CS3500: Operating Systems

Lab-3: Memory Management

19th August 2025

Page Table Exploration Lab: Introduction

In this lab, you will explore page tables in `xv6` and modify them to implement common operating system features.

Before You Start Coding:

- **Read Chapter 3 of the xv6 book**. This chapter is essential for understanding page tables and virtual memory in xv6.
- Review the following xv6 source files:
  - `kernel/memlayout.h` — captures the layout of memory.
  - `kernel/vm.c` — contains most of the virtual memory (VM) code.
  - `kernel/kalloc.c` — contains code for allocating and freeing physical memory.
- For deeper understanding, consult the RISC-V privileged architecture manual.

Lab Setup

To start the lab, clone into the following repo and switch to the `pgtbl` branch:

```
$ git clone git://g.csail.mit.edu/xv6-labs-2024
$ git fetch
$ git checkout pgtbl
$ make clean
```

- **You would need to start the docker with this new repo folder and not the old one)**. Refer to the Lab-1 handout for further reference.
- **IMPORTANT:** Replace `user/pgtbltest.c`, grade-lab-pgtbl, and the Makefile with the ones provided to you in "pgtbl_lab_util.zip"
- In case of any other issues, refer to an existing system call or function or test case from the initial version you cloned and make the necessary changes in the relevant files as suited to your needs in each question. This will help you gain a better understanding of the xv6 architecture.
- You can run the testcases yourself by running `make qemu` followed by `pgtbltest`. It will run tests for all the questions and will print "OK" or "FAIL" for each of them

---

1 Page Table Tree Printing in xv6

The goal of this task is to implement a function in `xv6` that prints the contents of a page table. The function should display each page table entry (PTE) along with its corresponding physical address. Entries should be indented to reflect the page table hierarchy across different levels.Only valid mappings should be printed. This functionality will help in understanding how virtual addresses are translated and how user processes are mapped to physical memory.

Steps:

1. Define the function in `kernel/vm.c`:

   ```
   void vmprint(pagetable_t pt);
   ```

2. Insert a call to `vmprint` in `exec.c` to print the page table for the first user process.
3. Add the function prototype to `defs.h`.

Output Requirements:

- The first line should print the address of the `pagetable_t` argument.
- For each mapped entry, print:
  - Index in page directory (PTE index).
  - Page table entry value (`pte`).
  - Corresponding physical address (`pa`).
- Indicate the page table level using dot notation:
  - Top-level entries are preceded by ".."
  - Entries in lower levels are indicated by additional dots (e.g., ".. ..", "...", etc.)
- Skip printing entries that aren't mapped.
- The output should look like:

```
page table 0x0000000087f22000
 ..0: pte 0x0000000021fc7801 pa 0x0000000087f1e000
 .. ..0: pte 0x0000000021fc7401 pa 0x0000000087f1d000
 .. .. ..0: pte 0x0000000021fc7c5b pa 0x0000000087f1f000
 .. .. ..1: pte 0x0000000021fc70d7 pa 0x0000000087f1c000
 .. .. ..2: pte 0x0000000021fc6c07 pa 0x0000000087f1b000
 .. .. ..3: pte 0x0000000021fc68d7 pa 0x0000000087f1a000
 ..255: pte 0x0000000021fc8401 pa 0x0000000087f21000
 .. ..511: pte 0x0000000021fc8001 pa 0x0000000087f20000
 .. .. ..509: pte 0x0000000021fd4013 pa 0x0000000087f50000
 .. .. ..510: pte 0x0000000021fd48c7 pa 0x0000000087f52000
 .. .. ..511: pte 0x000000002000184b pa 0x0000000080006000
```

Note: The particular "pte"/"pa" values may differ

Hints:

- Use the macros at the end of the file kernel/riscv.h
- Use recursion and hierarchical walking, similar to the `freewalk` function in `vm.c`.
- Use %p in your printf calls to print out full 64-bit hex PTEs and addresses as shown in the example
- Do not print unmapped entries.

2 Optimizing the `getpid()` System Call in `xv6` via Page Table Sharing

Modern operating systems (e.g., Linux) often speed up specific system calls by sharing data through read-only memory regions accessible from both userspace and the kernel. This technique avoids repeated kernel crossings, significantly improving performance for calls that only retrieve static information.

This task is to implement such an optimization for the `getpid()` system call in `xv6`:

1. When each process is created, map a single read-only page at `USYSCALL`, a virtual address defined in `memlayout.h`. At the beginning of this mapped page, store a `struct usyscall` (also defined in `memlayout.h`), initializing it with the PID of the current process.
2. The provided userspace function `ugetpid()` will automatically read the PID via the `USYSCALL` mapping, eliminating the need for a kernel trap.
3. Make the necessary changes in `proc.c`. Do not forget to add she usyscall structure in the relevant location in proc.h

Hints:

- Choose appropriate page table permission bits to ensure the shared page is *read-only* for userspace.
- Consider the lifecycle management of this shared page: allocation, mapping, initialization, and deallocation.
- For inspiration, study how the kernel handles `trapframe`s for new processes in `kernel/proc.c`.

---

3 Detecting Accessed Pages: Implementing `pgaccess()` in `xv6`

Enhance `xv6` by adding a new system call that detects and reports which user pages have been accessed, by inspecting the access bits in RISC-V page table entries (PTEs).

3.1 System Call Specification

You must implement `pgaccess()`, a system call that reports which pages have been accessed. Make the necessary changes to implement this syscall in the relevant files (use last lab as reference). The system call should take three arguments:

1. The starting virtual address of the first user page to check.
2. The number of user pages to examine.
3. A user address to a buffer, where results are written as a bitmask. Each bit of this bitmask indicates whether the corresponding page has been accessed (the least significant bit represents the first page).

3.2 Implementation Hints

- Review `pgaccess_test()` given below (and add it in `user/pgtbltest.c`) to understand how `pgaccess` is invoked from user-space.
- Begin by implementing `sys_pgaccess()` in `kernel/sysproc.c`. You will need to parse system call arguments using `argaddr()` and `argint()`.
- For the bitmask output, it is easiest to fill a temporary buffer in the kernel and then copy it out to user memory with `copyout()` after the bitmask has been constructed.

- Setting an upper limit on the number of pages that can be scanned is reasonable and encouraged for kernel robustness.
- Use the `walk()` function in `kernel/vm.c` to help locate the relevant page table entries (PTEs).
- You need to define the access bit macro, `PTE_A`, in `kernel/riscv.h`. Consult the RISC-V privileged architecture manual to determine its value.
- After checking the access bit, make sure to clear `PTE_A`. Otherwise, future calls to `pgaccess()` will always see the bit set and thus cannot distinguish recently accessed pages.
- The `vmprint()` function can be useful for debugging your page table management code and confirming correct access bit handling.

IMPORTANT

Following is the C-code that performs the testing for “pgaccess”. Refer this to gain better understanding of what is required from this task. (full file provided in the zip)

```
// Page Table Lab - Q3
void
pgaccess_test()
{
  char *buf;
  unsigned int abits;
  printf("pgaccess_test starting\n");
  testname = "pgaccess_test";
  buf = malloc(32 * PGSIZE);
  if (pgaccess(buf, 32, &abits) < 0)
    err("pgaccess failed");
  buf[PGSIZE * 1] += 1;
  buf[PGSIZE * 2] += 1;
  buf[PGSIZE * 30] += 1;
  if (pgaccess(buf, 32, &abits) < 0)
    err("pgaccess failed");
  if (abits != ((1 << 1) | (1 << 2) | (1 << 30)))
    err("incorrect access bits set");
  free(buf);
  printf("pgaccess_test: OK\n");
}
```

Further add the line “int pgaccess(void *base, int len, void *mask);” in `user/user.h` and “entry(”pgaccess”);” to `user/usys.pl`

NOTE: ATTEMPT THIS QUESTION AFTER SOLVING THE PREVIOUS QUESTIONS

Submission

- You are required to zip and submit the full folder and a report highlighting the changes you made and the codes you added for each question in the relevant files. Note that only your contributions/modifications to the code are required
- In the report, very briefly explain the logic/reasoning of the code you implemented for each question. We are looking for the core concept and not a line-by-line explanation of your code.