Chapter 9

Project 4: Virtual Memory

9.1 Introduction

The purpose of this project is to add paging to the GeekOS kernel. This will require many small, but difficult changes to your project. More than any previous project, it will be important to implement one thing, test it and then move to the next one.

9.2 Changing the Project to Use Page Tables

The first step is to modify your project to use page tables and segmentation rather than just segments to provide memory protection. To enable using page tables, every region of memory your access (both kernel and data segment) must have an entry in a page table. The way this will work is that there will be a single page table for all kernel only threads, and a page table for each user process. In addition, the page tables for user mode processes will also contain entries to address the kernel mode memory. The memory layout for this is shown in Figure 9.1.

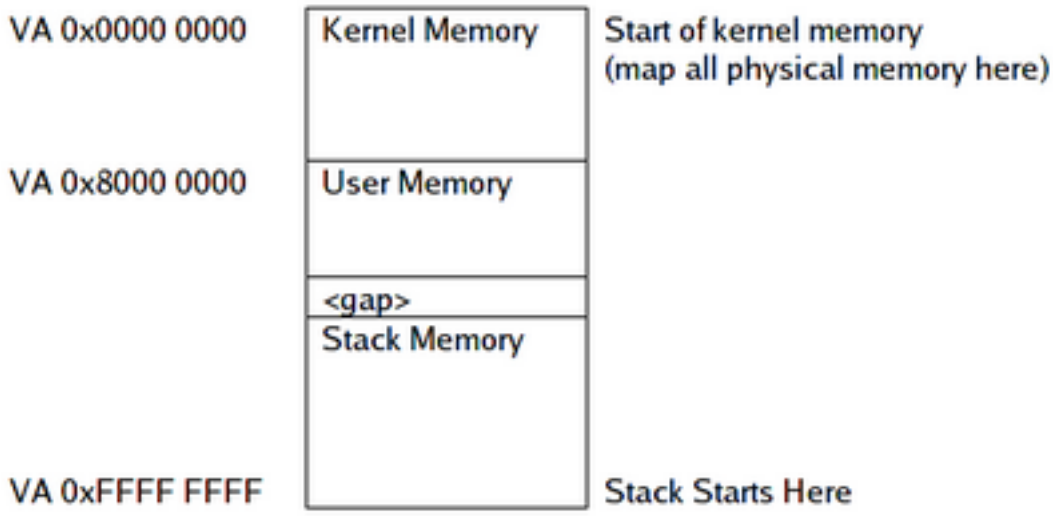

Figure 9.1: Virtual memory layout in GeekOS.

The kernel memory should be a one to one mapping of all of the physical memory in the processor (this limits the physical memory of the processor to 2GB, but this is not a critical limit for this project). The page table entries for this memory should be marked so that this memory is only accessible from kernel mode

(i.e. the `userMode` bit in the page directory and page table should be 0). To make this change, you should start by creating a page directory and page table entries for the kernel threads by writing a function that initializes the page tables and enables paging mode in the processor. You will do this in the `Init_VM()` function in `src/geekos/paging.c`.

To set up page tables, you will need to allocate a page directory (using `Alloc_Page()`) and then allocate page tables for the entire region that will be mapped into this memory context. You will need to fill out the appropriate fields in the page tables and page directories, which are represented by the `pte_t` and `pde_t` datatypes defined in `<geekos/paging.h>`. Finally, to enable paging for the first time, you will need to call an assembly routine, `Enable_Paging()`, that will take the base address of your page directory as a parameter and then load the passed page directory address into register `cr3`, and then set the paging bit in `cr0` (the MSB, bit 31).

The next step is to modify your user processes to all use pages in the user region. This is a two step process. First, you need to allocate a page directory for this user space. You should copy all of the mappings from the kernel mode page directory for those memory regions in the low range of memory. Next you need to allocate page table entries for the user processes text and data regions. Do not allocate extra space for the stack here. Finally, you should allocate space for one page of stack memory at the end of the virtual address range (i.e. the last entry in the last page table). For the user space page mappings, make sure to enable the `userMode` bits in both the page directory and page table entries.

You will also need to change some aspects of how the code from Project 1 sets things up. You should change the code and data segments for user processes so that the base address is be 0x80000000, and the limit is 0x80000000. This will allow the user space process to think that its virtual location 0 is the 2GB point in the page layout and will greatly simplify your kernel compared to traditional paged systems. You will also need to add code to `Switch_To_Address_Space()` to switch the PTBR register (`cr3`) as part of a context switch; where you load the LDT of the user context, you should also load the address of the page directory for the process, which is the `pageDir` field in the `User_Context` structure. [1]

When you are allocating pages of memory to use as part of a user address space, you should use a new function, `Alloc_Pageable_Page()` (prototype in `<geekos/mem.h>`. The primary difference is that any page allocated by this routine should have a special flag `PAGE_PAGEABLE` set in the flags field of its entry in the corresponding `Page` data structure. Having this flag set marks the page as being eligible to be stolen and paged out to disk by the kernel when a page of memory is needed elsewhere, but no free pages are available. Note that you should *not* allocate page tables or page directories using this function.

9.3 Handling Page Faults

One of the key features of using paging is to have the operating system handle page faults. To do this you will need to write a page fault interrupt handler. The first thing the page fault handler will need to do is to determine the address of the page fault; you can find out this address by calling the `Get_Page_Fault_Address()` function (prototype in `<geekos/paging.h>`. Also, the `errorCode` field of the `Interrupt_State` data structure passed to the page fault interrupt handler contains information about the faulting access. This information is defined in the `faultcode_t` data type defined in `<geekos/paging.h>`. Once the fault address and fault code have been obtained, the page fault handler will need to determine an appropriate action to take. Possible reasons for a page fault, and the action to take are shown in Figure 9.2.

9.4 Paging Out Pages

At some point, your operating system will run out of page frames to assign to processes. In this case, you will need to pick a page to evict from memory and write it to the backing store (paging file). You should

[1] You may choose to allocate the user code and data segment descriptors in the GDT (Global Descriptor Table) rather than having a separate LDT for each process. If you decide to use this approach, then the `User_Context` will not need to contain an LDT or selector fields. Instead, you should define user mode code and data segments in the GDT using the `Allocate_Segment_Descriptor()` function, *before any user process is created*, and create selectors for these segments to use for the code and data segment registers for each newly created process. Each user process can use the same user code and data segments; because each process uses a separate virtual address space, there is no way that a process can access another process's memory.

Cause	Indication	Action
Stack growing to new page	Fault is within one page of the current stack limit	Allocate a new page 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

Figure 9.2: Actions to be taken when a page fault occurs.

implement a version of pseudo-LRU. Use the reference bit in the page tables to keep track of how frequently pages are accessed. To do this, add a `clock` field to the `Page` structure in `<geekos/mem.h>`. You should update the clock on every page fault.

You will also need to manage the use of the paging file. The paging file consists of a group of consecutive 512 bytes disk blocks. Calling the routine `Get_Paging_Device()` (prototype in `<geekos/vfs.h>`) will return a `Paging_Device()` object; this consists of the block device the paging file is on, the start sector (disk block number), and the number of sectors (disk blocks) in the paging file. Each page will consume 8 consecutive disk blocks. To read/write the paging device, use the functions `Block_Read()` and `Block_Write()`.

9.5 Page Ins

When a page is paged out to disk, the kernel stores the index returned by `Find_Space_On_Paging_File()` in the `pageBaseAddr` field of the page table entry (`pte_t`), and also stores the value `KINFO_PAGE_ON_DISK` in the entry's `kernelInfo` field. In your page fault handler, when you find a non-present page that is marked as being on disk, you can use the value stored in `pageBaseAddr` to find the data for the page in the paging file.

9.6 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.

9.7 Implementation

In order to implementing virtual memory and paging, you will need to implement several functions.

9.7.1 Functions in `src/geekos/paging.c`

- `Init_VM()` (defined in) will set up the initial kernel page directory and page tables, and install a page fault handler function.
- `Init_Paging()` (defined in `src/geekos/paging.c`) should initialize any data structures you need to manage the paging file. As mentioned earlier, the `Get_Paging_Device()` function specifies what device the paging file is located on, and the range of disk blocks it occupies.
- `Find_Space_On_Paging_File()` should find a free page-sized chunk of disk space in the paging file. It should return an index identifying the chunk, or -1 if no space is available in the paging file.
- `Free_Space_On_Paging_File()` will free a chunk of space in the paging file previously allocated by `Find_Space_On_Paging_File()`.
- `Write_To_Paging_File()` writes the data stored in a page of memory to the paging file.
- `Read_From_Paging_File()` reads the data for a page stored in the paging file into memory.

9.7.2 Functions in `src/geekos/uservm.c`

- `Destroy_User_Context()` frees all of the memory and other resources (semaphores, files) used by a process.
- `Load_User_Program()` loads an executable file into memory, creating a complete, ready-to-execute user address space.
- `Copy_From_User()` copies data from a user buffer into a kernel buffer.
- `Copy_To_User()` copies data from a kernel buffer into a user buffer.
- `Switch_To_Address_Space()` switches to a user address space by loading its page directory and (if necessary) its LDT.

9.8 Extra Credit

The implementation of virtual memory in GeekOS is a very simple one. There are many ways that it can be extended and improved.

9.8.1 Improving `Find_Page_To_Page_Out()`

When a page of pageable memory is required and no pages are available, the kernel uses the `Find_Page_To_Page_Out()` function (in `src/geekos/mem.c`) to select a page to page out. While interrupts are disabled (meaning no other threads or interrupt handlers can run), this function traverses the array of all `Page` data structures, in order to find the one with the oldest clock field.

How can you make this function work more efficiently?