**HỆ ĐIỀU HÀNH**

**PROJECT 3 – Page Table**

# **Quy tắc chung:**

- Project được thực hiện theo nhóm: mỗi nhóm có tối đa **3 sinh viên**

- **Tất cả các bài tập giống nhau sẽ được chấm 0 điểm trong toàn bộ quá trình thực hành (mặc dù các bài tập và dự án thực hành khác cũng được chấm điểm).**

- Môi trường cài đặt: **Linux**

# **Nộp bài:**

***Nộp bài tập trực tiếp trên trang web của khóa học (MOODLE), không chấp nhận bài nộp qua email hoặc các hình thức khác.***

Tên tệp: **StudentID1\_StudentID2\_StudentID3.zip** (với StudentID1 < StudentID2< StudentID3)

VD: Nhóm của bạn có 2 sinh viên: 2312001 và 2312002, tên tệp là: **2312001\_2312002\_2312003.zip**

**Bao gồm:**

* **StudentID1\_StudentID2\_StudentID3\_Report.pdf:** Báo cáo nên ngắn gọn và súc tích. Không cố gắng đưa mã nguồn vào báo cáo. Mục đích của báo cáo là để bạn có cơ hội làm rõ giải pháp của mình, nêu ra những khó khăn gặp phải, và bổ sung thông tin hữu ích cho việc chấm điểm. Nếu bạn gặp vấn đề cụ thể, thử các phương pháp không thành công, hoặc có yêu cầu chưa được thực hiện đầy đủ, thì việc giải thích trong báo cáo có thể giúp chúng tôi đánh giá và chấm điểm một cách công bằng hơn.
* **Realease** : Tệp diff (bản vá diff, Ví dụ: $ git diff >

< **StudentID1\_StudentID2\_StudentID3** .patch)

* **Source** : Tệp Zip của xv6 (phiên bản được make clean)

***Lưu ý: Cần thực hiện đúng các yêu cầu trên, nếu không, bài làm sẽ không được chấm.***

# **Đánh giá**

Bài làm của nhóm sẽ được chấm điểm dựa trên tính hoàn thiện, tính chính xác, phong cách lập trình, mức độ kiểm tra kỹ lưỡng, giải pháp và sự hiểu biết về mã nguồn.

Khi quản lý khóa học này, chúng tôi cố gắng hết sức để đánh giá công bằng từng cá nhân dựa trên sự đóng góp của mỗi người cho project.

# **Yêu cầu**

Bài thực hành này sẽ giúp bạn tìm hiểu bảng trang, sửa đổi chúng để triển khai các tính năng phổ biến của Hệ Điều Hành

Trước khi bắt đầu viết code, hãy đọc Chương 3 của [sách xv6](https://pdos.csail.mit.edu/6.828/2024/xv6/book-riscv-rev4.pdf) và các tệp liên quan:

* kernel/memlayout.h , dùng để ghi lại cách bố trí bộ nhớ.
* kernel/vm.c , chứa hầu hết code bộ nhớ ảo (VM).
* kernel/kalloc.c , chứa code để phân bổ và giải phóng bộ nhớ vật lý.

Bạn cũng có thể tham khảo [hướng dẫn về kiến ​​trúc RISC-V](https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view?usp=drive_link) .

Nếu bài lab này có câu hỏi, bạn hãy viết câu trả lời của mình trong file có tên **answers-\*.txt.**

# **Tăng tốc system call**

Một số hệ điều hành (ví dụ như Linux) tăng tốc một số lệnh gọi hệ thống bằng cách chia sẻ dữ liệu trong một vùng chỉ đọc giữa không gian người dùng và kernel. Điều này giúp loại bỏ nhu cầu phải chuyển qua lại giữa kernel và người dùng khi thực hiện các lệnh gọi hệ thống này. Để giúp bạn học cách chèn ánh xạ vào bảng trang, nhiệm vụ đầu tiên của bạn là thực hiện tối ưu hóa này cho lệnh gọi hệ thống getpid() trong xv6.

#### **Yêu cầu chi tiết:**

* Khi một tiến trình được tạo, ánh xạ một trang chỉ đọc tại địa chỉ ảo USYSCALL (một địa chỉ ảo được định nghĩa trong memlayout.h).
* Ở đầu trang này, lưu trữ một cấu trúc struct usyscall (cũng được định nghĩa trong memlayout.h) và khởi tạo nó để lưu trữ PID (Process ID) của tiến trình hiện tại.
* Trong bài lab này, hàm ugetpid() đã được cung cấp ở phía người dùng và sẽ tự động sử dụng ánh xạ USYSCALL.
* Bạn sẽ nhận được điểm tối đa cho phần này của bài lab nếu bài kiểm tra ugetpid vượt qua khi chạy pgtbltest.

#### Một số gợi ý:

1. **Chọn các bit quyền truy cập** cho phép không gian người dùng chỉ được đọc trang này (chỉ đọc).
2. Có một số thao tác cần thực hiện trong suốt vòng đời của một trang mới. Để tham khảo, hãy hiểu cách xử lý trapframe trong kernel/proc.c.

**Câu hỏi thêm**: Những lệnh gọi hệ thống nào khác trong xv6 có thể được tăng tốc bằng cách sử dụng trang chia sẻ này? Hãy giải thích.

# **In bảng trang**

Để giúp bạn hình dung về các bảng trang trong hệ thống RISC-V và có thể hỗ trợ cho việc gỡ lỗi trong tương lai, nhiệm vụ thứ hai của bạn là viết một hàm để in ra nội dung của bảng trang.

### Mô tả nhiệm vụ:

1. **Định nghĩa hàm**vmprint():
   * Hàm này nhận vào một tham số kiểu pagetable\_t và in bảng trang đó theo định dạng mô tả dưới đây.
2. **Chèn lệnh gọi**vmprint()**trong**exec.c:
   * Chèn lệnh if (p->pid == 1) vmprint(p->pagetable) ngay trước câu lệnh return argc trong exec.c để in bảng trang của tiến trình đầu tiên.
   * Bạn sẽ nhận được điểm tối đa cho phần này nếu vượt qua bài kiểm tra in bảng trang của lệnh make grade.
3. **Kết quả khi chạy**xv6:
   * Khi bạn khởi động hệ thống **xv6**, nó sẽ in ra bảng trang của tiến trình đầu tiên khi tiến trình đó vừa hoàn thành việc gọi exec() để thực thi tiến trình **init**.

Ví dụ đầu ra có thể trông như sau:

|  |
| --- |
| 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 |

* + **Dòng đầu tiên** hiển thị đối số mà bạn truyền cho hàm vmprint(), tức là địa chỉ của bảng trang.
  + Sau đó, mỗi dòng tiếp theo sẽ hiển thị thông tin về một mục PTE.
    - Mỗi mục PTE bao gồm **chỉ số PTE** trong trang của bảng trang, **giá trị PTE** (bao gồm các bit quyền truy cập) và **địa chỉ vật lý** được lấy từ PTE.
    - Dấu .. thể hiện độ sâu của mục PTE trong cấu trúc cây bảng trang.
    - **PTE không hợp lệ** sẽ không được in ra.

Ví dụ trên cho thấy rằng:

* + Bảng trang cấp đầu tiên ánh xạ các mục từ chỉ số 0 đến 255.
  + Cấp tiếp theo của bảng trang cho mục 0 chỉ có mục 0 được ánh xạ, và bảng trang cấp thấp nhất cho mục 0 có các mục từ 0 đến 2.

1. **Một số gợi ý**:
   * Bạn có thể đặt hàm vmprint() trong file kernel/vm.c.
   * Sử dụng các macro đã có sẵn ở cuối file kernel/riscv.h.
   * Hàm freewalk() có thể giúp bạn hình dung cách duyệt qua các bảng trang.
   * Định nghĩa prototype cho vmprint() trong kernel/defs.h để có thể gọi hàm này từ exec.c.
   * Sử dụng %p trong các lệnh printf để in ra các PTE và địa chỉ 64-bit theo định dạng hexa, như ví dụ.
2. **Giải thích chi tiết các trang lá trong đầu ra của**vmprint():
   * Với mỗi trang lá trong đầu ra, bạn cần giải thích những gì nó chứa về mặt logic và các bit quyền của nó. Bạn có thể tham khảo Hình 3.4 trong sách xv6, mặc dù lưu ý rằng hình ảnh này có thể có một số sự khác biệt so với các trang được kiểm tra trong tiến trình init ở đây.

# **Phát hiện các trang nào đã được truy cập**

Một số bộ thu gom rác (garbage collectors) có thể tận dụng thông tin về các trang bộ nhớ đã được truy cập (đọc hoặc ghi) để tối ưu hóa việc quản lý bộ nhớ. Trong phần này của bài lab, nhiệm vụ của bạn là thêm một tính năng vào hệ điều hành xv6 để phát hiện và báo cáo thông tin này cho không gian người dùng bằng cách kiểm tra các bit truy cập trong bảng trang của RISC-V. Bộ dò trang (page walker) của phần cứng RISC-V sẽ đánh dấu các bit này trong PTE (Page Table Entry) mỗi khi xảy ra lỗi TLB (TLB miss).

### Nhiệm vụ của bạn:

1. **Triển khai hàm**pgaccess():
   * Đây là một lệnh gọi hệ thống (system call) để báo cáo các trang bộ nhớ đã được truy cập.
   * Hàm pgaccess() nhận ba tham số:
     1. **Địa chỉ ảo bắt đầu** của trang người dùng đầu tiên cần kiểm tra.
     2. **Số lượng trang** cần kiểm tra.
     3. **Địa chỉ người dùng** trỏ tới một bộ đệm để lưu trữ kết quả vào một bitmask (một cấu trúc dữ liệu sử dụng một bit cho mỗi trang, với trang đầu tiên tương ứng với bit ít quan trọng nhất).
2. **Đạt điểm tối đa**:
   * Bạn sẽ nhận được điểm tối đa nếu bài kiểm tra pgaccess vượt qua khi chạy bài test pgtbltest.

### Một số gợi ý để thực hiện:

1. **Đọc hàm**pgaccess\_test() trong user/pgtbltest.c để hiểu cách sử dụng hàm pgaccess().
2. **Triển khai hàm**sys\_pgaccess() trong kernel/sysproc.c.
3. **Phân tích các tham số**:
   * Sử dụng các hàm argaddr() và argint() để lấy địa chỉ và số lượng trang từ không gian người dùng.
4. **Xử lý bitmask đầu ra**:
   * Sử dụng một bộ đệm tạm thời trong kernel để lưu trữ bitmask, sau đó sao chép nó vào không gian người dùng (sử dụng copyout()) sau khi đã điền các bit.
5. **Giới hạn số trang quét**:
   * Bạn có thể đặt một giới hạn về số lượng trang có thể được quét để tránh quét quá nhiều trang.
6. **Sử dụng hàm**walk()**trong**kernel/vm.c:
   * Hàm này rất hữu ích để tìm các PTE và kiểm tra bit truy cập.
7. **Định nghĩa PTE\_A (bit truy cập)** trong kernel/riscv.h:
   * Tham khảo tài liệu về kiến trúc RISC-V để xác định giá trị của bit PTE\_A.
8. **Xóa bit PTE\_A sau khi kiểm tra**:
   * Nếu bit PTE\_A được đặt, bạn cần xóa nó để đảm bảo rằng chỉ các trang đã được truy cập kể từ lần cuối gọi pgaccess() mới được đánh dấu.
9. **Sử dụng**vmprint()**để gỡ lỗi**:
   * Hàm vmprint() có thể giúp bạn in ra bảng trang và kiểm tra các PTE trong quá trình phát triển.

### **Mục tiêu cuối cùng:**

Khi hoàn thành, bạn sẽ có thể kiểm tra các trang bộ nhớ đã được truy cập bằng cách sử dụng lệnh gọi hệ thống pgaccess(). Bitmask sẽ chứa thông tin về các trang đã được truy cập, giúp bộ thu gom rác (garbage collector) hoặc các cơ chế quản lý bộ nhớ khác hoạt động hiệu quả hơn.

# **Tài liệu tham khảo**

https://pdos.csail.mit.edu/6.828/2023/labs/pgtbl.html