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

BÁO CÁO ĐỒ ÁN SỐ 1:

System calls

Môn học : Hệ điều hành
Giảng viên lý thuyết : Trần Trung Dũng
Giảng viên hướng dẫn thực hành : Lê Giang Thanh
Nguyễn Thanh Quân

TP Hồ Chí Minh 11/2023

MỤC LỤC

Chapter 1

Bảng đánh giá thành viên

MSSV	Họ và tên	% đóng góp
21120075	Trần Minh Hoàng	100%
21120471	Phan Gia Huy	100%

Chapter 2

Tổng quan đồ án

Đồ án thứ hai của môn học hệ điều hành phần thực hành là làm quen công cụ gỡ lỗi gdb, tìm hiểu và thêm mới các system call mới cho hệ điều hành xv6 nhằm mục đích mở rộng chức năng cho xv6 cũng như tăng cường hiểu biết về cơ chế hoạt động của system call. Đồ án giúp sinh viên rút ra được những kiến thức và kỹ năng mới về hệ điều hành thông qua việc thực hành thiết kế và thêm system call cho xv6.

Trong bài báo cáo này, chúng em sẽ trình bày quá trình thực hiện và triển khai các system call mới vào kernel của xv6, đó chính là `system call tracing` và `sysinfo`. Chúng tôi sẽ trình bày chi tiết về mỗi system call, giải thích lý do tại sao chúng ta cần thêm chúng và cách chúng em đã thiết kế và triển khai chúng trong hệ thống xv6.

Chapter 3

Làm quen với công cụ GDB

Công cụ gdb (GNU Debugger) là một công cụ rất hữu ích để gỡ lỗi các chương trình được biên dịch từ ngôn ngữ C. Nó cho phép thiết lập các breakpoint, kiểm tra giá trị biến, truy vết quá trình thực thi thông qua các lệnh call stack.

Đối với hệ điều hành xv6, gdb có thể được sử dụng để gỡ lỗi các thay đổi mã nguồn hoặc module mới được thêm vào. Để sử dụng gdb với xv6 :

1. Trong cửa sổ terminal đầu tiên, chạy lệnh: `make qemu-gdb`. Điều này sẽ biên dịch xv6 với tùy chọn debug, sau đó khởi chạy qemu và kết nối với gdb.
2. Trong cửa sổ terminal thứ hai, chạy lệnh: `gdb-multiarch`.
3. Trong gdb, gõ lệnh: `target remote :26000`
4. Sau đó có thể sử dụng các lệnh gdb để set breakpoint, print biến để debug đoạn mã.

Chapter 4

Cài đặt các system calls

4.1 System call tracing

- **Mục đích**:
  Trace syscall cho phép theo dõi các system call được thực hiện bởi các tiến trình trong hệ thống. Việc theo dõi các system call có thể giúp chúng ta xác định chính xác nơi lỗi xảy ra và giúp trong việc gỡ lỗi.

- **Quá trình cài đặt**:

  – **Thêm mask**: Trong cấu trúc `struct proc` trong file *proc.h*, thêm một trường `mask` để lưu trữ mặt nạ (mask) của syscall được theo dõi cho mỗi tiến trình.

```
struct proc {
    // ... Cac truong khac
    int mask;  // Process mask
};
```

  – **Define SYS_trace**: Định nghĩa hằng số `SYS_trace` trong file *syscall.h* để đại diện cho số syscall của hàm `sys_trace` khi gọi hàm syscall.

```
// File syscall.h
#define SYS_trace  22
```

```
// File syscall.c

// Prototypes for the functions that handle system calls.
extern uint64 sys_trace(void);

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
[SYS_trace]   sys_trace
```

– **Thêm hàm sys_trace trong sysproc.c**: Hàm `sys_trace` được gọi khi syscall `SYS_trace` được gọi từ một tiến trình. Nó nhận một tham số `val` từ người dùng và lưu giữ giá trị này vào trường `mask` của tiến trình hiện tại.

```
uint64
sys_trace(void)
{
    int val;
    argint(0,&val);
    if(val<0) return -1;
    myproc()->mask=val;
    return 0;
}
```

– **Xử Lý syscall**: Trong hàm `syscall`, hệ thống kiểm tra xem syscall có nằm trong phạm vi của hệ thống không và có được định nghĩa không. Nếu có, hàm `syscall` được gọi và sau đó kiểm tra xem syscall đó có được theo dõi hay không bằng cách sử dụng giá trị mask của tiến trình. Nếu syscall đang được theo dõi, thông tin về syscall và giá trị trả về của nó được in ra màn hình.

```
 static char *syscall_name[] = {"","fork","exit","wait","pipe","read",
"kill","exec","fstat","chdir","dup","getpid","sbrk","sleep","uptime","open",
"write","mknod","unlink","link","mkdir","close","trace","sysinfo"};
 void
 syscall(void)
 {
     int num;
     struct proc *p = myproc();

     num = p->trapframe->a7;
     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
       // Use num to lookup the system call function for num, call it,
       // and store its return value in p->trapframe->a0
       p->trapframe->a0 = syscalls[num]();
       if ((p->mask >> num)%2==1) {
      printf("%d: syscall %s -> %d\n",
            p->pid, syscall_name[num], p->trapframe->a0);
       }
     } else {
       printf("%d %s: unknown sys call %d\n",
              p->pid, p->name, num);
```

```
                p->trapframe->a0 = -1;
        }
    }
```

– **Ví dụ với syscall fork**: Khi một tiến trình mới được tạo ra thông qua syscall fork, giá trị `mask` của tiến trình cha được sao chép sang tiến trình con để tiếp tục theo dõi các syscall.

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  //... Cac cai dat truoc

  //trace
  np->mask = p->mask;

  release(&np->lock);

  return pid;
}
```

4.2 Sysinfo

- **Mục đích:**
  Lệnh này sẽ thu thập thông tin về hệ thống đang chạy và điền vào các trường cần thiết của struct sysinfo. Mục tiêu là đặt giá trị cho trường freemem là số byte bộ nhớ trống và trường nproc là số lượng tiến trình không sử dụng. Chúng ta sẽ kiểm tra đúng sai bằng chương trình thử nghiệm sysinfotest.

- **Quá trình cài đặt:**

  – **Thêm struct sysinfo:** Tạo mới `struct sysinfo` trong file mới tạo *sysinfo.h*,thêm hai trường `uint64 freemem`,`uint64 nproc` để lưu bộ nhớ trống và số lượng quá trình.

```
            sturct sysinfo{
```

```
    uint64 freemem; // amount of free memory (bytes)
    uint64 nproc; // number of process
};
```

– **Thêm hàm getfreemem trong kalloc.c:**Hàm `getfreemem` trong đoạn mã trình bày là một phần của hệ thống quản lý bộ nhớ hoặc kernel. Chức năng chính của nó là tính toán và trả về tổng lượng bộ nhớ trống trong hệ thống.Để đảm bảo tính nhất quán và an toàn trong quá trình truy cập tài nguyên chia sẻ, hàm sử dụng một cơ chế khóa (’`acquire`’ và ’`release`’). Bằng cách duyệt qua danh sách liên kết các khối bộ nhớ không sử dụng và tính tổng kích thước của chúng, hàm đóng góp vào việc theo dõi và quản lý tình trạng bộ nhớ trống trong hệ thống.

```
uint64 getfreemem() {
      struct run *r;
      uint64 freemem = 0;
      // Acquire the lock to access the shared resource
      acquire(&kmem.lock);
      // Assign the pointer 'r' to the head of the
      //linked list kmem.freelist
      r = kmem.freelist;
      while (r) {
        // Increase the value of freemem by PGSIZE
        //for each node in the list
        freemem += PGSIZE;

        // Update the pointer 'r'
        //to the next node in the list
        r = r->next;
      }
      // Release the lock to access the shared resource
      release(&kmem.lock);
      // Return the total amount of free memory
      return freemem;
    }
```

– **Thêm hàm get_nproc trong proc.c:** Hàm `get_nproc` được thiết kế để đếm số lượng tiến trình đang hoạt động trong hệ thống. Bằng cách duyệt qua mảng các tiến trình và kiểm tra trạng thái của mỗi tiến trình, nó tăng biến đếm cho mỗi tiến trình không ở trạng thái chưa sử dụng. Việc sử dụng các hàm ’`acquire`’ và ’`release`’ giúp đảm bảo

tính toàn vẹn dữ liệu khi truy cập thông tin của mỗi tiến trình. Kết quả trả về là tổng số lượng tiến trình đang hoạt động trong hệ thống.

```
int get_nproc() {
  struct proc *p;
  int count = 0;
  // Loop through the array of processes
  //from proc[0] to proc[NPROC-1]
  for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    // Acquire lock to prevent access to the process
    // Check the state of the process
    if (p->state != UNUSED) ++count;
    // If the process is not in the UNUSED state,
    //increment the count
    release(&p->lock);
    // Release the lock to allow access to the process
  }
  // Return the total number of active processes
  return count;
}
```

– **Define SYS_info**: Định nghĩa hằng số `SYS_info` trong file *syscall.h* để đại diện cho số syscall của hàm `sys_info` khi gọi hàm syscall.

```
// File syscall.h
#define SYS_info  23
```

```
// File syscall.c

// Prototypes for the functions that handle system calls.
extern uint64 sys_info(void);

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
[SYS_infoz]   sys_info
```

– **Thêm hàm systeminfo trong sysinfo.c:** Hàm `systeminfo` là một phần quan trọng của hệ thống và cho phép các tiến trình ở user space truy cập thông tin hệ thống quan

trọng, như lượng bộ nhớ trống và số lượng tiến trình đang hoạt động.Thông qua hàm này, các ứng dụng hoặc tiến trình có thể theo dõi và tương tác với trạng thái hệ thống.

```
int systeminfo(uint64 addr) {
  // Get a pointer to the current process
  struct proc *p = myproc();
  // Create a struct to hold system information
  struct sysinfo info;
  // Retrieve the amount of free memory in the system
  info.freemem = getfreemem();
  // Retrieve the number of active processes in the system
  info.nproc = get_nproc();
  // Copy the system information to the user's address space
  if (copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
    return -1;
  return 0; // Return success
}
```

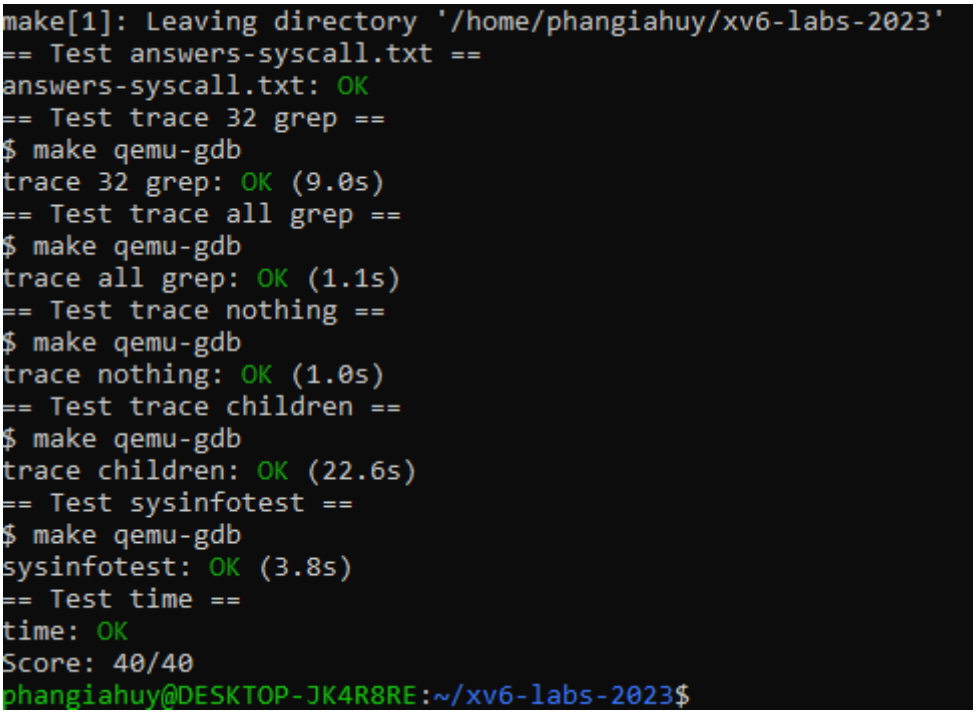
– **Thêm hàm sys_sysinfo trong sysproc.c**: Hàm `sys_sysinfo` được gọi khi syscall `SYS_sysinfo` được gọi từ một tiến trình.Nó nhận một tham số info từ người dùng và trả về tổng tiến trình thực hiện trong hàm `systeminfo` cho phép các tiến trình ở info có sự nhận biết về trạng thái và tài nguyên của hệ thống, giúp họ quản lý và tương tác với hệ thống một cách hiệu quả.

```
uint64 sys_sysinfo(void) {
  uint64 info;  // Initialize a variable to hold the user-provided
  //address for system information
  argaddr(0, &info);  // Get the user-provided
  //address from the first argument and store it in 'info'
  // Check if the user-provided address is invalid (less than 0)
  if (info < 0)
    return -1;
  // Call the 'systeminfo' function to retrieve
  //and copy system information to the user-provided address
  return systeminfo(info);
}
```

Chapter 5

Tổng kết đánh giá

Nhóm đã hoàn thành cài đặt 2 system calls được yêu cầu. Dưới đây là bảng kết quả sau khi dùng lệnh **make grade** :



Tài liệu

[1] Sử dụng gdb: https://pdos.csail.mit.edu/6.1810/2022/labs/guidance.html