

# OPERATING SYSTEMS

## ASSIGNMENT 3

### MEMORY MANAGEMENT

---

#### Introduction

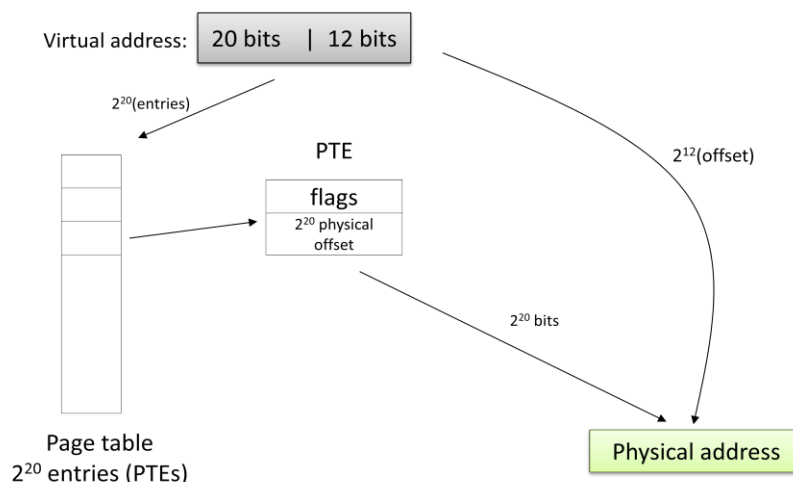
Memory management and memory abstraction is one of the most important features of any operating system. In this assignment we will examine how xv6 handles memory and attempt to extend it.

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

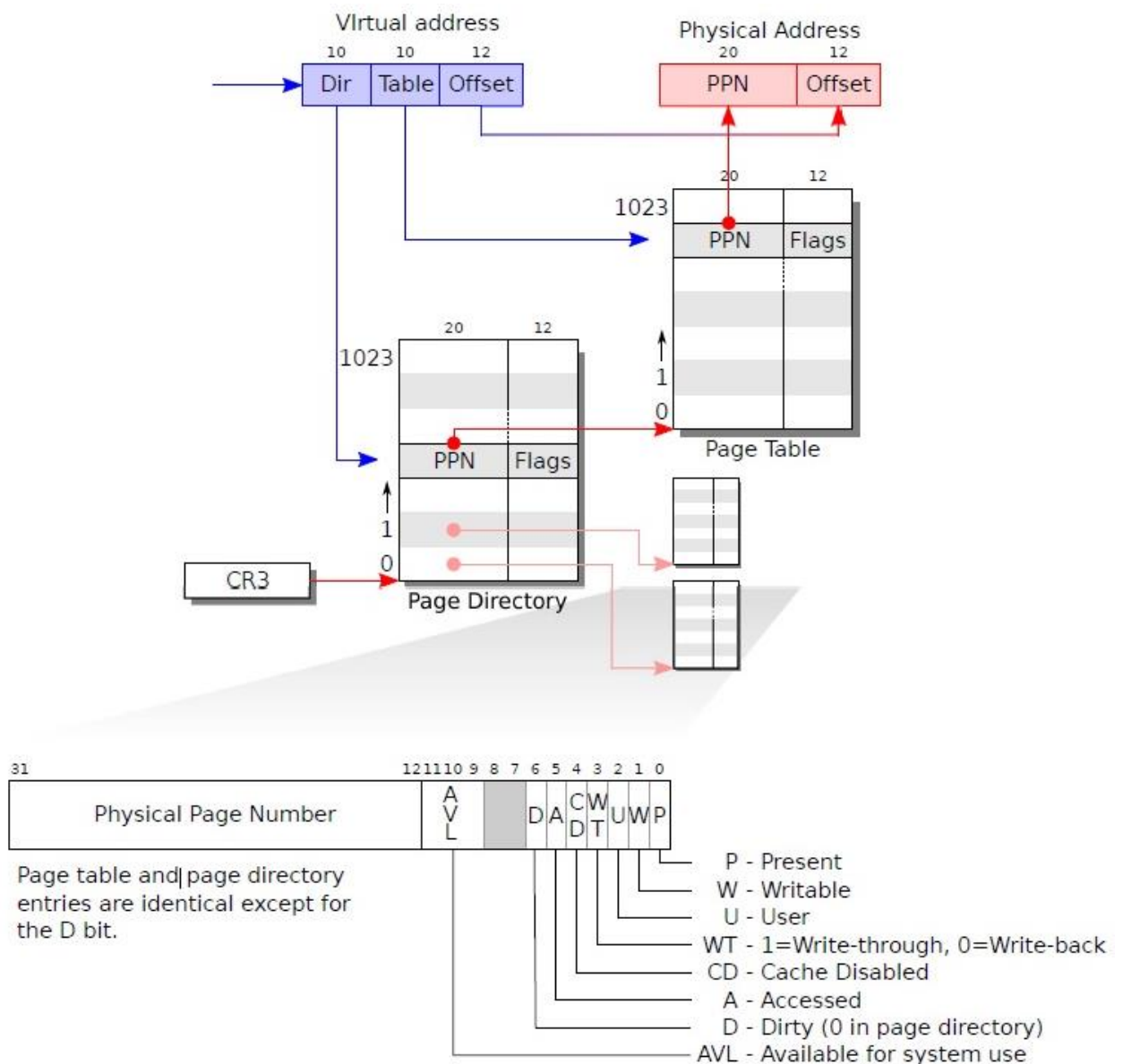
#### *Xv6 memory*

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 point 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 (taken from xv6 - a simple, Unix-like teaching operating system, Russ Cox, Frans Kaashoek, Robert Morris):



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 proceedings 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? Does xv6 memory mechanism limit the number of user processes? If so, what is the lowest number of processes xv6 can 'have' at the same time (assuming the kernel requires no memory whatsoever)?

- 💡 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 os112 **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-132-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: Process swapping**

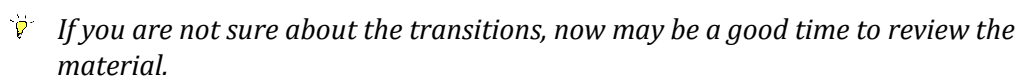
An important feature lacking in xv6 is the ability to swap out pages to a backing store. That is, at each moment in time all processes are held within the main (physical) memory. In this task, you are to implement a swapping mechanism for xv6 which is capable of swapping out pages and storing these to disk.

To keep things simple, we will swap entire processes pages in and out of memory. Each swapped out process pages will be saved on a dedicated file whose name is the same as the process' pid (e.g., "<pid>.swap").

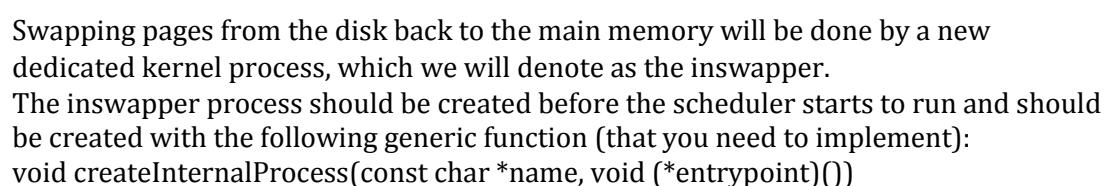
- 💡 *Note: there are good reasons for not writing to files from within kernel modules but in this assignment we ignore these.*
- 💡 For a few "good reasons", read this: <http://www.linuxjournal.com/article/8110>, <http://kernelnewbies.org/FAQ/WhyWritingFilesFromKernelsIsBad>
- 💡 And finally, if you really want to understand how swapping is done: <http://www.kernel.org/doc/gorman/pdf/understand.pdf>

In current XV6 version processes can be in one of three (Running, Runnable & Sleeping) states from the moment they are created until they are terminated (there are actually more states, but we are interested only in those when the process is "alive").

The following state diagram displays the process states:



We start by expanding process' states. We will add two more states which we call "sleeping\_suspended" and "runnable\_suspended". A process being blocked should be immediately swapped out by the kernel to the disk and have his state changed to sleeping\_suspended. When a sleeping\_suspended process is no longer blocked, his state should change to runnable\_suspended. At this point, having a runnable state, the process can be selected by the scheduler to run. Nevertheless, it cannot run because its memory is swapped out (stored at the disk and not at the main memory). Thus, it first must be swapped back to the main memory and only then it can be scheduled to run. The following state diagram displays the new states and transitions:



The following high level algorithms describe the swap in and out:

Swap out - once a process is blocked, the kernel executes:

1. Create a swap file for the selected process
2. for each user page of the process' pages:
  - 2.1. Write the current page to the swap file
3. Free the user pages

💡 *Tip: be sure to write and free 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: <http://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf>).*

💡 *Tip: writing / reading data to disk causes a process to sleep. Xv6 disk driver append the read / write request to the disk requests queue and wait for the request to finish. You should prevent infinite recursion from occurring (process sleep → swap out → sleep ...). In order to solve this problem, change the disk driver so it would carry out busy waiting instead of sleeping.*

💡 *Tip: When a process is swapped out its state cannot change (from sleeping to runnable or from suspended sleeping to suspended runnable) prior to the completion of the swapping out operation.*

Swap in - executed by the inswapper:

1. Open the relevant swap file
2. for each page in the file
  - 2.1. Allocate a new page and copy its content from the file
  - 2.2. Map the page to the page table in the proper location
3. Remove the swap file

💡 *Tip: there are several functions already implemented in xv6 that can assist you- reuse existing code.*

💡 *Tip: while there are no Runnable suspended processes, the inswapper should not be scheduled to run (no busy waiting!).*

### **Task 1.2: Enabling & disabling Swapping:**

In order to compare the OS performance with / without swapping, you need to implement two system calls:

- enableSwapping
- disableSwapping

The implementation is up to you, bear in mind that a swapped out process should always be able to be swapped back in (assuming there is enough memory).

## **Task 2: Shared memory**

In order to coordinate their activities, processes must communicate with each other. There are several ways processes can communicate. One way of achieving IPC (Inter Process Communication), as you learned in class, is through signals. Another way, already implemented in XV6 is through pipes. In this task you will implement additional mechanism which will enable processes to communicate through shared memory.

💡 *Tip: The internet is abundant with process shared memory explanation and examples. Before proceedings we strongly suggest you go over some of them to gain a better understanding of the subject.*

To make things simple your implementation will be very naïve and will only support some basic shared memory features:

1. Allocating (and freeing) shared memory
2. Linking (and delinking) shared memory to a process

### **Task 2.1: Allocating (and freeing) shared memory**

You should add the following system calls to your OS:

1. `int shmget(int key, uint size, int shmflg);`

`shmget()` returns the identifier of the shared memory segment associated with the value of the argument `key`.

The value `shmflg` is composed of:

`CREAT` - to create a new shared memory segment, with size equal to the value of `size` rounded up to a multiple of page size. If `shmflg` specifies `CREAT` and a shared memory segment already exists for `key`, then `shmget()` fails and returns -1.

`GET` - to find the segment associated with `key`. If `shmflg` specifies `GET` and a shared memory segment not exists for `key`, then `shmget()` fails and returns -1.

2. `Int shmdel(int shmid)`

If the shared memory mapped to `shmid` is not linked to any other process, the kernel free the previously allocated frames and return the number of frames freed. Otherwise, it returns -1.

### **Task 2.2: 2. Linking (and delinking) shared memory to a process**

You should add the following system calls to your OS:

1. `void *shmat(int shmid, int shmflg);`  
`shmat()` attaches the shared memory segment identified by `shmid` to the address space of the calling process. The system chooses a suitable (unused) address at which to attach the segment.

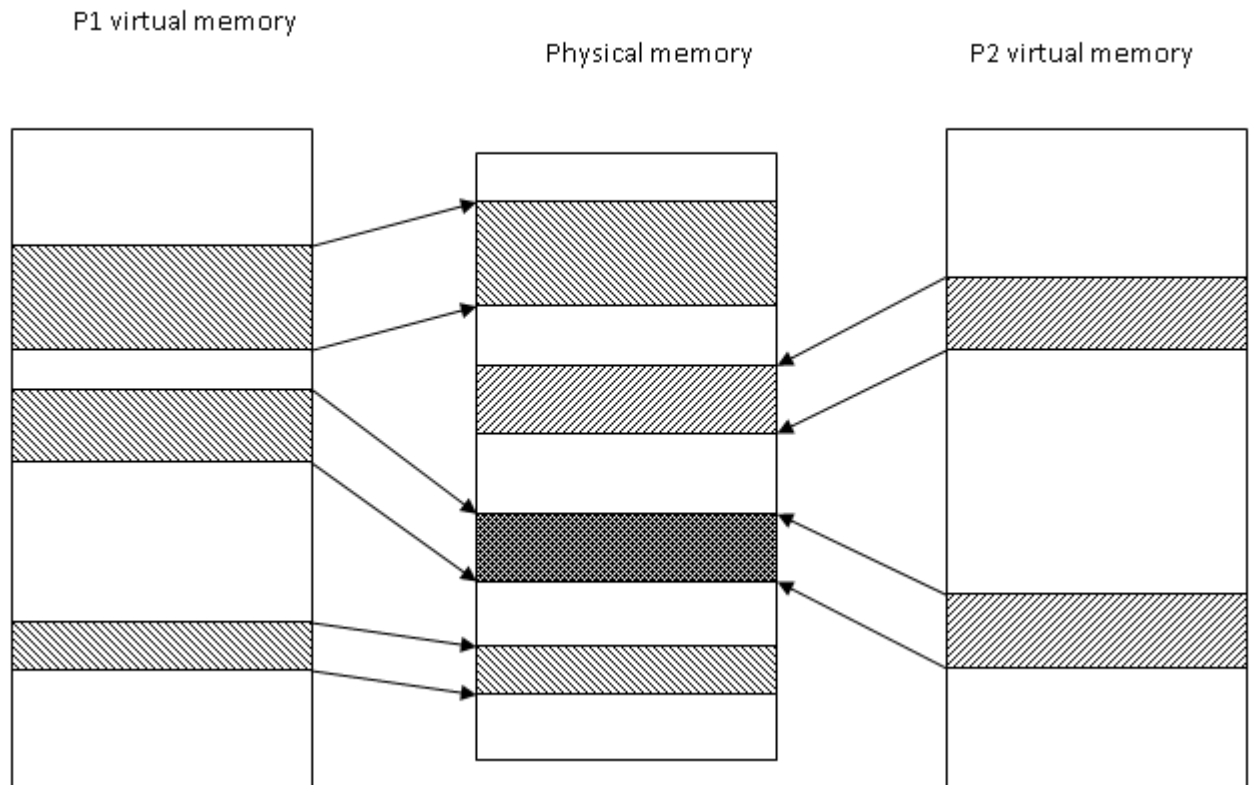
If `SHM_RDONLY` is specified in `shmflg`, the segment is attached for reading only. If `SHM_RDWR` is specified in `shmflg`, the segment is attached for read and write. There is no notion of a write-only shared memory segment.

On success `shmat()` returns the address of the attached shared memory segment. Otherwise -1 is returned.

2. `int shmdt(const void *shmaddr);`  
`shmdt()` detaches the shared memory segment located at the address specified by `shmaddr` from the address space of the calling process. The to-be-detached segment must be currently attached with `shmaddr` equal to the value returned by the attaching `shmat()` call.  
On success `shmdt()` returns 0. Otherwise -1 is returned.

- 💡 *Tip:* in order to implement the shared memory mechanism the kernel must maintain some related data.
- 💡 Shmdel description, as written above, is not the right way to implement a shared memory segment deletion (why?). Nonetheless, in order to keep it simple, you will implement it this way.
- 💡 Before implementing, think carefully in which scenarios shmat / shmdt must return -1.
- 💡 We encourage you to go over the man pages of these (or related) functions in order to gain a better understanding.

A conceptual view of shared memory:



### Task 3: User space scenarios

In this section you will add an application which tests the shared memory and swapping frameworks.

Write 2 programs that will convince the grader your implementation achieves:

Program 1: named **swaptest.c**

1. Blocked processes are swapped out.
2. After being not blocked anymore they are swapped back in.
3. Only the process's private data is swapped.
4. Swapping can be activated / deactivated.

Program 2: named **sharetest.c**

1. Processes can share memory.
2. Only the process's private data is swapped.

You can choose whatever scenarios you see fit, they do not have to be long nor complex. Please use exact names. The Grader may replace your code with his own code.

## Submission guidelines

Assignment due date: 6/6/2013 22:00

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.

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 change list and make sure it contains all the changes that you applied and noting 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.***

*Good luck!*