BÁO CÁO PROJECT 1 – SYSTEM CALL

Môn học: Hệ điều hành (OPERATING SYSTEM)

Nhóm thực hiện bao gồm:

- 23120095 - Lưu Đức Toàn
- 23120138 – Nguyễn Quốc Kỳ
- 23120183 – Nguyễn Hoàng Anh Tú

I. Trace

1. Giải pháp thực hiện

Nhóm đã triển khai system call trace theo các bước sau:

A. **Định nghĩa Kernel:**

1. Thêm `SYS_trace` vào `kernel/syscall.h`.
2. Thêm một trường `int mask;` vào `struct proc` trong `kernel/proc.h` để lưu mặt nạ (`mask`) theo dõi của từng tiến trình.
3. Triển khai `sys_trace()` trong `kernel/sysproc.c`. Hàm này sử dụng `argint(0, &mask)` để lấy đối số mask từ người dùng và gán nó vào trường `mask` của tiến trình hiện tại (`myproc()->mask`).

B. **Kế thừa mask:**

1. Sửa đổi hàm `fork()` trong `kernel/proc.c` để sao chép giá trị `mask` từ tiến trình cha sang tiến trình con khi một tiến trình mới được tạo. Điều này đảm bảo rằng các tiến trình con cũng được theo dõi.

C. **In thông tin theo dõi:**

1. Sửa đổi hàm `syscall()` trong `kernel/syscall.c`.
2. Nhóm đã thêm một mảng `syscallnames[]` để lưu tên của các system call tương ứng với số hiệu của chúng.
3. Trước khi hàm `syscall()` trả về, nhóm thêm một đoạn mã kiểm tra: nếu bit tương ứng với `num` (số hiệu system call) được bật trong `myproc()->tracemask`, hệ thống sẽ in ra một dòng thông tin bao gồm PID, tên system call, và giá trị trả về (lấy từ `myproc()->trapframe->a0`).

D. **Chương trình User-space:**

1. Tạo file `user/trace.c`.
2. Chương trình này đọc mask từ `argv[1]` , gọi system call `trace(mask)` , sau đó dùng `exec` để thực thi chương trình ở `argv[2]` với các đối số từ `argv[3]` trở đi.
3. Thêm `\$U/_trace` vào `UPROGS` trong `Makefile`.

2. Vấn đề gặp phải

Ban đầu, nhóm đã cố thử cài đặt biến `mask` 64-bit để `trace` có thể hỗ trợ nhiều system call hơn trong tương lai. Tuy nhiên, tham số của system call chỉ hỗ trợ nhận vào giá trị 32-bit `integer`. Do vậy, `trace` chỉ có thể tuân theo quy định của `kernel`

II. Sysinfo

1. Giải pháp thực hiện

Nhóm đã triển khai system call sysinfo như sau:

A. **Định nghĩa cấu trúc:**

1. Tạo file `kernel/sysinfo.h` và định nghĩa `struct sysinfo` với ba trường: `uint64 freemem`, `uint64 nproc`, và `uint64 nopenfiles`.

B. **Triển khai Kernel:**

1. Triển khai `sys_sysinfo()` trong `kernel/sysproc.c`.
2. Hàm này đầu tiên dùng `argaddr(0, &address)` để lấy con trỏ `struct sysinfo *` từ `user space`.
3. Tạo một biến `struct sysinfo _sysinfo` trong `kernel`.
4. Gọi các hàm trợ giúp để thu thập thông tin:
   - `freemem`: Thêm hàm `get_freemem()` vào `kernel/kalloc.c` để duyệt qua `freelist` và đếm tổng số byte bộ nhớ trống.
   - `nproc`: Thêm hàm `get_nproc()` vào `kernel/proc.c` để duyệt qua mảng `proc[NPROC]` và đếm số lượng tiến trình có trạng thái khác `UNUSED`.
   - `nopenfiles`: Thêm hàm `get_nopenfiles()` vào `kernel/file.c`. Giải pháp của nhóm là duyệt qua `ftable.file` (trong

kernel/file.c) và đếm số lượng file có ref (số tham chiếu) lớn hơn 0, vì đây là bảng file của toàn hệ thống.

5. Cuối cùng, dùng copyout() để sao chép struct _sysinfo từ kernel space về con trỏ address ở user space.

C. **Chương trình User-space:**

1. Tạo file user/sysinfotest.c để gọi sysinfo() và in ra các giá trị thu thập được.
2. Thêm \$U/_sysinfotest vào UPROGS trong Makefile.

2. Vấn đề gặp phải (hoặc làm rõ)

Ban đầu nhóm không rõ nên đếm nopenfiles bằng cách duyệt qua proc->ofile của từng tiến trình hay duyệt qua ftable chung. Sau khi cài đặt và thử nghiệm cả hai cách trên, nhóm đã quyết định chọn duyệt qua ftable để tránh đếm trùng lặp các file được mở bởi nhiều tiến trình.

III. Kết luận

Nhóm đã hoàn thành toàn bộ các yêu cầu của Project 1. Các chương trình trace và sysinfotest hoạt động đúng như các ví dụ và mô tả trong đề bài.