Lab 6. xv6 system call

1. Mục đích

Thông qua bài thực hành này, sinh viên sẽ hiểu được cách hệ điều hành xv6 xử lý các system call, và cách thêm một system call mới vào xv6

2. Tóm tắt lý thuyết

Các wrapper của **system call (syscall)** được viết bằng assembly và nằm trong file user/usys.S. File này được tạo ra tự động trong quá trình build make qemu bởi file user/usys.pl. Prototype của các syscall được khai báo trong file user/user.h

Bên dưới là syscall read() lấy từ file user/usys.S:

```
1  .global read
2  read:
3   li a7, SYS_read
4   ecall
5   ret
```

Khi user program gọi đến **read()** với các tham số **fd, buf, và n**, các tham số này lần lượt được gán vào registers a0, a1, a2 tương ứng theo quy ước của kiến trúc RISC-V. Bên trong read(), system call number tương ứng với read (SYS_read được định nghĩa trong file kernel/syscall.h) được gán vào register a7 (dòng 3) trước khi thực thi instruction **ecall** (dòng 4)

Instruction ecall gây ra trap ở user space. CPU RISC-V chuyển sang suppervisor mode và thực thi function **uservec()** (kernel/trampoline.S) nằm ở page TRAMPOLINE. Function này save trạng thái của CPU (context) vào page TRAPFRAME, chuyển sang sử dụng kernel stack và chuyển pagetable sang kernel pagetable, trước khi gọi đến function usertrap() (kernel/trap.c).

Khi **usertrap()** xác định trap được gây ra bởi instruction ecall, nó gọi đến function syscall() (kernel/syscall.c)

Function **syscall()** đọc giá trị register a7 lưu trong TRAPFRAME để xác định syscall nào đang được gọi, và gọi đến sys_read() (kernel/sysfile.c)

Function **sys_read()** đọc các tham số **fd, buf, và n** từ a0, a1, a2 lưu trong TRAPFRAME, sau đó tiến hành đọc tối đa n byte của file tương ứng với fd, và lưu vào buf. Tùy theo loại của file (regular file, dir, pipe hay device), sys_read() sẽ dùng các function tương ứng để đọc file. Do buf sử dụng địa chỉ virtual address của user program nên quá trình ghi dữ liệu vào buf phải sử dụng function **copyout()** (kernel/vm.c). Cuối cùng sys_read() return 0 nếu không có lỗi xảy ra, ngược lại return -1

Function syscall() gán giá trị return của sys_read() vào a0 lưu trong TRAPFRAME và return

Sau khi syscall() return, các function **usertrapret()** (kernel/trap.c) và **userret()** (kernel/trampoline.S) lần lượt được gọi đến để quay trở lại user program: Trạng thái của CPU được phục hồi từ TRAPFRAME,

pagetable chuyển sang user pagetable và instruction **sret** được thực thi để chuyển từ suppervisor mode sang user mode và quay trở lại read()

read() return về user program (dòng 5) với giá trị trả về của syscall đã được lưu trong register a0. User program tiếp tục thực thi như bình thường

3. Thực hành

Chuyển thư mục làm việc vào thư mục source code của xv6 và tạo branch mới lab6 từ branch origin/syscall, sau đó checkout branch lab6

Mở hai Terminal để tiến hành debug xv6 với GDB. Đa số các command GDB đều có thể viết tắt, ví dụ: break(b), continue(c), … Các command GDB sử dụng trong bài thực hành này sẽ dùng dạng viết tắt.

Trên Terminal GDB, đặt break point tại function syscall() b syscall sau đó tiếp tục chương trình c

```
(gdb) b syscall
Breakpoint 1 at 0x80001fe0: file kernel/syscall.c, line 133.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at kernel/syscall.c:133
133     {
(gdb)
```

Một trong các CPU sẽ gặp breakpoint và dừng lại. Hiển thị vị trí hiện tại của CPU trong source code layout src, và hiển thị nội dung callstack backtrace

```
(gdb) backtrace
#0  syscall () at kernel/syscall.c:133
#1  0x0000000080001d14 in usertrap () at kernel/trap.c:67
#2  0x0505050505050505 in ?? ()
```

Function gọi đến syscall() là function usertrap() (dòng 67 trong file kernel/trap.c). Stack không chứa thông tin của function trước đó bởi vì uservec() đã thiết lập register stack pointer ($sp) trỏ đến địa chỉ cuối của kernel stack trước khi gọi đến usertrap(). Do đó usertrap() trở thành function đầu tiên trong callstack

Gõ command next vài lần cho đến khi vượt qua đoạn code `struct proc *p = myproc();`

In ra Terminal thông tin của process hiện tại (struct proc trong file kernel/proc.h) theo kiểu hexa do con trỏ p trỏ tới p /x *p

```
(gdb) p /x *p
$1 = {
    lock = {locked = 0x0, name = 0x80008178, cpu = 0x0},
    state = 0x4, chan = 0x0, killed = 0x0, xstate = 0x0,
    pid = 0x1, parent = 0x0, kstack = 0x3fffffd000, sz = 0x1000,
    pagetable = 0x87f73000, trapframe = 0x87f74000,
    context = {
    ra = 0x80001466, sp = 0x3fffffde40, s0 = 0x3fffffde70, s1 = 0x80008d30, s2 = 0x80008900, s3 = 0x0,
    s4 = 0x5050505050505050, s5 = 0x5050505050505050, s6 = 0x5050505050505050, s7 = 0x5050505050505050,
    s8 = 0x5050505050505050, s9 = 0x5050505050505050, s10 = 0x5050505050505050, s11 = 0x5050505050505050
    },
    ofile = {0x0 <repeats 16 times>},
    cwd = 0x80016e40,
    name = {0x69, 0x6e, 0x69, 0x74, 0x63, 0x6f, 0x64, 0x65, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
}
```

In thông tin process id p p->pid và process name p p->name ra Terminal

```
(gdb) p p->pid
$2 = 1
(gdb) p p->name
$3 = "initcode\000\000\000\000\000\000\000"
(gdb)
```

initcode (user/initcode.S), pid bằng 1, chính là process đầu tiên chạy trong hệ thống. Nó được kernel khởi tạo trong function main() (kernel/main.c) khi gọi đến userinit() (kernel/proc.c). Nhiệm vụ của nó là thực thi system call exec() để chạy program **init**

init (user/init.c) sau đó sẽ tạo ra child process để chạy program shell **sh** (user/sh.c). Theo thiết lập của init, sh sử dụng device file **/console** làm standard I/O và cung cấp giao diện CLI cho người dùng

Command p p->trapframe->a7 in ra Terminal giá trị của register a7 chứa trong TRAPFRAME, đây chính là system call number. Số 7 tương ứng với syscall exec()

```
(gdb) p p->trapframe->a7
$4 = 7
(gdb)
```

Command p /t $sstatus in ra Terminal giá trị của register $sstatus theo dạng binary

```
(gdb) p /t $sstatus
$5 = 100010
(gdb)
```

Bit 8 của $sstatus bằng 0 cho biết mode của CPU trước khi xảy ra trap là user mode, bit 1 bằng 1 cho biết interrupt đang được enable (#define SSTATUS_SPP và #define SSTATUS_SIE trong kernel/riscv.h)

Tiếp tục gõ command next nhiều lần để xem cách syscall exec() thực thi. Gõ command quit để kết thúc phiên debug

Khi lỗi (**exception**) xảy ra trong kernel, xv6 sẽ in ra Terminal thông báo **panic…** kèm với thông tin hỗ trợ debug lỗi như ví dụ bên dưới:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
scause 0x000000000000000d
sepc=0x0000000080002158 stval=0x0000000000000000
panic: kerneltrap
```

scause cho biết loại của trap, **sepc** chứa địa chỉ của register program counter ($PC) khi xảy ra trap và **stval** chứa địa chỉ VA gây ra exception nếu loại của trap là page-fault. Sinh viên tham khảo tài liệu **riscv-privileged** để biết thêm thông tin về các registers này

Break point có thể được đặt tại địa chỉ chứa trong sepc để debug exception như ví dụ bên dưới:

```
(gdb) b *0x0000000080002158
Breakpoint 1 at 0x80002158: file kernel/syscall.c, line 247.
(gdb) layout asm
(gdb) c
```

```
Continuing.
[Switching to Thread 1.3]

Thread 3 hit Breakpoint 1, syscall () at kernel/syscall.c:247
```

BÁO CÁO THỰC HÀNH

Sinh viên: .....

MSSV: ..... Nhóm: .....

Lưu ý:

- Trước khi làm các bài tập, SV xem source code các system call có sẵn trong file kernel/sysproc.c
- Các bước để thêm một system call mới vào xv6:
  - System call number cần được định nghĩa trong file kernel/syscall.h
  - Thêm system call prototype vào file user/user.h và chỉnh sửa file user/user.pl để tạo ra system call wrapper function
  - System call nên được viết thêm vào file kernel/sysproc.c với địa chỉ function được lưu vào mảng **syscalls[ ]** trong file kernel/syscall.c
- Các lưu ý khi viết system call:
  - Sử dụng các function argaddr(), argint(), … để nhận tham số mà user program truyền vào system call. Các function này được định nghĩa trong file kernel/syscall.c
  - Sử dụng function copyout() (kernel/vm.c) nếu cần copy dữ liệu từ kernel address space sang user address space, hoặc copyin() (kernel/vm.c) nếu cần copy dữ liệu theo chiều ngược lại. Tham khảo cách dùng copyout() trong sys_fstat() (kernel/sysfile.c) và filestat() (kernel/file.c)
  - System call sẽ return 0 nếu thành công và return -1 nếu có lỗi xảy ra

Bài 1: Thời gian tồn tại của một process là số lượng clock tick trôi qua tính từ lúc process được tạo ra cho đến hiện tại. Thêm system call **howold()** có chức năng xác định thời gian tồn tại của một process dựa trên pid của nó

```
int howold(int pid, uint64* ticks);
```

howold() ghi thời gian tồn tại của process tương ứng với **pid** vào địa chỉ do **ticks** trỏ tới

Thêm user app **howoldtest** bên dưới để kiểm tra chức năng của howold()

```
 1 #include "kernel/types.h"
 2 #include "kernel/stat.h"
 3 #include "user/user.h"
 4
 5 void testparam(void){
 6
 7   int pid;
 8   uint64 ticks;
 9   uint64 upticks;
10
11   pid = 0;
12   if(howold(pid, &ticks) != -1){
13     fprintf(2, "error: succeeded with invalid pid %d\n", pid);
14     exit(1);
```

```
15    }
16
17    pid = 1;
18    if(howold(pid, (uint64*)(0xeaeb0b5b00002f5e)) != -1) {
19      fprintf(2, "error: succeeded with invalid address\n");
20      exit(1);
21    }
22
23    pid = 1;
24    if(howold(pid, &ticks) != 0){
25      fprintf(2, "error: pid %d should exist\n", pid);
26      exit(1);
27    }
28
29    upticks = uptime();
30    if(ticks > upticks + 5 || ticks < upticks - 5){
31      fprintf(2, "error: init process ticks %d"
32              " different from upticks %d\n",
33              ticks, upticks);
34      exit(1);
35    }
36  }
37
38  void testfunc(void){
39
40    int pid;
41    uint64 ticks;
42    uint8 sleeptime[] = {5, 10, 6, 7, 20, 13, 5, 8};
43
44    pid = fork();
45    if(pid == 0){ //child
46      for(int i = 0; i < sizeof(sleeptime); i++){
47        sleep(sleeptime[i]);
48      }
49    } else { //parrent
50      uint64 xticks = 0;
51      for(int i = 0; i < sizeof(sleeptime); i++){
52        sleep(sleeptime[i]);
53        xticks += sleeptime[i];
54        if(howold(pid, &ticks) != 0){
55          fprintf(2, "error: pid %d should exist with xticks "
56                  "%d\n", pid, xticks);
57          exit(1);
58        }
59        if(ticks > xticks + 5 || ticks < xticks - 5){
```

```
60          fprintf(2, "error: pid %d ticks %d different from "
61                  "xticks %d\n", pid, ticks, xticks);
62          exit(1);
63        }
64      }
65      wait(0);
66      printf("all tests passed\n");
67    }
68  }
69
70  int
71  main(int argc, char *argv[])
72  {
73    sleep(17); //sleep some random clock tick
74    testparam();
75    testfunc();
76    exit(0);
77  }
```

Gợi ý:

- Xem system call kill() để biết cách xác định process dựa trên pid
- System call uptime() cho biết số clock tick trôi qua tính từ thời điểm xv6 bắt đầu chạy
- Thêm biến vào struct proc (kernel/proc.h) để lưu lại thời điểm process được tạo ra
- p->state trong struct proc (kernel/proc.h) cho biết trạng thái của process
- Xác định thời điểm process được tạo ra trong function allocproc() (kernel/proc.c)
- Thời gian tồn tại của process cần được gán vào biến có địa chỉ ở user address space bằng function copyout() (kernel/vm.c)

Bài 2: Thêm system call **trace()** có chức năng trace (lần theo dấu vết, theo dõi) các system call mà một process gọi đến

```
int trace(int mask);
```

trace() nhận tham số **mask** là kiểu số nguyên 32-bit, mỗi bit của mask sẽ cho biết system call nào cần trace. Khi một process muốn trace system call X và Y, nó sẽ gọi đến trace() như sau:

```
trace((1 << SYS_X) | (1 << SYS_Y));
```

Trong đó SYS_X và SYS_Y là hai system call number tương ứng với X và Y, được định nghĩa trong file kernel/syscall.h

Từ lúc này, nếu system call X hoặc Y được gọi bởi process hoặc các child process sinh ra sau này, thông tin của nó sẽ được in ra Terminal theo định dạng:

<process id>: <system call name> -> <system call return value>

User program **trace** (user/trace.c) được viết sẵn để kiểm tra chức năng của system call trace()

Ví dụ:

```
 1 $ trace 32 grep hello README
 2 3: syscall read -> 1023
 3 3: syscall read -> 966
 4 3: syscall read -> 70
 5 3: syscall read -> 0
 6 $
 7 $ trace 2147483647 grep hello README
 8 4: syscall trace -> 0
 9 4: syscall exec -> 3
10 4: syscall open -> 3
11 4: syscall read -> 1023
12 4: syscall read -> 966
13 4: syscall read -> 70
14 4: syscall read -> 0
15 4: syscall close -> 0
16 $
17 $ grep hello README
18 $
19 $ trace 2 usertests forkforkfork
20 usertests starting
21 test forkforkfork:
22 407: syscall fork -> 408
23 408: syscall fork -> 409
24 409: syscall fork -> 410
25 410: syscall fork -> 411
26 409: syscall fork -> 412
27 410: syscall fork -> 413
28 409: syscall fork -> 414
29 411: syscall fork -> 415
30 ...
31 $
```

Trong ví dụ đầu tiên (dòng 1 – 5), system call read có system call number là 32 sẽ được trace khi thực thi command grep hello README

Ví dụ tiếp theo (dòng 7 – 15) thực thi command grep hello README và trace toàn bộ các system call bởi vì mask có 31 bit cuối bằng 1 (2147483647 = 0x7fffffff)

Ví dụ ở dòng 17, command grep hello README được thực thi bình thường, không có system call nào được trace

Ví dụ cuối cùng (dòng 19 – 30) trace system call fork khi thực thi command usertests forkforkfork. usertests (user/usertests.c) là chương trình đặc biệt được sử dụng để kiểm tra xv6 system call

Gợi ý:

- Xem file user/trace.c để hiểu cách user program dùng system call trace()
- Thêm biến vào struct proc (kernel/proc.h) để lưu lại các system call cần trace
- Thay đổi fork() (kernel/proc.c) để process con kế thừa các system call cần trace từ process cha
- Thay đổi syscall() (kernel/syscall.c) để in ra Terminal nếu system call đang thực thi là system call cần trace
- Cần định nghĩa 1 mảng chứa tên các system call

Bài 3: Thêm system call **tracepid**() có chức năng tương tự như trace() với những thay đổi sau:

- Nhận thêm tham số **pid** để xác định process và các child process đang chạy cần trace
- Không cần trace các child process sinh ra sau này

```
int tracepid(int pid, int mask);
```

Gợi ý:

- Xem system call kill() để biết cách xác định process dựa trên pid
- Xem system call wait() để biết cách tìm các child process đang chạy

Bài 4: Thêm system call sysinfo() có chức năng thu thập thông tin hệ thống

```
int sysinfo(struct sysinfo *info);
```

sysinfo() ghi thông tin của hệ thống vào địa chỉ do **info** trỏ tới

struct sysinfo được định nghĩa trong file kernel/sysinfo.h, trong đó **freemem** là số lượng memory trong hệ thống chưa được sử dụng theo byte, và **nproc** là số lượng process có trạng thái khác UNUSED

User program **sysinfotest** (user/sysinfotest.c) được viết sẵn để kiểm tra chức năng của system call sysinfo()

Gợi ý:

- Khai báo `struct sysinfo;` cần đứng trước khai báo prototype sysinfo() trong user/user.h
- Thêm một function mới vào file kernel/kalloc.c để tìm số lượng memory chưa sử dụng
- Thêm một function mới vào file kernel/proc.c để tìm số lượng process
- Thông tin hệ thống cần được gán vào biến có địa chỉ ở user address space bằng function copyout() (kernel/vm.c)

Bài 5*: Khi một page được truy cập (read hoặc write), access bit (bit 6 trong PTE) của page sẽ được gán giá trị 1. Một vài user program (ví dụ như garbage collectors) cần biết giá trị access bit của các page trong virtual address space của nó. Thêm system call **pgaccess()** vào xv6 để hỗ trợ tính năng này.

```
int pgaccess(void *base, int len, void *mask);
```

pgaccess() nhận tham số **base** là địa chỉ của page bắt đầu, tham số **len** là số lượng page cần kiểm tra tính từ page bắt đầu (giới hạn từ 1 đến 1024) và tham số **mask** là địa chỉ của mảng các số nguyên 8bit. Tổng số bit của mảng luôn lớn hơn hoặc bằng số page cần kiểm tra. Tham số base và mask là virtual address của user program

pgaccess() sẽ gán access bit của page bắt đầu vào bit-0 của mảng mask, access bit của page tiếp theo vào bit-1 của mảng mask,… Sau khi gán xong, pgaccess() sẽ xóa các access bit trong các PTE. Bằng cách này, user program sẽ xác định được các page nào đã được truy cập (có access bit bằng 1) trong khoảng thời gian giữa hai lần gọi đến pgaccess()

Thêm user program **pgtbltest** bên dưới để kiểm tra chức năng của pgaccess()

```
1 int
```

```
 2  main(int argc, char *argv[])
 3  {
 4    char *buf;
 5    unsigned int abits;
 6    printf("pgaccess_test starting\n");
 7    buf = malloc(32 * PGSIZE);
 8    if (pgaccess(buf, 32, &abits) < 0)
 9      printf("pgaccess failed\n");
10    buf[PGSIZE * 1] += 1;
11    buf[PGSIZE * 2] += 1;
12    buf[PGSIZE * 30] += 1;
13    if (pgaccess(buf, 32, &abits) < 0)
14      printf("pgaccess failed\n");
15    if (abits != ((1 << 1) | (1 << 2) | (1 << 30)))
16      printf("incorrect access bits set\n");
17    free(buf);
18    printf("pgaccess_test: OK\n");
19    exit(0);
20  }
```

Gợi ý:

- Sử dụng function walk() (kernel/vm.c) để tìm PTE tương ứng với virtual address của page
- Xác định vị trí của access bit trong PTE
- Lưu các bit access vào một mảng tạm trong kernel và copy nó vào mảng của user program bằng function copyout() (kernel/vm.c)
- Xóa bit access trong PTE sau khi đọc xong nếu nó đang bằng 1