CMSC 412 Project #4

Virtual Memory

Due Monday April 11, 2016, at 5:00pm

Introduction

The purpose of this project is to add paging to your GeekOS kernel. This will require many small, but difficult, changes. More than any previous project, it will be important to implement one thing, test it, and then move to the next one. A successful implementation of earlier projects is **not required** for this project. You should implement this code directly on a fresh checkout of the GeekOS system.

There are two parts to this project. First you will modify the operating system to use virtual address translation. Second, you will add memory mapped files to the operating system.

This project will be done in two person teams. You will be assigned a project partner by Dr. Hollingsworth. You should each turn in via submit a copy of your joint project.

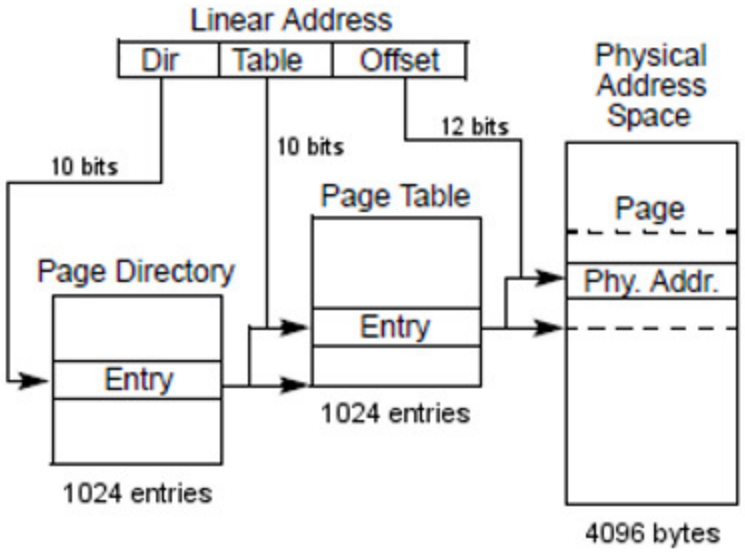
Background

In this project you will add paging to the GeekOS kernel, combined with the segmentation that is already present. For detailed information on paging, segmentation, and how they are combined on the Intel architecture, please refer to sections 8.4-8.7 in the text (pps. 288-309), and the Intel documentation specified above.

In a system that combines paging and segmentation, there are three kinds of addresses: logical addresses, linear addresses, and physical addresses. *Logical addresses* are those issued by a process. Logical addresses are mapped, via segmentation, to *linear addresses* by merely adding the base address of the relevant process segment to the logical address. A linear address is mapped to a *physical address*, which will be used by the processor to actually read from main memory, by using page tables set up and maintained by the operating system. Paging is similar to segmentation because it allows each process to run in its own memory space. However, paging allows a much finer granularity of control by specifiying per-page mappings rather than a constant value offset. This will allow us to implement two useful features: demand paging, which allows a user program's stack to grow dynamically, and paging to disk, which allows memory to be "paged out" to disk, enabling the system to use more physical memory than it actually has.

In GeekOS, your paging system will use a *page directory* and *page tables* which form what as known as a *two-level page table scheme* (see Section 8.5.1, and notably Fig. 8.14, in the text). Each memory address in the x86 system is 32 bits long. To translate this linear address to a physical one, the processor will use the current page directory (this is analagous to finding an LDT for a process):

1. It will use the most significant 10 bits of the linear address to index into the page directory to obtain a page directory entry. 10 bits allows you to write numbers between 0 and 1023, so therefore the page directory must have 1024 entries.
2. The page directory entry is then used to find the appropriate page table. The next 10 bits are used to index into the page table to obtain a page table entry. As above, the page table must have 1024 entries.
3. The page table entry is used to find the base address of the physical memory page. The last 12 bits of the linear memory address are used to index into the physical page. 12 bits allows you to write numbers between 0 and 4095, so therefore a page must be 4096 bytes in size.



Each page directory and page table are themselves pages in the system. Each page directory entry and page table entry is 4 bytes, and as explained above, each table contains 1024 entries. Therefore, each page directory and page table is 4096 bytes, which allows you to use pages of memory to represent the directories and tables. Each page directory contains a pointer to a page table which in turn contains a pointer to the physical memory.

Part I

The first step is to modify your project to use page tables and segmentation rather than just segments to provide memory protection.

Kernel Memory Mapping

The kernel currently starts running without paging enabled, and the segmentation hardware maps logical addresses directly to physical addresses. Adding paging introduces a level in between: a logical address is mapped by segmentation to a linear address, which paging then maps to a physical address. As a first step, we will introduce a page table that maps all linear addresses to themselves; that is each linear address X is mapped to physical address X. In Part II, this will become our page table for kernel-only threads.

To set up page tables, you will need to allocate a page directory (via `Alloc_Page`) and then allocate page tables for the entire region that will be mapped into this memory context. You will do this in the `Init_VM` in `paging.c`. You will need to fill out the appropriate fields in the page tables and page directories. The definition of paging tables and directories are found in `paging.h` (structs `pte_t` and `pde_t`). Finally, as mentioned earlier, GeekOS does not use paging by default, so to enable it, you will need to call the routine `Enable_Paging` which is already defined for you in `lowlevel.asm`. It takes the base address of your page directory as a parameter.

The final step of this function is to add a handler for page faults. Currently, a page fault can occur only when a user program attempts to access an invalid address. Therefore, we have provided a default handler, `Page_Fault_Handler` in `paging.c`, to terminate a user program that does this. You should register it for the page fault interrupt, interrupt 14, by calling `Install_Interrupt_Handler`. You should then add a call the `Init_VM` function from your `main.c` (after `Init_Interrupts`).

You should be able to complete this part and test it by temporarily giving user mode access to these pages by setting the `flags` field in the page table entries to include `VM_USER`. This allows user programs complete access to the pages referenced by the table. Refer to item 1 in the grading criteria to understand how to test what you should have running at this point. You can then submit this an intermediate submission, and you should pass the checkpaging public test.

User Memory Mapping

We will use the page directory and page tables you set up in Part I for all kernel-only threads, and will now add a page directory for each user process. As such, you should change the `flags` fields in this directory to *not* include `VM_USER`.

The page directory for user processes will contain entries mapping user logical memory to linear memory, *but will also contain entries to address the kernel memory*. This is not so user processes can access kernel memory directly (we will set the access flags of the memory to prevent them from doing so), but rather so that when an interrupt occurs, the page table does not need to be changed for the kernel to access its own memory; it will simply use the page table of the user process that was running when the interrupt occurred.

The memory layout for a user process is shown below, where the addresses on the left are linear addresses (which are mapped to physical addresses by the paging system). User processes still refer to logical addresses, which are mapped by the segmentation system to the linear addresses shown here. These addresses will be important in your paging implementation, and thus have constants defined in `paging.h` to represent them.

0x0000 0000	Kernel Memory	Start of kernel memory
		(map all physical memory here)
	<gap>	
0x8000 0000	User Memory	User address space begins here (unmapped page)
0x8000 1000	User Memory	Text segment usually loaded here (segment->startAddress)
	<gap>	
0xFFFF E000	User Memory	Initial stack at top of this page
0xFFFF F000	User Memory	Args in this page
0xFFFF FFFF		Memory space ends here

The next step is to modify your user processes to use pages in the user range of memory. To help you do this, there is a new file called `uservm.c` that will replace your `userseg.c` from previous projects. You should start by taking the implementations from `userseg.c` and copying them to `uservm.c` and then modifying them for paging. There is a line in the `Makefile.common` in the build directory which specifies whether it should use `userseg.c` or `uservm.c`. You can switch between them by modifying `USER_IMP_C` variable.

Setting up paging for user processes occurs in `Load_User_Program`, and takes two steps. First, you need to allocate a page directory for the process. You should copy all of the entries from the kernel page directory for the physical pages at the bottom of the address space.

Next, allocate page table entries for the user process's text, data, and stack regions. Each of these regions will consist of some number of pages allocated by the routine `Alloc_Pageable_Page`. This routine differs from `Alloc_Page` in that the allocated page it returns will have a special flag `PAGE_PAGEABLE` set in the `flags` field of its entry in the `struct Page` data structure (see `mem.h`). This marks the page as eligible for being stolen and paged out to disk by the kernel when a page of memory is needed elsewhere, but no free pages are available. All pages (but not page directories and page tables) for a user space process should be allocated using this routine.

Contrast this with current implementation: for segmentation, one big chunk of memory was allocated for the entire user process. Paging allows per-page mappings for user memory so that each page of the user process is now allocated and mapped individually, and so it need not be contiguous.

Although there is a much better way, if you're stuck, do this as follows: calculate the size of the text and data segment of the process as is done now, without including the size of the stack (do round to `PAGE_SIZE`, though), and `Malloc` it. Load the program into this memory, as now. Copy the image, page by page, into the newly allocated pages. Allocate two pages of memory at the end of the virtual address range (i.e., the last two entries in the last page table, as shown in the figure above). One is for the arguments, the other one is for stack. Make sure the `flags` bits in both the page directory and page table entries allow user mode access (contain the `VM_USER` flag). Unmap the first page to support null pointer checking.

Finally, you will need to tweak some aspects of the current segmentation implementation so that it works with paging. The base linear address for the user mode process (that is, the base address for the code and data segments set in `Create_User_Context`) should be 0x80000000 (`USER_VM_START` in `paging.h`), and the limit should be (`USER_VM_END` - `USER_VM_START`) / `PAGE_SIZE` , rounded up. This will allow the user process to think that its logical address 0 is the linear address at 2GB and will simplify your kernel compared to traditional paged systems. You will also need to add code to switch the PDBR (cr3) register as part of a context switch. For this, in `Switch_To_Address_Space` you should add a call to `Set_PDBR` (provided for you in `lowlevel.asm`), after you load the LDT. You will use the `pageDir` field in the `User_Context` structure that will store the address of the process's page directory.

The APIC and IO-APIC pages will need to be identity mapped into the address space of all processes at locations 0xFEE00000 and 0xFEC00000. These pages should be mapped READ/WRITE but only from ring 0 not ring 3.

At this point, you should test your memory mapping by running QEMU. If you are able to load and run shell, you have completed this correctly.

Part II

The next step is to add support for memory mapped files. Memory mapping files allows processes to treat a file as though it was memory. Thus individual bytes can be read or written. Also, you will need to support mapping in files that are larger than the amount of physical memory. Changes made to memory mapped files will also change the contents of the files (if the files are mapped writeable).

Memory Mapping System Calls

There are two additional system calls to support memory mapped files. They are:

```
void *Mmap(void *addr, unsigned int length, int prot, int flags, int fd)
```

This system call will map the file indicated by the passed file descriptor (fd) into the calling process's address space at the virtual address specified by addr (addr must be aligned on a 4KB boundary). It will map length bytes of memory starting from the 0^{th} byte of the file. If length is larger than the size of the current file, the system call will return NULL. The prot parameter determines the memory protection of the mapped file. PROT_READ means that file may only be read. PROT_WRITE means the file may only written. Passing PROT_READ| PROT_WRITE will allow the mapped file to be both read and written. If the passed mapping region (addr to addr+length) overlaps any other page mappings or the data, text, or stack of the user process mmap should return NULL. The flags mode is one of MAP_SHARED or MAP_PRIVATE. Only MAP_SHARED is used in this project.

Upon successful mapping, Mmap returns addr and any subsequent read to the map memory will return the appropriate values from the file. If the protection includes PROT_WRITE, writing to the mapped memory will update the contents of the file. Writing to a mapped region without PROT_WRITE, should result in a segmentation violation and the program being terminated.

If an attempt to read a file mapped page of memory is made and it is not currently loaded, the user program will take a page fault and your kernel will need to locate a page frame and read the corresponding page from the file.

```
int Munmap(void *addr)
```

This system call will unmap the mapped file that starts at addr. If addr is not the first address of a mapped file, the system call will return EINVALID. It should write back any dirty pages to the file, free the associated page frames, and clear the appropriate page table entries.

You will need to register the page fault trap handler and enable paging on each core seperately. The function Init_Secondary_VM is provided for this purpose and will be called at the right time in the boot sequence. These should be enabled for core #0 in the routine Init_VM.

Paging to Disk

While paging is useful for efficiently managing memory, the processes that can be active are still limited by the amount of physical memory. To remedy this, you will implement *paging to disk* as part of GeekOS, in which memory pages can be temporarily stored on disk so that the freed physical memory can be used by another process. To do this, the OS will need to pick a page to evict from memory and write it to the *paging file*, stored on disk or the file of a memory mapped file. You should implement a version of pseudo-LRU algorithm (see section 9.4.5 in text, pp. 336). Use the `accessed` bit in the page tables to keep track of how frequently pages are accessed. To do this, maintain a global (static) clock value that points to a page in memory (a `struct Page` in the array of pages). When a page needs to be paged out, increment that clock, and clear the accessed bit or claim the page if the accessed bit was already clear.

The paging file consists of consecutive disk blocks of `SECTOR_SIZE` bytes. Calling the routine `Get_Paging_Device` in `vfs.h` will return a `Paging_Device` structure containing the first disk block number

of the paging file and the number of disk blocks in the paging file. Each page will consume eight consecutive disk blocks (`PAGE_SIZE/SECTOR_SIZE`). To read and write the paging file, use the functions `Block_Read` and `Block_Write` provided. These functions write `SECTOR_SIZE` bytes at a time. How you manage your paging file is up to you. You may want to write a `Init_Pagefile` function in `paging.c` and call it from `main.c`

The code to page out a page is implemented for you in `Alloc_Pageable_Page` in `mem.c`, and works as follows:

- Find a page to page out using `Find_Page_To_Page_Out` which you will implement in `mem.c`.
- Find space on the paging file using `Find_Space_On_Paging_File` which you will implement in `paging.c`
- Write the page to the paging file using `Write_To_Paging_File` which you will implement in `paging.c`
- Update the page table entry for the page to clear the `present` bit.
- Update the `pageBaseAddr` in the page table entry to be the first disk block that contains the page.
- Update the `kernelInfo` bits (3 bits holding a number from 0-7) in the page table entry to be `KINFO_PAGE_ON_DISK` (used to indicate that the page is on disk rather than not valid).
- Flush the TLB using `Flush_TLB` from `lowlevel.asm`

Eventually, the page that was put on disk will be needed by some process again. At this point you will have to read it back off disk into memory (possibly while paging out another page to fit it into memory). Since the page that is paged out has its `present` bit set to 0 in the page table, an access to it will cause a page fault. Your page fault handler should then realize that this page is actually stored on disk and bring it back from disk (the `kernelInfo` field in the page table entry). When you bring a page in off disk, you may free the disk space used by the page. This will simplify your paging system, but will require that when a page is removed from memory it must always be written to the backing store. You will rely on the information stored when a page is paged out (such as `pageBaseAddr`) to find it on disk and page it back in.

The following table summarizes the actions of your page fault handler.

Cause	**Indication**	**Action**
Fault for mapped file page not in memory	File mapping table for process indicates this is a valid mapping	Read page from file and continue.
Fault for paged out page	Bits in page table indicate page is on disk	Read page from paging device (sector indicated in PTE) and continue.
Fault for invalid address	None of the other conditions apply	Terminate user process

Copying Data Between Kernel and User Memory

Because the GeekOS kernel is preemptible and user memory pages can be stolen at any time, some subtle issues arise when copying data between the kernel and user memory spaces. Specifically, the kernel must never read or write data on a user memory page if that page has the `PAGE_PAGEABLE` bit set at any time that a thread switch could occur. The reason is simple; if a thread switch did occur, another process could run and steal the page. When control returns to the original thread, it would be reading or writing the wrong data, causing serious memory corruption.

There are two general approaches to dealing with this problem. One is that interrupts (and thus preemption) should be disabled while touching user memory. This approach is not a complete solution, because it is not legal to do I/O (i.e., `Block_Read` and `Block_Write`) while interrupts are disabled. The second approach is to use

page locking. Before touching a user memory page, the kernel will atomically clear the `PAGE_PAGEABLE` flag for the page; this is referred to as locking the page. Once a page is locked, the kernel can then freely modify the page, safe in the knowledge that the page will not be stolen by another process. When it is done reading or writing the page, it can unlock the page by clearing the `PAGE_PAGEABLE` flag. Note that page flags should only be modified while interrupts are disabled.

Process Termination

As part of process termination, you will need to free the memory associated with a process. This includes freeing the pages used by the process, freeing the page tables and page directories. In addition, you will need to make sure any dirty (written to) pages from mapped files are saved back to the file system. You should modify your `Destroy_User_Context` function in `uservm.c` to do this.

Notes

In this project you do **not** need to handle the case of the same file being mapped by two processes at once.

If you put files into the directory ..../build/user (with a .txt suffix) they will end up on the C drive when geekOS boots.