

VNUHCM - UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY



LAB REPORT

COURSE: OPERATING SYSTEM LAB 2: PAGE TABLES

Instructor:
Le Viet Long
Pham Tuan Son

Students:

Le Ho Dan Anh	23127318	23CLC08
Le Trung Kien	23127075	23CLC08
Nguyen Huu Anh Tri	23127130	23CLC08

Ho Chi Minh City, April 7th 2025

Contents

1	Introduction	2
2	Print a Page Table	3
2.1	Kernel Space	3
2.2	User Space	3
2.3	Execution Flow of the vmprint Function	3
2.3.1	Step 1: User Program Execution	3
2.3.2	Step 2: Kernel Space Execution	3
2.3.3	Step 3: Printing the Page Table	4
2.4	Result	4
3	Detect Which Pages Have Been Accessed	6
3.1	Kernel Space	6
3.2	User Space	6
3.3	Execution Flow of the pgaccess System Call	6
3.3.1	Step 1: User Program Calls pgaccess()	6
3.3.2	Step 2: Kernel Identifies the System Call	7
3.3.3	Step 3: Processing the pgaccess System Call	7
3.3.4	Step 4: Checking Each Page	7
3.3.5	Step 5: Returning the Bitmask to User Space	7
3.4	Result	7

1 Introduction

The Page Tables lab provides an opportunity to gain a deeper understanding of the inner workings of the Xv6 operating system, particularly how it manages memory through page tables. Through this project, we explore how Xv6 handles page table entries, memory access, and system calls related to page tables.

Based on this knowledge, we implement two key functionalities: printing a page table and detecting accessed pages. The `vmprint` function allows users to print the structure of a page table, aiding in debugging and understanding memory layout. The `pgaccess` system call reports which pages have been accessed, providing essential information for performance analysis and debugging. By modifying the kernel's core components and designing user-space programs to test these functionalities, we gain practical experience in memory management and kernel-user space interactions.

2 Print a Page Table

2.1 Kernel Space

To implement the `vmprint` function, we need to modify the kernel space. The `vmprint` function is responsible for printing the structure of a page table. This involves recursively traversing the page table entries and printing their details.

- **Implementation of `vmprint` and `vmprint_helper` functions:** The `vmprint_helper` function is a recursive function that traverses the page table entries. It prints the details of each entry, including the physical address and the level of the page table. The `vmprint` function calls `vmprint_helper` to start the traversal.
- **Modification in `exec.c`:** To enable the printing of the page table, we need to modify the `exec` function in `exec.c`. We add code to check for the `--print_pagetable` option in the program arguments. If this option is present, we set a flag to indicate that the page table should be printed. After processing the arguments, we call the `vmprint` function if the flag is set.
- **Declaration in `defs.h`:** The `vmprint` function needs to be declared in `defs.h` to make it accessible from other parts of the kernel.

2.2 User Space

To print the page table, the user can run a program with the `--print_pagetable` option. This option triggers the `vmprint` function in the kernel, which prints the structure of the page table.

2.3 Execution Flow of the `vmprint` Function

The execution of the `vmprint` function follows a step-by-step process, involving transitions between user space and kernel space. The following sections outline this process in detail.

2.3.1 Step 1: User Program Execution

- A user program, such as `init`, `ls`, or `echo`, is executed by the user with the `--print_pagetable` option.
- The program processes the arguments and sets the `print_pagetable` flag if the option is present.

2.3.2 Step 2: Kernel Space Execution

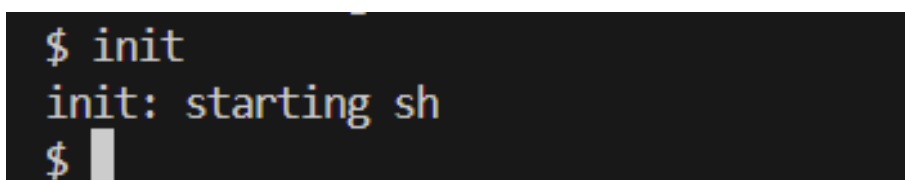
- The `exec` function in `exec.c` processes the arguments and check if the arguments include the `--print_pagetable`, then remove it.

- If the `print_pagetable` flag is set, the `vmprint` function is called to print the page table.

2.3.3 Step 3: Printing the Page Table

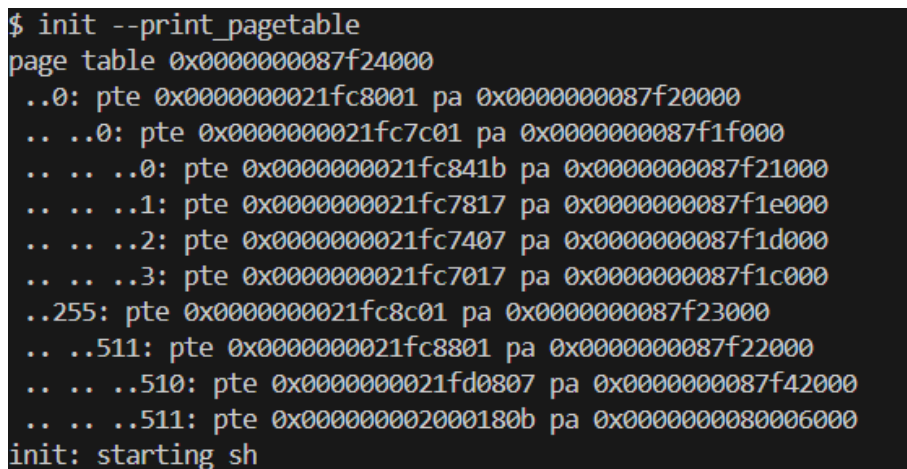
- The `vmprint` function calls `vmprint_helper` to recursively print the page table entries.
- `vmprint_helper` only visits the next level recursively if it meets the flag that the next page table entry is readable (`PTE_R`), writable (`PTE_W`), or executable (`PTE_X`).
- The page table structure is printed with indentation based on the level of the page table.

2.4 Result



```
$ init
init: starting sh
$
```

Figure 1: Result when running with no `--print_pagetable` option



```
$ init --print_pagetable
page table 0x0000000087f24000
..0: pte 0x0000000021fc8001 pa 0x0000000087f20000
.. ..0: pte 0x0000000021fc7c01 pa 0x0000000087f1f000
.. .. ..0: pte 0x0000000021fc841b pa 0x0000000087f21000
.. .. ..1: pte 0x0000000021fc7817 pa 0x0000000087f1e000
.. .. ..2: pte 0x0000000021fc7407 pa 0x0000000087f1d000
.. .. ..3: pte 0x0000000021fc7017 pa 0x0000000087f1c000
..255: pte 0x0000000021fc8c01 pa 0x0000000087f23000
.. ..511: pte 0x0000000021fc8801 pa 0x0000000087f22000
.. .. ..510: pte 0x0000000021fd0807 pa 0x0000000087f42000
.. .. ..511: pte 0x000000002000180b pa 0x0000000080006000
init: starting sh
```

Figure 2: Result when running with `--print_pagetable` option

With `--print_pagetable`, program prints the page table, followed by a list of entries with the following columns:

- **Index:** A sequential number (e.g., 0, 1, 2, ..., 511) that identifies each entry in the page table. These numbers appear to help the user track the position of each entry, especially since the output is paginated and contains at least 512 entries (as the last visible row is numbered 511).
- **pte:** A page table entry, represented as a hexadecimal memory address (e.g., `0x0000000021fc8801`). This maps a virtual address to a physical address in the system's memory management.

- **pa:** The physical address in memory (e.g., `0x0000000087f20000`) where the data is actually stored. These addresses show the result of the virtual-to-physical translation performed by the operating system.

This type of output is typically used by system administrators or developers to debug memory-related issues, inspect process memory mappings, or analyze system performance. The hexadecimal addresses provide a low-level view of how virtual memory is translated to physical memory in the operating system.

3 Detect Which Pages Have Been Accessed

3.1 Kernel Space

To implement the `pgaccess` system call, we need to modify the kernel space. The `pgaccess` system call reports which pages have been accessed by the process. This involves checking the accessed bit (`PTE_A`, which needs to be defined in `kernel/riscv.h`) in the page table entries and returning a bitmask indicating the accessed pages.

- **Definition of the `PTE_A` Bit:** The `PTE_A` bit is defined in `riscv.h` to represent the accessed bit in the page table entry. This bit is used to check if a page has been accessed.
- **Implementation of the `pgaccess` System Call:** The `pgaccess` system call is implemented in `sysproc.c`. It retrieves the starting virtual address, number of pages, and user address for the bitmask from user space. It then checks each page to see if it has been accessed by examining the `PTE_A` bit. If a page has been accessed, the corresponding bit in the bitmask is set. The `PTE_A` bit is then cleared to reset the state. Finally, the bitmask is copied from kernel space to user space.
- **Adding the `pgaccess` System Call:** The `pgaccess` system call is added to the system call table in `syscall.c` and `syscall.h`. This involves defining the system call number and mapping it to the `sys_pgaccess` function.

3.2 User Space

To use the `pgaccess` system call, we need to declare the `pgaccess` function in `user.h` and add an entry for it in `usys.pl`. This makes the system call accessible from user space.

3.3 Execution Flow of the `pgaccess` System Call

The execution of the `pgaccess` system call follows a step-by-step process, involving transitions between user space and kernel space. The following sections outline this process in detail.

3.3.1 Step 1: User Program Calls `pgaccess()`

- A user-space program `pgtbltest`, calls `pgaccess()`.
- The function `pgaccess()` prepares the system call request and passes the starting virtual address, number of pages, and user address for the bitmask.
- The syscall number is placed into a register, and an `ecall` instruction is executed to transfer control to the kernel.

3.3.2 Step 2: Kernel Identifies the System Call

- The system call handler in `kernel/syscall.c` reads the syscall number.
- It maps the request to `sys_pgaccess()` using the system call table.
- Execution is transferred to `sys_pgaccess()` in `kernel/sysproc.c`.

3.3.3 Step 3: Processing the pgaccess System Call

- The `sys_pgaccess()` function retrieves the starting virtual address, number of pages, and user address for the bitmask from user space using `argaddr()` and `argint()`.
- It limits the number of pages to avoid performance issues.
- The function initializes a bitmask to store the results.

3.3.4 Step 4: Checking Each Page

- For each page, the function calculates the virtual address and retrieves the page table entry (PTE) using `walk()`.
- If the PTE is valid and the page has been accessed (PTE_A is set), the function records this in the bitmask and clears the PTE_A bit.

3.3.5 Step 5: Returning the Bitmask to User Space

- The function copies the bitmask from kernel space to user space using `copyout()`.
- The function returns 0 to indicate success.

3.4 Result

```
$ pgtbltest
pgaccess_test starting
[pgaccess] Page 0 accessed (va = 0x4000)
[pgaccess] Page 2 accessed (va = 0x6000)
[pgaccess] Page 4 accessed (va = 0x8000)
pgaccess_test: OK
```

Figure 3: Result of the `pgaccess` system call

When `pgtbltest` runs, it calls `pgaccess_test()` to verify the `pgaccess()` system call. The test touches pages 0, 2, and 4, then uses `pgaccess()` to check which pages were accessed. The expected bitmask is `0b10101`, indicating that pages 0, 2, and 4 were accessed. If correct, the test prints `pgaccess_test: OK`, confirming the system call works as intended.

This type of test is typically used by system developers or administrators to debug memory management issues, verify the correctness of page table implementations.