ĐẠI HỌC QUỐC GIA TPHCM
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

Báo cáo bài tập

Đề tài: xv6

Môn học: Operating System

Sinh viên thực hiện:

Phạm Quốc Nam Anh	23120111
Nguyễn Xuân Duy	23120121
Nguyễn Minh Hiếu	23120124
Huỳnh Thái Toàn	23120175

Giáo viên hướng dẫn:

Lê Viết Long

Ngày 15 tháng 12 năm 2025

Mục lục

1 Tăng tốc syscall

1.1 Mô tả bài toán

Tối ưu hóa lệnh gọi hệ thống `getpid()` bằng cách chia sẻ một vùng nhớ chỉ đọc giữa kernel và user. Thay vì trap vào kernel mỗi lần gọi getpid, tiến trình user có thể đọc trực tiếp PID từ một trang bộ nhớ đặc biệt đã được ánh xạ sẵn. Trang này chứa một struct `usyscall` với trường `pid` lưu PID của tiến trình hiện tại.
Ngoài ra, viết chương trình kiểm tra `pidget()` để so sánh giá trị trả về từ getpid() (system call thông thường) và ugetpid() (đọc trực tiếp từ vùng chia sẻ). Bài tập đạt yêu cầu nếu hai giá trị này trùng nhau.

1.2 Phương pháp thực hiện

1. Trong `proc_pagetable()`, khởi tạo page table mới và ánh xạ các vùng bắt buộc (trampoline, trapframe).
2. Trong `allocproc()`, cấp phát và khởi tạo tiến trình, đồng thời xử lý cấp phát và ánh xạ trang `USYSCALL`.
3. Trong `freeproc()`, giải phóng tài nguyên tiến trình, bao gồm cả trang `USYSCALL` nếu tồn tại.

1.2.1 Khởi tạo page table

Trong hàm `proc_pagetable()`:

- Tạo một page table mới bằng `uvmcreate()`.
- Ánh xạ `trampoline` và `trapframe` vào không gian địa chỉ user.
- Đảm bảo các ánh xạ này có quyền truy cập phù hợp.

1.2.2 Cấp phát và khởi tạo tiến trình

Trong hàm `allocproc()`:

- Tìm một tiến trình rảnh trong mảng `proc[]` và khởi tạo các trường cơ bản (PID, trạng thái, trapframe).

- Cấp phát một trang cho `usyscall` bằng `kalloc()`.
- Khởi tạo trường `pid` trong `struct usyscall`.
- Ánh xạ địa chỉ ảo `USYSCALL` tới trang vật lý chứa `struct usyscall` bằng `mappages()` với quyền chỉ đọc cho user.

1.2.3 Giải phóng tiến trình

Trong hàm `freeproc()`:

- Giải phóng vùng nhớ được cấp cho `usyscall` nếu tồn tại.
- Đặt lại trạng thái tiến trình về `UNUSED`.

Trong hàm `proc_freepagetable()`:

- Bỏ ánh xạ vùng `USYSCALL` khỏi page table bằng `uvmunmap()`.
- Giải phóng toàn bộ page table của tiến trình.

2 In bảng trang

2.1 Mô tả bài toán

Viết hàm `vmprint(pagetable_t)` để in nội dung bảng trang trong RISC-V theo định dạng phân cấp. Mỗi cấp được thể hiện bằng dấu `..`, in ra chỉ số PTE, giá trị PTE (bao gồm các bit quyền truy cập), và địa chỉ vật lý của các PTE hợp lệ. Chèn lệnh gọi `vmprint()` vào `exec.c` để in bảng trang của tiến trình init (PID = 1).

2.2 Phương pháp thực hiện

1. Trong `kernel/vm.c`, viết hàm đệ quy `vmprintlevel()` để duyệt qua các cấp của bảng trang.
2. Viết hàm `vmprint()` để in địa chỉ bảng trang và gọi hàm đệ quy.
3. Trong `kernel/defs.h`, khai báo prototype của hàm `vmprint()`.
4. Trong `kernel/exec.c`, chèn lệnh gọi `vmprint()` cho tiến trình có PID = 1.

2.2.1 Duyệt qua bảng trang

Trong hàm `vmprintlevel()`:

- Duyệt qua 512 PTE trong bảng trang hiện tại.
- Chỉ xử lý các PTE hợp lệ (có bit `PTE_V` được set).
- In ra độ sâu (số lượng `..`), chỉ số PTE, giá trị PTE, và địa chỉ vật lý.
- Phân biệt PTE là bảng trang con (không có bit R|W|X) hay là trang lá (có ít nhất một bit R|W|X).
- Nếu PTE trỏ đến bảng trang con, đệ quy xuống cấp tiếp theo với độ sâu tăng thêm 1.

2.2.2 In bảng trang

Trong hàm `vmprint()`:

- In dòng đầu tiên chứa địa chỉ của bảng trang: `"page table 0x..."`.

- Gọi `vmprintlevel()` với tham số bảng trang và độ sâu ban đầu là 0.
- Sử dụng macro `PTE2PA()` từ `kernel/riscv.h` để chuyển đổi PTE thành địa chỉ vật lý.

2.2.3 Khai báo và sử dụng

Trong hàm `kernel/defs.h`:

- Thêm dòng khai báo `void vmprint(pagetable_t);` vào phần khai báo các hàm của `vm.c`.

Trong hàm `exec()` của `kernel/exec.c`:

- Sau khi gán `p->pagetable = pagetable` và trước câu lệnh `return argc`.
- Thêm điều kiện kiểm tra: `if(p->pid == 1) vmprint(p->pagetable)`.
- Điều này đảm bảo chỉ in bảng trang của tiến trình init (tiến trình đầu tiên).

2.3 Cấu trúc bảng trang và logic ánh xạ

2.3.1 Cấu trúc bảng trang phân cấp trong RISC-V

Hệ thống RISC-V sử dụng bảng trang phân cấp 3 cấp (3-level page table) để ánh xạ địa chỉ ảo 39-bit sang địa chỉ vật lý. Mỗi cấp bảng trang chứa 512 mục (Page Table Entry - PTE), mỗi PTE có kích thước 64-bit.
Cấu trúc địa chỉ ảo 39-bit được chia thành các phần:

- **Bits 38-30 (9 bits):** Chỉ số vào bảng trang cấp 2 (L2) - 512 mục
- **Bits 29-21 (9 bits):** Chỉ số vào bảng trang cấp 1 (L1) - 512 mục
- **Bits 20-12 (9 bits):** Chỉ số vào bảng trang cấp 0 (L0) - 512 mục
- **Bits 11-0 (12 bits):** Offset trong trang vật lý - 4096 bytes

2.3.2 Logic ánh xạ địa chỉ

Quá trình dịch địa chỉ ảo sang địa chỉ vật lý diễn ra như sau:

1. **Bắt đầu từ bảng trang gốc (root page table):** Địa chỉ bảng trang gốc được lưu trong thanh ghi `satp`.

2. **Cấp 2 (L2):** Sử dụng bits 38-30 của địa chỉ ảo làm chỉ số để tra cứu PTE tương ứng trong bảng trang cấp 2.

3. **Kiểm tra PTE:**

   - Nếu PTE không hợp lệ (bit `V` = 0): Lỗi page fault.
   - Nếu PTE là trang lá (có ít nhất một bit R, W, hoặc X = 1): Lấy địa chỉ vật lý từ PPN (Physical Page Number) và kết hợp với offset.
   - Nếu PTE là bảng trang con (R = W = X = 0): Chuyển sang cấp tiếp theo.

4. **Cấp 1 (L1):** Nếu PTE ở cấp 2 là bảng trang con, sử dụng PPN để tìm bảng trang cấp 1, sau đó dùng bits 29-21 làm chỉ số tra cứu PTE.

5. **Cấp 0 (L0):** Tương tự, nếu PTE ở cấp 1 là bảng trang con, tiếp tục với bits 20-12.

6. **Tính địa chỉ vật lý cuối cùng:** Khi tìm được trang lá, địa chỉ vật lý = (PPN « 12) | offset (12 bits cuối của địa chỉ ảo).

2.3.3 Trang lá (Leaf Page) và bảng trang con

Trang lá là PTE cuối cùng trong chuỗi tra cứu, trỏ đến trang vật lý thực sự chứa dữ liệu. Đặc điểm nhận biết trang lá:

- Có ít nhất một bit trong R (Read), W (Write), hoặc X (Execute) được set = 1.
- PPN của trang lá chứa địa chỉ vật lý của trang dữ liệu (data page).
- Trang lá có thể xuất hiện ở bất kỳ cấp nào (L2, L1, hoặc L0), cho phép superpage (trang lớn hơn 4KB).

Bảng trang con (intermediate page table) là PTE trỏ đến bảng trang ở cấp thấp hơn. Đặc điểm:

- Không có bit R, W, X nào được set (R = W = X = 0).
- PPN của bảng trang con chứa địa chỉ vật lý của bảng trang cấp tiếp theo.
- Chỉ xuất hiện ở cấp 2 (L2) hoặc cấp 1 (L1), không ở cấp 0 (L0).

2.3.4 Quyền truy cập (Permission Bits)

Mỗi PTE chứa các bit quyền truy cập quan trọng:

- **V (Valid, bit 0):** PTE có hợp lệ hay không. Nếu V = 0, mọi truy cập đều gây page fault.
- **R (Read, bit 1):** Cho phép đọc. Nếu R = 1, trang có thể được đọc.
- **W (Write, bit 2):** Cho phép ghi. Nếu W = 1, trang có thể được ghi. Thường W = 1 kéo theo R = 1.
- **X (Execute, bit 3):** Cho phép thực thi. Nếu X = 1, trang có thể chứa mã lệnh để thực thi.
- **U (User, bit 4):** Cho phép user mode truy cập. Nếu U = 1, trang có thể được truy cập từ user mode, nếu U = 0, chỉ kernel mode mới truy cập được.
- **G (Global, bit 5):** Ánh xạ toàn cục, không bị xóa khi chuyển ngữ cảnh.
- **A (Accessed, bit 6):** Đánh dấu trang đã được truy cập.
- **D (Dirty, bit 7):** Đánh dấu trang đã được ghi.

Các tổ hợp quyền thường gặp:

- `R=1, W=0, X=0, U=1`: Trang chỉ đọc của user (read-only data).
- `R=1, W=1, X=0, U=1`: Trang đọc-ghi của user (data, heap, stack).
- `R=1, W=0, X=1, U=1`: Trang mã lệnh của user (code/text segment).
- `R=1, W=1, X=0, U=0`: Trang đọc-ghi của kernel.
- `R=0, W=0, X=0, V=1`: Bảng trang con (không phải trang lá).

2.4 Phân tích output của vmprint()

Khi chạy hàm `vmprint()`, output có dạng:

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
```

```
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
```

Giải thích từng dòng:

- **Dòng 1:** `page table 0x0000000087f6e000` - Địa chỉ vật lý của bảng trang gốc (root page table).
- **Dòng 2:** `..0: pte 0x0000000021fda801 pa 0x0000000087f6a000`
  - `..0`: Cấp 2 (L2), PTE tại chỉ số 0.
  - `pte 0x0000000021fda801`: Giá trị PTE. Bit 0 (V) = 1 (hợp lệ), các bit R|W|X = 0 → Đây là bảng trang con.
  - `pa 0x0000000087f6a000`: Địa chỉ vật lý của bảng trang cấp 1.
- **Dòng 3:** `.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000`
  - `.. ..0`: Cấp 1 (L1), PTE tại chỉ số 0.
  - Giá trị PTE cho thấy đây cũng là bảng trang con (R|W|X = 0).
  - Trỏ đến bảng trang cấp 0 tại địa chỉ `0x87f69000`.
- **Dòng 4:** `.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000`
  - `.. .. ..0`: Cấp 0 (L0), PTE tại chỉ số 0.
  - `pte 0x0000000021fdac1f`: Giá trị PTE kết thúc bằng `0x1f` (binary: 11111) → V=1, R=1, W=1, X=1, U=1.
  - Đây là **trang lá** với quyền đọc, ghi, thực thi cho user mode.
  - `pa 0x0000000087f6b000`: Địa chỉ vật lý của trang dữ liệu.
- **Dòng 5:** `.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000`
  - `.. .. ..1`: Cấp 0 (L0), PTE tại chỉ số 1.
  - `pte 0x0000000021fda00f`: Kết thúc bằng `0x0f` (binary: 01111) → V=1, R=1, W=1, X=1, U=0.
  - Đây là **trang lá** với quyền đọc, ghi, thực thi nhưng chỉ cho kernel mode (U=0).

– Địa chỉ vật lý: `0x87f68000`.

Từ output này, ta thấy rõ cấu trúc phân cấp: Bảng trang gốc (L2) → Bảng trang con (L1) → Bảng trang con (L0) → Các trang lá chứa dữ liệu thực sự. Mỗi cấp được thể hiện bằng số lượng dấu `..`, và quyền truy cập được mã hóa trong các bit cuối của giá trị PTE.

3 Các trang đã được truy cập

3.1 Mô tả bài toán

Trong các hệ điều hành hiện đại, việc quản lý bộ nhớ hiệu quả đóng vai trò quan trọng đối với hiệu năng hệ thống. Một số bộ thu gom rác (garbage collector) và cơ chế quản lý bộ nhớ nâng cao có thể tận dụng thông tin về việc các trang bộ nhớ đã được truy cập (đọc hoặc ghi) để đưa ra các quyết định tối ưu, chẳng hạn như ưu tiên giữ lại các trang thường xuyên được sử dụng hoặc thu hồi các trang ít được truy cập.

Trên kiến trúc RISC-V, phần cứng hỗ trợ việc theo dõi truy cập bộ nhớ thông qua các bit trạng thái trong Page Table Entry (PTE). Cụ thể, bit `PTE_A` (Accessed bit) sẽ được phần cứng tự động thiết lập khi một trang bộ nhớ được truy cập và xảy ra TLB miss. Tuy nhiên, thông tin này không được cung cấp trực tiếp cho không gian người dùng.

Bài toán đặt ra là mở rộng hệ điều hành xv6 bằng cách cài đặt một lệnh gọi hệ thống mới, cho phép chương trình người dùng truy vấn các trang bộ nhớ đã được truy cập kể từ lần kiểm tra trước đó. Lệnh gọi hệ thống này phải đọc trạng thái bit truy cập trong bảng trang, tổng hợp kết quả và trả về cho không gian người dùng dưới dạng một bitmask.

3.2 Phương pháp thực hiện

3.2.1 Xử lý tham số hệ thống

Trong hàm `sys_pgaccess()`, các tham số được trích xuất từ không gian người dùng bằng các hàm hỗ trợ của xv6:

- `argaddr()` để lấy địa chỉ ảo bắt đầu và địa chỉ bộ đệm kết quả.
- `argint()` để lấy số lượng trang cần kiểm tra.

Để đảm bảo an toàn và tránh quét quá nhiều trang, số lượng trang được giới hạn ở một giá trị tối đa cho phép.

3.2.2 Duyệt bảng trang và kiểm tra bit truy cập

Hệ điều hành sử dụng hàm `walk()` trong `kernel/vm.c` để truy xuất Page Table Entry tương ứng với từng địa chỉ ảo của các trang cần kiểm tra. Với mỗi PTE hợp lệ:

- Kiểm tra bit `PTE_A` để xác định xem trang đã được truy cập hay chưa.
- Nếu bit `PTE_A` được thiết lập, bit tương ứng trong bitmask sẽ được đặt.

Bit `PTE_A` được định nghĩa trong `kernel/riscv.h` dựa trên đặc tả của kiến trúc RISC-V.

3.2.3 Xóa bit truy cập sau khi kiểm tra

Sau khi phát hiện một trang đã được truy cập, bit `PTE_A` trong PTE sẽ được xóa. Việc này đảm bảo rằng:

- Các lần gọi `pgaccess()` tiếp theo chỉ phản ánh các truy cập mới.
- Thông tin trả về luôn thể hiện chính xác các trang được truy cập kể từ lần kiểm tra gần nhất.

3.2.4 Trả kết quả về không gian người dùng

Bitmask kết quả được lưu tạm thời trong một biến của kernel. Sau khi hoàn tất quá trình quét:

- Hàm `copyout()` được sử dụng để sao chép bitmask từ không gian kernel sang bộ đệm trong không gian người dùng.
- Việc sao chép này đảm bảo tính an toàn bộ nhớ giữa kernel và user space.

—