Project 4b: Paging

Due April 17

1 Overview

In this part of the project, you will support demand paging (as in, paging to disk) and user Malloc(). Every process will have its own page tables (`userContext->pageDir`), and all processes will have the same segment base address, starting at 0x80000000.

As with part a, a functioning Fork, Pipe, and Serial port are *not* required for this project. Making Fork() work properly with page tables is a substantial challenge.

2 User VM

The following changes occur primarily in uservm.c, which can be based on the functions in userseg.c. Change Makefile.common (USER_IMP_C field) in order to compile with uservm.c instead of userseg.c.

Linear addresses greater than 0x80000000 shall be for users. The VM_USER flag should be set (except for the APIC hole described in 4a) only on these page table and directory entries. Addresses below shall be for the kernel.

Set the beginning of the segments for user processes to be 0x80000000, rather than the result of a kernel Malloc, and the size of the segments to be 0x70000000.

Each user page directory should refer to (shared) kernel page tables for the pages below 0x80000000. This allows us to continue using the user page directory during system calls, rather than switching to the kernel's page directory. Each user page directory should be unique above 0x80000000, with the exception of the APIC hole.

The argument block should be placed so that it ends just before 0xF0000000; Allocate 2 pages for the user stack and place it below the argument block. This should yield stack pointers in user space that are logically 0x6FFFF... and are linearly 0xEFFFF.... You will not need to manage the dynamic growth of the stack.

Leave the user's NULL page unmapped and set the pages which contain user code (beginning from logical address 4096) to be read only.

Switch_To_Address_Space should change the current paging directory to that of the current thread's. This can be done using the function Set_PDBR, defined in lowlevel.asm.

3 Malloc and bget

To allow the process to grow to use more memory than is physically present, we will use Malloc. Malloc in the geekos kernel is based on bget.c. The same file can be reapplied at user level, and it's already in src/common, ready to be built and included.

Edit src/libc/compat.c to implement Malloc using: bget, bpool, and Sbrk; implement Free using brel. The "b" functions are described in bget.c under the heading "BGET FUNCTION DESCRIPTIONS".

Sbrk should have the following semantics. It will return the end of the heap of the process. This should initially be located at the end of the data segment which was copied over from the executable file. If given a numeric parameter, Sbrk will extend the heap by that value. The numeric parameter may be zero, which means the end of the heap shall not change, but is merely returned.

You may find an authoritative description of the sbrk system call in its man page.

Implement Sys_Sbrk. Always extend the segment as much as requested by the program, unless by doing so the total heap size would exceed 163,840,000 bytes (40,000 pages). The Page_Fault_Handler() should enforce that the application does not access memory outside of its allocation. [1]

4 Allocating a page

Use Alloc_Pageable_Page() to get a page that is set up to be paged out if needed.

You will want to allocate pages when loading the code in the initial creation of a process. You will also want to allocate pages when a page fault occurs on an address that is valid for the process (e.g., had previously been Malloc'd.)

5 Paging to disk

The paging file consists of a group of consecutive disk blocks of size 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 8 consecutive disk blocks (PAGE_SIZE/SECTOR_SIZE). To read and write the paging file, use the functions Block_Read and Block_Write provided in blockdev.h. These functions write SECTOR_SIZE bytes at a time. How you manage your paging file is completely up to you. A good idea would be 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:

1. Find a page to page out using Find_Page_To_Page_Out which you will implement in mem.c. (This function relies on the clock field in the Page structure which you must manage).
2. Find space on the paging file using Find_Space_On_Paging_File which you will implement in paging.c
3. Write the page to the paging file using Write_To_Paging_File which you will implement in paging.c
4. Update the page table entry for the page to clear the present bit.
5. Update the pageBaseAddr in the page table entry to be the first disk block that contains the page.
6. 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).
7. Flush the TLB using Flush_TLB from lowlevel.asm. Note that this will only flush the TLB associated with the current processor, potentially causing problems if the stolen page was being used by another processor. When testing this functionality, you may wish to change the makefile to compile QEMU with the option "-smp 1" in order to turn off secondary processors.

Eventually, the page that was put on disk will probably 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 (since even pages that haven't been written since they were last brought into memory from disk are not already on disk). You will rely on the information stored when a page is paged out to find it on disk and page it back in.

[1] The segmentation hardware could probably do this too, but we will use the page fault handler.

6 Page Replacement

Ahh, but which page should you send to disk?

There is a simple page replacement algorithm in Find_Page_To_Page_Out(), however it is broken.

Implement a page replacement scheme that is based on clock, but has two watermarks:

1. If the process has fewer than 10 pages in memory, it will not choose its own page to evict.
2. If the process has more than 1000 pages in memory, it will not choose a page belonging to another process to evict.

A process with fewer than 10 pages in memory is unlikely to need to replace pages; a process can get more than 1000 pages in memory since there are many thousands more pages available, and this task is only invoked when they are all exhausted.

This design should work well in practice, since it discourages stealing a page from a process that may potentially be running on another processor at the same time.

The clock hand is a static variable that is an index into the array of Pages; it 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. The accessed bit will be cleared if set; the page evicted if not. By running through the page list twice, it is very likely that an unaccessed page will be found.

7 Dead Processes

As part of process termination, free the memory pages and page tables, and even any pages in the page file associated with a process. Modify Destroy_User_Context.

8 OOM

When there are no pages and no space on disk, terminate the faulting process and release its pages. There exist better out-of-memory killer algorithms.

9 Hints

Beware operating on a page table that does not belong to the process currently running, for example, when in Spawn(). The task switcher will reset the page table base register to that associated with the user context, and does not remember a temporary page table base register. Disable interrupts.

Paging often permits a "hard" way, directly manipulating the physical pages and working one page at a time, and an "easy" way, setting up the page table and using it. The "hard" way may be necessary if interrupts must be disabled, but the "easy" way means much less code.

Break the page table and even Qemu will become highly confused, possibly dumping register state to stderr. Interestingly, when gdb can make no sense of the address space of one core because of trouble with the page directory it has loaded, switching to the other thread allows it to make progress.

10 TODOs

Generally, the TODO macros associated with this project are VIRTUAL_MEMORY_B and MALLOC.

11 Test

The key test for this assignment is mlc and multimlc. Mlc tries to Malloc as many pages as given on an argument, then accesses those pages repeatedly to check that they are still as initialized. Multimlc spawns three concurrent copies of mlc.