ELL 783/ELL 405: Operating Systems
Assignment-2

Total: 50 Marks
Released: 23rd March 2021

Introduction

Memory management is one of the most important features of any operating system. In this assignment we will examine how Xv6 handles memory and attempt to extend it by implementing a paging infrastructure which will allow Xv6 to store parts of the process' memory in a secondary storage.

To help get you started we will first provide a brief overview of the memory management facilities of Xv6. We strongly suggest you read this section while examining the relevant Xv6 files (`vm.c`, `mmu.h`, `kalloc.c`, etc) and documentation.

Xv6 memory overview

Memory in Xv6 is managed in 4096 (= 2^{12}) bytes long pages (and frames). Each process has its own page table which is used to translate virtual addresses to physical addresses.

In Xv6, the process virtual address space is 2^{32} bytes long ($\tilde{4}$ GB). However, a user space process is limited to only 2 GB (from virtual address 0 to `KERNBASE`) of memory. The memory parts above `KERENBASE` are mapped to the kernel memory and allow kernel mode code to use the process' page table. Figure 1, taken from the Xv6 book, shows Xv6 s memory layout.

When a process attempts to access an address in its memory (i.e. provides a 32 bit virtual address) it must first seek out the relevant page in the physical memory. Xv6 uses the first (leftmost) 20 bits to locate the relevant Page Table Entry (PTE) in its page table. The PTE will contain the physical location of the frame – a 20 bits frame address (within the physical memory). To locate the exact address within the frame, the 12 least significant bits of the virtual

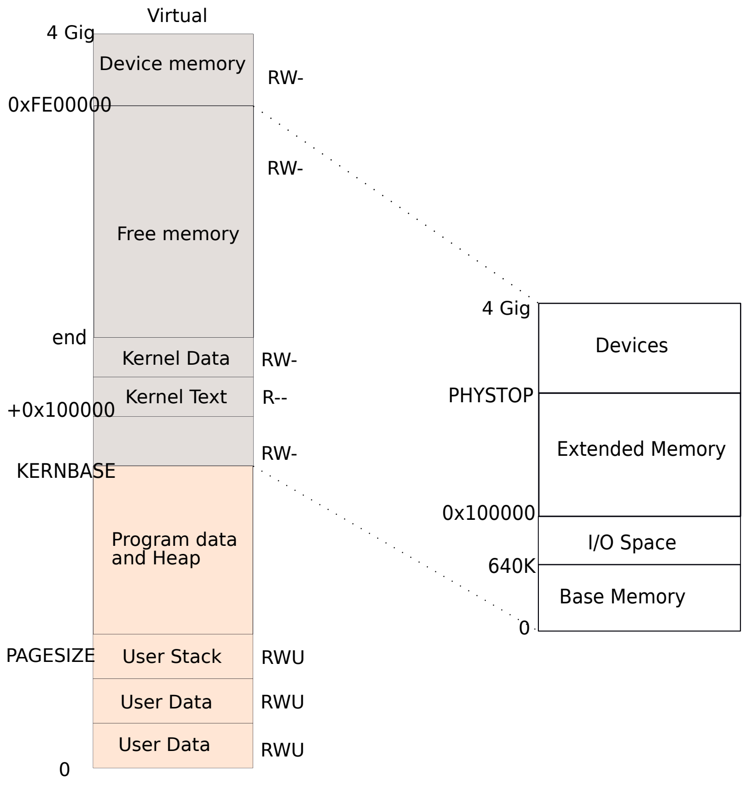

Figure 1: Layout of a virtual address space and the physical address space.

address, which represent the in-frame offset, are concatenated to the 20 bits retrieved from the PTE.

Maintaining a page table may require significant amount of memory as well, so a 2-level page table is used. Figure 2 describes the process in which a virtual address translates into a physical one.

Each process has a pointer to its page directory (`proc.h`). This is a single page sized (4096 bytes) directory which contains the page addresses and flags of the second-level table(s). This second-level table is spanned across multiple pages which are like the page directory (a page).

When seeking an address the first 10 bits will be used to locate the correct entry within the page directory (extracted using the macro `PDX(va)`). The physical frame address can be found within the correct index of the second level table (accessible via the macro `PTX(va)`). As explained earlier, the exact address may be found with the aid of the 12 LSB (offset).

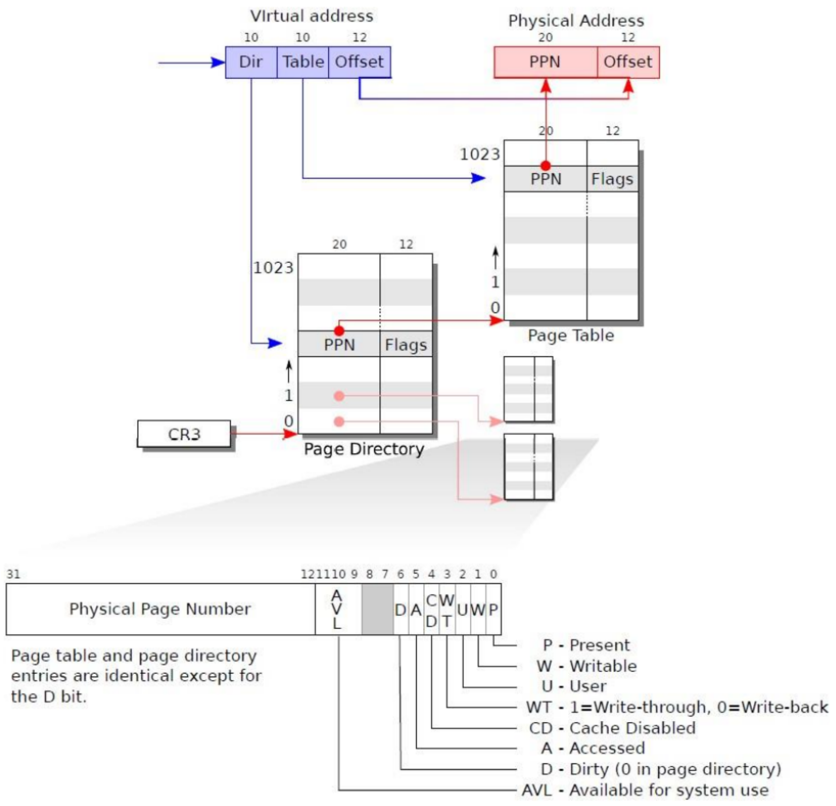

Figure 2: Page tables on x86.

Notice that x86 (Intel 80386 or later) processors have a paging hardware which translates virtual addresses to physical addresses. This enables an efficient abstraction of each memory access required by the execution of a process. The %CR3 register is used to point to the current process' page directory (see Figure 2).

Before proceeding, you should go over Xv6's documentation on memory management.

- Tip: we strongly suggest you go over the code again. Now, attempt to answer questions such as:
  1. How does the kernel know which physical pages are used and unused?
  2. What data structures are used to answer this question?
  3. Where do these reside? What does the `walkpgdir` function (from vm.c) do?

- A very good and detailed account of memory management in the Linux kernel can be found here: `http://www.kernel.org/doc/gorman/pdf/understand.pdf`
- Another link of interest: "What every programmer should know about memory" `http://lwn.net/Articles/250967/`
- Don't start the implementation before reading the entire assignment.

1 Part 1: Paging framework

An important feature lacking in Xv6 is the ability to swap out pages to a backing store (secondary storage). That is, at each moment in time all processes are held within the main (physical) memory. In the first part, you are to implement a paging framework for Xv6 which is capable of taking out pages and storing them to disk. In addition, the framework will retrieve pages back to the memory on demand.

We start by developing the process-paging framework. In our framework, each process is responsible for paging in and out its own pages (as oppose to managing this globally for all processes).

To keep things simple, we will use the file system interface supplied (described below) and create for each process a file in which swapped out memory pages are stored.

- Note: there are good reasons for not writing to files from within kernel modules but in this assignment we ignore these.
- For a few "good reasons", read this: `http://www.linuxjournal.com/article/8110`
- Well, memory is written to partitions and files anyway, no? Yep. A quick comparison of swap files and partitions: `http://lkml.org/lkml/2005/7/7/326`
- And finally, if you really want to understand how VM is done: `http://www.kernel.org/doc/gorman/pdf/understand.pdf`

1.1 Supplied File Framework

We supplied a framework for creating writing reading and deleting swap files. The framework was implemented in fs.c and uses a new parameter named `swapFile` that was added to the `struct proc` (proc.h). This parameter will hold a pointer to a file that will hold the swapped memory. The files will be named `"/.swap<id>"` where the id is the process id. Review the following functions and understand how to use them.

- `int createSwapFile(struct proc *p)` – Creates a new swap file for a given process p. Requires $p->pid$ to be correctly initiated.
- `int readFromSwapFile(struct proc *p, char* buffer, uint placeOnFile, uint size)` – Reads size bytes into buffer from the placeOnFile index in the given process p swap file.
- `int writeToSwapFile(struct proc *p, char* buffer, uint placeOnFile, uint size)` – Writes size bytes from buffer to the placeOnFile index in the given process p swap file.
- `Int removeSwapFile(struct proc *p)` – Delete the swap file for a given process p. Requires $p->pid$ to be correctly initiated.

1.2 Storing Pages in Files

We next detail some restrictions on processes. In any given time a process should have no more than MAX_PSYC_PAGES (= 15) pages in the physical memory. In addition, a process will not be larger than MAX_TOTAL_PAGES (= 30) pages. Whenever a process exceeds the MAX_PSYC_PAGES limitation it has to select (see Task 2) enough pages and move them to its dedicated file.

- These restrictions are necessary since Xv6's files system can generate only small size files (up to approximately 17 pages). You are allowed to assume that any given user process will not require more than MAX_TOTAL_PAGES pages (the shell and init should not be included, and would not be affected by our framework).

To know which page is in the process' page file and where it is located in that file (i.e., paging meta-data); you should maintain a data structure. We leave the exact design of the required data structure to you.

- Tip: you may want to enrich the `struct process` with the paging meta-data.
- Tip: be sure to write only the process' private memory (if you don't know what it means, then you haven't read the documentation properly. Go over it again.

- Tip: there are several functions already implemented in Xv6 that can assist you – **reuse existing code**.
- Tip: don't forget to free the page's physical memory.

Whenever a page is moved to the paging file, it should be marked in the process' page table entry that the page is not present. This is done by clearing the present (PTE_P) flag.

A cleared present flag does not imply that a page was swapped out (there could be other reasons). To resolve this issue we use one of the available flag bits (see Figure 2) in the page table entry to indicate that the page was indeed paged out. Add the following line to mmu.h:

```
#define PTE_PG 0x200 // Paged out to secondary storage
```

Now, whenever you move a page to the secondary storage set this flag as well.

1.3 Retrieving Pages on Demand

While executing, a process may require paged out data. Whenever the processor fails to access the required page, it generates a trap (interrupt 14, T_PGFLT). After verifying that the trap is indeed a page fault, use the %CR2 register to determine the faulting address and identify the page.

Allocate a new physical page, copy its data from the file, and map it back to the page table. After returning from the trap frame to user space, the process should retry executing the last failed command again (should not generate a page fault now).

- Tip: don't forget to check if you passed MAX_PSYC_PAGES, if so another page should be paged out.

1.4 Comprehensive changes in existing parts

These changes also affect other parts of Xv6. You should modify existing code to properly handle the new framework. Specifically, make sure that Xv6 can still support a fork system call. The forked process should have its own page file which is identical to the parent's file.

Upon termination, the kernel should delete the page file, and properly free the process' pages which reside in the physical memory.

2 Part 2: Page replacement schemes

Now that you have a paging framework, there is an important question which needs to be answered: **Which page should be swapped out?**

2.0.1 Page replacement algorithms

As seen in class, there are numerous alternatives to selecting which page should be swapped. Controlling which policy is executed is done with the aid of a *Makefile* macro. Add policies by using the C preprocessing abilities.

- Tip: You should read about `#IFDEF` macros. These can be set during compilation by GCC (see `http://gcc.gnu.org/onlinedocs/cpp/Ifdef.html`)

For the purpose of this assignment, we will limit ourselves to only a few simple selection algorithms:

1. FIFO, according to the order in which the pages were created [SELECTION=FIFO].
2. Second change FIFO, according to the order in which the pages were created and the status of the PTE_A (accessed/reference bit) flag of the page table entry [SELECTION=SCFIFO].
3. NFU, we will approximate LRU using the PTE_A (accessed/reference bit) flag of the page table entry. Implement the "aging" version of the algorithm which you saw in class [SELECTION=NFU].
4. The paging framework is disabled – No paging will be done and behavior should stay as in the original Xv6 [SELECTION=NONE].

Modify the Makefile to support 'SELECTION' – a macro for quick compilation of the appropriate page replacement scheme. For example, the following line invokes the Xv6 build with NFU scheduling:

```
make qemu SELECTION=NFU
```

1. If the SELECTION macro is omitted, NFU should be used as default. Again, it is up to you to decide which data-structures are required for each of the algorithms.
2. Tip: Your NFU implementation should use the timer interrupts to update its data structure
3. You can use the dirty (PTE_D) flag to improve your algorithm's performance, but this is not required.
4. Tip: Don't forget to clear the PTE_A flag

3 Part 3: Enhanced process details viewer

Now that you have implemented a paging framework, which supports configurable paging replacement schemes it is time to test it. Prior to actually writing the tests (see Task 4), we will develop a tool to assist us in testing.

In this task you will enhance the capability of the Xv6's process details viewer. Start by running Xv6. Press *ctrl* + *P* from within the shell. You should see a detailed account of currently running processes. Try running any program of your liking (e.g., stressfs) and press *ctrl* + *P* again (during and after its execution).

The *ctrl* + *P* command provides important information (what information is that?) on each of the processes. However it reveals no information regarding the current memory state of each process. Add the required changes to the function handling *ctrl* + *P* so that the number of allocated memory pages to each process is also printed. Print the number of pages which are currently paged out. In addition print the number of times the process had page faults and the total number of times in which pages were paged out. The new *ctrl* + *P* output should include a line for each process containing the 3 different sets of fields it had prior to your changes and new fields indicating the number of allocated memory pages and if the process is swapped:

```
<default fields><allocated memory pages><paged out><page faults>
<totalnumber of pagedout>
```

The number of pages allocated to a process is important information but it is often not enough. Add the required changes to the same function so that when *ctrl* + *P* is pressed the last line printed will notify the user on the percentage of free pages in the system:

```
<percentage>% free pages in the system
```

That is, the system should compute the ratio between the number of currently available pages and the number of pages that are available after the kernel is loaded.

The *ctrl* + *P* feature is invoked upon user demand, thus allowing the user to see the current process information. However, the final process information is important as well. Print the same information for each user process upon termination. Since in most cases this information is verbose, enable the final printing only if the quick compilation macro `'VERBOSE_PRINT'` is equal to TRUE (Similar to the SELECTION flag).

- If the `VERBOSE_PRINT` macro is omitted, FALSE is used as the default value.

3.1 Sanity Check

In this section you will add an application which tests the paging framework.

Write a simple user-space program to test the paging framework you created in Section 1. Your program should allocate some memory and then use it. Check your program with the different page replacement algorithms you created in Section 2. Analyze the program's memory usage using the *ctrl* + *P* tool you built in Section 3. This testing program should be named **myMemTest**.

- Be prepared to explain the fine details of your memory sanity test. Can you detect major performance differences between page replacement algorithms with it?
- Make sure your test covers all aspects of your new framework, including for example the fork changes you made

4 Report

The report should clearly mention the implementation methodology for all the parts of the assignment. Small code snippets are alright, additionally, the pseudo code should also suffice.

- Details that are relevant to the implementation.
- Say what you have done that is extra (this should be the last section in the document).

- Limit of 10 pages (A4 size) and must be in PDF format (name: report.pdf).

5 Submission Guidelines

- We will run MOSS on the submissions. We will also include last year's submissions. Any cheating will result in a zero in the assignment, a penalty as per the course policy and possibly much stricter penalties (including a fail grade and/or a DISCO).
- There will be NO demo for assignment 2. .

How to submit:
Create a tar ball using:

```
tar czvf assignment1_<entryNumber1_entryNumber2>.tar.gz *
```

This will create a tar ball with name, assignment1_<entryNumber1_entryNumber2>.tar.gz. Submit this tar ball on Moodle. **Entry number format: 2017ANZ8353**