PROJECT 4B: PAGING - USER

Minimum Requirements: 7 public tests

This should a short project but get familiar with paging so you are ready for project 4b.
TEST DISTRIBUTION

Public tests – 9 tests | 42 points
Release tests – 0 tests | 0 points
Secret tests – 12 tests | 50 points
INTRO

• In 4A:
  • We have already set up an identity mapping for kernel space.
  • But when setting up a user process, it still uses a kernel Malloc to malloc the user memory. (That is why VM_USER is set for kernel space in 4A, you need to unset it in 4B)

• In 4B:
  • Make each user process have its own linear address space, which means each user process is going to have its own page directory and page tables.
  • Implement demanding paging and paging to disk.
VIRTUAL MEMORY

• Address translation: Virtual Addr -> (segmentation address translation) -> Linear Addr -> (Paging address translation) -> Physical Addr.

• Memory is divided into page frames. To run any program, we need to map the address of this page frames to the address space of that user process, and load the corresponding binary into the page frames.

• When the OS runs out of page frames, it can swap out some pages and store them in secondary storage. When some missing pages are accessed again, it can swap them in.

• Because of swapping, user space size is no longer limited by the physical memory size. Each user process can view its user address space from 0x0000 0000 to 0x6fff ffff, which translates into 0x8000 0000 to 0xefff ffff in kernel address. (Note: the upper limit can be different and it depends on your implementation)
LINEAR ADDRESS SPACE LAYOUT – BEFORE PAGING

- When GeekOS only works with segmentation (Linear address is the same as physical address)
LINEAR ADDRESS SPACE LAYOUT – AFTER PAGING

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF FFFF</td>
<td></td>
<td>Memory space ends</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>Kernel</td>
<td>Memory space start</td>
</tr>
<tr>
<td>0x0F00 0000</td>
<td>User</td>
<td>Dynamically growing stack</td>
</tr>
<tr>
<td>0x0F000 0000</td>
<td>User</td>
<td>StackPointer</td>
</tr>
<tr>
<td>0x8000 0000</td>
<td>User</td>
<td>User memory start, first page is unmapped</td>
</tr>
<tr>
<td>0x8000 1000</td>
<td>User</td>
<td>Heap, data, text</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>Kernel</td>
<td>Kernel binary and gap</td>
</tr>
</tbody>
</table>

The argument block should be placed so that it ends just before 0x70000000; the initial stack pointer should be the address of the argument block.

Notes:
- 0x7000 0000 is user address which corresponds to 0xF000 0000 in kernel space.
- You don’t actually need to put them in the exact position the spec mentions.
- The point here is:
  a) Argument block is at the bottom of the stack.
  b) When the stack grows, it will not collide with APIC.
  c) The stack can expand large enough to pass the tests.
- But it is recommended to put it this way.
FILES TO MODIFY

- Paging.c
  - Map kernel space (4a)
  - Handle page fault (allocating a stack page or swapping)
  - Initialize page related data structures.
  - Manage paging file
- uservm.c
  - Replace userseg.c with uservm.c (replace segmentation with virtual memory).
  - All the functions implemented in userseg.c need to be re-implemented in uservm.c.
- Mem.c
  - Managing page frames.
- Corresponding headers
- Main.c
- Makefile.common
CHANGE MAKEFILE: MAKEFILE.COMMON

- There is a line which specifies which of userseg.c or uservm.c should be used. You can switch between them by USER_IMP_C.
INTRO

• Allocate page frames for the user process, fill in the two-level page tables for user processes.
  Load user programs into the page frames for the user processes.

• Make sure you understand the function in userseg.c before implementing its counterpart in uservm.c
USER_TO KERNEL()

• No need to change this function, copy over from userseg. But you need to understand why it works.

• Use offset 0x8000 0000 (we will set userContext->memory to 0x8000 0000 in create_user_context).
In addition to the original function, which sets segment registers, invoke Set_PDBR to use the page directory of the current process (userContext->pageDir, which will be initialized in create_user_context)
CREATE_USER_CONTEXT()

- Initialize the fields of User_Context considering paging.
- Most parts should be the same as in userseg.
- Instead of Malloc, context->memory should now point to the start of user address in linear address space. (spec: Set the beginning of the segments for user processes to be 0x80000000.) The size should also be changed to the size of the entire user address space in linear address space.
- Add a page directory for each user process. The page directory for a user process will contain entries that maps user linear memory to physical memory, but it will also contain entries to address the kernel memory. (The first half of the user page directory should be the same as the kernel page directory, as well as the entry for APIC and IOAPIC. You don’t need to worry about the user space part here because you will map this part in Load_User_Program)
CREATE_USER_CONTEXT()

- Notes about LDT: since different user processes have the same value for context->memory, the LDTs are initialized the same way for different user processes (check the original create_user_context if you don’t understand). You can now choose to have one global LDT for all user processes, or you can choose to preserve the one ldt-per-process approach, but now they are identical. (Refer to the LDT section in the spec.)
LOAD_USER_PROGRAM()

• Load the user program binary into memory and establish an address mapping.

• Go through the Load_User_Program in userseg.c, pay attention to how each segment is loaded into the main memory and how the argument block is formatted.

• Fill in the 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. (Hint: we need to use Alloc_Pageable_Page in order to fill in the baseAddress in the second level page table here, which is different from 4a where we can calculate the base address directly).
  • 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.

• Map the memory for the argument block.

• Map 2 more pages beyond argument block for the initial stack region,
LOAD_USER_PROGRAM()

• Load the binary segments into the memory.
  • Option1: You can copy the image page by page into the physical page frames you got from last step. (NOT recommended)
  • You can follow the instruction for Copy_To_User except that the PDBR should be set to userContext->pageDir here.
• Format the argument block, it should be copied to the address as the picture of linear address space layout indicates. (Hint: you might want to look at the definition of Format_Argument_Block.)
• When initializing userContext->argBlockAddr, userContext->stackPointerAddr, you need to notice that they refer to user addresses.
• If (segment->protFlags & VM_WRITE) == 0, do not give writing permission to the pages for that segment, for protection reasons.
• If you changed PDBR in this function, reset it to the original one.
• Round_Up_To_Page(), Round_Down_To_Page() in mem.h might be useful.
LOAD_USER_PROGRAM()

Hint: a helper function that maps one page at a time is useful
In addition to the original implementation in userseg.c, you need to free all the things you allocated (e.g. all the user pages (for now) and pages on the paging file (after implementing swapping)).

Useful functions: Free_Page
COPY_FROM_USER() & COPY_TO_USER()

- Set PDBR to CURRENT_THREAD->userContext->pageDir, so that cpu can access the user address space.
- Write a helper function to lock all the pages you are going to access (hint: clear the PAGE_PAGEABLE bit of struct page::flags), must be done with interrupts disabled or using synchronization tools to prevent the pages to be swapped out. If the page is not in memory, you need to bring it back into the memory and lock it. (You don’t need to worry about this part now, you can complete this part after finishing paging to file and page replacement.)
- Use memcpy to copy from source to destination, be careful about kernel address and user address.
- Reset the PAGE_PAGEABLE bit. (hint: useful function Get_Page())
TEST YOUR CODE!!!

• At this point, you should be able to boot up and run shell in your GeekOS.
• Make a submission, you should be able to pass all public tests except public test 7 – rec1000.
DEMAND PAGING

- A nice benefit of paging is that it is straightforward to dynamically allocate physical memory to processes. For example, you can allow the process's stack to grow beyond its initial allocation.
- To implement stack growth, you need to modify the default page fault handler from 4a. The fault handler reads register cr2 to determine the faulting address. It also prints the errorCode defined in InterruptState and the fault defined in the struct faultcode_t in paging.h.
• Implement demand paging based on the original function.
• Use state->errorCode to determine the type of page fault.
• If the address is within one page of the **current stack limit**, allocate a new physical page frame, map the appropriate virtual page (which expands the stack) to this physical page frame; and return normally from the handler.
• **Current stack limit**: you need to keep track of this.
**PAGE_FAULT_HANDLER() – PAGE ON DISK**

- Implement this after paging to disk is implemented.
- If it is a page that is swapped out to disk (pte->kernellInfo == KINFO_PAGE_ON_DISK), you need to bring it back to disk.
- Function might be useful: GET_PDBR, Get_Page, Free_Space_On_Paging_File
- Enable interrupts before Alloc_Pageable_Page/Read_From_Paging_File.
  - If it is disabled, it will not be able to do IO, and cannot swap in/out pages if there are no more free page frames.
- If you want to swap in a page, first you need to Alloc_Pageable_Page, before Read_From_Paging_File, you need to set the PAGE_PAGEABLE bit to 0, because this page could potentially get swapped out while you are writing stuff into this page. The last step is to Free_Space_On_Paging_File and fill in the page table entries for the page you brought in.
PAGE_FAULT_HANDLER() – PAGE ON DISK

- Allocate a new page (Alloc_Pageable_Page)
- Read the contents of the indicated block of space in the paging file into the allocated page (Read_From_Paging_File)
- Update the relevant page table entry
- Free the page-sized chunk of disk space in the paging file (Free_Space_On_Paging_File)
# PAGE_FAULT_HANDLER() – SUMMARY

<table>
<thead>
<tr>
<th>Cause</th>
<th>Indication</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack growing to new page</td>
<td>Fault is within one page of the current stack limit</td>
<td>Allocate a new page can continue.</td>
</tr>
<tr>
<td>Fault for page on disk</td>
<td>Bits in the page table indicate page is on disk</td>
<td>Read page from paging device and continue.</td>
</tr>
<tr>
<td>Fault for invalid address</td>
<td>None of the above is true</td>
<td>Terminate this user process.</td>
</tr>
</tbody>
</table>
PAGING TO DISK & PAGE REPLACEMENT

Functions to implement:

- In paging.c
  - Init_Paging
  - Find_Space_On_Paging_File
  - Free_Space_On_Paging_File
  - Write_To_Paging_File
  - Read_From_Paging_File

- In mem.c
  - Alloc_Or_Reclaim_Page
  - Find_Page_To_Page_Out

- In main.c
  - Add_Init_Paging

- Also, anywhere you may want to use KINFO_PAGE_ON_DISK to check if the page is on disk
PAGING TO DISK & PAGE REPLACEMENT

Physical Memory

1 page

Swap

1 page = 8 blocks

startSector

A total of numSectors blocks

1 block

Track/
Cylinder

Sector

Heads
8 Heads,
4 Passes
PAGING TO DISK

• If it is a page that is swapped out to disk (pte->kernelInfo == KINFO_PAGE_ON_DISK), you need to bring it back from disk.

• How are paging files stored on disk?
  • The paging file consists of a group of consecutive disk blocks of size SECTOR_SIZE bytes.
  • Each page will consume 8 consecutive disk blocks.
  • Call Get_Paging_Device() will return a Paging_Device structure, which contains startSector, numSectors you can use.

• Disk read/write is block by block. (Block_Write() and Block_Read() in GeekOS)

• How to manage the paging files is up to you. A possible way is to make an array of pages on the disk.
The code to page out a page is partially 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 meme.c. (This function relies on the clock field in the Page structure which you must manage)
- 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 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 is on disk rather than not valid).
- Flush the TLB using Flush_TLB from lowlevel.asm.
PAGING TO DISK

• In paging.c:
  • Init_Paging
    • Initialize your paging file structure with Get_Paging_Device() and the variables (numSectors) in the Paging_Device structure. The structure can simply be an int/boolean array, representing if a page on disk is free or not.
  • Find_Space_On_Paging_File
    • Use the paging file structure you defined above to find a free page on disk.
  • Free_Space_On_Paging_File
    • Use the paging file structure you defined above to mark the page on disk as free.
  • Write_To_Paging_File
    • Use Block_Write to write contents in the physical memory to disk blocks. Remember disk read/write is block by block.
  • Read_From_Paging_File
    • Similar with Write_To_Paging_File, but use Block_Read
• In main.c
  • Call Init_paging()
PAGE REPLACEMENT

- In mem.c:
  - Alloc_or_Reclaim_Page
    - The routine is on page 27
  - Find_Page_To_Page_Out
    - The clock algorithm.
    - g_pageList contains all pages in the system.
    - Clock hand is incremented as long as the page table entry that the Page structure refers to has the accessed bit set or the page is not pageable.
    - Add a new field to userContext and mark how many pages the context has in memory. If less than 10 pages, don’t page out its own pages; if more than 1000 pages, don’t page out another process’ pages.
CLOCK ALOGORITHM

Page 0
struct Page *curr->entry->accessed == 1
Not Pageable

Page 1
struct Page *curr->entry->accessed == 1
Pageable

Page 2
struct Page *curr->entry->accessed == 1
Pageable

Clock hand
CLOCK ALGORITHM

Page 0
struct Page *curr->entry->accessed == 1
Not Pageable

Page 1
struct Page *curr->entry->accessed == 1
Pageable

Page 2
struct Page *curr->entry->accessed == 1
Pageable

Clock hand
CLOCK ALGORITHM

Page 0
struct Page *curr->entry->accessed == 1
Not Pageable

Page 1
struct Page *curr->entry->accessed == 1
Pageable

Page 2
struct Page *curr->entry->accessed == 1
Pageable

Clock hand
CLOCK ALOGORITHM

Page 0
struct Page *curr->entry->accessed == 1
Not Pageable

Page 2
struct Page *curr->entry->accessed == 1
Pageable

Page 1
struct Page *curr->entry->accessed == 0
Pageable

Clock hand
CLOCK ALGORITHM

Page 0
struct Page *curr->entry->accessed ==
1
Not Pageable

Page 2
struct Page *curr->entry->accessed ==
1
Pageable

Page 1
struct Page *curr->entry->accessed ==
0
Pageable

Clock hand
CLOCK ALOGORITHM

Page 0
struct Page *curr->entry->accessed == 1
Not Pageable

Page 2
struct Page *curr->entry->accessed == 0
Pageable

Page 1
struct Page *curr->entry->accessed == 0
Pageable

Clock hand
CLOCK ALOGORITHM
CLOCK ALGORITHM

Page 0
struct Page *curr-
>entry->accessed ==
0
Pageable

Page 1
struct Page *curr-
>entry->accessed ==
0
Pageable

Page out!

Page 2
struct Page *curr-
>entry->accessed ==
0
Pageable
HINT

- There are many ways to finish this project as long as you can implement a VM system. And they might need fewer lines of code. No worries if you didn’t follow the slides strictly.
Page Fault Error Codes

Interrupt 14—Page-Fault Exception (#PF) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>P 0 The fault was caused by a non-present page.</td>
</tr>
<tr>
<td></td>
<td>1 The fault was caused by a page-level protection violation.</td>
</tr>
<tr>
<td>3</td>
<td>W/R 0 The access causing the fault was a read.</td>
</tr>
<tr>
<td></td>
<td>1 The access causing the fault was a write.</td>
</tr>
<tr>
<td>2</td>
<td>U/S 0 The access causing the fault originated when the processor was executing in supervisor mode.</td>
</tr>
<tr>
<td></td>
<td>1 The access causing the fault originated when the processor was executing in user mode.</td>
</tr>
<tr>
<td>1</td>
<td>RSVD 0 The fault was not caused by reserved bit violation.</td>
</tr>
<tr>
<td></td>
<td>1 The fault was caused by reserved bits set to 1 in a page directory.</td>
</tr>
</tbody>
</table>

Figure 5-7. Page-Fault Error Code