CS 333: Operating Systems Lab
Autumn 2023
Lab5: dipping in the pensieve

This lab explores xv6's memory management magic.

To setup xv6 refer to the installation guide.

Following is a partial list of important files for the lab:
syscall.c, syscall.h, sysproc.c, user.h, usys.S, vm.c, proc.c, trap.c, defs.h, mmu.h, memlayout.h, kalloc.c

- **sysproc.c, syscall.c, syscall.h, user.h, usys.S** link user system calls to system call implementation code in the kernel.
- **mmu.h, memlayout.h** and **defs.h** are header files with various useful definitions pertaining to memory management.
- **vm.c** contains most of the logic for memory management in the xv6 kernel, and **proc.c** contains process-related system call implementations.
- **trap.c** contains trap handling code for all traps including memory access related exceptions (page faults).

Task1: what is your address?—virtual and physical !

1. is the virtual address space real? 🔢

Write a system call **getvasize()** that returns the size of the virtual memory used by a process. Specifically, the system call should have the following interface:

int getvasize(int pid); **//** pid is argument to the call and amount of virtual memory
// used by the process as return value.

Hints:
(i) Look up and understand implementation of the **sbrk** system call in **proc.c**.
Also, check the **struct proc** data structure in **proc.h**
(ii) Refer to Sheet 38 of **xv6 source code booklet** for sbrk() system call in xv6.
(iii) Refer to discussion on Page 34 of the xv6 book .
(iv) To count up the virtual address space in the user part of the memory, check **struct proc** declaration and also the **PAGESIZE** constant.

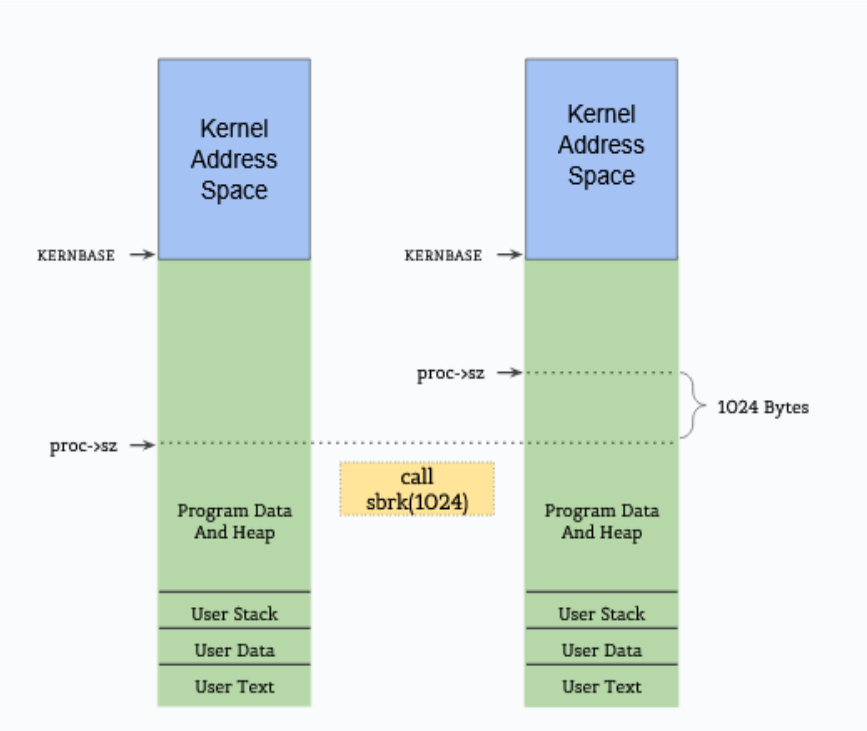

Figure 1. After calling **sbrk(1024)** to increase the process size by 1024.

The **sbrk**(n) system call is implemented in the function **sys_sbrk**() in sysproc.c that 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).

A sample program **t_getvasize.c** is available for testing.

```
$ t_getvasize
Pid of the process is 4
Size of process:          12288 Bytes
Address returned by sbrk: 0x3000
Size of process:          13312 Bytes
Address returned by sbrk: x03000
```

---

2. What is your postal address?

Write a system call **va2pa** that returns the virtual address to physical address mapping from the page table of the current process. Specifically, the system call should have the following interface:

uint va2pa(uint virtual_addr); // virtual address is the argument
// corresponding physical address is the return value

Hints:
(i) Lookup and understand the **walkpgdir() function** and understand usage of this function in the system calls implemented in **vm.c**
(ii) Refer to Sheet 17 of xv6 source code booklet for sbrk() system calls in xv6.

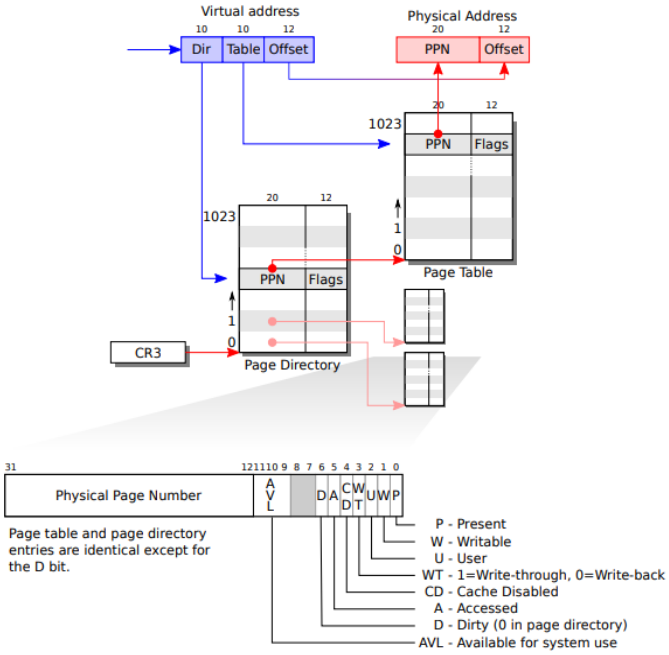

Figure 1: Page table layout

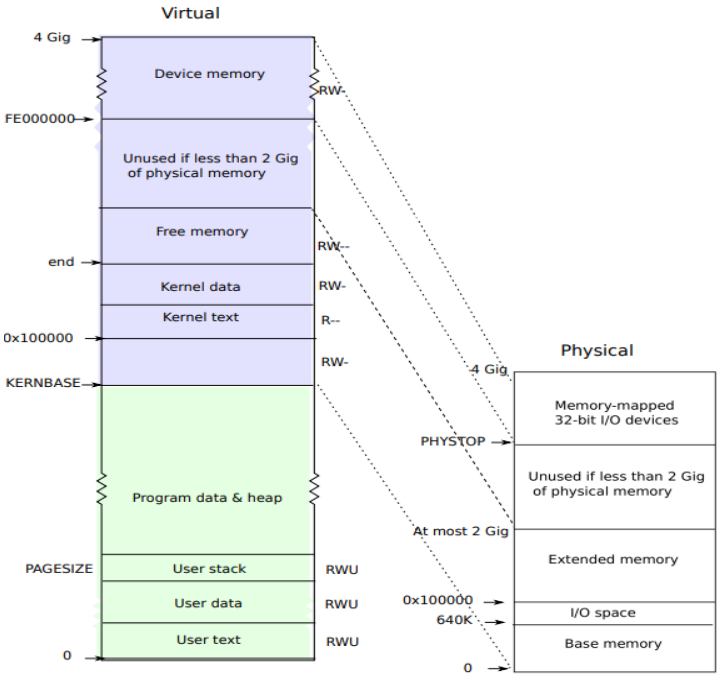

Figure 2: Virtual to physical address mapping

(iii) Refer to discussion on Page 29-32 of [the xv6 book](#).

Figure 1. A page table is stored in physical memory as a **two-level tree.** The root of the tree is a 4096-byte page directory that contains 1024 PTE references to page table pages. Each page table page is an array of 1024 32-bit PTEs. The paging hardware uses the top 10 bits of a virtual address to select a page directory entry. If the page directory entry is present, the paging hardware uses the next 10 bits of the virtual address to select a PTE from the page table page that the page directory entry refers to. **If either the page directory entry or the PTE is not present, the paging hardware raises a fault.**

Figure 2. A process's address space starts at virtual address zero and can grow up to **KERNBASE**, allowing a process to address up to 2 GB of memory. The file **memlayout.h** declares the constants for xv6's memory layout, and macros to convert virtual to physical addresses. When a process asks xv6 for more memory, xv6 first finds **free physical pages** from the free page list and then adds PTEs to the process's page table that point to the new physical pages. xv6 sets the **PTE_U**, **PTE_W**, and **PTE_P** flags in these PTEs. xv6 includes all mappings needed for the **kernel** to run in every process's page table; these mappings all appear above **KERNBASE**.

Use the above information to traverse the page table of a process and convert virtual address to physical address.

Sample programs **t_va2pa.c** and **t_va2pa2.c** are provided for testing.

Sample usage:

```
$ t_va_to_pa1
Physical Address of user   virtual address 2FCC is DF24FCC
Physical Address of Kernel virtual address 80100400 is 100400
Physical Address of Kernel virtual address 80100800 is 100800
Physical Address of Kernel virtual address 80101000 is 101000
Physical Address of Kernel virtual address 80101448 is 101448
```

```
$ t_va_to_pa2
Virtual address of Var N in child    2FCC
Physical address on Var N in child   DF1FFCC
Virtual address of Var N in parent   2FCC
Physical address on Var N in parent  DF76FCC
```

Q. Can you see what is interesting in the two outputs?

---

3. enter the page table!

Implement the following system calls to get details of the page table of a process.

int get_pgtb_size();
The system call has no arguments and returns the number of page table pages allocated to the current process.

int get_usr_pgtb_size();
The system call has no arguments and returns the number of page table pages allocated for user space memory for the current process.

int get_kernel_pgtb_size();
The system call has no arguments and returns the number of page table pages allocated for kernel space memory for the current process. **Recall** kernel pages are mapped for virtual addresses above **KERNBASE.**

A user-level program t_**getpgtablesz.c** is provided for testing. This program will call all the above system calls before and after multiple sbrk() system calls.

Hint:
Walk the page table of a process by using the **walkpgdir** function and consider only those entries which indicate mapping that are present (the present bit is set).

Sample usage

```
$ t_getpgtablesz
Page Table Size 65 Pages
User Pages Table Size 1 Pages
kernel Pages Table Size 64 Pages
------------Doing sbrk-----------
Page Table Size 69 Pages
User Pages Table Size 5 Pages
kernel Pages Table Size 64 Pages
```

---

4. no escape from reality!

Next, report the physical memory (in pages) allocated for a process via a system call

int getpasize(int pid)

The call takes pid as an argument and prints the number of physical pages mapped to the virtual addresses of a process (process virtual addresses).

NOTE: Count the number of mapped pages by walking the process page table and counting the number of page table entries that have a valid physical address assigned.

You are provided with **t_getpasize.c** for testing,

Sample usage

```
$ t_getpasize
Virtual Address Space of process: 3 Pages
Number of physical pages allocated to process: 3 Pages
Virtual Address Space of process after sbrk: 14 Pages
Number of physical pages allocated to process after sbrk: 14 Pages
```

We will use these system calls to test your implementation of Task2 of this lab.

Hints:
(i) 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.

(ii) 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.

Task2: the scam revealed 🛌

Step 1: faulting page, page faulting, who is handling?

The default xv6 distribution does not handle the page fault trap explicitly.
Extended implementation of trap handler function in **trap.c** to explicitly handle a page fault. The handler should print details of the page fault — pid of the process and faulting address which was accessed for the trap.
The page fault trap defined in traps.h is **T_PGFLT**.

Refer to Sheet 34 of [xv6 source code booklet] for trap() handler in xv6.

Sample program **t_pagefault.c** is provided for testing.

Sample usage

```
$ t_pagefault
Pid of the process is 8
Simulating a page_fault on addr 4000
```

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.
- Once you correctly handle the page fault, do break or return in order to avoid the cprintf and the add proc->killed = 1 statement.

Step 2: mmap() 🗺️

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. The mmap call grows the size of the process (virtual) address space and expects a mapped physical address. ***However, mapping from a virtual to physical address is required only when the virtual address is accessed!***.

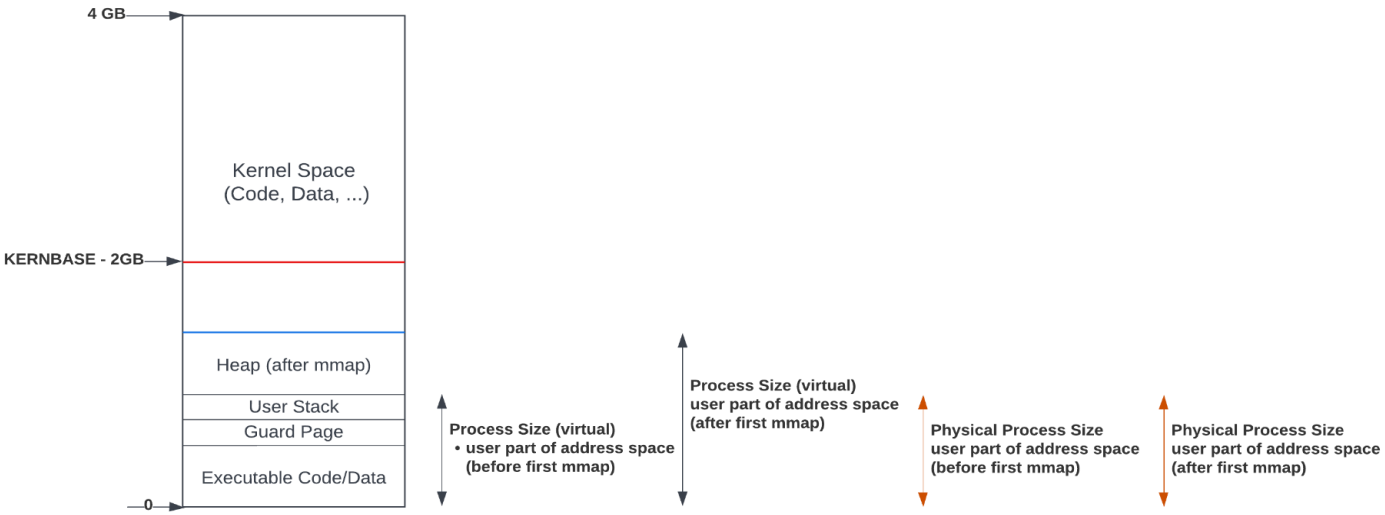

Figure 3: Virtual and physical addresses before and after mmap().

The figure shows the working of mmap system call. when user call say mmap(1024) the virtual address space of the process increases however the physical address space remains the same.

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.

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. 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 (lazy page allocation and mapping) … this Step 3.

Hints:
(i) **mmap**() system call is similar to **sbrk**() (with code related to memory allocation and mapping pages … kalloc, **growproc**, allocuvm, mappages etc.)
(ii) Understand the implementation of the **sbrk** system call and **mmap**() system call will follow a similar logic.
(iii) Refer to Sheets 19, 25, 38 of **xv6 source code booklet** for the related system calls.
(iv) Refer to Page 34 of the xv6 book.

Source file **t_mmap.c** is provided for testing.

Sample usage:

```
$ mmap
Process size before mmap 12288 Bytes
Process size after mmap 16384 Bytes
Accessing a address allocated by mmap 3F80
Pagefault occured at address eip 0x6a addr 0x3f80--kill proc
```

Step 3: the big reveal

Next, modify the page fault handler logic, to allocate memory on demand for the page (need to check if the page faulting address is a valid address!). Once a physical page is allocated and mapped for the virtual address being accessed, the handler returns and the access is re-attempted and should not result in a page fault.
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.
- You can add your page fault handler in vm.c and call it from trap.c.
- 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 t_lazy.c is provided for testing.

Sample usage:

```
VAS: 3 Pages
PAS: 3 Pages
------Mapping 10 Pages-----
VAS: 6 Pages
PAS: 3 Pages
------Accessing Page 0-----
VAS: 6 Pages
PAS: 4 Pages
------Accessing Page 1-----
VAS: 6 Pages
PAS: 5 Pages
------Accessing Page 2-----
VAS: 6 Pages
PAS: 6 Pages
```

Submission Instructions

- Submission is via moodle.
- Name your submission as <rollnumber>_lab5.tar.gz,
  e.g., 190050096_lab5.tar.gz
- The tar should contain the all the files in the xv6_public directory structure:
  - Run **make clean**
  - tar -czvf 190050096_lab5.tar.gz xv6-public/
- Your modified code/added code should be well commented and readable.

Deadline: 08th September Friday 2023, 11:59 PM via moodle.

Task3 : to be lazy is human, to share is humane! (Optional task)

note: solving this question is optional, rest is not optional.

xv6 does not support shared memory by default in this Part. We would like you to implement a mechanism for Shared Memory. For this part we want you to implement Shared Memory support for xv6. You will need to implement 4 System Calls namely

`uint attach_shm(uint key);` → Allocated one page shared memory identified by KEY to process address space.

`uint create_shm(uint key);` → Attaches Shared memory identified by KEY to process add space.

`int detach_shm(uint key);` → Deattaches Shared memory from process add space.

`int destroy_shm(uint key);` → Destroy Shared memory identified by KEY

The program in **t_shm_client.c** is provided for testing.

```
$ shm_client
Creating Shared Memory with key 10
Writing Data Into Shared Memory From Parent: Systems are vital to WORLD PEACE !!
Accessing Shared Memory In Child Process: Systems are vital to WORLD PEACE !!
Detached Shared Memory From Child Address Space
Destoryed Shared Memory From Parent
```

Hints:

(i) need to maintain some form of table in the kernel in order to keep track of all the shared memory identified by key and when you do call attach this table can be used to give you PA associated with the shared memory identified by key.

(ii) need to call **mappages**() from trap.c in order to map the physical address in shared memory table to virtual address for the shared memory in attach_shm() .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);
