Trường Đại học Khoa học tự nhiên, ĐHQG-HCM

Khoa Công nghệ thông tin

HỆ ĐIỀU HÀNH – CQ2023_22

BÁO CÁO ĐỒ ÁN 2

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 11 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 1	100%
23120197	Trà Văn Sỹ	Trưởng nhóm	Bài 1 Tổng hợp báo cáo	100%
23120244	Nguyễn Nhật Duy	Thành viên	Bài 1	100%
23120283	Phạm Quốc Khánh	Thành viên	Bài 2	100%
23120314	Phạm An Ninh	Thành viên	Bài 2	100%

B. BÁO CÁO ĐỒ ÁN

Bài 1: Trace

1. Yêu cầu bài toán

Thêm một tính năng vào kernel xv6 cho phép theo dõi các system call. Chương trình yêu cầu tạo một system call mới là trace, nhận vào một đối số nguyên "mask" có các bit chỉ định các system call nào cần theo dõi. Khi một system call được gọi, kernel sẽ kiểm tra xem bit tương ứng với system call đó có được bật trong mask của tiến trình hay không. Nếu có, kernel sẽ in ra một dòng thông tin bao gồm: ID của tiến trình (PID), tên system call (fork, read…) và giá trị trả về của system call đó. Tính năng theo dõi này phải được bật cho tiến trình gọi trace và cũng tự động kế thừa cho tất cả các tiến trình con mà nó tạo ra sau đó.

2. Cách giải quyết

Để giải quyết, ta cần can thiệp ở cả user-space (cung cấp chương trình trace để người dùng gọi) và kernel-space (thêm system call trace mới và logic theo dõi.

Định nghĩa giao diện system call

- Đầu tiên, ta định nghĩa một số hiệu syscall mới SYS_trace cho system call trace trong kernel/syscall.h. Do các system call đã có trong hệ điều hành có số hiệu từ 1 đến 21, ta sẽ gán cho SYS_trace số hiệu là 22.

- Tiếp theo, trong user/user.h, ta sẽ thêm một prototype *int trace(int)* giúp compiler biết về system call trace để các chương trình người dùng có thể sử dụng nó nhằm giao tiếp với kernel.

- Trong user/usys.pl, ta thêm vào đó *entry("trace")*. Lệnh này sẽ tự động tạo mã assembly trong usys.S nhằm gọi system call tương ứng bằng cách nạp số hiệu syscall vào thanh ghi a7 rồi gọi lệnh ecall để chuyển quyền điều khiển từ user mode sang kernel mode.

Triển khai logic kernel

- Trong kernel/proc.h, ta sẽ sửa đổi struct proc bằng cách thêm một trường *int trace_mask* vào đó, trường này sẽ lưu trữ mask theo dõi cho mỗi tiến trình.

- Với mỗi tiến trình, ta cần đảm bảo trace_mask trong tiến trình đó sẽ được xử lí chính xác khi tiến trình được tạo, fork hoặc giải phóng.

- Trong kernel/proc.c, tại hàm allocproc(), khi một tiến trình mới được cấp phát, ta phải khởi tạo giá trị trace_mask của nó. Ta gán p->trace_mask = 0 để đảm bảo tiến trình mới không theo dõi bất cứ system call nào trừ khi được chỉ định.
- Ở hàm fork() trong proc.c, để các tiến trình con có thể kế thừa được mask của cha, sau khi tiến trình con (np) được tạo, ra sẽ sao chép mask từ cha (p) sang con: np->trace_mask = p->trace_mask.
- Tại hàm freproc trong proc.c ta cũng nên dọn dẹp trace_mask bằng cách gán về 0 để giải phóng tiến trình.

- Tiếp theo, trong kernel/sysproc.c, ta triển khai hàm kernel thực sự sẽ được gọi khi user gọi system call trace() bằng hàm sys_trace(void). Tại đây, ta sẽ dùng argịn(0, &mask) để lấy đối số mask từ thanh ghi a0 mà user đã truyền vào khi gọi tới system call ở hàm trace() trong trace.c đã mô tả ban đầu. Sau đó ta gán mask này cho tiến trình hiện tại: myproc()->trace_mask = mask. Cuối cùng hàm trả về 0 để báo hiệu thành công.

- Trong kernel/syscall.c, ta thêm prototype của system call trace() (extern uint64 sys_trace(void)) ở đầu file. Tiếp theo, thêm [SYS_trace] sys_trace() vào mảng syscalls[] để ánh xạ số hiệu system call SYS_trace với hàm sys_trace() mà ta vừa viết. Ta cũng thêm một mảng syscall_names[] giúp ánh xạ số hiệu các system call trong hệ điều hành với tên gọi của system call tương ứng giúp viện in ra thông tin system call tiện hơn.

- Tiếp đến, ta sẽ thực hiện sửa đổi hàm xử lí system call trung tâm syscall(void) để in ra thông tin syscall được gọi trong tiến trình theo yêu cầu của đề bài.

- Đầu tiên, ta lấy số hiệu của lệnh gọi system call chứa trong thanh ghi a7 của trapframe lưu vào biến num. Nếu biến num là hợp lệ, system call tương ứng sẽ được gọi và kết quả trả về của nó sẽ được lưu vào thanh ghi a0 của trapframe.
- Sau đó, ta sẽ thêm logic kiểm tra xem system call được gọi có phải là system call cần được theo dõi không. Ta dùng một biểu thức bitwise: if ((p->trace_mask & (1 << num)) != 0), với num là số hiệu của system call vừa chạy. 1 << num tạo ra một bitmask chỉ có bit thứ num được bật, và phép toán & (AND) kiểm tra xem bit đó có được bật trong p->trace_mask hay không. Nếu đúng, tức system call này

cần được theo dõi, ta sẽ in ra thông tin theo định dạng yêu cầu: p->pid, syscall_names[num] và giá trị trả về p->trapframe->a0.

Triển khai chương trình user-space

Sau khi kernel đã có khả năng theo dõi system call, ta tạo file trace.c trong thư mục user để người dùng có thể kích hoạt tính năng này. Hàm này sẽ nhận vào các đối số dưới dạng *"trace mask <lệnh thực hiện>"* , trong đó mask chính là một số nguyên mà mỗi bit của nó chỉ định các system call cần theo dõi như đã mô tả.

- Do đó, cần kiểm tra số lượng đối số mà người dùng nhập vào tương ứng và tính hợp lệ của mask, nếu không thỏa thì báo lỗi và kết thúc chương trình. Ngược lại, chương trình sẽ chuyển mask từ chuỗi sang số nguyên sau đó gọi system call trace() mới mà ta sẽ tạo ở phía sau. Lệnh gọi system call này sẽ đăng kí với kernel rằng tiến trình này và các tiến trình con của nó sẽ theo dõi các system call theo mask đã cung cấp. Quá trình đăng kí trên sẽ được nhắc tới ở phần sau.

- Sau khi gọi trace(), chương trình sẽ tách lấy phần lệnh thực hiện từ các đối số đã truyền vào, và dùng exec() để thực thi lệnh được chỉ định đó. exec() sẽ thay thế code và data của tiến trình trace hiện tại nhưng struct proc của tiến trình sẽ được giữ nguyên, bao gồm cả trace_mask đã được thiết lập.

Bài 2: Sysinfo

1. Yêu cầu bài toán

Thêm một system call mới là sysinfo, để thu thập thông tin về thông tin system đang chạy. Lệnh gọi hệ thống này lấy một đối số: là con trỏ struct sysinfo. Kernel sẽ điền vào các trường của struct này: trường freemem sẽ được đặt thành số byte bộ nhớ trống và trường nproc sẽ được đặt thành số tiến trình có trạng thái không phải là UNUSED.

2. Cách giải quyết

- Ta tạo file kernel/sysinfo.h để định nghĩa struct sysinfo, chứa các thông tin mà system call sẽ trả về.

- Trong user/user.h ta khai báo struct sysinfo và int sysinfo (struct sysinfo*) với mục đích cung cấp các hàm system call mà user space có thể sử dụng

- Tiếp đến ta thêm ID cho syscall để kernel có thể phân biệt được các syscall với nhau. Ở đây ta sẽ gán SYS_sysinfo với mã 23 trong file kernel/syscall.h. Lưu ý không được trùng với những system call trước đó.

- Trong user/usys.pl, ta thêm vào entry("sysinfo") với mục đích tự động sinh assembly code chứa instruction ecall để trap từ user mode sang kernel mode.

- Trong kernel/syscall.c ta cần đăng ký system call bằng cách thêm khai báo extern uint64 sys_sysinfo(void); ở phần đầu file để báo cho compiler biết hàm sys_sysinfo() được định nghĩa ở file khác (sysproc.c), sau đó thêm [SYS_sysinfo] sys_sysinfo vào trong mảng static uint64 (*syscalls[])(void) để ánh xạ system call number 23 với handler function sys_sysinfo, khi đó hàm syscall() sẽ đọc số system call từ thanh ghi a7 và tra trong mảng này để gọi đúng handler tương ứng.

- Trong file kernel/sysproc.c, ta thêm mới hàm uint64 sys_sysinfo(void) để lấy thông tin system. Cụ thể, hàm này sẽ gọi đến hai hàm là getfreemem() để lấy tổng dung lượng bộ nhớ còn trống (byte) và getnproc() để lấy số lượng process đang được dùng (state != UNUSED) (hai hàm này sẽ được mô tả ở phía sau), sau đó hàm này sẽ trả về 2 thông số lấy được từ hai hàm getfreemem() và getnfroc() để trả về cho tiến trình gọi nó thông qua hàm copyout và địa chỉ được truyền vào từ user space. Chi tiết như sau:

+ Đầu tiên, chúng ta sẽ lấy địa chỉ space để trả dữ liệu: argaddr(0, &addr) lấy tham số đầu tiên (argument 0) mà tiến trình truyền vào khi gọi system call. Tham số này là địa chỉ trong user space nơi kernel sẽ ghi thông tin hệ thống.

+ Sau đó, tạo một struct sysinfo để lưu dữ liệu. Gán freemem bằng tổng bộ nhớ còn trống. Gán nproc bằng số lượng process đang chạy bằng hai hàm getfreemem() và getnfroc().

+ Cuối cùng, ta sẽ lấy con trỏ tới tiến trình hiện tại bằng hàm myproc() để truy cập bảng trang và truyền dữ liệu tới user space thông qua hàm copyout với các tham số p->pagetable: bảng trang của tiến trình hiện tại, addr: địa chỉ đích trong user space, (char *)&info: con trỏ tới dữ liệu cần sao chép, sizeof(info): kích thước dữ liệu. Nếu copy lỗi sẽ trả về -1 và ngược lại sẽ trả về 0.

Lí do tại sao ta phải dùng copyout chính là kernel không bao giờ được phép ghi trực tiếp vào con trỏ addr mà user truyền vào (ví dụ như *addr = info). Vì addr là một địa chỉ ảo do user cung cấp, nó có thể là một địa chỉ trỏ vào vùng nhớ của chính kernel hoặc là một địa chỉ không hợp lệ, gây ra lỗ hổng bảo mật. Giải pháp là dùng hàm copyout(), giúp vận chuyển dữ liệu một cách an toàn. Nó sử dụng p->page table của tiến trình để xác thực addr là một địa chỉ hợp lệ và an toàn bên trong user space trước khi sao chép. Nếu địa chỉ không an toàn, copyout sẽ trả về lỗi thay vì làm sập kernel.

- Trong kernel/defs.h, thêm 2 khai báo hàm uint64 getfreemem(void) và uint64 getnproc(void), hàm getfreemem có nhiệm vụ đếm số trang bộ nhớ còn trống và trả về tổng dung lượng bộ nhớ trống trong hệ thống; hàm getnproc có nhiệm vụ đếm số lượng process hiện đang được sử dụng trong hệ thống. Việc thêm khai báo cho hai hàm này trong file kernel/defs.h là cần thiết để đảm bảo rằng chúng có thể được gọi từ những nơi khác trong kernel.

- Bây giờ ta sẽ bắt đầu định nghĩa hàm getfreemem trong file kernel/kalloc.c. Cụ thể hàm này sẽ khóa kmem.lock trước khi bắt đầu duyệt (để tránh CPU khác có thể gọi kalloc() hoặc kfree() khi ta đang duyệt làm thay đổi freelist dẫn đến sai kết quả), sau đó duyệt qua freelist để đếm tất cả các page còn trống, cuối cùng mở khóa kmem.lock và

trả về (số trang trống * pagesize) chính là số byte còn trống để phục vụ cho system call sysinfo.

- Tiếp đến, trong file kernel/proc.c, ta định nghĩa hàm getnproc(). Hàm này sẽ duyệt qua table proc[], với mỗi tiến trình p trong mảng, ta cần lock tiến trình lại để tránh xung đột dữ liệu (do một CPU khác có thể thay đổi state của p làm trạng thái không nhất quán), thực hiện đếm các tiến trình có trạng thái khác UNUSED (nghĩa là đang được sử dụng) và mở khóa cho tiến trình sau khi đếm xong.

- Cuối cùng ta tạo file user/sysinfotest.c để kiểm tra system call hoạt động đúng bằng ba test case: testcall() kiểm tra system call có gọi được không, testmem() kiểm tra freemem có giảm khi cấp phát bộ nhớ bằng sbrk(4096) không, và testproc() kiểm tra nproc có tăng khi fork() tạo tiến trình con không, sau đó thêm $U/_sysinfotest\ vào phần UPROGS trong Makefile để biên dịch chương trình test cùng với hệ thống.

C. TÀI LIỆU THAM KHẢO

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