Project 4 part 2: Big Stack

Consult the submit server for deadline date and time

1 Overview

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

As for part 1, a functioning Fork, Pipe, and Signals are *not* required for this project. Making Fork() work properly with page tables is a substantial challenge. Consider adding a "return EUNSUPPORTED" at the top of Sys_Fork().

2 User VM

The following changes occur primarily in uservm.c, which can be based on the functions in userseg.c. Change the Makefile to refer to uservm.c instead of userseg.c to choose this functionality.

Linear addresses greater than 0x80000000 shall be for users. VM_USER 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 segment to be 0x70000000.

Each user page directory may refer to (shared) kernel page tables for the pages below 0x80000000. Each user page directory should be unique above, with the exception of the APIC hole, which may be handled either way. No shared memory is expected.

The stack should start just below the argument block, which ends at 0x70000000, yielding stack pointers in user space that are logically 0x6ffff... and are linearly 0xeffff.... This low value is meant to avoid eventual conflict with the APIC hole, which, at 0xFEC, is pretty likely to get in the way of a large stack.

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.

3 The LDT

When using segmentation, each process needs its own LDT in the userContext. The LDT includes base and limit for code and data segments. Typically, it is easiest to just change how the LDT in each userContext is constructed in userseg.c and preserve the one-ldt-per-process approach. However, you might consider creating a single, global LDT for all processes to use, since they are all identical. A reasonable place to declare and reference this is in Init_GDT in gdt.c.

You might look at `http://wiki.osdev.org/GDT_Tutorial`, if the intel manual is a bit rough. Figure 3-10 in the Intel manual is a good representation of how the GDT and LDT work.

4 Expanding the stack

To implement stack growth, you need to modify the default page fault handler from Part I. 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.

You will modify this page fault handler to determine an appropriate action to take based on the address. If the address is within one page of the current stack limit, allocate a new physical page, map the appropriate virtual page (which expands the stack) to this physical page, and return normally from the handler. The program will now be able to use the memory that you just allocated to grow the stack beyond its old page. In order to test your new page fault handler, run the provided program rec.c.

If the address is too low (more than a few pages below the previous stack page), allow the fault (print the page fault information) and exit the program.

5 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.)

6 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

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.

7 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. Stealing a page from another process is tricky due to the TLB.

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.

Testing for reasonable page replacement is based on how many faults are necessary to complete the tests that use more virtual memory than there is physical memory. Repeatedly choosing the same page for eviction will likely complete (perhaps slowly), but require too many faults.

8 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.

9 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.

10 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.

11 TODOs

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

12 Test

The key test for this assignment is rec.c. It takes an argument of the number of invocations to recurse; with a large enough argument, stack pages will need to go to disk. As the recursive function returns, it will check to see that its state hasn't been corrupted.