ECE592 – Operating Systems Design: Homework #4

Due date: November 28, 2017

Objectives

- To understand Xinu's memory management.
- To understand how to implement a virtual memory. This includes understanding in details interactions between the hardware and the software.
- To understand how interrupt handling works (including the related hardware and software interactions).

Overview

The goal of this project is to implement virtual memory management and demand paging in Xinu. Refer to the chapters of the Intel Manual (available in Moodle) related to Paging and Interrupt Handling. In particular, as you read the Intel Manual, focus on the following: (i) registers used to support virtual memory and paging, (ii) PDE and PTE format, (iii) interrupt handling support.

Please read carefully the whole description below before starting the implementation.

System call implementation

Your solution must implement the system calls listed below.

1. `pid32 vcreate (void *funcaddr, uint32 ssize, uint32 hsize, pri16 priority, char *name, int nargs, …)`

   This system call creates a Xinu process with a specified heap size. *The process's heap must be **private** and exist in the process's own virtual memory space*. Parameter `hsize` indicates the heap size (in number of pages). Use a *4KB page size*. The heap size limitation is determined by macro `MAX_HEAP_SIZE` that you must define in `paging.h`. Set `MAX_HEAP_SIZE` to 4096 frames.

   Note: While processes created with Xinu's `create` function should not have a private virtual heap, enabling virtual memory and paging will require you to modify the `create` function as well.

2. `char* vmalloc (uint32 nbytes)`

   This function allocates the desired amount of memory (in bytes) off a process's virtual heap space, and returns `SYSERR` if the allocation fails.

3. `char* vmalloc_shared (pid32 pid, uint32 nbytes, char *ptr)`

   This function allows processes to allocate and share heap space. It can be invoked in two ways: (i) to allocate some memory space that will be shared with other processes, and (ii) to map a memory space that was previously allocated (as shared) by another process to the caller's virtual address space. We call the process that first allocates a shared space the "owner" of that space.

The arguments of `vmalloc_shared` are as follows:

- `pid` is the identifier of the owner of the space;
- `nbytes` is the size (in bytes) of the shared heap space allocated;
- In case (i), `ptr` is unused (and can be NULL); in case (ii), `ptr` is the pointer to the shared space in the virtual address space of the owner.

The function returns a pointer to the shared space within the virtual address space of the caller in case of success, and `SYSERR` in case of failure.

For example, say that process 0 allocates a shared space of `N` bytes that will be later shared with processes 1 and 2. Process 0 will be the owner of the shared space. This will be accomplished by the following code:

Process 0: `char *vptr0 = vmalloc_shared (0, N, NULL);`
Process 1: `char *vptr1 = vmalloc_shared (0, N, vptr0);`
Process 2: `char *vptr2 = vmalloc_shared (0, N, vptr0);`

In the example, `vptr0`, `vptr1` and `vptr2` are pointers within the virtual address spaces of process 0, process 1, and process 2, respectively.

4. `syscall vfree (char* ptr, uint32 nbytes)`

This function frees heap space (previously allocated with `vmalloc` or `vmalloc_shared`). If heap space was allocated as shared, it must be de-allocated by the owner process. `vfree` returns `OK` in case of success, and `SYSERR` in case of failure.

Additional requirements

1. Memory initialization
   The original Xinu memory initialization has a bug. Replace the original `meminit.c` and `i386.c` files in the `system` folder with the ones attached, which fix the bug. You are recommended to look into these files and understand what has been fixed. As you can see in Xinu's code, besides the TEXT, DATA, and BSS regions, Xinu statically reserves a so-called HOLE area for boot-loading purposes.
   Hint: After paging is enabled, all memory accesses are through the virtual address space. Therefore, as we discussed in class, you need to map the static segments (TEXT, DATA, etc.) in the virtual memory space of each process.

2. Disk Space Simulation
   As you know, virtual memory typically uses disk space to extend the physical memory of the machine. However, the Virtual Box version of Xinu that you are using has no file system. Thus, you need to simulate the disk space using free memory. Reserve a total of `MAX_HEAP_SIZE` frames from the free space to emulate the disk space. The disk space should be mapped to the virtual memory space of all the processes.

3. Free Frame Space (FFS)
   The FFS is the physical memory space where processes map their virtual heap space. FFS must be released when no longer required. The maximum size of FFS that can be allocated by a process is determined by

macro `MAX_FSS_SPACE` defined in `paging.h`. Set `MAX_FSS_SPACE` to 2048 frames. *Unless the virtual space is shared, when a process maps a FFS frame to a virtual page, that FFS frame should not be visible to other processes* (that is, there should be no mapping between virtual pages of other processes to that FFS frame).

4. Page directory, Page Tables and Virtual Heap Pages
   Page directories and page tables must be always *resident* in physical memory (i.e., they should never be swapped to disk). Physical memory used by page directories and page tables must be released when no longer required.

5. Heap allocation
   You must use a **lazy allocation policy** for heap allocation. That is, the physical space should be reserved not at allocation time, but when the virtual page is first accessed. Accesses to pages that have not been previously allocated must cause a SEGMENTATION_FAULT.

6. Page replacement and swapping
   You must use the **Approximate LRU replacement policy** (specifically, Corbato's clock algorithm). In addition, you must have **global replacement** (that is, a process might cause the eviction of pages belonging to other processes). On page eviction, only dirty pages must be copied to disk.

Hints

Before you start coding, make sure you have a clean design of all aspects of the problem. To this end, you may want to take a piece of paper and sketch down the following.

1. Where should the page directories and page tables be stored? How can a process access them? Draw the processes' memory map.
2. What operations need to be performed by the operating system at **process creation**?
3. What operations need to be performed by the operating system at **process destruction**?
4. What operations should be performed at **context switch**?
5. What operations should be performed at **system initialization**?
6. What operations should be performed at **heap allocation**?
7. What operations should be performed at **heap de-allocation**?
8. Can the hardware raise a segmentation fault?
9. In which circumstances will the hardware raise a page fault?
10. What operations should be performed by the page fault handler depending on the circumstances under which it is invoked?
11. x86 PDE and PTE have bits that are reserved for software use. You can use those bits as you please in your implementation.
12. **Interrupt handling** – Study the code in `clkinit.c`, `clkdisp.S` and `clkhandler.c` to find out how to install an interrupt service routine (ISR) in Xinu. Implement the page fault handler in the following two files: `pagefault_handler_disp.S` and `pagefault_handler.c`. As indicated in the Intel Manual, a page fault triggers interrupt 14. Thus, you need to install your page fault handler to interrupt 14. Intel processors automatically push an error code on the stack when an ISR is called, so you need to pop out the error code off the stack within your Page Fault Handler.

13. When a process A creates a process B, it needs to initialize its page directory and page table. Similarly, a process A needs to access the page directory and page table of a process B when evicting its pages. To simplify the implementation, you can map all the page directories and tables to the virtual space of all the processes.
14. For convenience, we have added a `control_reg.c` and `paging.h` that you can extend.

Questions for extra points:

Note: it is **not** allowed to post questions on the extra credits problems on the class mailing list. Work on these problems only if you can do it individually or within your team.

1. **(4 points)** Implement a **swap daemon** that runs in background. Specifically, the daemon should sleep for a predefined `SD_SLEEP_TIME` time (in ms). When it wakes up, it should start evicting pages if the amount of free pages available is less that a low watermark (LW) threshold, and it should evict pages until HW pages are free (HW is a high watermark threshold). After eviction, the daemon should go back to sleep for `SD_SLEEP_TIME` time. When necessary, the page fault handler may also awaken the swap daemon. Macros `SD_SLEEP_TIME`, `LW` and `HW` should be defined in `paging.h`.
2. **(6 points)** Modify your implementation so to make the page tables and directories visible only to one system process (called `memory_management_process`). That is, your implementation should map the physical memory space containing the page directories and page tables only to the virtual space of the `memory_management_process` (and should not follow hint #13).

Submissions instructions

1. **Report:** Your report should:
   - contain an illustration of the memory map of the processes, including the start and end address of each section (e.g., text, data, bss, disk space, etc.). If different types of processes have different memory maps, draw them all;
   - explain the salient aspects of your design and implementation;
   - explain what works and what not in your implementation;
   - indicate if your implementation also covers the extra points requirements.
2. **Test cases:** For this homework assignment, you do **not** need to submit your test cases. We will test your implementations using different test cases (i.e., different `main.c` files). Therefore, do not implement any essential functionality (other than your test cases) in the `main.c` file. Also, turn off debugging output before submitting your code.
3. **Extra points:** Submit **only one implementation**. You can choose to either submit a code that implements only the basic requirements (for no extra points), or a code that also implements the requirements for extra points.
   - Submit the implementation for extra points only if you have tested it extensively and are certain that it works properly. In other words, make sure that all the working features of your basic implementation still operate correctly (otherwise you will lose credits).
   - If only one team member worked on the extra points, make two separate submissions (two codes and reports): one team member will submit the implementation of the required portion, and the other team member will submit the implementation covering also the requirements for extra points.
4. Go to the `xinu/compile` directory and invoke `make clean`.

5. As for previous homework assignments, create a `xinu/tmp` folder and **copy** all the files you have modified/created into it (the `tmp` folder should have the same directory structure as the `xinu` folder).
6. Go to the parent folder of the `xinu` folder. Compress the whole `xinu` directory into a *tgz* file.

```
tar czf xinu_homework4.tgz xinu
```

7. Submit your assignment through Moodle.
8. Print your report and bring it to class on the due date.