OPERATING SYSTEMS, ASSIGNMENT 3
MEMORY MANAGEMENT

Responsible TAs: Dolav Nitay, Or Dinari

Introduction

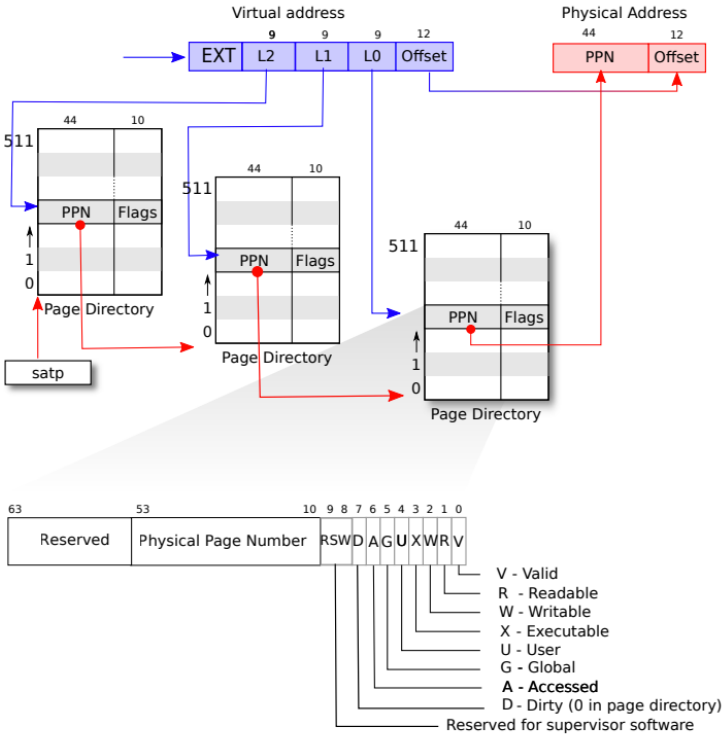
Memory management is one of the key features of every operating system. In this assignment, we will examine how xv6 handles memory and attempt to extend it by implementing a paging infrastructure which will allow xv6 to store parts of the process' memory in secondary storage.

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, memlayout.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 user space addresses to physical addresses, all processes share the same page table for the kernel.

In xv6 riscv, only the bottom 39 bits of a 64-bit virtual address are used, using a triple paging mechanism. The first 9 bits are *pte (Page Table Entry)* index in the process page table, the second 9 bits are the *pte* index in the second page table, followed by the 9 bits for the index in the last page table, the last 12 bits in the VA are the offset in the physical page.

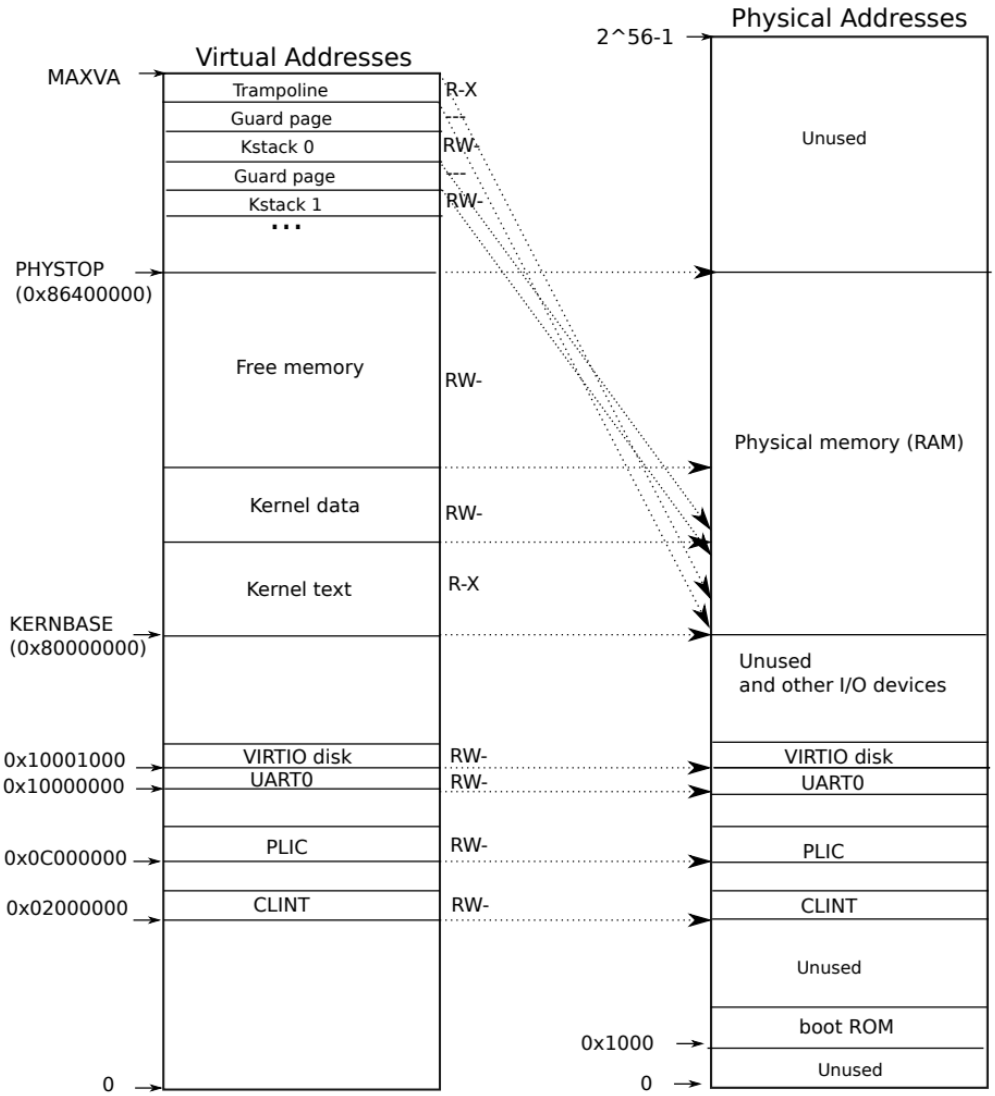


(Figure 1 taken from xv6-riscv book)

Each *pte* is 64 bits (8 bytes) long, and each page table contains 512 *pte's* , in each *pte* the first 10 bits are reserved and not used, followed by 44 bits which are the physical page number, and 10 flag bits (see the above figure).

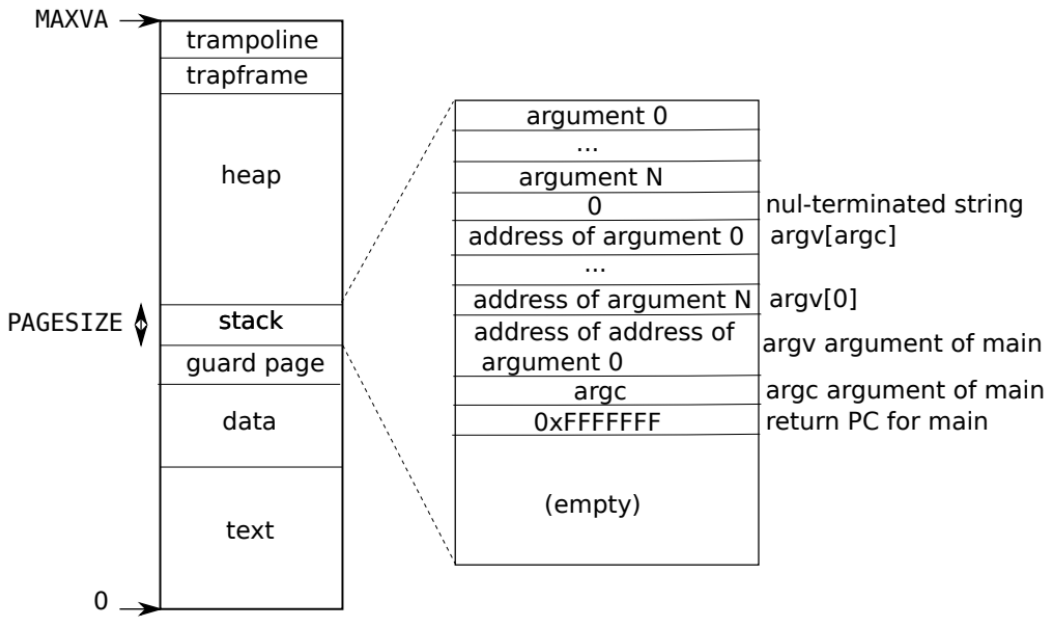
The xv6 kernel uses direct mapping, that is, kernel VA is also it's PA (Note that this does not overwrite the paging system), direct mapping simplifies the kernel code when handling memory. There are however some exceptions: The trampoline page is in the top of the VA (*MAXVA)*, and just below it are each process kernel stack pages, each page followed by a guard page, which serves as a barrier between the different stack pages.

The kernel memory starts at *KERNBASE*, and apart from the above exceptions, continues until *PHYSTOP*.



(Figure 2 taken from xv6-riscv book)

While in principle each process virtual memory is limited by *MAXVA* (256GB), in xv6 runtime allocations occur between the kernel data segment, and *PHYSTOP*, there exists double mapping - When a process asks the kernel to allocate memory, the memory is allocated from the kernel free list using *kalloc*, thus you can access the memory from the process page directory after the allocation, and from the kernel using its direct memory mapping.



(Figure 3 taken from xv6-riscv book)

The trampoline page location is constant in both the kernel and user space, however in the userspace the trampoline page and below it the process trapframe. in VA 0 is the process text segment, followed by the data segment, a guard page and the stack.

Task 0: running xv6

Begin by downloading our revision of xv6, from the os ***git*** repository:

- Open a shell, and traverse to the desired working directory.
- Execute the following command (in a single line):
  ```
  > git clone https://github.com/BGU-CS-OS/os212-assignment3
  ```
  This will create a new folder called os212-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: 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, the kernel implements a local paging policy for each process separately (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 really want to understand how VM is done: http://www.kernel.org/doc/gorman/pdf/understand.pdf*

1.0– Supplied file framework

We supplied a framework for creating, writing, reading, and deleting swap files. The framework was implemented in *fs.c* and used a new parameter named *swapFile* that was added to the proc struct (*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 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.

Note that if *fileOffset* == *n*, the file size should be at least *n*.

1.1 – 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 (= 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, the kernel must select (see Task 2) enough pages and move them to a dedicated file for this process.

- These restrictions are necessary since xv6's file system can generate only small size 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/6.828/2020/xv6/book-riscv-rev1.pdf*.)*
- *Tip: there are several functions already implemented in xv6 that can assist you – **reuse existing code**.*
- *Tip: when swapped out, don't forget to free the page's physical memory.*
- *Remember xv6 double memory mapping, a page's physical address is its virtual address when accessing it from the kernel space.*

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 valid (PTE_V) flag.

A cleared present flag does not imply that a page was swapped out (there could be other reasons for this flag being reset). To resolve this issue, we use one of the available flag bits (see Figure 1) in the page table entry to indicate that the page was indeed paged out. Add the following line to riscv.h:

~~`#define PTE_PG (1L << 10) // Paged out to secondary storage`~~

```
#define PTE_PG (1L << 10) // 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. The TLB is refreshed when moving between user space and kernel space, via the method `sfence_vma()`.

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 a trap. Use *r_scause()* to determine the reason for the trap (should be either 13 or 15 for pagefault), and `r_stval()` to determine the faulting address and identify the page. Check the PTE to identify if the page was paged out, a valid page not yet allocated (see next section) or if this is just a segmentation fault.

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****.*

1.3 – 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 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 yet learn to handle files in the kernel.

Task 2: Page replacement schemes

Now that you have a paging framework, 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 remained without visits, 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 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 (as learned in class): 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. 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 3: 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. Your program should allocate some memory and then use it. Check your program with the different page replacement algorithms you created in Task 3.

- *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.*

Task 4 (Bonus): Implementing Lazy Allocation

When a process asks the kernel for heap space, by using *malloc* or directly *sbrk*, it will not necessarily use the entire allocated amount of memory, examples for this could be sparse arrays, pre-allocated buffers. In this task you are required to implement lazy

memory allocation. Using lazy memory allocation, a process heap memory will only be allocated on the first access to it.

For example, using *malloc* to allocate a 16KB array of chars (4 pages) will increase the process size by 16KB, however it will not allocate any heap memory. The memory will only be allocated when the process tries to access this array, and even then, only the necessary pages will be allocated. Accessing the array at an address in the third page will cause the kernel to allocate the third page only.

In order to implement lazy allocation, you will need to make the following modifications:

1. Disable the memory allocation when calling *sbrk*, however you should still modify the process size!
2. Catch pagefaults in *trap.c* correctly, examine the *usertrap* function, when *r_scause()* is either 13 or 15 it is a pagefault. Note that after handling a pagefault, if *p->killed = 0*, the process should simply try again to access the faulting page after returning to user space. After catching a pagefault you should allocate the required page, and update the relevant *pte*. You can examine what happens when calling *growproc* to see the required actions.
3. Make sure *sbrk* works with negative size, e.g. it is able to correctly deallocate memory.
4. Examine the process lifecycle, specifically, its creation and its end, make sure memory is being allocated and deallocated correctly.

When all the tasks are done, you should be able to pass the “lazytests” when the paging framework is disabled (SELECTION=NONE, see previous section).

Submission guidelines

Assignment due date: 27/05 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!