location of the physical memory with the testcase. (It's OK for your code to save and use the physical memory buffers, but testcases will not.)

2.2 Hooks from Phase 1

In Phase 1, we had several "hooks" into Phase 5 - these were functions called by the Phase 1 code, which would be implemented in Phase 5. I have **changed** these functions; you will need to **download the updated Phase 1 library file,** and design your code against the new version.

In this Phase, you will need to provide:

- `USLOSS_PTE *phase5_mmu_pageTable_alloc(int pid)`

  Called by `fork1()`; you should allocate new memory[1], which will be the page table of the new process. The `pid` is the PID of the child that is being created.

[1]Or, you could have a fixed-length array in the shadow process table, and return a pointer to it...

The page table is an array of `USLOSS_PTE` entries; there will be one entry for each virtual page known to the process. This page table should be empty (all zeroes) at first; your page fault handler will fill it in later.

You are encouraged (not required) to have "shadow process table" for Phase 5, and to store the page table pointer in there, as well as returning it.

- `void phase5_mmu_pageTable_free(int new_pid, USLOSS_PTE *page_table)`

  This function must free the memory allocated for the page table.

  Eventually, you will also need to write code that frees all resources used by the process - such as any frames it has in use, or any pages in the swap disk. However, feel free to **defer this "cleanup" code until late** - you don't have to write it all at once.

(spec continues on the next page)

3 Task: Page Fault Handler

Define an interrupt handler, for the `USLOSS_MMU_INT` interrupt. The first argument to the handler will always be `USLOSS_MMU_INT`; the second, will be an offset in bytes (encoded as a pointer) into the virtual memory region; this tells you **where** the user tried to access memory. So if the offset is zero, the user tried to touch the first byte in `vmRegion`; if it is 4099, they tried to touch the 4th byte in the 2nd page of the virtual memory region.

3.1 Page Fault Causes

Inside the handler, you should call `USLOSS_MmuGetCause()` to find out what caused the page fault. If the cause is `USLOSS_MEM_FAULT`, then this means that the virtual memory didn't have a valid mapping - this means that the `incore` flag, in the `PTE`[2], is zero. On the other hand, if the page table entry was valid (that is, `incore` is one) but the user tried some type of access which was not allowed by the `read` or `write` bits in the PTE, then we will get a cause of `USLOSS_MEM_ACCESS`

You should have **different code** to handle the `ACCESS` and `FAULT` page fault types. For `ACCESS`, I suggest that you write a function; it should be able to finish its work synchronously, and return to the program. **NOTE:** You can make this a stub function at first - in your first version of the system, I recommend that you set `read=1,write=1` for every PTE you create - meaning that `ACCESS` faults will never occur.

For the `FAULT` cause, you should compose a mail message, and send it to the "pager mailbox" (see below). Then you must **block** (since the program can't make forward progress until the pager has finished its work).

3.2 After You Update Your Page Table

Typically, when you change a page table entry in the Real World, the CPU doesn't know about it immediately. You often need to flush something inside the CPU, to make sure that it will re-read the page table entry from memory.

In our simulation, we don't have the abiltity to invalidate only a single entry from our cache; instead, we must ask USLOSS to completely wipe all of the virtual memory mappings, and rebuild them. We do this by calling `USLOSS_SetPageTable()`.

Make it a habit to call this function at the end of your interrupt handler.

(spec continues on the next page)

[2]Page Table Entry. The standard type of this, in our projects, is `USLOSS_PTE`.

4 Task: Pager Daemons

You will fork off a set of Pager Daemons. Read the variable `vm_num_pagerDaemons`, which is defined by the testcase, to tell you how many pager daemon processes you should create.

Pager daemons communicate with the ordinary proceses through a mailbox; I recommend that you only have a single mailbox, which all processes can write to, and which all of the pager daemons read from.

We have multiple pager daemons because we might have, in some testcases, multiple pager processes blocked on disk I/O. Many pager daemons makes it possible to be doing multiple write and/or reads to the swap disk at the time.

4.1 Handling a Fault

When handling a `FAULT` interrupt, the first step is to decide which physical page will be associated with the virtual page. Scan through the list of frames, looking for an available one. Later, you will use a clock algorithm to perform this scan (meaning that you pick up, in a new search, where the last search ended); I encourage you to add it now.

Eventually, you will add more features to your pager daemon, including the ability to evict pages to swap, and to then bring them back in. But you can defer that until later; for your first version of Phase 5, I recommend that you only fill up empty pages.

(spec continues on the next page)

5 Task: Zero Pages

When your code has found a physical page that you want to associate with the virtual page, you now have to fill it up with the proper data. There are two things that you might do: zero a page, or read from swap. At first, I recommend that you ignore the swap disk altogether; mark it as a TODO for the future.

In your earliest version of the code, however, it is easy to zero out a new page. So if you find a physcial page that you plan to use, and you notice that the virtual page doesn't have anything sitting in the swap file, then you can simply call `memset()` to set the physical page to zero.

Note that your code should **not** attempt to share the zero pages across multiple processes. While this is what we would do in the Real World - and it should work just fine in USLOSS - it would make your code much more complex. Instead, each time that a process touches a page for the first time, allocate space for it, all filled with zeroes.

How do you touch the physical memory, if it's not in the `vmRegion` yet? It's easy: just use the pointer to physical memory, which you read (and saved) when you initialize the MMU. While this memory is invisible to your student, from the perspective of your kernel code, it's perfectly ordinary memory. Feel free to read and write it as much as you want.

(spec continues on the next page)

6 Task: `REF` and `DIRTY` Bits

For each physical page in the system, keep track of two flags: `REF` and `DIRTY`. `REF` is set each time that we touch the page as either a read or a write; `DIRTY` is set each time that we write to the page.

These two bits can be easily tracked by controlling the bits in our page table. If, in the PTE, you set both `read=0` and `write=0` (even though you also set `frame`, and set `incore=1`), then you will get an `ACCESS` fault the first time that the user attempts to access the virtual page with any sort of operation. Therefore, if you get an `ACCESS` fault while `read=0`, you should set `read=1` in the PTE, and set the `REF` flag in the physical page's metadatat. Then return from the page fault handler.

If the user was attempting a read, then turning on read permissions is sufficient. If the user was actually attempting a write, we will **immediately** get another page fault; again, it will be a `ACCESS` fault. This time, when you look at the permissions bits, you will see that `read=1`, meaning that the failed op had to be a write. Turn on write access by setting `write=1`, and also set the `DIRTY` flag in the frame.

(spec continues on the next page)

7 Task: Pager Clock, and Using the `REF` Flag

Your pager daemons should use a "clock" variable when searching for available frames to use. A clock makes the search fairer (because all frames get checked regularly) - but it works best if the clock is shared across all of the pagers. So store the clock in some global variable - and be careful to use a lock to protect **all** of the shared variables - including the pager state, the process and virtual memory states, the physical memory state, and the swap disk state.

7.1 First, Clear the `REF` Flag

As the clock moves through the frames, you should check the `REF` flag on each page. If a page has the `REF` flag set, then clear the flag and also turn of `read` (and if necessary `write`) in the PTE.

In this way, the `REF` flag allows us to identify commonly-used pages. Any page that has been used recently will not be stolen way and sent to the swap disk; however, a page that has not been used in a long time will (soon) have its `REF` bit cleared, and it will be evicted on the next pass of the clock.

7.2 Second, Discard the Page

On the other hand, if, as the clock sweeps through the array of frames, it finds a frame which does **not** have `REF` set, then it will immediately choowe that page as the "target" physical page, which will be used for other purposes.

But how do we discard a page? Certainly we don't want to throw away any valuable data - but are there some circumstances where we could discard the page without loss? Yes! If the `DIRTY` bit is not set, then that frame has not been written to since it was most recently initialized. Therefore, we can just discard it without loss.

If the page that we are discarding was filled with zeroes (meaning that it was **not** read from swap), then we can return to the previous state; we can pretend that it was never in memory at all, since it never changed.

On the other hand, if the frame is something that we read up from disk (but we haven't changed it since we read it), then we can also discard it - but we "discard" it back into the swap file! That is, we are not required to actually write anything to disk, but we can simply say, "That block, which is on disk, is the data we want."

(spec continues on the next page)

8 Task: Write Pages to Swap

Eventually, you will need to upgrade your pager daemon so that it has the ability to write pages to swap when you find that you need to discard a `DIRTY` physical page.

When writing pages to swap, you may use the same disk functions as you implemented in Phase 4 - but of course, the functions that you used required that the caller be in user mode. To get around this problem, I have updated the Phase 4 library to add three new functions, which were not part of the Phase 4 requirement. `kernDiskSize(), kernDiskRead(), kernDiskWrite()` have the same arguments, return values, and semantics as the old user-mode functions - except that they can run in the kernel.

When writing pages to swap, you can address the data using the "physical memory" pointer that your code knows about. **Make sure not to use the virtual memory pointer,** since this points to different pages in different processes!

Remember that the functions that I've provided are **synchronous** - meaning that they will block the pager daemon until they complete. Of course, you just wrote Phase 4, and so you know that down deep, they are using asynchronous operations and interrupts - but the user interface is synchronous.

8.1 Where Did It Go?

Until this point in the project, you probably didn't need any virtual page metadata, beyond the fields of each PTE. That will change now. When a virtual page has been swapped to disk, there cannot possibly be any PTE that stores its data - and yet, you need to be able to remember where the page resides on disk. So add the metadata table now, if you haven't done so already.

8.2 "Shooting Down" Remote PTEs

Pause for a moment and think: which process is performing the write, when we send a page out to swap? It's a pager daemon. And was this triggered by the process that owns the page? Probably not! In fact, the **ordinary** case is that some process runs along and hits a page fault - and the page that it kicks out of memory belong to **another process.**

But we need to make sure that, when that other process runs, we don't use the old, no-longer-valid PTEs! For this reason, you should do a "shoot down" of the PTE in the owning process. In a "shoot down," you reach into the page table of a different process, and you disable entries.

In a single-CPU system, PTE shoot down is easy; you know that the other process isn't running. In a multi-CPU system, it's a lot more complex - but thankfully, it's not something that you need to worry about in this simulation.

8.3 Transient States

Writing a page to the swap disk takes time. While this is happening, you absolutely **must not** allow the other process to run and write to the page. Theoretically, you could allow it to read - there's nothing really wrong with that - but for simplicity, my implementation doesn't even allow that.

But you are going to have to be careful with page faults! What happens if the other process runs, and tries to touch the page that we are writing out to swap? Might you accidentally read in the wrong value?

I would recommend that, at first, you simply **detect** this condition: if you are writing a given page to swap (or reading it up from swap), if you hit a page fault on the same virtual page, crash the program. If you then find that you're hitting it in a testcase, you can add a fix for it, later.

(spec continues on the next page)

9 Task: Read Back From Swap

When you have a `FAULT` on a virtual page, and you select a physical page to store it, you still have to figure out what will go into that physical page. Eventually, you will have to support the ability to read pages up from the swap disk.

For the most part, this code will be a mirror of what you did in swap-out; you will need to perform a (blocking) call to `kernDiskRead()`. While you are in the middle of doing a swap-in, make sure that your physical page is marked as busy (so that no-one will try to use it). And once the page is up in physical memory, set up the PTE and the physical memory flags.

Remember that every time we read (or zero out) a physical page, we can start over with the `REF` and `DIRTY` bits. That is, the new page has not been touched (yet), and it has not been modified. So I recommend that you set `read=0` and `write=0` in the PTE, and that you **clear** the `REF` and `DIRTY` bits on the physical page.[3]

10 Task: `vmStats`

You must declare a variable named `vmStats`; see `phase5.h` for information about it. Some of its fields are basic information about the system; you can set them once, during init, and never change them. One of them is something that you have to read from disk (the disk size). Several of them are counters, telling us how many times various events have occurred.

You should be able to pass most testcases without worrying too much about the `vmStats` variable; fill it in, as you work through the testcases, when you run across problems.

11 Turning in Your Solution

You must turn in your code using GradeScope. While the autograder cannot actually assign you a grade (because we don't expect you to match the "correct" output with perfect accuracy), it will allow you to compare your output, to the expected output. Use this report to see if you are missing features, crashing when you should be running, etc.

[3]Since you're in a page fault already, you might notice that you could go ahead, and set `read=1` and the `REF` flag, without waiting for a future page fault to do it. That's OK, too. Your choice.