CS 333: Operating Systems Lab
Autumn 2022
Lab5: Memory Management in xv6

In this lab, we explore how xv6 does memory management.

Before You Begin:

- **Setting up xv6 (part of lab4)**
  - Follow the instructions given **here** for the xv6 installation. xv6 runs on an x86 emulator called QEMU that emulates x86 hardware on your local machine. In the xv6 folder, run the following command sequence
  - **make**
    make qemu or make qemu-nox
    to boot xv6 and open a shell.
- For this lab, you will need to understand the following files: **syscall.c, syscall.h, sysproc.c, user.h, usys.S, vm.c, proc.c, trap.c, defs.h, mmu.h, kalloc.c**
  - The files **sysproc.c, syscall.c, syscall.h, user.h, usys.S** link user system calls to system call implementation code in the kernel.
  - **mmu.h** and **defs.h** are header files with various useful definitions pertaining to memory management.
  - The file **vm.c** contains most of the logic for memory management in the xv6 kernel, and proc.c contains process-related system call implementations.
  - The file **trap.c** contains trap handling code for all traps including page faults.
  - Understand the implementation of the **sbrk** system call that spans all of these files.
- Download, read and use as reference the xv6 source code companion book.
  - https://pdos.csail.mit.edu/6.828/2017/xv6/book-rev10.pdf (page no - 29 to 35)
- The xv6 OS book is here
  - https://pdos.csail.mit.edu/6.828/2017/xv6/xv6-rev10.pdf

Part1: Displaying memory information

1. Implement a new system call **`freememstat`** that will print the available system memory (in bytes) in xv6. Specifically, it should have the following interface:
   `int freememstat(void);`
   It takes no arguments and returns the amount of memory available in the system.

   A user-level program freememtestcase.c is provided which takes size(in bytes) as a command line argument and allocates the physical memory of the given size, calls the new system call(i.e. freememstat) and prints the result on qemu-terminal (assume that the number of bytes is a positive number and is a multiple of page size). Add _freememtestcase to the UPROGS and freememtestcase.c to the EXTRA definition in Makefile if it is not already there.

Sample Output:

```
$ freememtestcase
Usage: freememtestcase size(in bytes)

$ freememtestcase 0
Available memory: 232611840

$ freememtestcase 4096
Available memory: 232607744

$ freememtestcase 8192
Available memory: 232603648
```

Hint: To count up the available system memory, you should walk the linked list used by the memory allocator(kmem.freelist) and count how many pages are still available on that list.
You may find the kalloc function in kalloc.c helpful. Also, look up the init2 function and the PHYSTOP variable.

2. Next, we want to print memory information for an ***active*** process in the xv6 system (either in embryo, running, runnable, sleeping, or zombie states). For that, you should implement another system call **`void getmeminfo(int pid)`** that prints the stats shown below.

   a. *Size of virtual address space*
   the number of virtual/logical pages in the user part of the address space of the process, up to the program size stored in struct proc. You must count the stack guard page as well in your calculations.

   b. *Size of allocated physical address space*
   the number of physical pages in the user part of the address space of the process. You must count this number by walking the process page table, and counting the number of page table entries that have a valid physical address assigned.

   c. *Page Table Size*
   the number of the page table pages(i.e. directory pages and the page table pages).

`void getmeminfo(int pid)`

If **`pid`** is greater than 0 print details for specified **`pid`**, if it equals zero print stats for all pids (i.e., all currently active processes) and error message in case of invalid pid (i.e., pid less than 0) or failure.

Note: xv6 does not use **demand paging** by default, you can expect the number of virtual and physical pages to be the same initially. However, part 2 of this lab will change this property.

Also, you may want to implement functionality by adding details of one stat at a time to the system call.

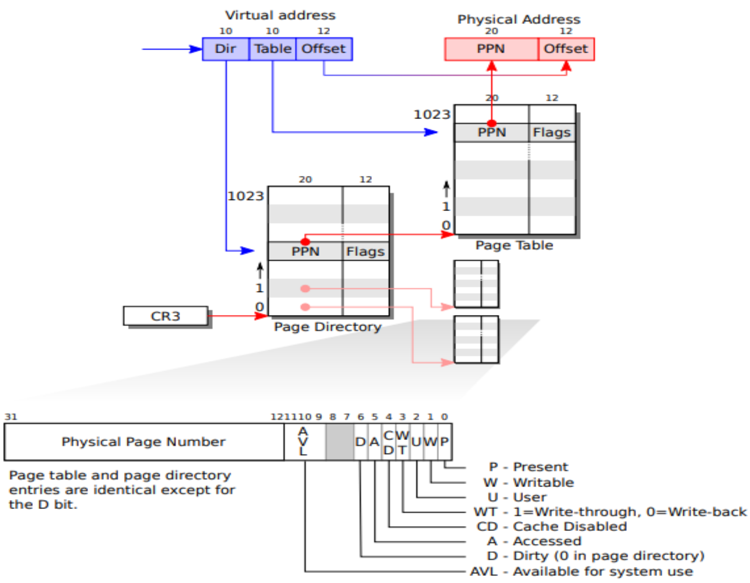

Figure 2-1. x86 page table hardware.

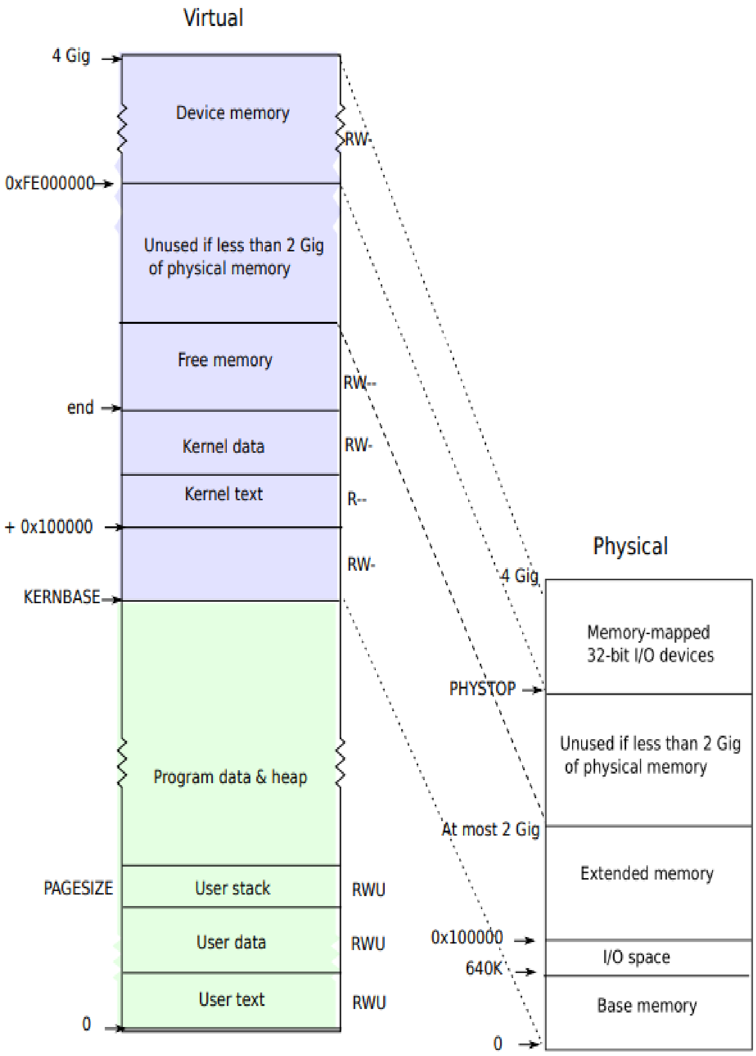

Figure 2-2. Layout of the virtual address space of a process and the layout of the physical address space. Note that if a machine has more than 2 Gbyte of physical memory, xv6 can use only the memory that fits between KERNBASE and 0xFE00000.

Hints:

A. You can use cprintf for printing in kernel mode.

B. To iterate over all active processes in the xv6 system(i.e when pid equals 0 in the getmeminfo argument) and print their information to the screen, you should iterate over ptable. Look up the code for the kill function in proc.c to understand how to iterate over ptable.

C. To count up the virtual pages in the user part of the memory, check struct proc declaration and also the PAGESIZE constant.

D. You can walk the page table of the process by using the walkpgdir function which is present in vm.c. You can look up loaduvm and deallocuvm in vm.c to see how to invoke the walkpgdir function. To compute the number of physical pages in a process, you can write a function that walks the page table of a process in vm.c and invoke this function from the system call handling code.

E. xv6 has a 2-level page table organization. You need to calculate the size of the page table (total level 0 and level 1 pages). You need to iterate over the Page Directory Entries (PDEs) to check if a page is assigned for storing Page Table Entries (PTEs) for that PDE.

Note: It is important to keep in mind that the process table struct ptable is protected by a lock. You must acquire the lock before accessing this structure for reading or writing and must release the lock after you are done.

You are provided with test-meminfo1.c and test-meminfo2.c to test your implementation.

Sample Output:

```
init: starting sh
$ test-meminfo1
*Case1: invalid pid*
Invalid pid: -1
-----------------------------------------------------
*Case2: pid = 0*
pid: 1, name: init
Memory usage in pages || Virtual: 3 | Physical: 3
Page Table Size in pages: 66
pid: 2, name: sh
Memory usage in pages || Virtual: 4 | Physical: 4
Page Table Size in pages: 66
pid: 3, name: test-meminfo1
Memory usage in pages || Virtual: 3 | Physical: 3
Page Table Size in pages: 66
$
```

```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2
init: starting sh
$ test-meminfo2
*Case3: specified valid pid(pid > 0)[getpid() > 0]*
----------------------------------------------------
Memory information before sbrk system call
pid: 3, name: test-meminfo2
Memory usage in pages || Virtual: 3 | Physical: 3
Page Table Size in pages: 66
----------------------------------------------------
Memory information after sbrk system call
pid: 3, name: test-meminfo2
Memory usage in pages || Virtual: 2051 | Physical: 2051
Page Table Size in pages: 68
$
```

Part2: Sometimes it's ok to be lazy

Implement a simple version of the **mmap** system call in xv6. The **mmap** system call should take one argument: the number of bytes to add to the ***size*** of the process. The process size in this context refers to the heap size.

You may assume that the number of bytes is a positive number and is a multiple of the page size. The system call should return a value of 0 if any invalid inputs are provided.

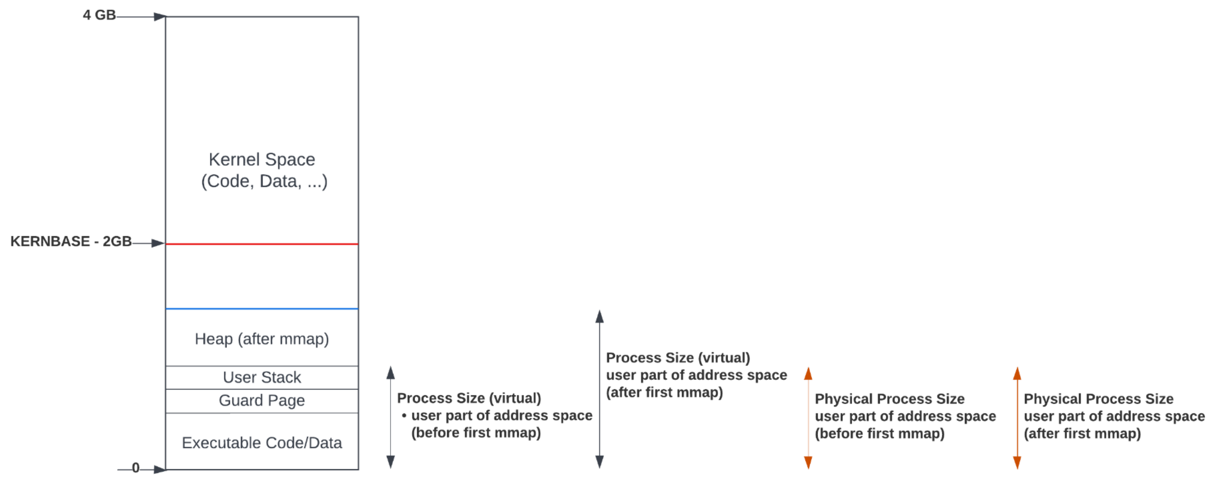

Figure 3: memory image of a user program (before and after first mmap)

In the valid case, the system call should expand the process's size by the specified number of bytes, and return the starting virtual address of the newly added memory region.

However, the system call should **NOT** allocate any physical memory corresponding to the new virtual pages, as we will allocate memory on demand. When the user accesses a memory-mapped page, a page fault will occur, and physical memory should only be allocated as part of the page fault handling.

Step 1: mmap() system call, similar to sbrk() but should not call growproc()

Understand the implementation of the sbrk system call. mmap() system call will follow a similar logic. The sbrk(n) system call is implemented in the function sys_sbrk() in sysproc.c allocates physical memory and maps it into the process's virtual address space. The sbrk(n) system call grows the process's memory size by n bytes, and then returns the start of the newly allocated region (i.e., the old size). Your new mmap(n) should only increment the process's size (myproc()->sz) by n and return the old size. It should not allocate memory—so you shouldn't invoke the growproc() (but you still need to increase the process's size! The implementation of sbrk() invokes the growproc function).

Step 2: Lazy Allocation

The original version of xv6 does not handle the page fault trap. For this assignment, you must write extra code to handle the page fault trap in **trap.c**, which will allocate memory on demand for the page that has caused the page fault return from the trap handler, so that the process can access the virtual address originally accessed. You can check whether a trap is a page fault by checking if tf->trapno is equal to T_PGFLT. Once you write code to handle the page fault, do break or return in order to avoid the processing of other traps.

Note: Now, you will need to understand how xv6 gets to the faulting virtual address.

Some helpful hints:

- Look at the arguments to the cprintf statements in **trap.c** to figure out how one can find the virtual address that caused the page fault.
- Use PGROUNDDOWN(va) to round the faulting virtual address down to the start of a page boundary.
- You may invoke allocuvm (or write another similar function) in **vm.c** in order to allocate physical memory upon a page fault.
- Once you correctly handle the page fault, do break or return in order to avoid the cprintf and the proc->killed = 1 statement.
- You will need to call mappages() from **trap.c** in order to map the newly allocated page. In order to do this, you'll need to delete the static in the declaration of mappages() in **vm.c**, and you'll need to declare mappages() in the **trap.c**. Add this declaration to the **trap.c** before any call to mappages(): int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
- You should check whether the page fault was actually due to a lazy allocated page or an actual page fault (For example - illegal memory access)

Note: it is important to call switchuvm to update the CR3 register and TLB every time you change the page table of the process. This update to the page table will enable the process to resume execution when you handle the page fault correctly

A user program test-mmap-partial.c(step 1 only) and test-mmap.c(complete implementation) is provided to test your implementation.

Sample Output:

- ***Partial implementation (Step 1 only)***

```
init: starting sh
$ test-mmap-partial
Initial memory information
pid: 3, name: test-mmap-parti
Memory usage in pages || Virtual: 3 | Physical: 3
Page Table Size in pages: 66
--------------------Partial Testcase (implemented only step1)------------------
mmap failed for wrong inputs(i.e. -1234)
mmap failed for wrong inputs(i.e. 1234)
----------------------------------------------------
After mmap one page
pid: 3, name: test-mmap-parti
Memory usage in pages || Virtual: 4 | Physical: 3
Page Table Size in pages: 66
pid 3 test-mmap-parti: trap 14 err 6 on cpu 0 eip 0xdb addr 0x3000--kill proc
$
```

- ***Complete Implementation (Both step 1 and step 2)***

```
init: starting sh
$ test-mmap
Initial memory information
pid: 3, name: test-mmap
Memory usage in pages || Virtual: 3 | Physical: 3
Page Table Size in pages: 66
----------------------------------------------------
mmap failed for wrong inputs(i.e. -1234)
mmap failed for wrong inputs(i.e. 1234)
----------------------------------------------------
After mmap one page
pid: 3, name: test-mmap
Memory usage in pages || Virtual: 4 | Physical: 3
Page Table Size in pages: 66
After access of one page
pid: 3, name: test-mmap
Memory usage in pages || Virtual: 4 | Physical: 4
Page Table Size in pages: 66
----------------------------------------------------
After mmap two pages
pid: 3, name: test-mmap
Memory usage in pages || Virtual: 6 | Physical: 4
Page Table Size in pages: 66
After access of first page
pid: 3, name: test-mmap
Memory usage in pages || Virtual: 6 | Physical: 5
Page Table Size in pages: 66
After access of second page
pid: 3, name: test-mmap
Memory usage in pages || Virtual: 6 | Physical: 6
Page Table Size in pages: 66
$
```

Submission Instructions

- All submissions to be done on moodle only.
- Name your submission as **<rollnumber>_lab5.tar.gz** (e.g **190050096_lab5.tar.gz**)
- The tar should contain the following files in the following directory structure:
  <rollnumber>_lab5/

  |_ _< all modified files in xv6 such as

  syscall.c, syscall.h, sysproc.c, user.h, usys.S, proc.c, trap.c, defs.h, kalloc.c, vm.c, . . . >

  |_ _Makefile

  Please adhere to it strictly.
- Your modified code/added code should be well commented on and readable.
- tar -czvf <rollnumber>_lab5.tar.gz <rollnumber>_lab5

Deadline: Monday 09th September Friday 2022, 06:30 PM via moodle.