

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN  
KHOA CÔNG NGHỆ THÔNG TIN



## PROJECT 03 – PAGE TABLES

Môn : Hệ Điều Hành

### GIẢNG VIÊN HƯỚNG DẪN

Thầy Lê Giang Thanh  
Thầy Nguyễn Thanh Quân  
Thầy Lê Hà Minh

### SINH VIÊN THỰC HIỆN

22127151 - Lâm Tiến Huy  
22127290 - Nguyễn Thị Thu Ngân  
22127408 – Kha Vĩnh Thuận

LỚP : 22CLC08

Thành Phố Hồ Chí Minh – 2024

# Lời Cảm Tạ

Lời đầu tiên, cho phép chúng em được cảm ơn chân thành và sâu sắc nhất muốn gửi đến thầy Nguyễn Thanh Quân vì đã nhiệt tình hướng dẫn chúng em trong cả quá trình làm đồ án này. Ngoài ra, chúng em cũng muốn gửi lời tri ân đến thầy Lê Giang Thanh với công sức và thời gian thầy bỏ ra để phổ cập kiến thức bộ môn Hệ điều hành. Đây sẽ là tiền đề vững chắc cho những năm học sau trong ngành Công nghệ Thông tin.

Đồ án Page Tables khám phá các bảng trang và sửa đổi chúng để tăng tốc độ cho một số system call cụ thể và để phát hiện ra những trang nào đã được truy cập, thực hiện trên hệ điều hành xv6.

Với quá trình làm việc, chúng em cam đoan rằng công việc được phân chia hợp lý, có minh chứng rõ ràng, quá trình làm việc ổn định và linh hoạt. Về kết quả làm việc nói chung, như bảng phân công công việc hay những điều đúc kết được sau đồ án sẽ được ghi chú vào chương cuối cùng của bản báo cáo.

Trong thời gian làm bản cáo cáo, sai sót là điều chúng em không thể tránh khỏi, vì thế chúng em rất mong các thầy xem xét và bỏ qua. Bản báo cáo dựa trên những kiến thức đã được dạy và tự tích lũy nên cũng có phần thiếu kinh nghiệm thực tiễn, trình độ lý luận, chúng em rất mong nhận được những ý kiến, đóng góp của thầy, cô để chúng em có thể học hỏi và hoàn thiện hơn.

Nếu file gửi qua Moodle có lỗi, các thầy có thể truy cập GitHub của tụi em: <https://github.com/tnstanHCMUS/xv6-labs-hcmus>

Chúng em xin chân thành cảm ơn.

# Mục Lục

Lời Cảm Tạ.....	2
Mục Lục.....	3
Danh Mục Hình .....	4
Chương 1. Setup .....	5
Chương 2. Thực Hành .....	6
2.1 Speed Up System Calls.....	6
2.1.1 Quá trình thực hiện.....	6
2.1.2 Kết quả .....	9
2.2 Print A Page Table.....	10
2.2.1 Quá trình thực hiện.....	10
2.2.2 Kết quả .....	12
2.3 Detect Which Pages Have Been Accessed .....	14
2.3.1 Quá trình thực hiện.....	14
2.3.2 Kết quả .....	15
Chương 3. Hoàn Thành .....	16
Chương 4. Tổng Kết .....	17
Tài liệu tham khảo .....	18

# Danh Mục Hình

Hình 1. Chuyển qua branch pgtbl.....	5
Hình 2. Thêm con trỏ usyscall.....	6
Hình 3. Mappages trong Pagetable.....	6
Hình 4. Thêm usyscallpage vào allocproc .....	7
Hình 5. Giải phóng trang trong freeproc.....	7
Hình 6. Lỗi khi make qemu.....	7
Hình 7. Dò lỗi freewalk: leaf trong hàm freewalk.....	8
Hình 8. Kiểm tra lỗi trong GDB.....	8
Hình 9. Thêm câu lệnh uvmunmap để giải phóng.....	8
Hình 10. Điểm của ugetpid.....	9
Hình 11. Cấu trúc Page Table .....	10
Hình 12. Hàm printpgtb() và vmprint().....	11
Hình 13. Thêm prototype của hàm.....	11
Hình 14. Kiểm tra pid để in page table .....	12
Hình 15. Kết quả của make qemu.....	12
Hình 16. Điểm của pte printout.....	12
Hình 17. User address space của process .....	13
Hình 18. Định nghĩa cờ PTE_A.....	14
Hình 19. Hàm sys_pgaccess.....	15
Hình 20. Điểm của pgaccess trong pgtbltest .....	15
Hình 21. Make grade.....	16
Hình 22. Bảng phân công công việc.....	17

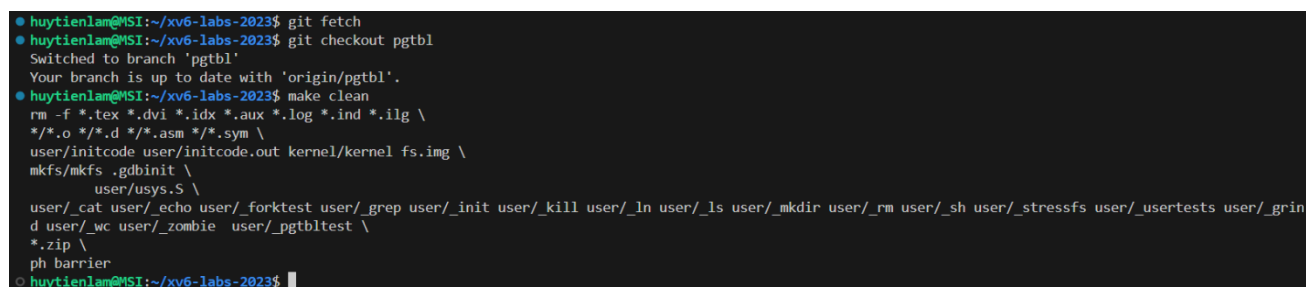
# Chương 1

## Setup

Trước khi bước vào công việc chính thì việc đầu tiên cần làm đó chính là setup và tạo ra môi trường để có thể hoạt động. Với hai thành viên sử dụng **Windows** và một thành viên sử dụng **macOS**, sẽ có sự khác biệt trong việc cài đặt môi trường. Tuy nhiên, vì đây là **bài thực hành số ba**, nên tất cả đều sẽ tận dụng tiếp hệ điều hành **xv6** mà chúng ta đã cài thêm các tính năng ở **bài thực hành số một và hai**.

Để bắt đầu bài thực hành, chúng ta sẽ chuyển từ branch **syscall** qua branch **pgtbl**. Các câu lệnh sẽ lần lượt mang tính **cập nhật**, **đổi branch** và **dọn dẹp**.

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



```
huytienlam@MSI:~/xv6-labs-2023$ git fetch
huytienlam@MSI:~/xv6-labs-2023$ git checkout pgtbl
Switched to branch 'pgtbl'
Your branch is up to date with 'origin/pgtbl'.
huytienlam@MSI:~/xv6-labs-2023$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grin
d user/_wc user/_zombie user/_pgtbltest \
*.zip \
ph barrier
huytienlam@MSI:~/xv6-labs-2023$
```

Hình 1. Chuyển qua branch *pgtbl*

Đề án lần này khám phá các **bảng trang** và sửa đổi chúng để **tăng tốc độ** cho một số **cuộc gọi hệ thống** cụ thể và để phát hiện ra những trang nào đã được **truy cập**.

**kernel/memlayout.h** chứa cấu trúc của bộ nhớ.

**kernel/vm.c** chứa hầu hết code virtual memory.

**kernel/kalloc.c** chứa code để phân bổ và giải phóng bộ nhớ vật lý.

## Chương 2

# Thực Hành

### 2.1 Speed Up System Calls

#### 2.1.1 Quá trình thực hiện

Đầu tiên, chúng ta cần ánh xạ một **struct usyscall** đến địa chỉ **USYSCALL** trong **kernel/memlayout.h**. Trong **kernel/proc.h**, thêm dòng **struct usyscall\* usyscallpage** vào trong **struct proc**.

```
106 char name[16];           // Process name (debugging)
107
108 struct usyscall* usyscallpage;
109 };
```

Hình 2. Thêm con trỏ usyscall

Quá trình ánh xạ được thực hiện thông qua **proc\_pagetable** trong **kernel/proc.c**, đòi hỏi việc sử dụng hàm **mappages** với quyền chỉ đọc cho **user space**. Chúng ta ánh xạ một trang từ không gian **physical memory** vào **virtual memory**. **USYSCALL** là địa chỉ bắt đầu của **virtual memory** mà chúng ta muốn ánh xạ trang vào, và **(uint64)(p->usyscallpage)** là địa chỉ của trang trong **physical memory** mà chúng ta muốn ánh xạ vào **virtual memory**. Sử dụng **PTE\_U | PTE\_R** để thiết lập các cờ cho trang, chúng ta cho phép truy cập dạng đọc và truy cập bởi user code.

```
C proch M • C proc.c 3, M X
kernel > C proc.c > proc_freepagetable(pagetable_t, uint64)
186 // Create a user page table for a given process, with no user memory,
187 // but with trampoline and trapframe pages.
188 pagetable_t
189 proc_pagetable(struct proc *p)
190 {
191     pagetable_t pagetable;
192
193     // An empty page table.
194     pagetable = uvmcreate();
195     if(pagetable == 0)
196         return 0;
197
198     if(mappages(pagetable, USYSCALL, PGSIZE,
199                (uint64)(p->usyscallpage), PTE_U | PTE_R) < 0){
200         uvmfree(pagetable, 0);
201         return 0;
202     }
203     // map the trampoline code (for system call return)
204     // at the highest user virtual address.
205     // only the supervisor uses it, on the way
206     // to/from user space, so not PTE_U.
```

Hình 3. Mappages trong Pagetable

Theo trang 33 của sách **xv6: a simple, Unix-like teaching operating system (2022)**, mỗi **Page Table Entry (PTE)** có các bit cờ để nói cho phần cứng là **virtual address** đó được quyền sử dụng như thế nào. Trường hợp này, cờ **PTE\_R** được thiết lập nên các chỉ thị có thể đọc từ trang. Cờ **PTE\_U** được thiết lập, các chỉ thị từ **user mode** có thể sử dụng PTE đó, nếu không chỉ có **supervisor mode** mới có thể sử dụng nó.

Tiếp theo, chúng ta tìm đến **struct allocproc**, khởi tạo trang và lưu Process ID hiện tại vào **allocproc**. Chúng ta kiểm tra xem việc cấp phát bộ nhớ cho **usyscallpage** của tiến trình **p** thành công hay không. Nếu không đủ bộ nhớ, tiến trình **p** sẽ được giải phóng bằng hàm **freeproc** và trả về giá trị **0**. Sau đó, khóa của nó sẽ được giải phóng bằng cách gọi hàm **release**. Nếu cấp phát bộ nhớ cho **usyscallpage** thành công, tiến trình **p** sẽ lưu Process ID hiện tại vào, giúp **usyscallpage** biết rằng nó thuộc về tiến trình nào.

```

128 // Allocate a trapframe page.
129 if((p->trapframe = (struct trapframe *)kalloc()) == 0){
130     freeproc(p);
131     release(&p->lock);
132     return 0;
133 }
134
135 if ((p->usyscallpage = (struct usyscall *)kalloc()) == 0) {
136     freeproc(p);
137     release(&p->lock);
138     return 0;
139 }
140 p->usyscallpage->pid = p->pid;

```

Hình 4. Thêm **usyscallpage** vào **allocproc**

Đồng thời, chúng ta giải phóng trang trong hàm **freeproc**. Nếu **usyscallpage** của tiến trình **p** đã được cấp phát bộ nhớ (khác **NULL**), tiến trình **p** sẽ giải phóng bộ nhớ của **usyscallpage** bằng cách gọi hàm **kfree()**. Sau đó, **usyscallpage** của **p** sẽ được gán lại giá trị **NULL** để chỉ ra rằng bộ nhớ đã được giải phóng, đảm bảo không rò rỉ bộ nhớ.

```

163 static void
164 freeproc(struct proc *p)
165 {
166     if(p->trapframe)
167         kfree((void*)p->trapframe);
168     p->trapframe = 0;
169     if(p->usyscallpage)
170         kfree((void *)p->usyscallpage);
171     p->usyscallpage = 0;

```

Hình 5. Giải phóng trang trong **freeproc**

Chạy thử lệnh **./grade-lab-pgtbl ugetpid** thì xảy ra lỗi treo. Chúng ta tiếp tục thử **make qemu** thì hiện ra lỗi sau.

```

huytienlan@MSI:~/xv6-labs-2023$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
panic: freewalk: leaf

```

Hình 6. Lỗi khi **make qemu**

Để giải thích cho lỗi **freewalk: leaf**, chúng ta tìm thấy trong hàm **freewalk** thuộc **kernel/vm.c**. Hàm **freewalk** được sử dụng để **giải phóng** các **page table** một cách **đệ quy** khi không còn cần thiết nữa. Trong quá trình giải phóng, nếu một mục nhập trong **page table** là một **leaf**, tức là nó **không** trở đến một **bảng trang con** mà chỉ trực tiếp trở đến một trang dữ liệu, trang không hợp lệ, trang không có quyền đọc, ghi hoặc thực thi, thì hàm sẽ gây ra lỗi **panic** này. Trong trường hợp này, vì khi kiểm tra cờ **PTE\_V**, mang ý nghĩa xác nhận sự tồn tại của **PTE**, đã trả ra giá trị **1** chứng tỏ rằng **trang vẫn tồn tại**. Với lỗi này khi một lá được phát hiện, nó không được xử lý đúng cách bằng **kfree** trong hàm **freewalk**.

```

276 // Recursively free page-table pages.
277 // All leaf mappings must already have been removed.
278 void
279 freewalk(pagetable_t pagetable)
280 {
281     // there are 2^9 = 512 PTEs in a page table.
282     for(int i = 0; i < 512; i++){
283         pte_t pte = pagetable[i];
284         if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
285             // this PTE points to a lower-level page table.
286             uint64 child = PTE2PA(pte);
287             freewalk((pagetable_t)child);
288             pagetable[i] = 0;
289         } else if(pte & PTE_V){
290             panic("freewalk: leaf");
291         }
292     }
293     kfree((void*)pagetable);
294 }

```

Hình 7. Dò lỗi **freewalk: leaf** trong hàm **freewalk**

Khi này, chúng ta sẽ làm tương tự như **Lab 02**, sử dụng **make qemu-gdb** ở một terminal và **gdb-multiarch -x .gdbinit** ở terminal còn lại để chạy **GDB**. Sau đó, chúng ta đặt **breakpoint** tại **freewalk**, tiếp tục và **backtrace** để xem lỗi.

```

224 // Free a process's page table, and free the
225 // physical memory it refers to.
226 void
227 proc_freepagetable(pagetable_t pagetable, uint64 sz)
228 {
229     uvmunmap(pagetable, USYSCALL, 1, 0);
230     uvmunmap(pagetable, TRAMPOLINE, 1, 0);
231     uvmunmap(pagetable, TRAPFRAME, 1, 0);
232     uvmfree(pagetable, sz);
233 }
234

```

Hình 9. Thêm câu lệnh **uvmunmap** để giải phóng

```

#5 0x00000000000020e0 in syscall () at kernel/syscall.c:156
#6 0x0000000000001dd6 in usertrap () at kernel/trap.c:67
#7 0x0505050505050505 in ?? ()

```

Hình 8. Kiểm tra lỗi trong **GDB**

Theo dòng lỗi, chúng ta phát hiện lỗi nằm ở hàm **proc\_freepagetable** ở **kernel/proc.c**. Hàm này dùng để **giải phóng** bộ nhớ và các tài nguyên liên quan đến bảng trang của một tiến trình. Có thể thấy rằng, việc chúng ta ánh xạ **USYSCALL** của **virtual memory** mà chưa được giải phóng đã gây lỗi. Vì vậy, chúng ta thêm dòng **uvmunmap (pagetable, USYSCALL, 1, 0)** để thực hiện việc giải phóng ánh xạ.



Linux tăng tốc độ cho một số **system call** bằng cách **chia sẻ dữ liệu** trong một khu vực **read-only** giữa **user space** và **kernel**, này loại bỏ nhu cầu về các lần chuyển kernel khi thực hiện các cuộc gọi hệ thống này. Task này thực hiện tối ưu hóa cho **getpid()** trong xv6.

Khi mỗi tiến trình được tạo ra, ánh xạ một trang chỉ đọc tại **USYSCALL** (một địa chỉ ảo được định nghĩa trong **memlayout.h**). Tại đầu trang này, lưu trữ một **struct usyscall** (cũng được định nghĩa trong **memlayout.h**), và khởi tạo nó để lưu trữ **PID** của tiến trình hiện tại. Đối với lab này, hàm **ugetpid()** đã được cung cấp trong **user space** và sẽ tự động sử dụng ánh xạ **USYSCALL**.

### 2.1.2 Kết quả

Khi này lệnh **./grade-lab-pgtbl ugetpid** đã trả ra kết quả **OK**, chứng tỏ thành công trong task đầu tiên.

```
huytienlam@MSI:~/xv6-labs-2023$ ./grade-lab-pgtbl ugetpid
make: 'kernel/kernel' is up to date.
== Test pgtbltest == (0.7s)
== Test pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
```

Hình 10. Điểm của ugetpid

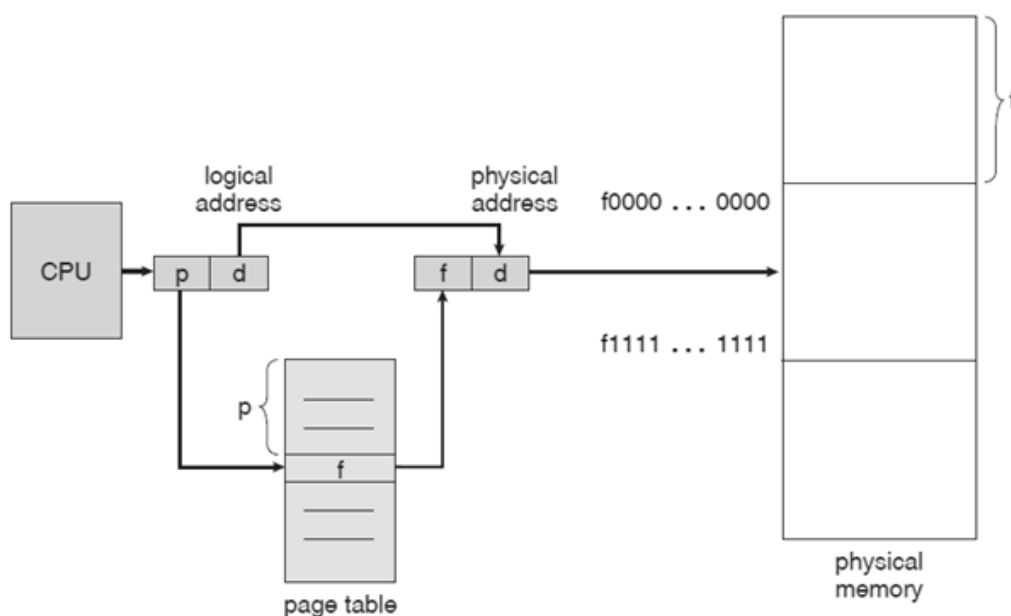
- *Các system call nào trong xv6 cũng có thể được thực hiện nhanh hơn bằng cách sử dụng trang chia sẻ này. Vì sao?*

Các system call nào trực tiếp hoặc gián tiếp gọi hàm **copyout** sẽ được thực hiện nhanh hơn, vì nó **tiết kiệm thời gian** trong quá trình sao chép dữ liệu, vì hàm **copyout** sao chép dữ liệu từ một **buffer** vào một **địa chỉ** trong **bộ nhớ**. Ngoài ra, các system call chỉ được sử dụng chỉ để **truy xuất thông tin** cũng sẽ được tăng tốc, như **getpid** trong phần trước. Một số hàm tương tự khác có thể kể đến **fork**, **time**, **getsid**, **getpgid**, **getuid**, **geteuid**, etc. Điều này là do việc **chuyển tiếp** vào hệ điều hành không còn cần thiết nữa, và dữ liệu tương ứng có thể được **đọc trực tiếp** từ trong **user mode** thay vì phải qua **kernel mode**.

## 2.2 Print A Page Table

### 2.2.1 Quá trình thực hiện

Page Table hoạt động như một bản đồ ánh xạ giữa không gian địa chỉ ảo và không gian địa chỉ vật lý trong hệ thống bộ nhớ ảo. Khi một quá trình trong hệ thống cần truy cập đến một địa chỉ ảo, hệ điều hành sử dụng Page Table để dịch địa chỉ ảo thành địa chỉ vật lý tương ứng. Khi in bảng trang, các PTEs được truy cập để trích xuất thông tin về mỗi trang trong không gian bộ nhớ ảo.



Hình 11. Cấu trúc Page Table

Để viết hàm in Page Table, ta có thể tham khảo ý tưởng từ hàm **freewalk** trong file **kernel/vm.c** – có chức năng giải phóng bộ nhớ cho tất cả các page table trong hệ thống quản lý bộ nhớ, hàm này tiếp cận đệ quy đến tất cả các page table. Từ đó, hàm **printpgtb(pageable\_t pageable, int depth)** trong **kernel/exec.c** dùng để in ra các mục nhập của bảng trang một cách đệ quy, hiển thị cấu trúc và mối quan hệ giữa các cấp độ khác nhau trong cấu trúc bảng trang. Hàm này nhận 2 tham số gồm **pageable** là con trỏ đến page table và số nguyên **depth** thể hiện độ sâu trong đệ quy (tương ứng với cấp độ trong cấu trúc page table). Hàm để in page table vừa nêu có thể được làm rõ như sau:

- Hàm dùng vòng lặp để duyệt qua 512 PTEs của page table.
- Lấy từng PTE và kiểm tra xem nó có hợp lệ hay không. Nếu có, in độ sâu tương ứng của PTE ra màn hình, sau đó in PTE và Physical Address tương ứng.
- Trong trường hợp PTE cũng là một page table, gọi hàm đệ quy để in các PTE của page table đó ra màn hình với và tăng biến **depth** lên 1.

Để quản lý hàm in **printpgtb**, ta dùng hàm **vmprint** nhận tham số là con trỏ đến page table để gọi hàm in và truyền các tham số cần thiết vào hàm.

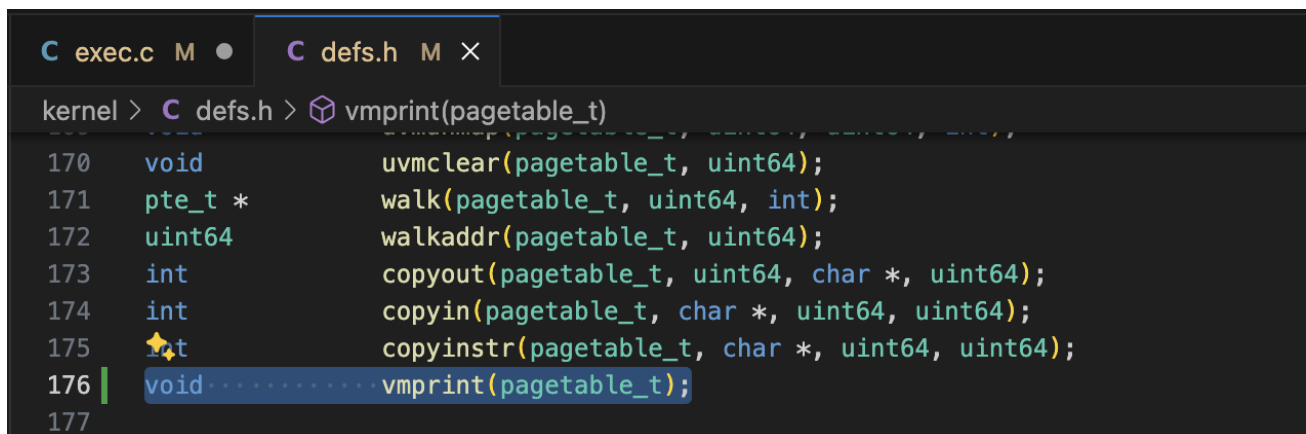
```
void
printpgtb(pagetable_t pagetable, int depth)
{
    // there are 2^9 = 512 PTEs in a page table.
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i]; // get the PTE
        if(pte & PTE_V){ // check if the PTE is valid
            printf("...");
            for(int j=0;j<depth;j++) {
                printf(" .."); // print the depth of the page table
            }
            printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte)); // print the PTE and the physical address

            if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0) { // check if the PTE is a page table
                uint64 child = PTE2PA(pte); // get the physical address of the child page table
                printpgtb((pagetable_t)child, depth+1); // recursively print the child page table
            }
        }
    }
}

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

Hình 12. Hàm `printpgtb()` và `vmprint()`

Tiếp đến, vào **kernel/defs.h** để thêm prototype của hàm.



```
kernel > C defs.h > vmprint(pagetable_t)
170 void uvmclear(pagetable_t, uint64);
171 pte_t * walk(pagetable_t, uint64, int);
172 uint64 walkaddr(pagetable_t, uint64);
173 int copyout(pagetable_t, uint64, char *, uint64);
174 int copyin(pagetable_t, char *, uint64, uint64);
175 int copyinstr(pagetable_t, char *, uint64, uint64);
176 void vmprint(pagetable_t);
177
```

Hình 13. Thêm prototype của hàm

Cuối cùng, ta thêm dòng lệnh sau vào **kernel/exec.c** trước **return argc** để in ra màn hình page table của process đầu tiên.

```

kernel > C exec.c > exec(char *, char **)
23  exec(char *path, char **argv)
124  oldpagetable = p->pagetable;
125  p->pagetable = pagetable;
126  p->sz = sz;
127  p->trapframe->epc = elf.entry; // initial program counter = main
128  p->trapframe->sp = sp; // initial stack pointer
129  proc_freepagetable(oldpagetable, oldsz);
130
131  if(p->pid==1)
132  .. vmprint(p->pagetable); // print the page table of the init process
133
134  return argc; // this ends up in a0, the first argument to main(argc, argv)
135
136  bad:

```

Hình 14. Kiểm tra pid để in page table

### 2.2.2 Kết quả

Khi thao tác booting xv6 với lệnh **make qemu**, ta sẽ nhìn thấy page table được hiển thị như hình bên dưới.

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
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
$ 

```

Hình 15. Kết quả của make qemu

Khi thực hiện lệnh **./grade-lab-pgtbl pte** đã trả ra kết quả **OK**, chứng tỏ thành công trong task thứ hai.

```

huytienlam@MSI:~/xv6-lab3$ ./grade-lab-pgtbl pte
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (1.3s)

```

Hình 16. Điểm của pte printout

- *Giải thích output của `vmprint()` dựa vào hình 3.4 trong sách xv6. Page 0 chứa những gì? Có gì ở page 2? Khi chạy user mode, tiến trình đọc/ghi bộ nhớ có thể được định vị bằng page 1 được hay không? Từ page 3 đến page cuối chứa cái gì?*

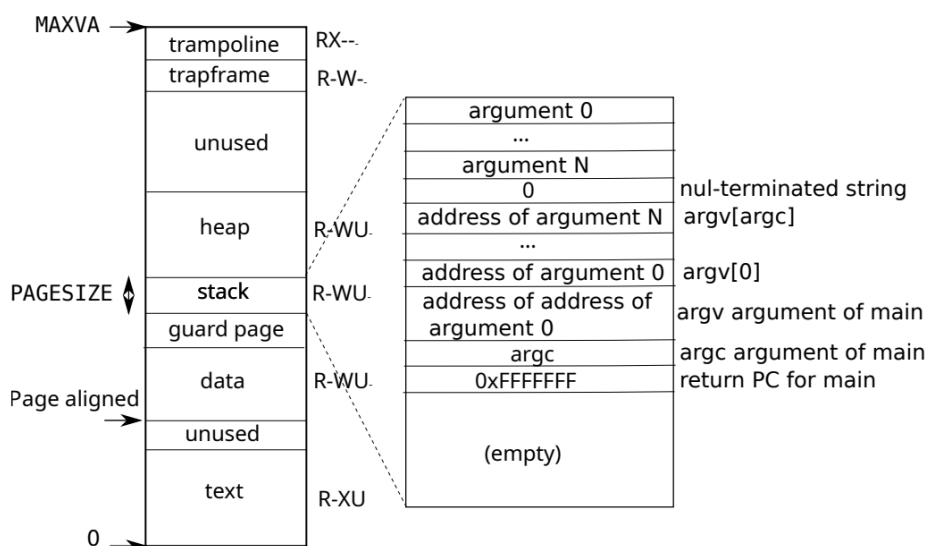


Figure 3.4: A process's user address space, with its initial stack.

Hình 17. User address space của process

Dựa vào **Hình 3.4** trong sách xv6, chúng ta có thể giải thích kết quả đầu ra của **vmprint** như sau:

**Page 0:** Chứa **data** và **text** của process. Đây là nơi chứa **main** code và **data** khởi tạo.

**Page 1:** Hoạt động như một **guard page** để bảo vệ **stack** khỏi bị **overflow**, phục vụ như cơ chế bảo vệ chống lại các **stack overflow** bằng cách đánh dấu **kết thúc** của ngăn xếp.

**Page 2:** Biểu thị **stack** của process. Đây là nơi các **biến cục bộ**, **tham số hàm**, và **địa chỉ trả về** được lưu trữ trong khi gọi hàm.

Khi chương trình đang chạy ở **user mode**, nó không thể đọc hoặc ghi vào **page 1** (**guard page**). Cụ thể, phần **PTE\_U** của **guard page** được **uvmclear** đặt thành **0**, có nghĩa là nếu process ở **user mode** cố gắng truy cập vào nó, **page fault** sẽ được kích hoạt. Mục đích của nó là để **bảo vệ page 2** (**stack**) khỏi sự **truy cập trái phép** của người dùng.

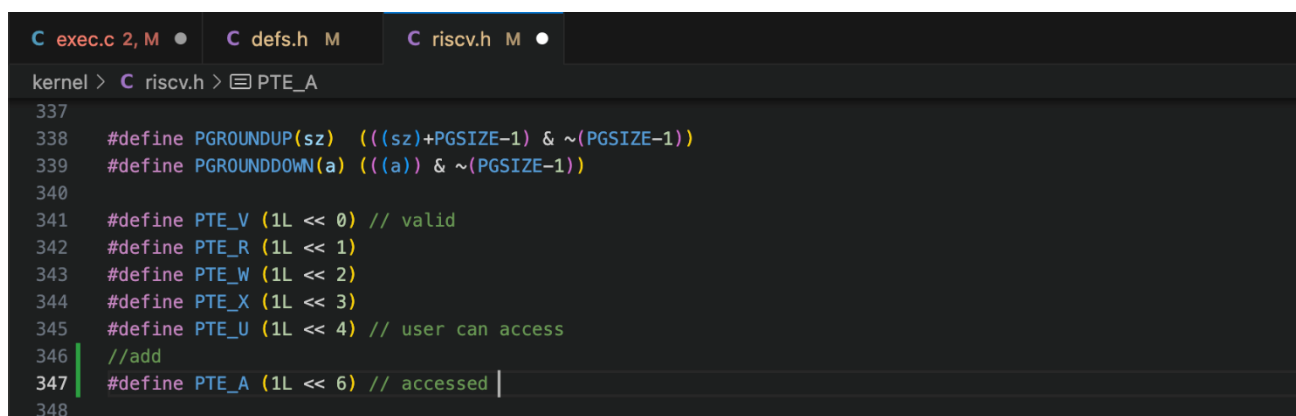
**Page 3 đến page cuối cùng:** Chứa **heap**, **trapframe** và **trampoline**. **Heap** được cấp phát động bộ nhớ. **Trapframe** chứa thông tin về **trạng thái thực thi** của tiến trình tại thời điểm bị **trap** hoặc bị **interrupt**. **Trampoline** được sử dụng để chuyển đổi giữa **user mode** và **kernel mode**.

## 2.3 Detect Which Pages Have Been Accessed

### 2.3.1 Quá trình thực hiện

Yêu cầu cuối cùng của lab đòi hỏi chúng ta triển khai hàm **pgaccess**, một **system call** để báo cáo những trang đã được truy cập. **System call** này nhận ba đối số: **địa chỉ ảo bắt đầu** của trang người dùng, **số lượng trang cần kiểm tra** và một **địa chỉ người dùng của buffer** để lưu kết quả vào **bitmask**. Quá trình thực hiện hàm này có các bước lần lượt như sau:

Đầu tiên trong **kernel/riscv.h**, tiến hành định nghĩa cờ **PTE\_A** dùng để đánh dấu page đã được truy cập.



```

337
338 #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
339 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
340
341 #define PTE_V (1L << 0) // valid
342 #define PTE_R (1L << 1)
343 #define PTE_W (1L << 2)
344 #define PTE_X (1L << 3)
345 #define PTE_U (1L << 4) // user can access
346 //add
347 #define PTE_A (1L << 6) // accessed
348

```

Hình 18. Định nghĩa cờ **PTE\_A**

Hàm **sys\_pgaccess(void)** đã được định nghĩa trước trong **kernel/sysproc.c** nên chúng ta có thể đi thẳng vào việc triển khai. Hàm này nhiệm vụ của nó là **báo cáo những trang đã được truy cập** với cơ chế hoạt động như sau: hàm **nhận thông tin** từ người dùng thông qua con trỏ **myproc**, sau đó lặp qua các **page** cần kiểm tra (nếu page được gắn cờ **PTE\_A** thao tác tăng **abits** và xóa cờ khỏi **PTE**), sau cùng copy **abits** vào user space.

Khởi tạo biến **p** là con trỏ đến cấu trúc **proc**, **abits** là biến để lưu trữ kết quả, sử dụng một bit cho mỗi trang và trang đầu tiên tương ứng với bit ít trọng nhất, **addr** là địa chỉ ảo bắt đầu của trang người dùng đầu tiên để kiểm tra, **num** là số lượng trang cần kiểm tra, và **dest** là địa chỉ người dùng để lưu kết quả dưới dạng một bitmask.

Chúng ta tiến hành duyệt qua mỗi trang cần kiểm tra, sử dụng hàm **walk** để tìm kiếm **PTE** của mỗi trang. Nếu không tìm thấy **PTE** cho trang, được giả định là trang không hợp lệ và tiếp tục với trang tiếp theo. Nếu **PTE tồn tại** và cờ **PTE\_A** được đặt, bit tương ứng trong **abits** được đặt thành **1** và cờ **PTE\_A** được xóa để đánh dấu là **truy cập đã được xác nhận**. Sử dụng hàm **copyout** để sao chép bitmask **abits** từ kernel space vào user space được chỉ định bởi **dest**. Nếu quá trình sao chép gặp lỗi, hàm trả về **-1**. Nếu mọi thứ diễn ra thành công, hàm trả về **0**.

```

kernel > C sysproc.c > sys_pgaccess(void)
73  #ifdef LAB_PGTBL
74  int
75  sys_pgaccess(void)
76  {
77      struct proc *p = myproc();
78      unsigned int abits=0;
79
80      uint64 addr;
81      argaddr(0, &addr);
82
83      int num;
84      argint(1,&num);
85
86      uint64 dest;
87      argaddr(2, &dest);
88
89
90      for(int i=0;i<num;i++){ // iterate through the number of pages
91          uint64 query_addr = addr + i * PGSIZE ;
92
93
94          pte_t *pte=walk(p->pagetable, query_addr, 0); // get the PTE of the address
95          if(*pte&PTE_A) // check if the accessed bit is set
96          {
97              abits=abits|(1<<i); // set the accessed bit
98              *pte=(*pte)&(~PTE_A); // clear the accessed bit
99          }
100     }
101
102     if(copyout(p->pagetable,dest,(char*)&abits, sizeof(abits)) < 0) // copy the accessed bits to the user space
103         return -1;
104
105     return 0;
106 }

```

Hình 19. Hàm sys\_pgaccess

### 2.3.2 Kết quả

Khi này phần **pgaccess** trong lệnh **./grade-lab-pgtbl pgtbltest** đã trả ra kết quả **OK**, chứng tỏ thành công trong task đầu tiên.

```

huytienlam@MSI:~/xv6-lab3$ ./grade-lab-pgtbl pgtbltest
make: 'kernel/kernel' is up to date.
== Test pgtbltest == (0.6s)
== Test pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
== Test pgtbltest: pgaccess ==
pgtbltest: pgaccess: OK

```

Hình 20. Điểm của pgaccess trong pgtbltest



## Chương 3

# Hoàn Thành

Quá trình **make grade** hiển thị các test chạy ra đều OK và số điểm được chấm là **46/46**. Tuy nhiên, chưa chắc chắn file **answers-pgtbl.txt** sẽ đúng hoàn toàn.

```

huytienlam@MSI:~/xv6-lab3$ make grade
make clean
make[1]: Entering directory '/home/huytienlam/xv6-lab3'
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grin
d user/_wc user/_zombie user/_pgtbltest \
*.zip \
ph barrier
make[1]: Leaving directory '/home/huytienlam/xv6-lab3'
./grade-lab-pgtbl -v
make[1]: Entering directory '/home/huytienlam/xv6-lab3'
riscv64-unknown-elf-gcc -c -o kernel/entry.o kernel/entry.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/kalloc.o kernel/kalloc.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/string.o kernel/string.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/main.o kernel/main.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/vm.o kernel/vm.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/proc.o kernel/proc.c
riscv64-unknown-elf-gcc -c -o kernel/switch.o kernel/switch.S
riscv64-unknown-elf-gcc -c -o kernel/trampoline.o kernel/trampoline.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/trap.o kernel/trap.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/syscall.o kernel/syscall.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/sysproc.o kernel/sysproc.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/fs.o kernel/fs.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/log.o kernel/log.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/sleeplock.o kernel/sleeplock.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/file.o kernel/file.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/pipe.o kernel/pipe.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/exec.o kernel/exec.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/sysfile.o kernel/sysfile.c
riscv64-unknown-elf-gcc -c -o kernel/kernelvec.o kernel/kernelvec.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/plic.o kernel/plic.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/virtio_disk.o kernel/virtio_disk.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/start.o kernel/start.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/console.o kernel/console.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/print.o kernel/print.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/uart.o kernel/uart.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/spinlock.o kernel/spinlock.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTLBL -DLAB_PGTLBL -MD -mcmodel=medany -ffreestanding -fno-common -nostd
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -march=rv64g -nostdinc -I. -Ikernel -c user/initcode.S -o user/initcode.o
riscv64-unknown-elf-objcopy -S -O binary user/initcode.out user/initcode
riscv64-unknown-elf-objdump -S user/initcode.o > user/initcode.asm
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/kalloc.o kernel/string.o kernel/main.o kernel/vm.o ker
nel/proc.o kernel/switch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/
file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o kernel/start.o kernel/console.o kernel/print.o kern
el/uart.o kernel/spinlock.o
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .*/ /; /$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/huytienlam/xv6-lab3'
== Test pgtbltest ==
$ make qemu-gdb
(2.0s)
== Test pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
== Test pgtbltest: pgaccess ==
pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.7s)
== Test answers-pgtbl.txt ==
answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(41.7s)
== Test usertests: all tests ==
usertests: all tests: OK
== Test time ==
time: OK
Screen: 46/46

```

Hình 21. Make grade



## Chương 4

# Tổng Kết

Qua đồ án **Page Tables**, chúng em được tìm hiểu sâu hơn về các bảng trang. Quá trình làm việc của nhóm chúng em khá rõ ràng, rành mạch, và có logic, phân chia công việc đều đặn. Chúng em xin chân thành cảm ơn!

MSSV	Họ và tên	Công việc	Hoàn thành
22127151	Lâm Tiến Huy	Speed Up System Calls. Làm báo cáo chính.	100%
22127290	Nguyễn Thị Thu Ngân	Print A Page Table. Làm báo cáo phụ.	100%
22127408	Kha Vĩnh Thuận	Detect Which Page Have Been Accessed.	100%

Hình 22. Bảng phân công công việc

# Tài liệu tham khảo

*Lab: System calls.* (n.d.). <https://pdos.csail.mit.edu/6.1810/2023/labs/syscall.html>

Cox, R., Kaashoek, F., & Morris, R. (2022). *xv6: a simple, Unix-like teaching operating system.*

<https://pdos.csail.mit.edu/6.1810/2023/xv6/book-riscv-rev3.pdf>

Waterman, A., Asanovic, K., Hauser, J., & RISC-V International. (2020). *The RISC-V Instruction Set Manual: Vol. Volume II: Privileged Architecture* (Document Version 20211203).

<https://drive.google.com/file/d/1EMip5dZlnypTk7pt4WWUKmtjUKTOkBqh/view>

*[Lab Report] MIT 6.S081 Lab: page tables (v2022).* (2023, February 13).

[https://jinzhec2.github.io/blog/post/6.s081\\_2022\\_lab3/?fbclid=IwAR1JRmEUe0RnBzAhnYP9TgqIUK5TYDkCC2y268yGkRdrvz4FxrHSzBx4Bw#print-a-page-table](https://jinzhec2.github.io/blog/post/6.s081_2022_lab3/?fbclid=IwAR1JRmEUe0RnBzAhnYP9TgqIUK5TYDkCC2y268yGkRdrvz4FxrHSzBx4Bw#print-a-page-table)

*xv6-labs-2022 Lab3 page tables 答案与解析.* (2022, November 7). <https://www.chens.life/posts/mit-xv6-lab3/>

*[MIT 6.S081] Lab 3: page tables\_ymprint-CSDN博客.* (n.d.).

<https://blog.csdn.net/LostUnravel/article/details/121340933>