

OPERATING SYSTEMS

ASSIGNMENT 3

MEMORY MANAGEMENT

Introduction

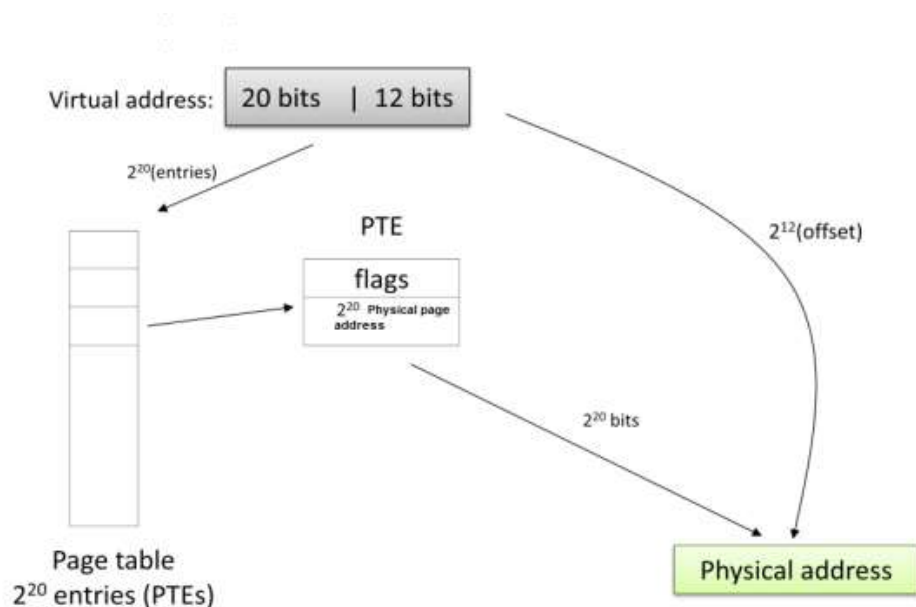
Memory management and memory abstraction is one of the most important features of any operating system. In this assignment we will examine how xv6 handles memory and attempt to extend it.

To help get you started we will provide a brief overview of the memory management facilities of xv6. We strongly encourage you to read this section while examining the relevant xv6 files (vm.c, mmu.h, kalloc.c, etc).

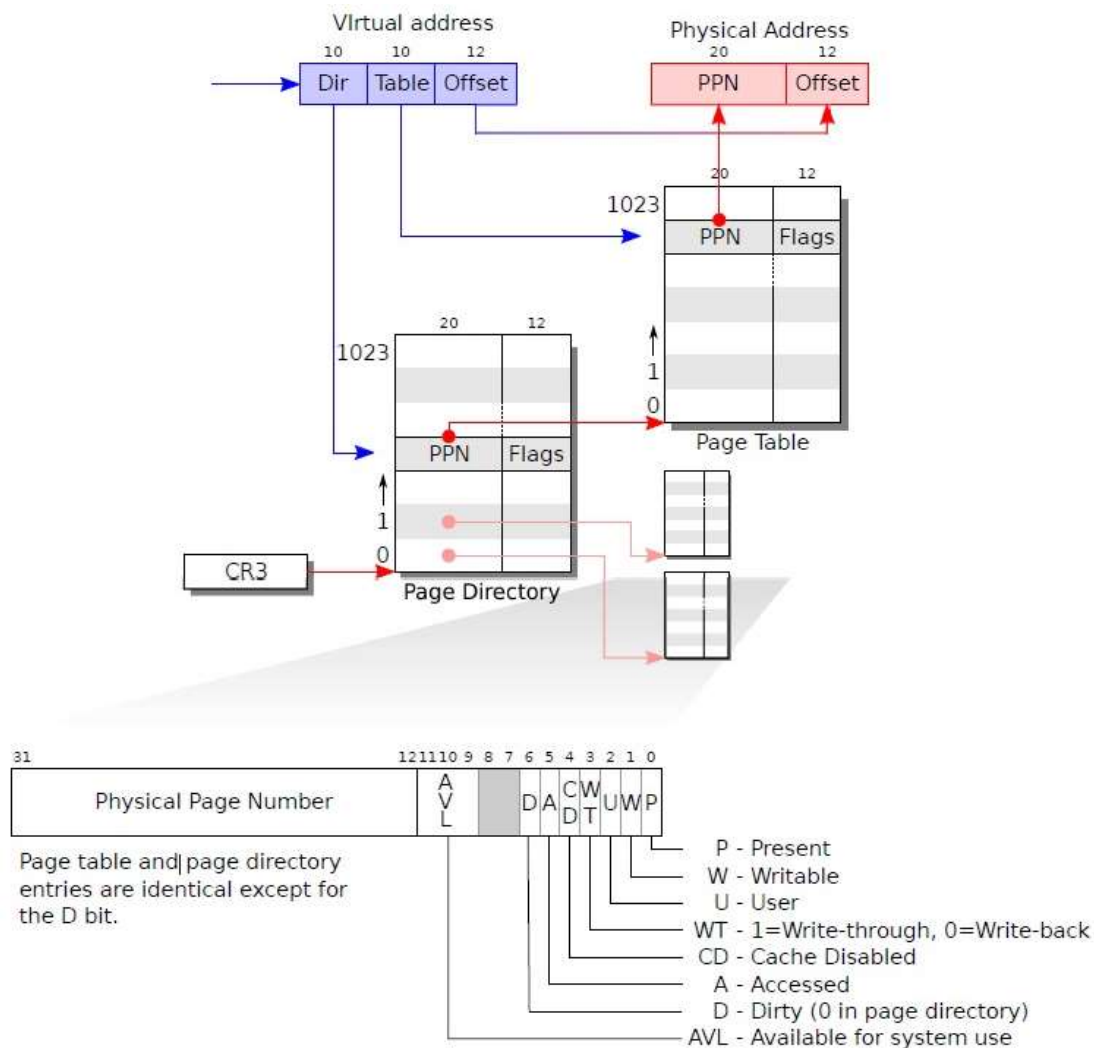
Xv6 memory

Memory in xv6 is managed in pages (and frames) where each such page is 4096 ($=2^{12}$) bytes long. Each process has its own page table which is used to translate virtual to physical addresses.

The virtual address space can be significantly larger than the physical memory. In xv6, the process address space is 2^{32} bytes long while the physical memory is limited to 16 MB only. When a process attempts to access some 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 the page table. The PTE will contain the physical location of the frame – a 20 bits frames address (within the physical memory). These 20 bits will point to the relevant frame 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. Roughly speaking, this process can be described by the following illustration:



Maintaining a page table may require 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:



Each process has a pointer to its page directory (line 59, 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 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 (by 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).

At this point you should go over the xv6 documentation on this subject:

<http://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf> (chapter 2)

- 💡 *Tip: before proceeding further we strongly recommend that 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? Does xv6 memory mechanism limit the number of user processes? What is the lowest number of processes xv6 can 'have' at the same time?*
- 💡 *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/>*

Task 0: running xv6

As always, begin by downloading our revision of xv6, from the os114 **svn** repository:

- Open a shell, and traverse to the desired working directory.
- Execute the following command (in a single line):

```
svn checkout http://bgu-os-142-xv6.googlecode.com/svn/trunk/assignment3
```

This will create a new folder called assignment3 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: Enhanced process details viewer

Prior to updating the memory system of xv6, you will develop a tool to assist in testing the current memory usage of the processes in the system. To do so, enhance the capability of the xv6's process details viewer. Start by running xv6 and then 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?) 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 `^P` so that the memory state of all the processes in the system will also be printed as written below. The memory state includes the mapping from virtual pages to physical pages (for all the used pages), and also information regarding the used page tables.

For each process, the following should be displayed (words in *italics* should be replaced by the appropriate values):

Process information currently printed with ^P

Page tables:

memory location of page directory = *location*

pdir PTE entry number, PPN:

memory location of page table = *location*

ptbl PTE entry number, PPN, memory location of physical page pointed to by PPN

...

ptbl PTE entry number, PPN, memory location of physical page pointed to by PPN

...

pdir PTE entry number, PPN:

memory location of page table = *location*

ptbl PTE entry number, PPN, memory location of physical page pointed to by PPN

...

ptbl PTE entry number, PPN, memory location of physical page pointed to by PPN

Page mappings:
virtual page number -> physical page number
...
virtual page number -> physical page number

For example, in a system with 2 processes, the information should be displayed as follows:

First process

1 sleep init 80104907 80104647 8010600a 80105216 801063d9 801061dd

Page tables:

Page tables:

memory location of page directory = 800000

pdir PTE 1, 63:

memory location of page table = 258048

ptbl PTE 1, 73, 299008

ptbl PTE 2, 80, 327680

Page mappings:

1 -> 300, y, n

200 -> 500, y, y

Second process

2 sleep sh 80104907 80100966 80101d9e 8010113d 80105425 80105216 801063d9 801061dd

Page tables:

[etc etc..]

Page mappings:

1 -> 306

200 -> 500

Showing that the first process has 2 pages. Virtual page number 1 is mapped to physical page number 300 and virtual page number 200 is mapped to physical page number 500. The page directory of the first process is in physical memory address 800000, the first PTE is unused, the second PTE is used and its PPN is 63. This PPN points to physical memory address 258048. That page table has 2 used PTEs (the first is unused). The PPNs of these 2 page table PTEs are 73 and 80, which point to physical memory addresses 299008 and 327680 respectively.

Please notice:

- **The values above are made up.**
- Don't print anything for pages or page tables that are currently unused.
- Only print pages and page tables that are user pages (PTE_U).
- This will be used for the frontal grading.

Task 2: Null-pointer protection

Many software exceptions occur when trying to dereference null points (accessing memory at location 0). While in certain situations it may be desirable to access the memory at address 0, it usually points to a bug in the code. For this reason modern operating systems crash after dereferencing a null pointer, making it quicker to find bugs. XV6, on the other hand, does not offer such protection and allows access to address 0, which contains the program's text segment (go ahead and try to tamper with it).

Your task is to implement such functionality for user programs (and fork_test). To do so, make sure that the first memory page is never used (unallocated), and that the text segment begins at the second page. Find and update all the relevant places. Notice that the Makefile needs to be updated as well, have a look at flag -T.

As a simple sanity test, write a small user program that prints out the contents of memory address 0. While this should work before the changes, when you have completed this task, accessing the memory at that location should cause an error.

Task 3: Protection of read-only segments

The protection described in task 2 is not the only protection missing in XV6. Another common protection is to set segments such as the text segment as read only. An ELF file specifies which segments should be loaded as read only, however, the current makefile in xv6 links programs so that all the different sections have the same program header, and read-write-execute permissions. This stands in contrast to the way programs are usually linked. Use `readelf` to compare executable files built for xv6 and other regular executable files in your Linux operating system.

In this task as well, the Makefile should be updated (along with several other files of course). Have a look at the `-N` linker flag and how it relates to the information described in the previous paragraph. Make sure to handle alignment issues that may arise. Also, be sure not to allow a read-only page to become writeable following a `fork` command. The `readelf` command should prove rather useful.

As a simple sanity test, write a user program that tries to violate the new protection. In order to do so, try and rewrite program commands. Remember that the name of a function in C is actually a pointer. More specifically, the variable `main` is just a pointer to the location of the beginning of the code of function `main`. Use `main` as a pointer and try to read and write new values to that location. Before your change both of these operations should execute well. After adding protection, reading the code of `main` should work, while writing over it should cause an error.

Task 4: Copy on write (COW)

Upon execution of a `fork` command, a process is duplicated together with its memory. The XV6 implementation of `fork` entails copying all the memory pages to the child process, a scheme which may require a long time to complete due to the many memory accesses needed. In a normal program, many memory pages will have no change at all following `fork` (e.g., the text segment), furthermore, many times `fork` is followed by a call to `exec`, in which copying the memory becomes redundant. In order to account for these cases, modern operating systems employ a scheme of copy-on-write, in which a memory page is copied only when it needs to be modified. This way, the child and parent processes share equal memory pages and unneeded memory accesses are prevented.

In this task you are required to implement the COW functionality in XV6. To accomplish this task, add a new system call named `cowfork` that will work like `fork` but will not copy memory pages, and the virtual memory of the parent and child processes will point to the same physical pages. Note that it is ok to copy page tables of a parent process (though you may also share/cow the page tables if you would like to). In order to prevent a change to a shared page (and be able to copy it), render each shared page as read-only for both the parent and the child. Now, when a process would try and write to the page, a page fault will be raised and then the page should be copied to a new place (and the previous page should become writeable once again). In order to be able to distinguish between a shared read-only page and a non-shared read-only page, add another flag to each virtual page to mark it as a shared page.

Notice that a process may have many children, which in turn, also have many children, thus a page may be shared by many processes. For this reason it becomes difficult to decide when a page should become writeable. To facilitate such a decision, add a counter for each physical page to keep track of the number of processes sharing it (it is up to you to decide where to keep these counters and how to handle them properly). Since a limit of 64 processes is defined in the system, a counter of 1 byte per page should suffice.

- When a page fault occurs, the faulty address is written to the cr2 register.
- After copying a page, the virtual to physical mapping for that page has changed, therefore, be sure to refresh the TLB using the following commands:

```
movl    %cr3,%eax
movl    %eax,%cr3
```

Task 5.0: Updating ^P

Now that you have made changes to the system, update the output of ^P (task 0) to include more relevant information. A part of the output you were asked to print is the virtual to physical page mapping:

Page mappings:

virtual page number -> physical page number

...

virtual page number -> physical page number

Now, update this output by adding more information so that it will look as the following:

Page mappings:

virtual page number -> physical page number, is read-only, is cow

...

virtual page number -> physical page number, is read-only, is cow

More specifically, for each page mapping also print the status of the read-only flag (also known as the protection flag), and the cow flag that marks whether the page is shared by several processes (the flag you have created in task 4).

So, the meaning of the following output:

First process

...

Page mappings:

1 -> 300, y, n

200 -> 500, y, y

Second process

...

Page mappings:

1 -> 306, y, n

200 -> 500, y, y

is that the first process has 2 pages. Virtual page number 1 is mapped to physical page number 300, is read only and is not shared with any other process. Virtual page number 200 is mapped to physical page number 500 and is read only and shared (with the second process).

Task 5: Sanity test

To make sure that all tasks have been completed properly write a simple user program and use ^P to see that everything works properly. In this program use the command *malloc* to create new values that sit in different pages (in the heap). Use the normal *fork* command and see that the newly created process copied all the memory pages. Now use the new *cowfork* command instead of the normal *fork* and see that the pages are shared. After the *cowfork* command try and update a variable in the parent process. Was the proper page copied properly?

Such a scenario, together with the information printed by ^P, can be used to check all the tasks above. More specifically, make sure that:

- The first page is never used (task 1).
- The right pages are read-only (task 2).
- Pages shared due to COW are indeed shared, and those that should not be shared are indeed separated. Also, make sure that a page that is shared by several processes (more than 2), after being written to from a process, is copied to a different page for that process and remains shared between the other processes that have not written to it.

Submission guidelines

Assignment due date: 25/5/2014 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 comments – these are often handy when discussing your code with the graders.

Due to our constrained resources, assignments are only allowed in pairs. Please note this important point and try to match up with a partner as soon as possible.

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 changelist and make sure it contains all the changes that you applied and nothing more. Modified files are automatically detected by svn but new files should be added explicitly with the 'svn add' command:

```
>svn add<filename>
```

In case you need to revert to a previous version:

```
>svn revert <filename>
```

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

```
>svn diff
```

Alternatively, if you have a diff utility such as kompare:

```
>svn diff | kompare -o -
```

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

```
>svn diff>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.*

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

Enjoy !!!