Trường Đại học Khoa học tự nhiên – Khoa Công nghệ thông tin.

Đồ án thực hành 03.

Operating System – Hệ điều hành.

Nhóm 3TT
Tháng 12, 2024.

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN THỰC HÀNH SỐ 03

Bộ môn: Hệ điều hành.

Tên đề tài:

"Page Table."

Tên nhóm: 3TT.

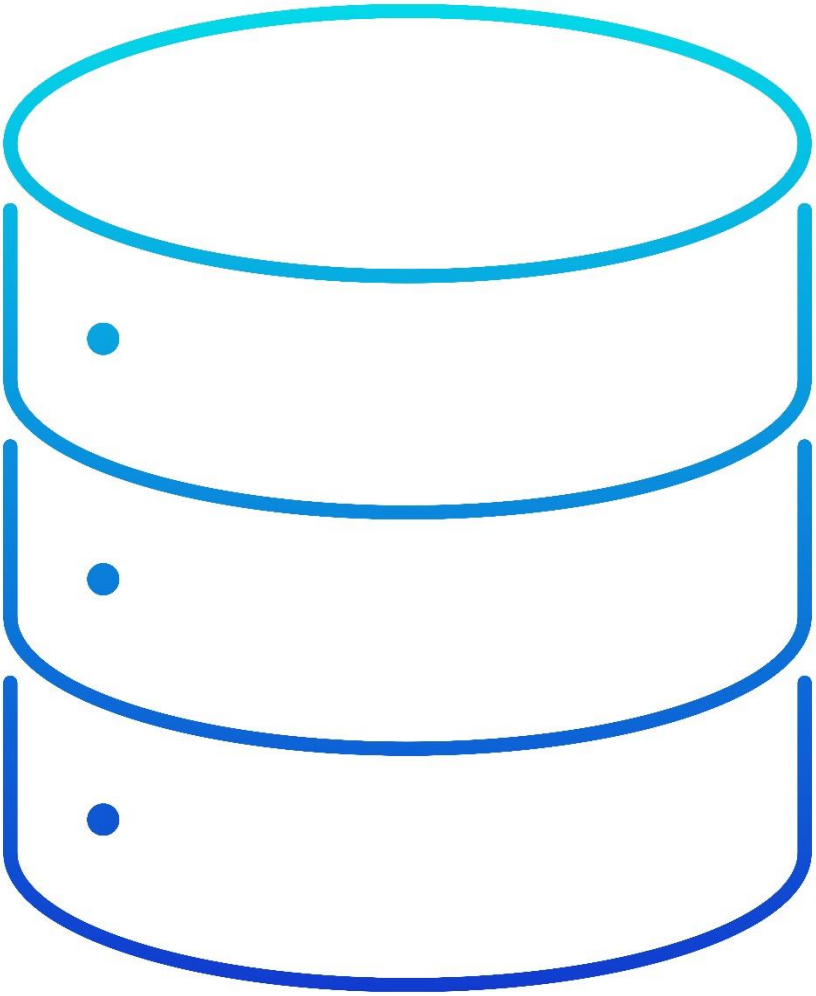
Thành viên:

1. 22120384 – Nguyễn Đình Trí.
2. 22120398 – Vũ Hoàng Nhật Trường.
3. 22120412 – Nguyễn Anh Tường.

Thông tin chung:

1. **Bộ môn:** Hệ điều hành.
2. **Giảng viên lý thuyết:** Thầy Trần Trung Dũng.
3. **Giảng viên thực hành:** Thầy Nguyễn Thanh Quân.
4. **Mã lớp:** 22_4.
5. **Tên nhóm:** 3TT.
6. **Danh sách thành viên:**
    a. 22120384 – Nguyễn Đình Trí.
    b. 22120398 – Vũ Hoàng Nhật Trường.
    c. 22120412 – Nguyễn Anh Tường.
7. **Link github repository:** "Click here to go to our github repository."

Section 0: *Bảng phân công công việc.*



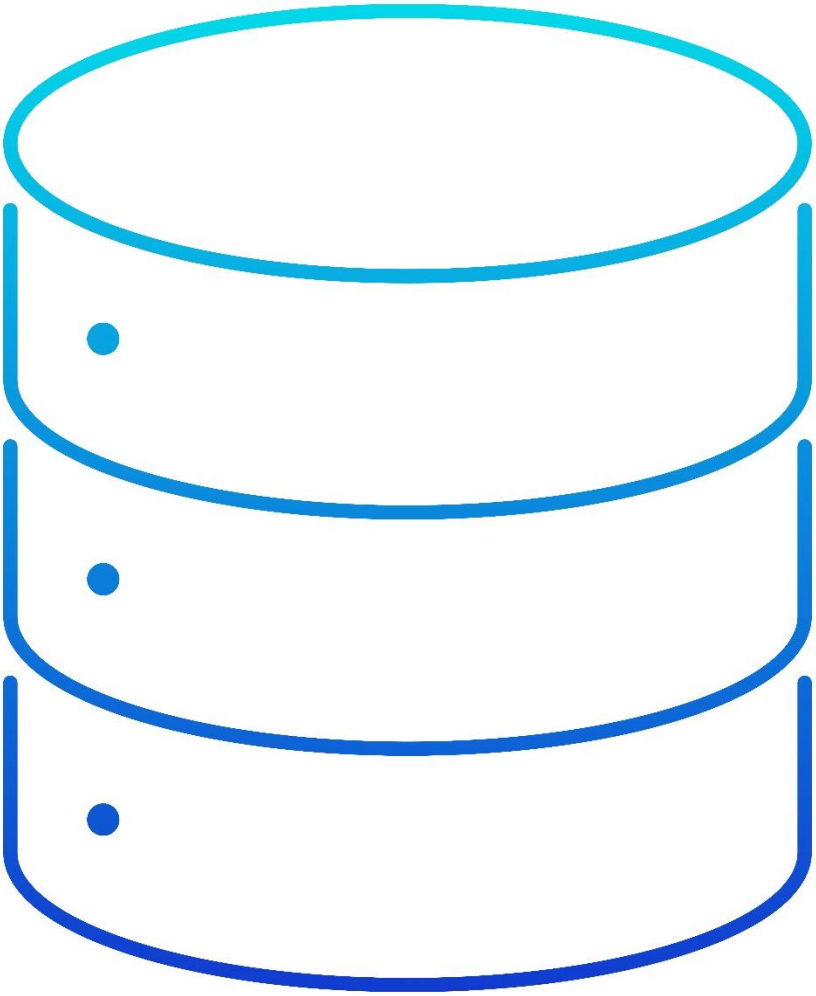
I. Bảng phân công công việc:

MSSV	Họ và Tên	Câu đảm nhiệm	Tiến độ
22120384	Nguyễn Đình Trí	2	100%
22120398	Vũ Hoàng Nhật Trường	3	100%
22120412	Nguyễn Anh Tường	1	100%

II. Tiến độ đồ án:

100%

Câu 1: *Tăng tốc system call.*



I. Kernel/Memlayout.h

Định nghĩa struct usyscall với thành phần process ID kiểu số nguyên.

II. Kernel/proc.h

Định nghĩa struct usycall *usyscall để sử dụng trong proc.c

```
struct usyscall *usyscall;
```

III. Kernel/Proc.c

Trong hàm allocproc – khởi tạo tiến trình:

Bước 1: Cấp phát bộ nhớ cho cấu trúc usyscall của tiến trình bằng kalloc (khoảng 4096 byte).

- Nếu không cấp phát được thì giải phóng tài nguyên liên quan đến tiến trình p bằng hàm freeproc.
- Giải phóng luôn khóa bằng hàm release (khóa này đảm bảo rằng tài nguyên không bị xung đột).

```
if ((p->usyscall = (struct usyscall *) kalloc()) == 0) {
    freeproc(p);
    release(&p->lock);
    return 0;
}
```

Bước 2: Khởi tạo PID và sao chép vào trang được chia sẽ.

```
p->usyscall->pid = p->pid;
```

Trong hàm freeproc – giải phóng tiến trình:

Bước 1: Kiểm tra tồn tại của usyscall, nếu tồn tại thì giải phóng bằng hàm kfree.

```
if (p->usyscall) {
    kfree((void *) p->usyscall);
}
```

Bước 2: Đặt giá trị này trở về bằng 0 (tương ứng NULL).

```
p->usyscall = 0;
```

Định nghĩa macro USYSCALL:

```
#define USYSCALL (TRAPFRAME - PGSIZE)
```

➔ **USYSCALL** được tính bằng cách lùi một trang bộ nhớ (PGSIZE); tức là nó nằm ngay trước **TRAPFRAME** trong không gian địa chỉ của tiến trình.

➔ **USYSCALL** là một địa chỉ ảo được sử dụng để ánh xạ trang chỉ đọc, cung cấp dữ liệu từ kernel đến không gian người dùng.

Trong bảng tiến trình – proc_pagetable:

Ánh xạ trang vật lý (p->usyscall) vào địa chỉ ảo USYCALL với quyền *chỉ đọc* (**PTE_R | PTE_U**).

- Nếu ánh xạ không thành công thì ta sẽ giải phóng các tài nguyên liên quan đến tiến trình, trang hiện tại.
    - Hủy ánh xạ trang tại địa chỉ ảo TRAPFRAME. (tránh xung đột tài nguyên).
    - Hủy ánh xạ tại địa chỉ ảo TRAPPOLINE. (tránh xung đột tài nguyên).
        - ⇨ Chức năng của TRAPOLINE là chuyển đổi không gian người dùng và kernel.
    - Giải phóng toàn bộ trang.

```
if (mappages(pagetable, USYSCALL, PGSIZE,
                    (uint64) (p->usyscall), PTE_R | PTE_U) < 0) {
    uvmunmap(pagetable, TRAPFRAME, 1, 0);
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
}
```

Giải phóng trang – freepagetable:

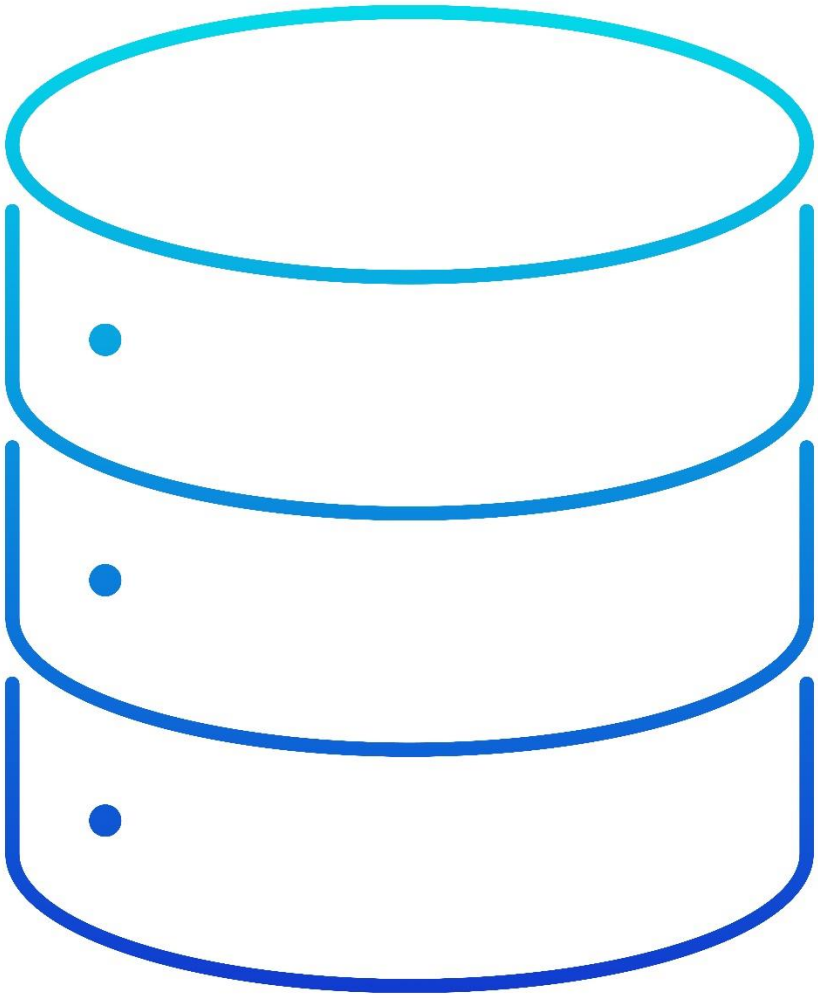
Giải phóng USYSCALL.

```
uvmunmap(pagetable, USYSCALL, 1, 0);
```

IV. Kết quả:

```
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
```

Câu 2: *In bảng trang.*



I. Yêu cầu

In bảng trang bằng cách viết hàm vmprint() nhận một tham số có kiểu dữ liệu pagetable_t và in nó ra theo định dạng yêu cầu.

II. Viết hàm vmprint() và recursive_vmprint()

- Hàm recursive_vmprint():
  - Duyệt đệ quy bảng trang ba cấp (3-level page table) với số lượng 512 mục (PTE) trong mỗi bảng trang.
  - In ra thông tin của mỗi mục hợp lệ (PTE có cờ PTE_V).

Dựa vào cấp độ (depth) để xác định số lượng ký hiệu ".." hiển thị trước mỗi dòng in.

```
void
recursive_vmprint(pagetable_t pagetable, uint64 depth)
{
    if(depth > 2){
        return;
    }
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        if(pte & PTE_V){
            uint64 child = PTE2PA(pte);
            if(depth == 0){
                printf(" ..%d: pte %p pa %p\n", i , pte, child);
                recursive_vmprint((pagetable_t)child, depth + 1);
            }else if(depth == 1){
                printf(" .. ..%d: pte %p pa %p\n", i , pte, child);
                recursive_vmprint((pagetable_t)child, depth + 1);
            }else{
                printf(" .. .. ..%d: pte %p pa %p\n", i , pte, child);
            }
        }
    }
    return;
}
```

- Hàm vmprint():
  - Gọi hàm recursive_vmprint() để duyệt và in thông tin bảng trang bắt đầu từ cấp 0.

```c
void
vmprint(pagetable_t pagetable)
{
    printf("page table %p\n", pagetable);
    recursive_vmprint(pagetable, 0);
    return;
}
```

III. Định nghĩa nguyên mẫu hàm trong kernel/defs.h

Thêm nguyên mẫu của hàm vmprint() vào file kernel/defs.h:

```c
void            vmprint(pagetable_t);
```

IV. Sử dụng vmprint() trong hàm exec()

Trong file kernel/exec.c, thêm đoạn mã sau vào trước return argc;:

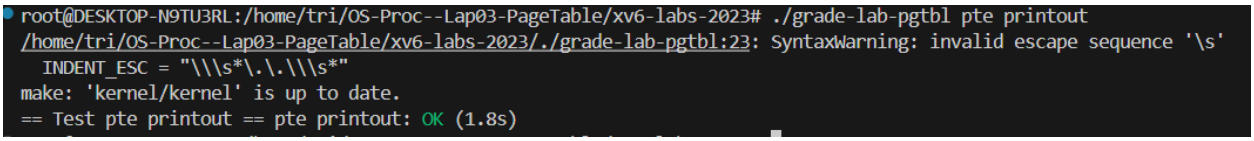
```c
if(p->pid == 1){
  vmprint(p->pagetable);
}
return argc; // this ends up in a0, the first argument to main(argc, argv)
```

V. Kiểm tra chương trình

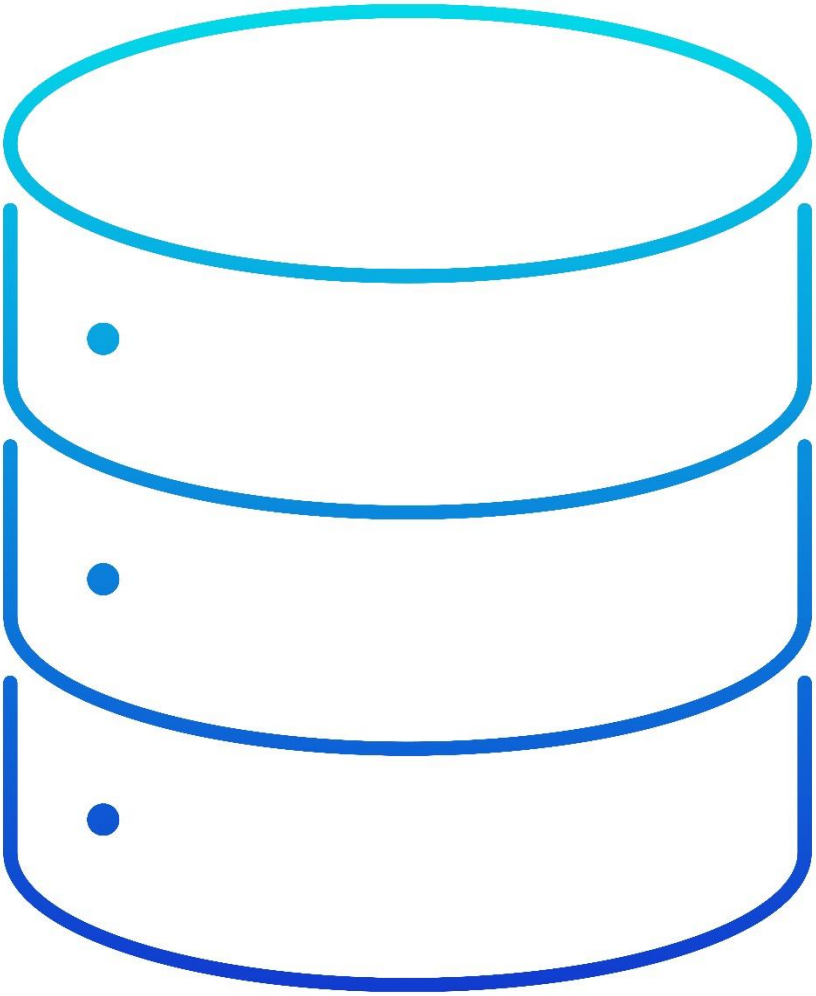
Chạy lệnh ./grade-lab-pbtbl pte printout để kiểm tra kết quả.

Kết quả in ra từ vmprint() hiển thị đúng thông tin bảng trang, và pte printout báo "OK".

```
page table 0x0000000087f6b000
 ..0: pte 0x0000000021fd9c01 pa 0x0000000087f67000
 .. ..0: pte 0x0000000021fd9801 pa 0x0000000087f66000
 .. .. ..0: pte 0x0000000021fda01f pa 0x0000000087f68000
 .. .. ..1: pte 0x0000000021fd941f 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
```


root@DESKTOP-N9TU3RL:/home/tri/OS-Proc--Lap03-PageTable/xv6-labs-2023# ./grade-lab-pgtbl pte printout
/home/tri/OS-Proc--Lap03-PageTable/xv6-labs-2023/./grade-lab-pgtbl:23: SyntaxWarning: invalid escape sequence '\s'
INDENT_ESC = "\\\s*\.\.\\\s*"
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (1.8s)

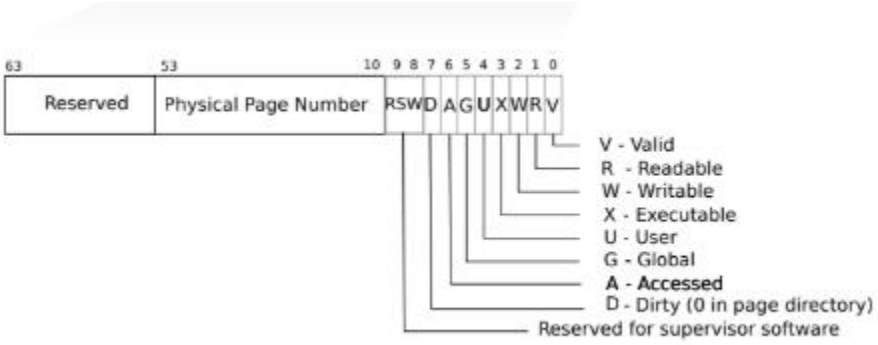
Câu 3: *Phát hiện trang được truy cập.*



I. kernel/riscv.h

```
#define PTE_A (1L << 6) // Access bit
```

Định nghĩa bit PTE_A (là bit kiểm tra xem trang có được truy cập hay chưa) - là bit thứ 6 trong kiến trúc PTE.



II. kernel/sysproc.c

Xây dựng hàm sys_pgaccess()

<u>Bước 1</u>: Lấy các tham số của người dùng trong hàm pgaccess()

```
uint64 base;     // Địa chỉ cơ sở của vùng nhớ cần kiểm tra
int num;         // Số lượng trang
uint64 bitmap_u;  // Địa chỉ để lưu bitmap kết quả

// Lấy tham số từ tiến trình người dùng
argaddr(0, &base);
argint(1, &num);
argaddr(2, &bitmap_u);
```

Ở đây, ta sẽ sử dụng các hàm **argaddr()**, **argint()** để lưu tham số vào 3 biến: **base** (địa chỉ cơ sở của vùng nhớ), **num** (số trang cần kiểm tra) và **bitmap_u** (địa chỉ để lưu kết quả).

<u>Bước 2</u>: Kiểm tra xem số trang có hợp lệ không

```
if (num < 0)
  return -1;
```

Nếu số trang quá nhiều, ta sẽ định nghĩa **MAX_PAGES (= 32)** để giới hạn lại.

```
#define MAX_PAGES 32
```

```
if (num > MAX_PAGES)
  num = MAX_PAGES; //
```

<u>Bước 3</u>: Lấy thông tin của tiến trình hiện tại (sử dụng myproc()) và tạo 1 biến tạm (bitmap) để lưu giá trị

```
struct proc *p = myproc();
uint64 bitmap = 0;
```

<u>Bước 4</u>: Duyệt từng trang

Ở đây ta sẽ tính địa chỉ của từng trang, sau đó sử dụng hàm walk() để tìm PTE của trang đó.

```
for (int i = 0; i < num; i++) {
  uint64 va = base + i * PGSIZE; // Tính địa chỉ từng trang
  pte_t *pte = walk(p->pagetable, va, 0); // Tìm PTE của trang
```

Kiểm tra xem PTE_V của trang đó có được bật không (là bit kiểm tra sự hợp lệ của trang (V - Valid)). Nếu không thì continue sang trang khác. Nếu có thì chạy tiếp.

```
if (!pte || (*pte & PTE_V) == 0)
  continue;
```

Tiếp theo, ta sẽ xem PTE_A có được bật không. Nếu có thì cập nhật số trang vào biến bitmap (sử dụng **bitwise** |). Sau khi cập nhật bitmap xong thì phải tắt PTE_A (để có thể kiểm tra cho những lần gọi hàm pgaccess() tiếp theo, nếu không ta sẽ không biết nó được bật kể từ khi nào) – sử dụng **bitwise & ~PTE_A** để tắt.

```
if (*pte & PTE_A) // Ki
{
  bitmap |= (1L << i);
  *pte &= ~PTE_A;
}
```

<u>Bước 5</u>: Sao chép bitmap vào địa chỉ bitmap_u từ kernel space sang user space bằng cách sử dụng hàm copyout()

```
if (copyout(p->pagetable, bitmap_u, (char *)&bitmap, sizeof(bitmap)) < 0)
  return -1;
```

III. Kết quả

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$
```

Khi chạy pgtbltest trong qemu, các test đều thực thi thành công.