ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC - KỸ THUẬT MÁY TÍNH

Course: OPERATING SYSTEM

Assignment #1 - System Call Report

GVHD: Huỳnh Nam

TP. HỒ CHÍ MINH, THÁNG 4/2019

Danh sách thành viên

1. Nguyễn Tiến Phát - 1712572
2. Nguyễn Lục Sâm Bảo - 1710598

Mục lục

1 Adding new system calls

SYSTEM CALL - procsched:

- Tại địa chỉ thư mục **arch/x86/entry/syscalls/**, các system calls được liệt kê trong các file **.tbl**. Để thông báo về system calls mới, mở file **syscall_32.tbl**, thêm dòng sau vào cuối file: [number] i386 procsched sys_procsched.
  Tương tự với file **syscall_64.tbl**: [number] x32 procsched sys_procsched.

  QUESTION: Ý nghĩa của **i386**, **procsched** và **sys_procsched** ?
  ANSWER:

  - **i386** là ABI, hay còn gọi là giao diện nhị phân ứng dụng, là hai giao diện giữa hai module chương trình nhị phân.Reference
  - **procsched** đơn giản là tên của system call, cụ thể ở đây là procshed.
  - **sys_procsched** là entry point. Entry point là tên của một hàm gọi đến để xử lý một syscall.Quy ước đặt tên của entry point là tên syscall với tiền tố là sys_.

- Mở file the đường dẫn include/linux/syscalls.h và thêm dòng sau:
  struct proc_segs;
  asmlinkage long sys_procsched(int pid, struct proc_segs* info);

  QUESTION: Ý nghĩa của 2 dòng trên ?
  ANSWER:

  - Hai dòng trên thông báo cho hệ thống xác nhận định nghĩa của struct **proc_segs** và hàm **sys_procsched()**.
  - **asmlinkage** là một #define cho trình dịch gcc với ý thông báo rằng tham số sẽ được lấy trực tiếp trên stack chứ không phải từ thanh ghi.

- Tại địa chỉ arch/x86/kernel, tạo 1 file source **sys_procsched.c** và hiện thực code theo hướng dẫn, hoàn chỉnh source code tại //TODO. Sau khi hiện thực xong ,dùng **Vim editor** để chỉnh sửa Makefile, bằng cách thêm dòng : **obj-y += sys_procsched.o** ở cuối file Makefile để build **sys_procsched**.

2 System calls Implementation

Các bước thực hiện:

- Để tiến hành, chúng ta cần chuẩn bị một máy ảo để cài đặt **Ubuntu OS**. Ở đây chúng tôi dùng máy ảo **Oracle VM VirtualBox** để cài đặt **Ubuntu version 12.04**
- Tiếp theo tiến hành cài đặt package bằng một số câu lệnh:

```
$ sudo apt-get update
$ sudo apt-get install build-essential
$ sudo apt-get install kernel-package
```

  QUESTION: Tại sao chúng ta cần cài đặt kernel-package?
  ANSWER:
  - **kernel-package** là một package được phát triển bên ngoài mong muốn dúng để tự động hóa một số bước thường xuyên trong việc yêu cầu biên dịch và cài đặt một customer kernel.
  - Nếu không có thì chúng ta không thể đụng vào cái file hệ thống khi cài đặt kernel.
- Tiếp theo chúng ta tạo thư mục **kernelbuild** ở thư mục **Home**. Sau đó thực hiện download kernel source và giải nén trong thư mục **kernelbuild**. Chúng ta chọn bản **4.4.56**, thực hiện qua 2 command line sau:

```
$ mkdir ~/kernelbuild
$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.4.56.tar.xz
$ tar -xvJf linux-4.4.56.tar.xz
```

  QUESTION: Tại sao chúng ta phải sử dụng một source kernel khác mà không sử dụng trực tiếp kernel gốc(kernel đang chạy trên OS)?
  ANSWER:
  - Để giảm bớt độ rủi ro khi build trực tiếp trên OS. Vẫn có thể build trực tiếp được nhưng tỉ lệ xảy ra lỗi rất cao, khó fix bug.
- Sau khi thực hiện cấu hình thành công customer kernel, chúng ta tiến hành hiện thực system call:
  - Đặc tả struct proc_segs với các biến bao gồm:

```
struct proc_segs {
unsigned long mssv;
unsigned long pcount;
unsigned long long run_delay;
unsigned long long last_arrival;
unsigned long longlast_queued;
};
```

  - Tiếp đến là hiện thực: asmlinkage long sys_procsched(int pid, struct proc_segs * info);. Hàm sys_procsched nhận vào một *pid*. Ta dùng marco for_each_process(task) để duyệt tất cả các process hiện đang chạy. Sau đó dùng if(int(task->pid) == pid) để kiểm tra xem *pid* của process có bằng với *pid* được truyền vào. Nếu bằng nhau thì thực hiện các lệnh gán thông tin sched từ task đến info:

```c
asmlinkage long sys_procsched(int pid,struct proc_segs * info){
    struct task_struct * task;
    for_each_process(task){
        if((int)task->pid == pid){
            info->mssv = 1712572;
            info->pcount = task->sched_info.pcount;
            info->run_delay = task->sched_info.run_delay;
            info->last_arrival = task->sched_info.last_arrival;
            info->last_queued = task->sched_info.last_queued;
        return 0;
        }
    }
    return 1;
}
```

3 Compilation and Installation process

1. **Build the configured kernel**

   - Đầu tiên, để build kernel, t chạy lệnh "Make" để tạo file *vmlinuz*. Vì đây là giai đoạn tốn thời gian, ta có thể chạy song song với tag "-j np" với np là số process ta muốn dùng thực thi công việc.
   - Sau khi biên dịch, *vmlinuz* là một kernel có khả năng khởi động thực tế, "z" để chỉ ra hạt nhân đã được nén.
   - Tiếp đến, tạo modules cho kernel mới, ta dùng lệnh "Make modules", ta cũng có thể sử tag "-j np" với np là số process ta muốn dùng thực thi công việc để tiết kiệm thời gian.

   QUESTION: Ý nghĩa của lệnh **Make** và **Make modules** ?
   Make được dùng để build kernel mới, trong khi Make modules để build các thư viện, mô đun khác.

2. **Installing the new kernel**

   - Với quyền sudo, ta chạy lệnh "Make modules_install" để copy các file kernel modules vào địa chỉ/lib/modules . Lệnh "Make install" để cài đặt kernel mới tạo vào hệ thống.
   - Sau khi đã install, ta thực hiện khởi động lại máy, lúc này, nếu không vì các lý do đặc biệt, hệ thống của chúng ta sẽ đang sử dụng kernel vừa được install. Vì chúng ta đã thay đổi số hiệu kernel trong khi thiết lập nên câu lệnh uname -r sẽ cho chúng ta số hiệu kernel giống vậy. Ở trường hợp này, ta sẽ thấy kết quả là 4.4.56.*MSSV*.

3. **Testing**
   Sau khi thực hiện xong các hàm, ta viết một test nhỏ sử dụng hàm syscall và kiểm chứng lại kết quả. Nếu kết quả là MSSV của mình chứng tỏ đã thành công.

```
#include <sys/syscall.h>
#include <stdio.h>
#define SIZE 10
int main() {
long sysvalue;
unsigned long info[SIZE];
sysvalue = syscall([number_32], 1, info);
printf("My MSSV: %ul", info[0]);
}
```

   QUESTION: Tại sao chương trình có thể kiểm tra được hệ thống của ta có hoạt động được hay không ?
   Vì ta dùng syscall() thực hiện gọi gián tiếp vào hệ thống và lấy một syscall cụ thể từ tham số nhập vào, tham số ở đây là số thứ tự của syscall mà ta vừa thêm vào. Nếu procsched đã thực sự tồn tại trong hệ thống thì nó sẽ được gọi một cách thành công.

4 Making API for system calls

- Tuy ta đã hiện thực system call sys_procsched thành công, nhưng việc kích hoạt vẫn khá bất tiện do phải gọi qua số thứ tự. Cho nên để đơn giản hơn, ta sẽ tạo một thư viện mới để liên kết động khi có ứng dụng nào đó muốn sử dụng procsched.
- Tạo file prosched.h ở địa chỉ khác với kernel, và khai báo struct proc_segs một lần nữa, giống với trong file mà được hiện thực trong kernel.

  QUESTION: Tại sao ta lại phải định nghĩa struct trong khi ta đã định nghĩa trước đó ?
  Ta cần định nghĩa lại struct proc_segs bởi vì đang viết wrapper ở bên ngoài thư mục kernel cho nên file procsched.h không có đường dẫn nào đến struct proc_segs trong kernel cho nên cần phải định nghĩa lại struct này để đảm bảo rằng có thể lấy được dữ liệu và gán cho nó.
- Tạo file procsched.c để chứa đoạn code cho wrapper, hoàn thành đoạn code tại # TODO như sau :

```c
#include "procsched.h"
#include<linux/kernel.h>
#include<sys/syscall.h>

long procsched(pid_t pid, struct proc_segs *info){
    long sysvalue;
    unsigned long INFO[5];
    sysvalue = syscall(546, pid, INFO);
    info->mssv = INFO[0];
    info->pcount = INFO[1];
    info->run_delay = INFO[2];
    info->last_arrival = INFO[3];
    info->last_queued = INFO[4];
    return 0;
}
```

- Ta đã có file h, nhưng để mọi người có thể sử dụng được (include được), ta cần copy tới /usr/include, nơi mà biên dịch gcc tìm khi ta dùng lệnh #include<>:

```
$ sudo cp <path to procsched.h> /usr/include
```

  QUESTION: Tại sao ta lại cần root để copy file tới /usr/include?
  Vì đó là file của hệ thống, người dùng thông thường chỉ có quyền xem không thể sửa đổi. Nên cần root để ta có thể copy file vào /usr.
- Và để có thể sử dụng trong chương trình khác, ta cần có một object có thể hện thực được:

```
$ gcc -shared -fpic procsched.c -o libprocsched.so
```

  QUESTION: Tại sao lại có -shared và -fpic trong câu lệnh ?
  -shared để tạo các shared object cho các thư viện liên kết. -fpic để đảm bảo trình biên dịch tạo ra code không phụ thuộc địa chỉ. Cả 2 được dùng để tạo thư viện động.
- Nếu thành công copy file libprocsched.so đến /usr/lib

- Và cuối cùng, để kiểm tra kết quả, tạo một file mới có sử dụng hàm procsched với pid là id của chính chương trình. Compile với -lprocsched.
- So sánh kết quả với /proc/<pid>/maps file. Kết quả giống nhau là thành công.

5 Conclusion

Bảng phân chia công việc

Thành viên	Công việc	Tỉ lệ đóng góp
Nguyễn Tiến Phát	Hiện thực sys_procsched, build kernel, viết báo cáo	50%
Nguyễn Lục Sâm Bảo	Hiện thực wrapper, build kernel, viết báo cáo	50%

Reference

1. *Linux Kernel In A Nutshell* by Greg Kroah-Hartman
2. *Adding a Syscall to Linux 3.14* - Shane Tully
3. *https://kernelnewbies.org/KernelBuild*