OPERATING SYSTEM

PROJECT 3 – Page tables

1. General rule:

- The project is done in groups: each group has a maximum of **2 students**

- **The same exercises will all be scored 0 for the entire practice (even though there are scores for other exercises and practice projects).**

- Programming environment: **Linux**

2. Submission:

Submit assignments directly on the course website (MOODLE), not accepting submissions via email or other forms.

Filename: **StudentID1_StudentID2.zip** (with StudentID1 < StudentID2)

Ex: Your group has 2 students: 2312001 and 2312002, the filename is: **2312001_2312002.zip**

Include:

- **StudentID1_StudentID2_Report.pdf:** Writeups should be short and sweet. Do not spend too much effort or include your source code on your writeups. The purpose of the report is to give you an opportunity to clarify your solution, any problems with your work, and to add information that may be useful in grading. If you had specific problems or issues, approaches you tried that didn't work, or concepts that were not fully implemented, then an explanation in your report may help us to assign partial credit
- **Release**: File diff (diff patch, Ex: $ git diff > **<StudentID1_StudentID2>**.patch)
- **Source**: Zip file of xv6 (the version is made clean)

Lưu ý: Cần thực hiện đúng các yêu cầu trên, nếu không, bài làm sẽ không được chấm.

3. Demo Interviews

Your implementation is graded on completeness, correctness, programming style, thoroughness of testing, your solution, and code understanding.

When administering this course, we do our best to give a fair assessment to each individual based on each person's contribution to the project

4. Requirements

In this lab you will explore page tables and modify them to implement common OS features.

Before you start coding, read Chapter 3 of the xv6 book, and related files:

- kernel/memlayout.h, which captures the layout of memory.
- kernel/vm.c, which contains most virtual memory (VM) code.
- kernel/kalloc.c, which contains code for allocating and freeing physical memory.
- It may also help to consult the RISC-V privileged architecture manual.

To start the lab, switch to the pgtbl branch:

```
$ git fetch
$ git checkout pgtbl
$ make clean
```

4.1. Inspect a user-process page table

To help you understand RISC-V page tables, your first task is to explain the page table for a user process.

Run make qemu and run the user program pgtbltest. The print_pgtbl functions prints out the page-table entries for the first 10 and last 10 pages of the pgtbltest process using the pgpte system call that we added to xv6 for this lab. The output looks as follows:

```
va 0 pte 0x21FCF45B pa 0x87F3D000 perm 0x5B
va 1000 pte 0x21FCE85B pa 0x87F3A000 perm 0x5B
...
va 0xFFFFD000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFE000 pte 0x21FD80C7 pa 0x87F60000 perm 0xC7
va 0xFFFFF000 pte 0x20001C4B pa 0x80007000 perm 0x4B
```

For every page table entry in the print_pgtbl output, explain what it logically contains and what its permission bits are. Figure 3.4 in the xv6 book might be helpful, although note that the figure might have a slightly different set of pages than the process that's being inspected here. Note that xv6 doesn't place the virtual pages consecutively in physical memory.

4.2. Speed up system calls

Some operating systems (e.g., Linux) speed up certain system calls by sharing data in a read-only region between userspace and the kernel. This eliminates the need for kernel crossings when performing these system calls. To help you learn how to insert mappings into a page table, your first task is to implement this optimization for the getpid() system call in xv6.

When each process is created, map one read-only page at USYSCALL (a virtual address defined in memlayout.h). At the start of this page, store a struct usyscall (also defined in memlayout.h), and initialize it to store the PID of the current process. For this lab, ugetpid() has been provided on the userspace side and will automatically use the USYSCALL mapping. You will receive full credit for this part of the lab if the ugetpid test case passes when running pgtbltest.

Some hints:

- Choose permission bits that allow userspace to only read the page.
- There are a few things that need to be done over the lifecycle of a new page. For inspiration, understand the trapframe handling in kernel/proc.c.

  Which other xv6 system call(s) could be made faster using this shared page? Explain how.

4.3. Print a page table

To help you visualize RISC-V page tables, and perhaps to aid future debugging, your second task is to write a function that prints the contents of a page table.

We added a system call kpgtbl(), which calls vmprint() in vm.c. It takes a pagetable_t argument, and your job is to print that pagetable in the format described below.

```
page table 0x0000000087f22000
 ..0x0000000000000000: pte 0x0000000021fc7801 pa 0x0000000087f1e000
 .. ..0x0000000000000000: pte 0x0000000021fc7401 pa 0x0000000087f1d000
 .. .. ..0x0000000000000000: pte 0x0000000021fc7c5b pa 0x0000000087f1f000
 .. .. ..0x0000000000001000: pte 0x0000000021fc70d7 pa 0x0000000087f1c000
 .. .. ..0x0000000000002000: pte 0x0000000021fc6c07 pa 0x0000000087f1b000
 .. .. ..0x0000000000003000: pte 0x0000000021fc68d7 pa 0x0000000087f1a000
 ..0xffffffffc0000000: pte 0x0000000021fc8401 pa 0x0000000087f21000
 .. ..0xffffffffffe00000: pte 0x0000000021fc8001 pa 0x0000000087f20000
 .. .. ..0xfffffffffffff000: pte 0x0000000021fd4c13 pa 0x0000000087f53000
```

```
.. .. ..0xffffffffffffe000: pte 0x0000000021fd00c7 pa 0x0000000087f40000
.. .. ..0xffffffffffffe000: pte 0x000000002000184b pa 0x0000000080006000
```

The first line displays the argument to vmprint. After that there is a line for each PTE, including PTEs that refer to page-table pages deeper in the tree. Each PTE line is indented by a number of " .." that indicates its depth in the tree. Each PTE line shows the PTE index in its page-table page, the pte bits, and the physical address extracted from the PTE. Don't print PTEs that are not valid. In the above example, the top-level page-table page has mappings for entries 0 and 255. The next level down for entry 0 has only index 0 mapped, and the bottom-level for that index 0 has entries 0, 1, and 2 mapped.

Your code might emit different physical addresses than those shown above. The number of entries and the virtual addresses should be the same.

Some hints:

- You can put vmprint() in kernel/vm.c.
- Use the macros at the end of the file kernel/riscv.h.
- The function freewalk may be inspirational.
- Define the prototype for vmprint in kernel/defs.h so that you can call it from exec.c.
- Use %p in your printf calls to print out full 64-bit hex PTEs and addresses as shown in the example.

For every leaf page in the vmprint output, explain what it logically contains and what its permission bits are. Figure 3.4 in the xv6 book might be helpful, although note that the figure might have a slightly different set of pages than the init process that's being inspected here.

4.4. Use superpages

The RISC-V paging hardware supports two-megabyte pages as well as ordinary 4096-byte pages. The general idea of larger pages is called superpages, and (since RISC-V supports more than one size) 2M pages are called megapages. The operating system creates a superpage by setting the PTE_V and PTE_R bits in the level-1 PTE, and setting the physical page number to point to the start of a two-megabyte region of physical memory. This physical address must be two-mega-byte aligned (i.e., a multiple of two megabytes). You can read about this in the RISC-V privileged manual by searching for megapage and superpage; in particular, the top of page 112. Use of superpages decreases the amount of physical memory used by the page table, and can decrease misses in the TLB cache. For some programs this leads to large increases in performance.

Your job is to modify the xv6 kernel to use superpages. In particular, if a user program calls sbrk() with a size of 2 megabytes or more, and the newly created address range includes one or more areas that are two-megabyte-aligned and at least two megabytes in size, the kernel should use a single superpage (instead of hundreds of ordinary pages). You will receive full credit for this part of the lab if the superpg_test test case passes when running pgtbltest.

Some hints:

- Read superpg_test in user/pgtbltest.c.
- A good place to start is sys_sbrk in kernel/sysproc.c, which is invoked by the sbrk system call. Follow the code path to the function that allocates memory for sbrk.
- Your kernel will need to be able to allocate and free two-megabyte regions. Modify kalloc.c to set aside a few two-megabyte areas of physical memory, and create superalloc() and superfree() functions. You'll only need a handful of two-megabyte chunks of memory.
- Superpages must be allocated when a process with superpages forks, and freed when it exits; you'll need to modify uvmcopy() and uvmunmap().

Real operating systems dynamically promote a collection of pages to a superpage.

5. Grade

No.	Exercise	Grade
1	Inspect a user-process page table	2
2	Speed up system calls	2
3	Print a page table	2
4	Use superpages	4

6. Reference

- https://pdos.csail.mit.edu/6.1810/2024/labs/pgtbl.html
- Navarro et al., *Practical, transparent OS support for superpages*, SIGOPS, 2002.