OPERATING SYSTEM

PROJECT 3 – Page Tables

1. General rule:

- The project is done in groups: each group has a maximum of **3 students**

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

- Môi trường lập trình: **Linux**

2. Submission:

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

Filename: **StudentID1_StudentID2_StudentID3.zip** (with StudentID1 < StudentID2 < StudentID3)

Ex: Your group has 3 students: 2312001, 2312002 and 2312003, the filename is: **2312001_2312002_2312003.zip**

Include:

- **StudentID1_StudentID2_StudentID3_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_StudentID3>**.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 speed up certain system calls and to detect which pages have been accessed.

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
```

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.

> Define a function called vmprint(). It should take a pagetable_t argument, and print that pagetable in the format described below. Insert if(p->pid==1) vmprint(p->pagetable) in exec.c just before the return argc, to print the first process's page table. You receive full credit for this part of the lab if you pass the pte printout test of make grade.

Now when you start xv6 it should print output like this, describing the page table of the first process at the point when it has just finished exec()ing init:

```
page table 0x0000000087f6b000
```

```
 ..0: pte 0x0000000021fd9c01 pa 0x0000000087f67000
 .. ..0: pte 0x0000000021fd9801 pa 0x0000000087f66000
 .. .. ..0: pte 0x0000000021fda01b pa 0x0000000087f68000
 .. .. ..1: pte 0x0000000021fd9417 pa 0x0000000087f65000
 .. .. ..2: pte 0x0000000021fd9007 pa 0x0000000087f64000
 .. .. ..3: pte 0x0000000021fd8c17 pa 0x0000000087f63000
 ..255: pte 0x0000000021fda801 pa 0x0000000087f6a000
 .. ..511: pte 0x0000000021fda401 pa 0x0000000087f69000
 .. .. ..509: pte 0x0000000021fdcc13 pa 0x0000000087f73000
 .. .. ..510: pte 0x0000000021fdd007 pa 0x0000000087f74000
 .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
```

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.

Detect which pages have been accessed

Some garbage collectors (a form of automatic memory management) can benefit from information about which pages have been accessed (read or write). In this part of the lab, you will add a new feature to xv6 that detects and reports this information to userspace by

inspecting the access bits in the RISC-V page table. The RISC-V hardware page walker marks these bits in the PTE whenever it resolves a TLB miss.

> Your job is to implement pgaccess(), a system call that reports which pages have been accessed. The system call takes three arguments. First, it takes the starting virtual address of the first user page to check. Second, it takes the number of pages to check. Finally, it takes a user address to a buffer to store the results into a bitmask (a datastructure that uses one bit per page and where the first page corresponds to the least significant bit). You will receive full credit for this part of the lab if the pgaccess test case passes when running pgtbltest.

Some hints:

- Read pgaccess_test() in user/pgtbltest.c to see how pgaccess is used.
- Start by implementing sys_pgaccess() in kernel/sysproc.c.
- You'll need to parse arguments using argaddr() and argint().
- For the output bitmask, it's easier to store a temporary buffer in the kernel and copy it to the user (via copyout()) after filling it with the right bits.
- It's okay to set an upper limit on the number of pages that can be scanned.
- walk() in kernel/vm.c is very useful for finding the right PTEs.
- You'll need to define PTE_A, the access bit, in kernel/riscv.h. Consult the RISC-V privileged architecture manual to determine its value.
- Be sure to clear PTE_A after checking if it is set. Otherwise, it won't be possible to determine if the page was accessed since the last time pgaccess() was called (i.e., the bit will be set forever).
- vmprint() may come in handy to debug page tables.

5. Grade

No.	Exercise	Grade
1	Speed up system calls	2
2	Print a page table	3
3	Detect which pages have been accessed	5

6. Reference

- https://pdos.csail.mit.edu/6.1810/2023/labs/pgtbl.html