Trường Đại học Khoa học tự nhiên, ĐHQG-HCM
Khoa Công nghệ thông tin

HỆ ĐIỀU HÀNH – CQ2023_22

BÁO CÁO ĐỒ ÁN 3

GVHD: Lê Viết Long

Thành viên:

23120194 – Võ Cao Tâm Chính
23120197 – Trà Văn Sỹ
23120244 – Nguyễn Nhật Duy
23120283 – Phạm Quốc Khánh
23120314 – Phạm An Ninh

Tháng 12 năm 2025

MỤC LỤC

A. BÁO CÁO NHÓM

MSSV	Họ và tên	Vai trò	Phân công	Mức độ hoàn thành
23120194	Võ Cao Tâm Chính	Thành viên	Bài 2	100%
23120197	Trà Văn Sỹ	Trưởng nhóm	Bài 1 Tổng hợp báo cáo	100%
23120244	Nguyễn Nhật Duy	Thành viên	Bài 2	100%
23120283	Phạm Quốc Khánh	Thành viên	Bài 3	100%
23120314	Phạm An Ninh	Thành viên	Bài 3	100%

B. BÁO CÁO ĐỒ ÁN

Bài 1: Tăng tốc system call

1. Yêu cầu bài toán

Một số hệ điều hành hiện đại tối ưu hóa hiệu năng bằng cách chia sẻ một vùng dữ liệu chỉ đọc (read-only) giữa kernel-space và user-space. Điều này giúp giảm thiểu chi phí chuyển đổi ngữ cảnh (context switch) khi thực hiện system call, vì người dùng có thể đọc dữ liệu trực tiếp từ bộ nhớ mà không cần trap vào kernel. Nhiệm vụ của bài tập là áp dụng kỹ thuật này cho system call getpid() trong xv6.

2. Cách giải quyết

Thay vì gọi getpid() để lấy id của 1 tiến trình bằng system call, ta cấp phát một trang nhớ vật lí chứa pid trong trong tiến trình đó, rồi map trang đó vào không gian user ở một địa chỉ ảo cố định USYSCALL với quyền chỉ đọc. User chỉ cần đọc trực tiếp dữ liệu từ vùng nhớ này để lấy pid, nhờ đó tránh trap vào kernel và tăng tốc đáng kể.

- Đầu tiên, trong kernel/memlayout.h, ta định nghĩa hằng số USYSCALL. Đây chính là địa chỉ ảo nơi trang chia sẻ được ánh xạ tới. Địa chỉ này được đặt ngay dưới trang TRAPFRAME đúng 1 trang.

- Tại file này, ta cũng sẽ định nghĩa một cấu trúc struct *usyscall* chứa một trường số nguyên pid lưu id của tiến trình. Sau đó, trong kernel/proc.h, ta bổ sung một con trỏ struct usyscall *usyscall vào trong struct proc. Con trỏ này sẽ giữ địa chỉ vật lí (trong kernel) của trang bộ nhớ được cấp phát.

- Tiếp theo, trong kernel/proc.c, ta sẽ tiến hành chỉnh sửa các hàm allocproc(), proc_pagetable(), proc_freepagetable(), freeproc() để thực hiện các yêu cầu tiếp theo:

+ Quá trình cấp phát và khởi tạo trang nhớ sẽ diễn ra trong hàm allocproc(). Tại đây, ta dùng hàm kalloc() để cấp phát một trang vật lí cho con trỏ p->usyscall. Nếu cấp phát thất bại ta giải phóng tài nguyên, mở khóa tiến trình và trả về lỗi. Ngay sau khi cấp phát thành công và tiến trình đã có pid (p->pid), ta thực hiện gán giá trị pid này vào trường pid của trang vừa tạo (p->usyscall->pid = p->pid).

+ Thao tác ánh xạ trang vào không gian user sẽ thực hiện bên trong proc_pagetable(). Sau khi map TRAPFRAME và TRAMPOLINE, ta dùng hàm mappages() để ánh xạ địa chỉ ảo USYSCALL tới trang vật lí chứa struct usyscall (p->usyscall) của tiến trình. Để đảm bảo an toàn, ta thiết lập cờ PTE_R (read) và PTE_U (user) cho phép người dùng chỉ đọc chứ không được thay đổi pid. Nếu mappages() không thành công, ta chạy lệnh uvmunmap để hủy ánh xạ cho địa chỉ TRAMPOLINE và TRAPFRAME đã ánh xạ trước đó, rồi chạy uvmfree() để giải phóng page table.

+ Khi tiến trình kết thúc, ta cần giải phóng bộ nhớ để tránh rò rỉ (memory leak). Trong proc_freepagetable(), ta gọi uvmunmap() để gỡ bỏ ánh xạ của USYSCALL khỏi bảng trang (sau khi thực hiện với TRAPFRAME và TRAMPOLINE trước đó), rồi gọi uvmfree() để giải phóng page table. Tiếp đến, trong freeproc(), ta sẽ gọi kfree() cho con trỏ p->usyscall để giải phóng trang bộ nhớ đã được cấp cho usyscall trước đó.

- Lúc này, thao tác đọc pid của 1 tiến trình sẽ không gọi lệnh assembly ecall để trap vào kernel nữa. Thay vào đó, nó ép kiểu địa chỉ USYSCALL thành con trỏ struct usyscall*, sau đó đọc và trả về giá trị của trường pid. Việc này nhanh hơn nhiều so với quy trình system call truyền thống vì không tốn chi phí chuyển đổi context giữa user mode và kernel mode.

Câu hỏi bổ sung: Những lệnh gọi hệ thống nào khác có thể được tăng tốc?

Ngoài getpid, các lệnh gọi hệ thống sau rất phù hợp để sử dụng cơ chế trang chia sẻ:

1. uptime() (Lấy thời gian hệ thống):

Biến đếm thời gian (ticks) được Kernel cập nhật liên tục qua ngắt đồng hồ. Thay vì user phải gọi system call để đọc biến này, Kernel có thể ánh xạ biến ticks vào một trang nhớ chỉ đọc dùng chung cho tất cả tiến trình. Điều này giúp giảm tải đáng kể cho các ứng dụng cần đo đạc thời gian thực.

2. sysinfo() (Thông tin thống kê hệ thống):

Các thông số như dung lượng RAM trống (freemem) hay số lượng tiến trình đang chạy (nproc) là các biến toàn cục. Kernel có thể cập nhật chúng vào trang chia sẻ mỗi khi tài nguyên thay đổi. Các công cụ giám sát hệ thống có thể đọc trực tiếp để hiển thị real-time mà không cần thực hiện bẫy (trap) vào kernel liên tục.

3. getuid() / getgid() (Lấy định danh người dùng/nhóm):

Tương tự như getpid, thông tin về chủ sở hữu của tiến trình (User ID) thường không thay đổi trong suốt vòng đời tiến trình. Việc lưu trữ nó trong cấu trúc usyscall tại trang chia sẻ cho phép truy xuất cực nhanh phục vụ việc kiểm tra quyền truy cập.

Bài 2: In bảng trang

1. Yêu cầu bài toán

Cài đặt hàm **vmprint()** để in ra cấu trúc bảng trang của một tiến trình. Hàm này nhận vào một tham số có kiểu **pagetable_t** và in ra bảng trang theo định dạng phân cấp. Để kiểm tra kết quả, ta thêm lệnh gọi hàm vmprint() trong tệp tin **kernel/exec.c** với điều kiện kiểm tra pid bằng một, nhằm in ra bảng trang của tiến trình đầu tiên trong hệ thống.

2. Cách giải quyết

Để giải quyết bài toán, ta cần hiểu rằng bảng trang trong xv6 là một cấu trúc cây 3 tầng (Level 2 -> Level 1 -> Level 0). Do đó, giải pháp tự nhiên và hiệu quả nhất là sử dụng thuật toán đệ quy.

Xây dựng hàm in ấn và duyệt cây (kernel/vm.c)

- Ta cần định nghĩa hàm vmprint() trong file kernel/vm.c. Hàm này đóng vai trò là điểm bắt đầu, in ra địa chỉ của bảng trang gốc, sau đó gọi một hàm phụ trợ để thực hiện việc duyệt đệ quy.

- Chiến lược duyệt đệ quy: Ta xây dựng một hàm phụ trợ (vmprint_walk) nhận vào bảng trang hiện tại và độ sâu (depth). Hàm này sẽ hoạt động như sau:

  - Sử dụng vòng lặp để duyệt qua tất cả 512 mục (entry) trong trang của bảng trang hiện tại.
  - Với mỗi mục, ta kiểm tra bit hợp lệ (PTE_V). Nếu mục này không hợp lệ, ta bỏ qua.
  - Nếu mục hợp lệ, ta in ra thông tin theo định dạng yêu cầu: sử dụng vòng lặp dựa trên biến level để in số lượng dấu .. tương ứng (thể hiện độ sâu), tiếp theo là chỉ số index, giá trị PTE dạng hexa, và địa chỉ vật lý được trích xuất từ PTE (sử dụng macro PTE2PA).
  - Quyết định đệ quy hay dừng: Dựa vào các bit quyền (PTE_R, PTE_W, PTE_X), ta xác định xem PTE hiện tại trỏ đến một bảng trang con (Directory) hay trỏ đến một trang dữ liệu vật lý thực sự (Leaf).

- Nếu các bit R, W, X đều bằng 0, đây là một nút trung gian (Directory). Ta trích xuất địa chỉ vật lý của bảng con, chuyển đổi sang địa chỉ ảo kernel và gọi đệ quy hàm duyệt với cấp độ sâu tăng lên (level + 1).
- Nếu một trong các bit R, W, X được bật, đây là trang lá (Leaf). Ta chỉ in thông tin và không đệ quy tiếp.

Khai báo nguyên mẫu hàm (kernel/defs.h)

Để hàm vmprint() có thể được gọi từ các file khác trong kernel (cụ thể là exec.c), ta cần thêm nguyên mẫu (prototype) của nó vào file kernel/defs.h. Việc này giúp trình biên dịch nhận biết sự tồn tại và kiểu dữ liệu của hàm trước khi liên kết.

Tích hợp vào tiến trình hệ thống (kernel/exec.c)

- Mục tiêu là in bảng trang của tiến trình init (tiến trình đầu tiên được tạo ra bởi kernel).

- Trong hàm exec() tại file kernel/exec.c, ngay trước khi hàm trả về thành công (trước dòng return argc), ta chèn một đoạn mã kiểm tra PID của tiến trình hiện tại (p->pid).

- Nếu p->pid == 1, nghĩa là đây là tiến trình init vừa khởi tạo xong không gian bộ nhớ của nó, ta thực hiện gọi hàm vmprint(p->pagetable). Điều này đảm bảo bảng trang được in ra ngay khi cấu trúc bộ nhớ của tiến trình đầu tiên đã hoàn thiện nhưng chưa bắt đầu thực thi mã người dùng.

- Trong kết quả đầu ra của vmprint đối với tiến trình init, các dòng có độ sâu lớn nhất (ví dụ: .. .. ..0, ...1) đại diện cho các *Trang lá (Leaf Pages)*. Đây là nơi ánh xạ địa chỉ ảo sang bộ nhớ vật lý chứa dữ liệu thực sự của chương trình. Cụ thể đối với một tiến trình đơn giản như init:

1. Trang chứa Mã lệnh và Dữ liệu (Text & Data Segment):

- Về mặt logic: Trang lá đầu tiên (thường ở chỉ số 0) chứa mã nhị phân (instructions) của chương trình init để CPU thực thi, cùng với các biến dữ liệu đã được khởi tạo.

- **Bit quyền:** Các PTE này thường có bit **PTE_R (Read)** và **PTE_X (Execute)** được bật (bit set = 1). Quyền thực thi (X) là bắt buộc để CPU nạp lệnh từ trang này. Nếu trang chứa cả dữ liệu thay đổi được, nó cũng có thể có bit **PTE_W (Write)**. Ngoài ra, bit **PTE_U (User)** phải được bật để chương trình chạy trong chế độ người dùng có thể truy cập.

2. Trang chứa Ngăn xếp (Stack Segment):

- Về mặt logic: Một trang lá khác (thường ở địa chỉ cao hơn hoặc được cấp phát tiếp theo) đóng vai trò là Stack của tiến trình. Nó chứa các biến cục bộ, khung ngăn xếp (stack frames) cho các lời gọi hàm và các tham số dòng lệnh.

- Bit quyền: Trang này bắt buộc phải có bit **PTE_R (Read)** và **PTE_W (Write)** để chương trình có thể lưu và đọc dữ liệu tạm thời. Bit **PTE_X (Execute)** thường bị tắt để ngăn chặn các cuộc tấn công thực thi mã trên stack (như buffer overflow). Bit **PTE_U** cũng được bật.

3. Trang Trapframe (Lưu ý đặc biệt):

- Trong xv6, có một trang đặc biệt là Trapframe được ánh xạ ở địa chỉ rất cao trong không gian ảo (không nằm liền kề với code/data). Trang này dùng để lưu trữ trạng thái các thanh ghi của user khi chuyển sang kernel mode.

- Bit quyền: Trang này có quyền **PTE_R** và **PTE_W**, nhưng **không** có bit PTE_U (User không được phép truy cập trực tiếp), chỉ kernel mới có quyền truy cập khi xử lý ngắt/bẫy.

Bài 3: Phát hiện các trang đã được truy cập

1. Yêu cầu bài toán

- Hệ điều hành cần cung cấp thông tin cho người dùng biết những trang bộ nhớ nào đã được truy cập (đọc hoặc ghi) kể từ lần kiểm tra cuối cùng. Để làm điều này, ta cần cài đặt một system call mới tên là pgaccess.

- System call này nhận vào 3 tham số: địa chỉ ảo bắt đầu (base), số lượng trang cần kiểm tra (len), và địa chỉ bộ đệm phía user để lưu kết quả bitmask (mask_addr).

- Cơ chế dựa trên việc phần cứng RISC-V tự động bật bit "Accessed" khi có TLB miss. Nhiệm vụ của phần mềm là kiểm tra bit này, báo cáo lại vào bitmask và xóa nó đi để phục vụ lần đo tiếp theo.

2. Cách giải quyết

Bước 1: Định nghĩa cờ PTE_A (Bit Accessed)

- Ta cần hiểu cấu trúc của một Page Table Entry (PTE). Kiến trúc RISC-V sử dụng bit thứ 6 để đánh dấu "đã truy cập", nhưng bit này chưa được định nghĩa trong xv6.

- Trong file kernel/riscv.h, ta định nghĩa thêm cờ PTE_A với giá trị là (1L << 6), tạo nền tảng để kernel có thể đọc/ghi trạng thái truy cập của trang.

Bước 2: Đăng ký System Call phía Kernel

- Ta cần đăng ký số hiệu và hàm xử lý để kernel nhận diện được lệnh gọi từ user:

+ Tại kernel/syscall.h: Định nghĩa mã số #define SYS_pgaccess 30.

+ Tại kernel/syscall.c: Khai báo extern uint64 sys_pgaccess(void); và ánh xạ nó vào mảng syscalls[] tại vị trí [SYS_pgaccess]

+ Tại kernel/defs.h: Khai báo prototype uint64 sys_pgaccess(void);

Bước 3: Chuẩn bị giao diện ở User Space

Để chương trình người dùng gọi được kernel:

- Tại user/user.h: Khai báo hàm int pgaccess(void *base, int len, void *mask);.

- Tại user/usys.pl: Thêm dòng entry("pgaccess");. Script Perl này sẽ tự động sinh ra file assembly usys.S chứa mã cầu nối ecall để chuyển ngữ cảnh từ user sang kernel.

Bước 4: Nhận tham số trong sys_pgaccess

- Việc cài đặt chính nằm trong file kernel/sysproc.c. Hàm sys_pgaccess sử dụng các hàm hỗ trợ để lấy tham số từ useuser:

1. Địa chỉ ảo bắt đầu (argaddr).
2. Số lượng trang cần kiểm tra (argint).
3. Địa chỉ buffer nhận kết quả (argaddr).

- Ta phải kiểm tra tham số số lượng trang. Vì kết quả trả về là một số nguyên 64-bit (bitmask), nên hệ thống chỉ hỗ trợ kiểm tra tối đa 64 trang một lần. Nếu len > 64 hoặc len < 0, hàm sẽ trả về -1 (lỗi).

Bước 5: Xử lý logic kiểm tra và xóa cờ

- Khởi tạo một biến bitmask tạm thời uint64 kmask = 0.

- Duyệt vòng lặp từ trang i = 0 đến len - 1:

+ Tính địa chỉ ảo của trang thứ i: va = base + i * PGSIZE.

+ Gọi hàm walk(p->pagetable, va, 0) để lấy con trỏ tới PTE tương ứng.

+ Kiểm tra PTE: Nếu PTE hợp lệ (PTE_V) và bit "Accessed" (PTE_A) đang bật:

- Bật bit thứ i trong biến kmask: kmask |= (1L << i).
- Xóa bit PTE_A trong PTE đó (*pte &= ~PTE_A) để reset trạng thái. Việc này đảm bảo lần kiểm tra sau sẽ chỉ báo cáo các truy cập mới phát sinh

Bước 6: Trả kết quả về User

- Sau khi duyệt xong, sử dụng hàm copyout(p->pagetable, mask_addr, ...) để sao chép giá trị kmask từ kernel space sang địa chỉ bộ nhớ của user.

- Hàm trả về 0 nếu thành công, ngược lại trả về -1.

C. TÀI LIỆU THAM KHẢO

Các file hướng dẫn được upload trên website môn học (Moodle).