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.

Specifically, you will find the information you need in the **Intel Architecture Software Developer's Manual Volume 3: System Programming Guide**:

- Memory management:
○ Memory management registers – Section 2.4
○ Invalidating TLBs – Section 2.6.4
○ Paging and Virtual Memory – Section 3.6
○ Translation Lookaside Buffers – Section 3.7
- Interrupt handling: Chapter 5

In addition, see attached slides **xinu_memory_management.pdf** for an overview of Xinu's memory management.

1**Important:** Please read the entire project description carefully (including the **simplifying assumptions** and **hints** on page 5 and 6) before getting started.

System call implementation

Your solution must implement the system calls listed below.

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

This system call creates a "user" process with a virtual heap**.** The process's heap must be private and exist in the process's own virtual memory space. Your implementation should use paging and a **4KB page size.** All processes can still use the `getmem` system call to allocate *shared* heap space.

Note: While "system" processes created with Xinu's `create` function should not have a private virtual heap, enabling virtual memory and paging might 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.

ECE465 students: Virtual heap space should be allocated using the next-fit policy from lower to higher addresses. For simplicity, you can assume that the memory allocations performed by a process never exceed the size of the virtual address space.

ECE565 students: Virtual heap space should be allocated using the first-fit policy from lower to higher addresses. Note that this can cause external fragmentation of the virtual space.

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

This function frees heap space (previously allocated with `vmalloc`). `vfree` returns `OK` in case of success, and `SYSERR` in case of failure.

ECE465 students: For simplicity, your implementation can assume that *vfree* is always invoked correctly (i.e., on pages that had been allocated previously using a *vmalloc*).

ECE565 students: Your implementation should also handle the case where *vfree* is invoked incorrectly (i.e., on pages that had not been allocated before). In case of failure, none of the pages involved should be freed.

Note: *vfree* releases only heap space, and not page tables. Page tables are released only when a process is terminated.

2**Additional requirements**

1. Debugging

For debugging purposes, provide the following utility functions (which will be used in the test cases):

- `uint32 free_ffs_pages()` – number of free frames in the FFS space (see below)
- `uint32 free_swap_pages()` – number of free frames in the swap space (see below)
- `uint32 allocated_virtual_pages(pid32 pid)` – number of virtual pages allocated by a particular process (including pages allocated but not mapped to physical memory space)
- `uint32 used_ffs_frames(pid32 pid)` – number of FFS frames in use by a given process

Note: function `free_swap_pages` must be provided only if you implement swapping (optional)

2. Handling of segmentation and protection faults

In case of segmentation fault your code should:

- Print the process triggering the fault as follows:

`P<pid>:: SEGMENTATION_FAULT`

- Kill the faulting process and continue execution

Your code does not need to handle protection faults.

3. Memory initialization

The original Xinu memory initialization does not allow the whole 4GB address space to be available for paging. Replace the original `meminit.c` and `i386.c` files in the `system` folder with the ones attached, which fix the problem.

Important hint*:* After paging is enabled, all memory accesses are through virtual addresses. Therefore, you need to map the static segments (TEXT, DATA, etc.) into the virtual memory space of each process.

4. 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 (in other words, heap frames should be released upon heap deallocation). The total amount of FSS frames available is determined by macro `MAX_FSS_SIZE` defined in `paging.h`. `MAX_FSS_SIZE` is not a per-process limitation – it indicates the total amount of FSS available to all processes. *When a user process maps an FFS frame to a virtual page, that FFS frame should not be visible to other user processes* (i.e., user processes can map to their virtual address space only the FFS frames they use). Unless you decide to implement swapping (optional), you can assume that the number of physical frames allocated never exceeds the FFS size.

5. Page directory and Page Tables

A total of `MAX_PT_SIZE` frames can be reserved for page directory and page tables.

ECE465 students: Physical memory used by 2nd-level page tables must be released when a process terminates.

3**ECE565 students:** Physical memory used by page directory *and* 2nd-level page tables must be released when a process terminates. *The area of memory where page directories and page tables are stored cannot be accessed by user processes (i.e., this area must not be mapped to user processes' virtual address space).*

6. 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.

7. Page replacement and swapping **[OPTIONAL: 10 extra points]**

Assume you have a total of `MAX_SWAP_SIZE` frames to emulate the swap space. Macro `MAX_SWAP_SIZE` is defined in `paging.h`. Your implementation must keep track of the mapping between FFS frames and swap frames. Only FFS frames can be evicted. However, *you can omit physically copying data between FFS and swap space*. You must use the **approximate LRU policy** (irrespective of the value of the dirty bit) for replacement. In addition, you must have **global replacement** (that is, a process might cause the eviction of pages belonging to other processes). Unless an FFS frame is already present in the swap space, on eviction you will use the first-fit policy to identify a free swap frame. For simplicity, assume that the swap space will never be exhausted.

For debugging purpose:

- Introduce a `DEBUG_SWAPPING` macro definition (`#define DEBUG_SWAPPING`)
- If `DEBUG_SWAPPING` is set, print the following information:

On FFS eviction:

```
eviction:: FFS frame <FFS frame #>, swap frame <swap frame #> [copy]
```

where `copy` indicates that the content of FFS must be copied to the swap space

On loading from swap space into FFS:

```
swapping:: swap frame <swap frame #>, FFS frame <FFS frame #>
```

The frame numbers should be in hexadecimal format. For example:

`eviction:: FFS frame 0x0, swap frame 0x10` (eviction without copy to swap space)

`eviction:: FFS frame 0x10, swap frame 0x20 copy` (eviction & copy to swap space)

`swapping:: swap frame 0x20, FFS frame 0x30`

- To limit the amount of debugging information printed out, add an unsigned `debug_swapping` global variable to indicate the number of debugging statements printed out (have a look at `swapping_testcases.c/out` to see how this variable is used).

8. Constant definitions in paging.h

Your code should work independent of the constant definitions in `paging.h` with the following exceptions:

- The page size won't be modified (you can assume a 4KB page size).
- `MAX_PT_SIZE`, `MAX_FFS_SIZE` and `MAX_SWAP_SIZE` will always be set to a multiple of 1024 frames.

4**Simplifying assumptions**

1. You can assume that the operating system performs allocations and triggers faults **at a page granularity.** For example, assuming 8-byte pages, the following allocation operations:

```
char *ptr1 = vmalloc(16);
char *ptr2 = vmalloc(15);
```

will be treated the same way. In other words, the OS will not keep track of the number of bytes within a page that have been effectively allocated. In addition, the OS won't trigger a segmentation fault if the code accesses `ptr2[15]` even if the allocation was just for 15 bytes.

2. Similarly, you can assume that allocation operations **are aligned to the beginning of a page**. For example, you can assume that the following allocations:

```
char *ptr1 = vmalloc(16);
char *ptr2 = vmalloc(15);
```

will go to two different pages.

Hints

1. 2. 3. 4. 5. 6. 7. Paging should be enabled at the end of system initialization. A good place to enable it is the end of the `sysinit()` function in `initialize.c`

x86 PDE and PTE have bits that are reserved for software use. You can use those bits as you please in your implementation.

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. The page fault handler should be implemented within 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. For convenience, `pagefault_handler_disp.S` is provided to you. Intel processors automatically push an error code on the stack when an ISR is called, so the code in `pagefault_handler_disp.S` pops out the error code off the stack within your Page Fault Handler.

For convenience, we have added a `control_reg.c` and `paging.h` that you can extend.

Use the same page table for all system processes.

Even if Xinu does not have a clear difference between user mode and system mode, you should aim at a design that conceptually distinguished "user mode" and "kernel mode". To this end, assume that, when a user process issues a system call, that call is executed in kernel mode. On the other hand, the rest of the user process's code (that is, the code within the function executed by the process) is executed in user mode.

Test your code "incrementally", following the testing plan below.

5**TESTING PLAN:**

A. Start by testing the code with paging enabled but **without invoking the three system calls above**. At this point, you will have only system processes, and you can use the default Xinu's `main.c` file. This test will allow verifying that the page directories and page tables have been setup correctly. If this is not the case, the system will hang on startup. To be able to debug problems related to the way you have initialized the page table(s), add a function to dump the content of page directories and page tables in a format that is easy for you to read and interpret.

B. When (A) works, test user process creation and termination. Start with a single process. No need to invoke `vmalloc` and `vfree` yet.

C. Add calls to `vmalloc` and `vfree` from one process. The page fault handler does not need to be implemented yet.

D. E. F. Add heap accesses (to trigger lazy allocation).

Add more processes, so to test context switch between user processes.

Add swapping (if you decide to perform the optional part of the project).

NOTE: We have included three categories of test cases: (1) *small heap allocations*, (2) *large heap allocations*, and (3) *large heap allocations with swapping*. Please make sure to read the `README` file for details on the test cases. Additionally, review the code in the test case files. You can selectively enable or disable test cases by commenting out the `#define` macros at the beginning of each file. Finally, as indicated in the `README` file, ECE465 students should add the following line at the beginning of `paging.h`:

```
#define ECE465
```