OPERATING SYSTEMS, ASSIGNMENT 3
MEMORY MANAGEMENT

Responsible TAs: Or Dinari

Introduction

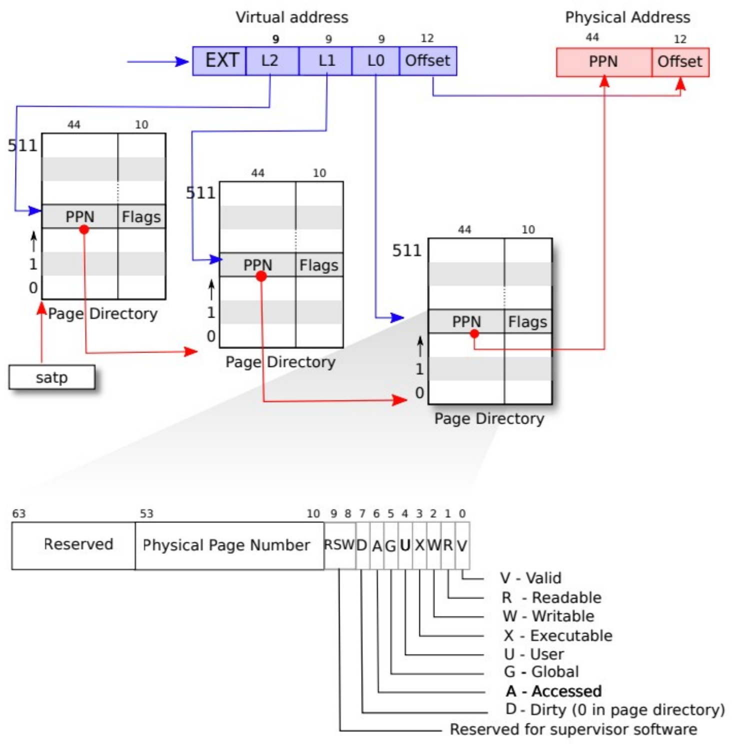
Memory management is one of the key features of every operating system. In this assignment, we will examine how xv6 handles memory and will extend it by implementing Copy-On-Write (COW).

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 3-level 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.

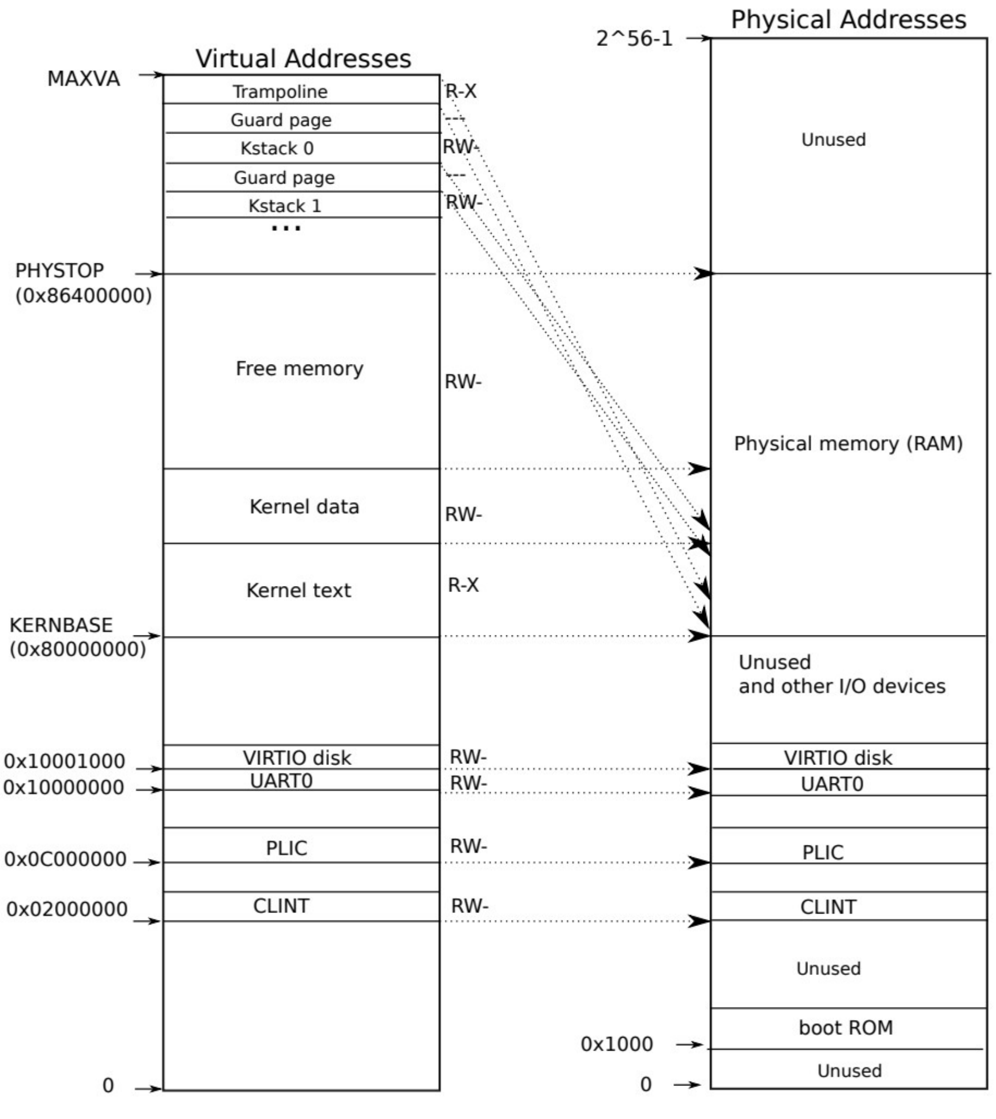


(Figure 1 taken from xv6-riscv book)

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).

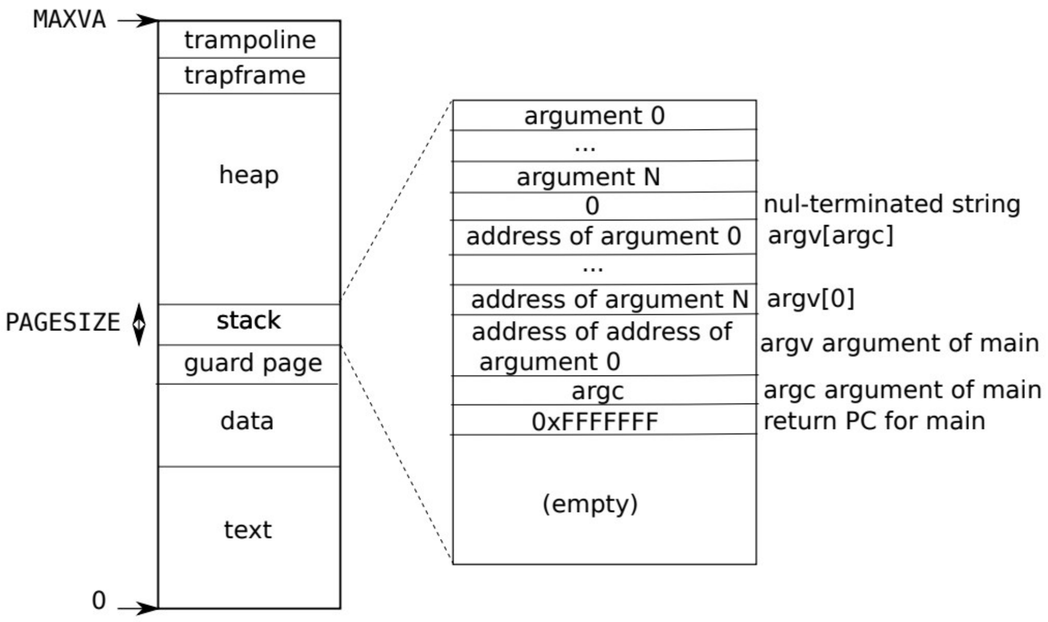
The xv6 kernel uses direct mapping, that is, kernel VA is also its PA (note that the direct mapping still used the paging system), direct mapping simplifies the kernel code when handling memory. There are however some exceptions: The trampoline page is in the top of the VA (*MAXVA)*, and just below it 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 (see figure 2)*, 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 the same in both the kernel and user space, followed by the trapframe (in the user space). 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/mit-pdos/xv6-riscv.git
  ```
  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: *Copy-On-Write*

The fork system call in xv6 copies the entire memory image of the calling 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 typically 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 we don't copy the pages during the fork call, but instead, we mark each page as read-only and as COW.

If an attempt to write to such a read-only page occurs, you need to catch the protection-fault and create a writable copy of the page for the process that caused the page fault.

Add the following definition to *riscv.h*:

#define PTE_COW (1L << 9) // copy-on-write

You will need to modify the user trap such that in the case of a pagefault, it will be handled explicitly by a function you will provide. The releveant *r_cause()* are 13 and 15.

In order to avoid memory leaks, you should maintain a reference count per each pyhsical page (which was used in COW), and when the refrence drop to 0, make sure you free the page.

In order to keep such reference counts, you should create an array with an entry for each physical page, make sure when updating this array to make it synchronized, you should use CAS from the previous assignment.

The number of the physical pages that can be dynamically alloacted can be calculated using the following formula:

#define NUM_PYS_PAGES ((PHYSTOP-KERNBASE) / PGSIZE)

You will write most of your code in *vm.c*, follow the path of process creation to see when and where the memory is copied. Also, you should pay special attetion to the times when memory is freed.

Task 2: Testing

To test your assignment, you should be able to pass the usertests with no errors, this will also be used to test it during the frontal check.

Submission guidelines

Assignment due date: 09/06 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 moodle. 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
```

Good luck!