OPERATING SYSTEMS, ASSIGNMENT 3
MEMORY MANAGEMENT

Responsible TAs: Amit Portnoy, Ron Korine

Introduction

Memory management is one of the key features of every operating system. In this assignment, we will examine how xv6 handles memory and will attempt to extend it by implementing a paging infrastructure which will allow xv6 to store parts of the process' memory in secondary storage.

To help you get 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 page table that maps virtual to physical addresses.

In xv6 rev11, the process virtual address space is 2^{32} bytes long (~ 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. The following figure, taken from the xv6 book, shows xv6's memory layout.

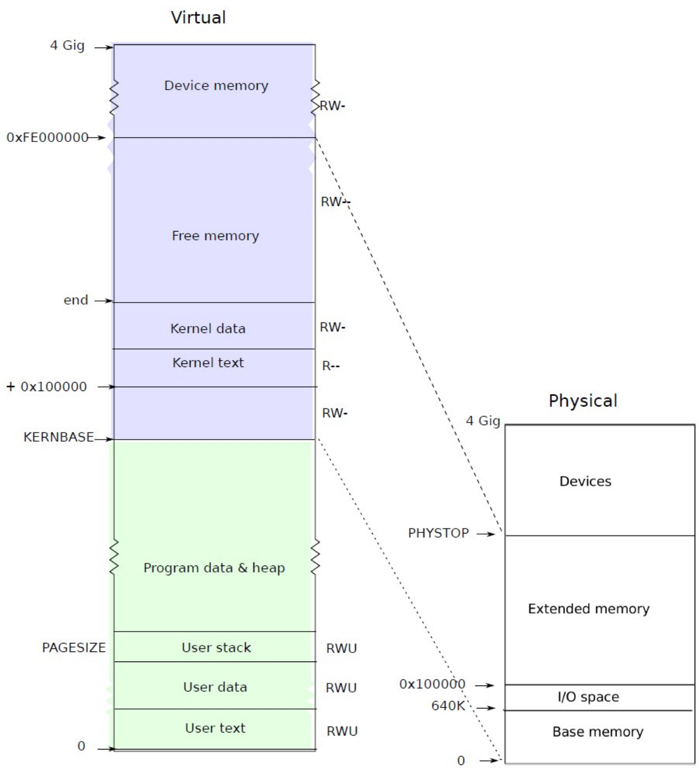

Figure 1 - Layout of a virtual address space and the physical address space.

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

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

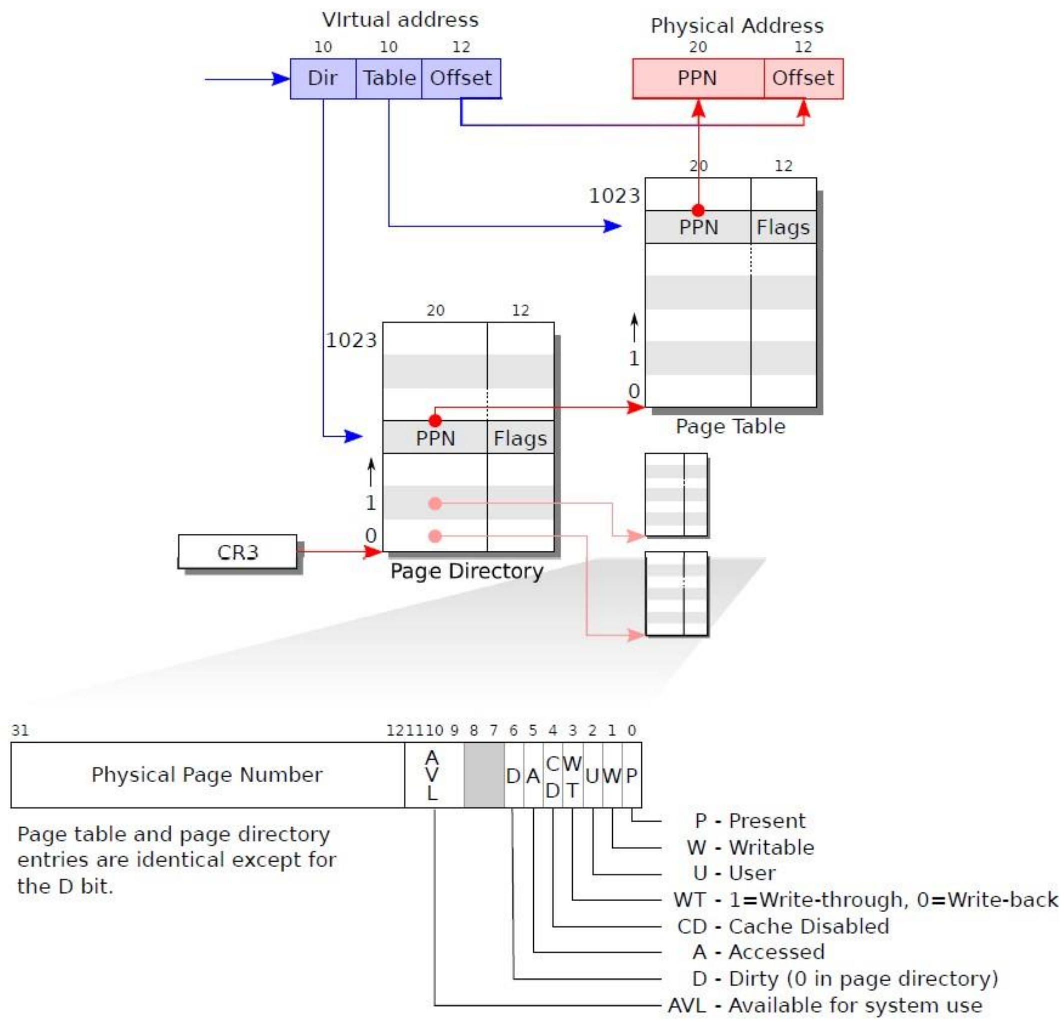

Figure 2 - Page tables on x86.

Each process has a pointer to its page directory (see 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 spans across multiple pages, which are very much like the page directory.

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).

Notice that x86 (Intel 80386 or later) processors have paging hardware that translates virtual addresses to physical addresses. This enables an efficient translation of each memory access required by the execution of a process. To activate the hardware paging mechanism, one must set the flag CR0_PG in the control register %CR0. 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: https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf

- *Tip: we strongly suggest you go over the code again. Now, attempt to answer questions such as: how does the kernel know which physical pages are used and unused? What data structures are used to answer this question? 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 description!*

Task 0: running xv6

As always, begin by downloading our revision of xv6, from the os202 ***git*** repository:

- Open a shell, and traverse to the desired working directory.
- Execute the following command (in a single line):
  `> git clone http://www.cs.bgu.ac.il/~os202/git/Assignment3`
  This will create a new folder called xv6-a3 that will contain all the project's files.
- Build xv6 by calling:
  `> make`
- Run xv6 on top of QEMU by calling:
  `> make qemu`

Task 1: Paging framework

Note that after finishing the following task, some of the **test programs** in xv6 (such as usertests) might not work. This is because some use more than 32 memory pages.

Developing a paging framework

An important feature lacking in xv6 is the ability to swap out pages to a backing store. That is, at each moment in time, all processes are held within the main (physical) memory. In the first subtask, you are to implement a paging framework for xv6, which can take out pages and store 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 opposed 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 such "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 want to understand more deeply how VM is done:* *http://www.kernel.org/doc/gorman/pdf/understand.pdf*

1.0 – Supplied file framework

We supply 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 proc struct (*proc.h*). This parameter holds 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 fileOffset, uint size)* – Reads *size* bytes into *buffer* from the *fileOffset* index in the given process *p* swap file.
- *int writeToSwapFile(struct proc *p, char* buffer, uint fileOffset, uint size)* – Writes *size* bytes from *buffer* to the *fileOffset* 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.

1.1 – Storing pages in files

We next describe some restrictions on processes. In any given time, a process should have no more than MAX_PSYC_PAGES (= 16) pages in the physical memory. In addition, a process will not be larger than MAX_TOTAL_PAGES (= 32) pages. Whenever a process exceeds the MAX_PSYC_PAGES limitation, it must select (see Task 2) enough pages and move them to its dedicated file.

- These restrictions are necessary since xv6's file system can generate only small files (up to ~17 pages). You can 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 pages are in the process' swap file and where they are 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 PCB with the paging meta-data.*
- *Tip: be sure to swap only the process' private user memory pages (if you don't know what it means, then you haven't read the documentation properly. Go over it again: https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf).*
- *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. It can be done using the kfree function described in the practical session, which gets a physical address of a page we desire to free, and adds it back to the free-pages list. Freeing a page that way should be done in several cases (e.g. upon a swap).*

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 for this flag being reset as will be explained later). 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.

NOTE: After you perform a page-out operation to a given page, the TLB might still hold a reference to its old mapping. To refresh the TLB, just refresh the rc3 register. This can be done with lcr3(v2p (p->pgdir)); as seen in the switchuvm function

1.2 – Retrieving pages on demand

While executing, a process may require paged out data. Whenever the MMU fails to access the required page, it generates an interrupt (interrupt 14, T_PGFLT). Use the %CR2 register to determine the faulting address and identify the page. Check the PTE to identify if the page was paged out or if this is just a segmentation fault (that is, an access of an illegal address).

Allocate a new physical page, copy the page’s data from the file to it, 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 (which 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**.*

1.3 – Comprehensive changes in existing functions

These changes also affect other parts of xv6. You should modify the existing code to handle the new framework properly. Specifically, make sure that xv6 can still support a fork system call. The forked process should have its own swap file whose initial content is identical to the parent's swap file.

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

You may assume ELF file size is smaller than 13 pages (which as exec works mean maximum 15 pages post exec. To support larger ELF files, you would have to create a temporary swap file, and you did not learn to handle files in the kernel, so this is not required).

Task 2: Implementing COW (Copy On Write)

The fork system call in xv6 copies the entire memory image of the calling the process, to be used as a memory image of the child process it creates.

As we saw in the practical sessions, a call to fork is often followed by a call to exec, which replaces the memory image with a new memory image. Therefore, the current implementation of the fork system call is highly inefficient. Thus, you are required to implement the COW mechanism, in which you should not copy the pages during the fork call, but instead, make sure that both the parent and the child use the same memory image, marking each page as read-only.

If an attempt to write to such a page occurs, you need to catch the page-fault and create a writable copy of the page for the faulting process. If and when the second process traps in an attempt to write to this page, no new pages should be allocated. In this case, you only need to remove the read-only restriction of the page, and proceed with the writing attempt.

In order to test your implementation, we strongly suggest you to implement a system call: *getNumberOfFreePages*, to retrieve the total number of free physical pages in the system. This system call can be very helpful for you in order to test your COW implementation [as you can use it to track when pages are being consumed].

Another simple test that can be done upon removing/raising the limitation on the maximum number of pages (both physical pages and maximum number of pages) applied in the first task is to attempt to fork a process which is using more than half of the available physical memory. Such a fork should fail with the regular fork implementation as there won't be enough physical memory to give the child to complete the copy of the memory image. Notice that using the previously suggested system call, you can create better tests.

- *Tip: You should kill a process if it attempts to write to memory, but there is insufficient memory to allocate a COW page.*
- *Tip: make sure your implementation doesn't cause any memory leaks*

Task 3: Page replacement schemes

Now that you have a paging framework and improved the inefficient implementation of the fork call with COW, there is an important question that needs to be answered: **Which page should be swapped out?**

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 this assignment, we will limit ourselves to only a few simple page replacement algorithms:

1. NFU + AGING: for each page, manage a counter (uint). When a page got accessed (check the status of the PTE_A), the counter is shifted right by one bit, and then the digit 1 is added to the most significant bit (the leftmost bit). If a page was not accessed, the counter is just shifted right by one bit (a leading zero should appear in the MSB). The page with the lowest counter should be removed. Note: when a page is created or loaded into the RAM, reset its counter to 0. [SELECTION= NFUA].
2. Least accessed page + AGING: manage a counter (uint) that is shifted similarly to the rules of the NFU + AGING algorithm, but the page with the smallest number of "1"s will be removed. If there are several such pages, the one with the lowest counter value should be removed. Note: when a page is created or loaded into the RAM, reset its counter to 0xFFFFFFFF. [SELECTION= LAPA].
3. Second chance 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].
4. Advancing queue: maintains a queue of the order in which pages should be replaced. Every time a page gets accessed, it should switch places with the page preceding it the queue (unless it is already in the first place). When more space is required, the last page in the queue is replaced. Note: when a page is created or loaded into the RAM, it takes the first place in the queue. [SELECTION=AQ].
5. 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 a quick compilation of the appropriate page replacement scheme. For example, the following line invokes the xv6 build with SCFIFO scheduling:

```
make qemu SELECTION=SCFIFO
```

- *If the SELECTION macro is omitted, second chance FIFO should be used as default. You can do that by using the following code snippet (in the make file):*

```
ifndef SELECTION
 SELECTION=SCFIFO
endif
```

Again, it is up to you to decide which data-structures are required for each of the algorithms.

- *Tip: Your AGING algorithms and your Advancing queue algorithm implementations should update their data structures every time a process returns to the scheduler function.*
- *Tip: Don't forget to clear the PTE_A flag*

Task 4: Enhanced process details viewer

Now that you have implemented a paging framework that 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 (^P) from within the shell. You should see a detailed account of the currently running processes. Try running any program of your liking (e.g., stressfs) and press ^P again (during and after its execution).

The ^P command provides important information (what information is that?) about each of the processes. However, it presents no information regarding the current memory consumption of each process.

Add the required changes to the function handling ^P so that the number of allocated ***memory pages*** to each process is also printed. Print also 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 ^P output should include a line for each process containing the 3 different sets of fields it had prior to your changes, a new field indicating the number of allocated memory pages and if the process has swapped pages, and if so, how many:

```
<field 1><field 2><allocated memory pages><paged out><page faults><total number of paged out pages><field set 3>
```

Note that upon process creation (call to fork), the number of `<allocated memory pages>` and `<paged out>` should be the same as the parent. The number of `<page faults>` and `<total number of paged out>` should be set to 0.

The number of page frames allocated to a process is valuable information but it is often not enough. Add the required changes to the same function so that when ^P is pressed the last line printed will notify the user on the number of free page frames in the system:

```
<current free page frames> / <total free page frames> free page frames in the system
```

That is, the system should compute the ratio between the number of currently available page frames and the number of page frames that are available after the kernel is loaded. For example if the system initially had 1000 page frames in the free memory section and

50 are currently in use the print out should show: `950 / 1000 free page frames in the system`

The ^P feature command is invoked by the user, thus allowing the user to see the current process information. However, we would like the system to also present this information regarding a user process when it terminates, so implement this as well. Since in most cases, this information is verbose, enable this final printing only if the quick compilation macro 'VERBOSE_PRINT' equals TRUE (Similar to the SELECTION flag).

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

Task 5: Sanity test

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

Write a simple user-space program to test the paging framework you created in Task 1. Add additional tests to check your COW implementation. Note that your program should allocate some memory and then use it. In addition, check your program with the different page replacement algorithms you created in Task 3. Analyze the program's memory usage using the ^P tool you enhanced in Task 4. This testing program should be named ***ass3Tests***.

- *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.*

Submission guidelines

Assignment due date: 04/06/2020 23:59

Make sure that your makefile is properly updated and that your code compiles with *no warnings whatsoever*. We strongly recommend documenting your code changes with remarks – these are often handy when discussing your code with the graders.

Submissions are only allowed through the submission system. To avoid submitting a large number of xv6 builds, you are required to submit a patch (i.e., a file which patches the original xv6 and applies all your changes).
You may use the following instructions to guide you through the process:

- *Back-up your work before proceeding!*

Before creating the patch, review the change list and make sure it contains all the changes that you applied and nothing more. Git automatically detects modified files, but new files must be added explicitly with the 'git add' command:

```
> make clean

> git add . -Av

> git commit -m "commit message"
```

At this point, you may examine the differences (the patch):

```
> git diff origin
```

Once you are ready to create a patch simply make sure the output is redirected to the patch file:

```
> git diff origin > ID1_ID2.patch
```

- *Tip: although graders will only apply your latest patch file, the submission system supports multiple uploads. Use this feature often and make sure you upload patches of your current work even if you haven't completed the assignment.*

To test, download a clean version of xv6 from our repository (as specified in Task 0) and use the following command to apply the patch:

```
> patch -p1 < ID1_ID2.patch
```

Finally, you should note that graders are instructed to examine your code on lab computers only (!) - ***Test your code on lab computers before submission.***

Good luck!