

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN



PROJECT 02 – SYSTEMCALL XV6

Môn : Hệ Điều Hành

GIẢNG VIÊN HƯỚNG DẪN

Thầy Lê Giang Thanh
Thầy Nguyễn Thanh Quân
Thầy Lê Hà Minh

SINH VIÊN THỰC HIỆN

22127151 - Lâm Tiến Huy
22127290 - Nguyễn Thị Thu Ngân
22127408 – Kha Vĩnh Thuận

LỚP : 22CLC08

Thành Phố Hồ Chí Minh – 2024

Lời Cảm Tạ

Lời đầu tiên, cho phép chúng em được cảm ơn chân thành và sâu sắc nhất muốn gửi đến thầy Nguyễn Thanh Quân vì đã nhiệt tình hướng dẫn chúng em trong cả quá trình làm đồ án này. Ngoài ra, chúng em cũng muốn gửi lời tri ân đến thầy Lê Giang Thanh với công sức và thời gian thầy bỏ ra để phổ cập kiến thức bộ môn Hệ điều hành. Đây sẽ là tiền đề vững chắc cho những năm học sau trong ngành Công nghệ Thông tin.

Đồ án Systemcall Xv6 lần này là một cơ hội để cho các sinh viên tìm hiểu thêm về các lệnh gọi hệ thống trong một hệ điều hành dựa trên Unix.

Với quá trình làm việc, chúng em cam đoan rằng công việc được phân chia hợp lí, có minh chứng rõ ràng, quá trình làm việc ổn định và linh hoạt. Về kết quả làm việc nói chung, như bảng phân công công việc hay những điều đúc kết được sau đồ án sẽ được ghi chú vào chương cuối cùng của bản báo cáo.

Trong thời gian làm bản cáo cáo, sai sót là điều chúng em không thể tránh khỏi, vì thế chúng em rất mong các thầy xem xét và bỏ qua. Bản báo cáo dựa trên những kiến thức đã được dạy và tự tích lũy nên cũng có phần thiếu kinh nghiệm thực tiễn, trình độ lí luận, chúng em rất mong nhận được những ý kiến, đóng góp của thầy, cô để chúng em có thể học hỏi và hoàn thiện hơn.

Nếu file gửi qua Moodle có lỗi, các thầy có thể truy cập GitHub của tụi em: <https://github.com/tnstanHCMUS/xv6-labs-hcmus>

Chúng em xin chân thành cảm ơn.

Mục Lục

Lời Cảm Tạ.....	2
Mục Lục.....	3
Danh Mục Hình.....	4
Chương 1. Setup.....	5
Chương 2. Thực Hành.....	6
2.1 Using GDB.....	6
2.1.1 Hàm gọi syscall.....	6
2.1.2 Giá trị của p->trapframe->a7.....	7
2.1.3 Trạng thái của CPU.....	9
2.1.4 Kernel Panic.....	9
2.1.5 Vì sao Kernel lại Crash?.....	9
2.1.6 Tên nhị phân khi Kernel Panic.....	11
2.2 System Call Tracing.....	12
2.2.1 Quá trình thực hiện.....	12
2.2.2 Kết quả.....	13
2.3 Sysinfo.....	15
2.3.1 Quá trình thực hiện.....	15
2.3.3 Kết quả.....	17
Chương 3. Thực Hành.....	18
Chương 4. Tổng Kết.....	19
Tài liệu tham khảo.....	20

Danh Mục Hình

Hình 1. Chuyển qua branch Syscall	5
Hình 2. Make qemu-gdb.....	6
Hình 3. Khởi động GDB	6
Hình 4. b syscall và c	7
Hình 5. Kết quả của layout src và backtrace	7
Hình 6. Code của initcode.S	8
Hình 7. Giá trị của p->trapframe->a7	8
Hình 8. Mã lệnh 7.....	8
Hình 9. Trạng thái trước đó của CPU	9
Hình 10. Thay câu lệnh	9
Hình 11. Kernel Panic.....	9
Hình 12. Mã lỗi từ sepc	9
Hình 13. Đặt breakpoint ở mã kernel panic.....	10
Hình 14. Tìm ra mã lỗi.....	10
Hình 15. Tên nhị phân của mã lỗi	11
Hình 16. Kiểm tra thông tin của quá trình.....	11
Hình 17. Khai báo prototype cho trace.....	12
Hình 18. Thêm stub cho trace	12
Hình 19. Định nghĩa system call cho trace.....	12
Hình 20. Thêm biến trace_mask	12
Hình 21. Hàm sys_trace	12
Hình 22. Chỉnh sửa hàm fork và freeproc	13
Hình 23. Thay đổi syscall cho trace.....	13
Hình 24. Kết quả system call tracing.....	14
Hình 25. Khai báo prototype cho sysinfo.....	15
Hình 26. Thêm stub cho sysinfo.....	15
Hình 27. Định nghĩa system call cho sysinfo.....	15
Hình 28. Tạo struct sysinfo trong sysinfo.h.....	15
Hình 29. Thêm system call cho sysinfo	16
Hình 30. Hàm sys_sysinfo	16
Hình 31. Hàm freemem trong kalloc.c.....	16
Hình 32. Hàm procnum trong proc.c	17
Hình 33. Thêm prototype vào defs.h.....	17
Hình 34. Kết quả của sysinfo.....	17
Hình 35. Make grade.....	18

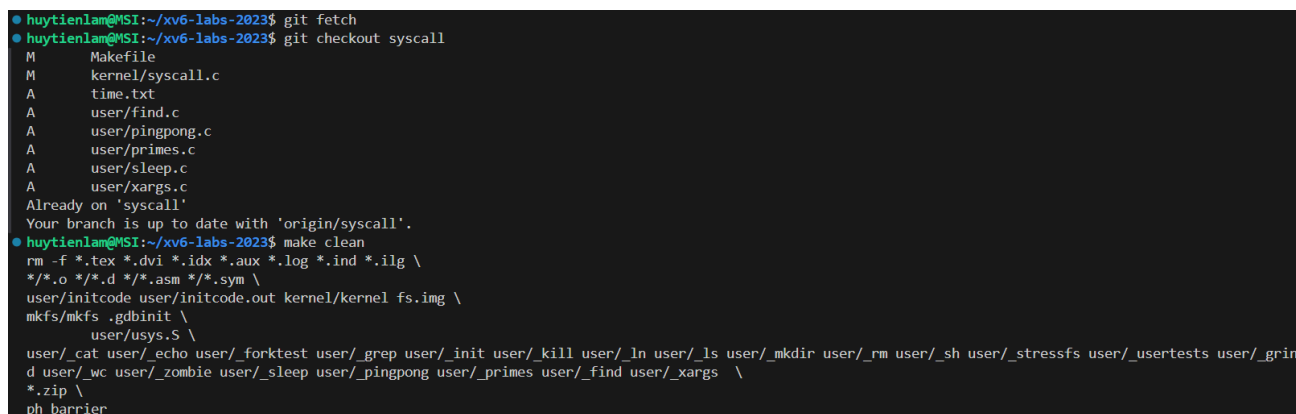
Chương 1

Setup

Trước khi bước vào công việc chính thì việc đầu tiên cần làm đó chính là setup và tạo ra môi trường để có thể hoạt động. Với hai thành viên sử dụng **Windows** và một thành viên sử dụng **macOS**, sẽ có sự khác biệt trong việc cài đặt môi trường. Tuy nhiên, vì đây là **bài thực hành số hai**, nên tất cả đều sẽ tận dụng tiếp hệ điều hành **xv6** mà chúng ta đã cài thêm các tính năng ở **bài thực hành số một**.

Để bắt đầu bài thực hành, chúng ta sẽ chuyển từ branch **util** qua branch **syscall**. Các câu lệnh sẽ lần lượt mang tính **cập nhật**, **đổi branch** và **dọn dẹp**.

```
$ git fetch
$ git checkout syscall
$ make clean
```



```
huytienlan@MSI:~/xv6-labs-2023$ git fetch
huytienlan@MSI:~/xv6-labs-2023$ git checkout syscall
M       Makefile
M       kernel/syscall.c
A       time.txt
A       user/find.c
A       user/pingpong.c
A       user/primes.c
A       user/sleep.c
A       user/xargs.c
Already on 'syscall'
Your branch is up to date with 'origin/syscall'.
huytienlan@MSI:~/xv6-labs-2023$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
  */*.o */*.d */*.asm */*.sym \
  user/initcode user/initcode.out kernel/kernel fs.img \
  mkfs/mkfs .gdbinit \
  user/usys.S \
  user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grin
  d user/_wc user/_zombie user/_sleep user/_pingpong user/_primes user/_find user/_xargs \
  *.zip \
  ph barrier
```

Hình 1. Chuyển qua branch Syscall

Trong bài thực hành này, chúng sẽ thêm một số lệnh **syscall** mới vào **xv6**, giúp chúng ta hiểu cách chúng hoạt động và biết thêm về **kernel** của **xv6**.

Với không gian **user**, code định tuyến các **syscall** vào **kernel** nằm trong **user/usys.S**, được tạo bởi **user/usys.pl** khi chạy **make**, với sự khai báo có trong **user/user.h**.

Code của không gian **kernel** định tuyến các **syscall** đến hàm **kernel** triển khai nó nằm trong **kernel/syscall.c** và **kernel/syscall.h**.

Code liên quan đến các **tiến trình** nằm ở **kernel/proc.h** và **kernel/proc.c**.

Chương 2

Thực Hành

2.1 Using GDB

2.1.1 Hàm gọi syscall

Chúng ta sẽ sử dụng hai cửa sổ terminal song song trong Visual Studio Code, với sự truy nhập vào xv6 sử dụng WSL tương tự như Lab 1.

Ở cửa sổ đầu tiên, nhập **make qemu-gdb**. Khi chúng ta thấy ***** Now run 'gdb' in another window**, chúng ta mở tiếp cửa sổ thứ hai.

```
huytienlam@MSI:~/xv6-labs-2023$ make qemu-gdb
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::26000
```

Hình 2. Make qemu-gdb

Ở cửa sổ thứ hai, nhập **gdb-multiarch -x .gdbinit** để khởi động GDB. Nếu **gdb-multiarch** chưa tồn tại, chúng ta sử dụng lệnh **sudo apt install gdb-multiarch**. Đây là GDB sử dụng phiên bản hỗ trợ nhiều kiến trúc (multi-architecture). Tên tập tin **.gdbinit** là một quy ước trong GDB để tự động chạy các lệnh cấu hình mỗi khi GDB được khởi động. Điều này giúp cấu hình môi trường gỡ lỗi của GDB một cách tự động và linh hoạt.

```
huytienlam@MSI:~/xv6-labs-2023$ gdb-multiarch -x .gdbinit
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
warning: File "/home/huytienlam/xv6-labs-2023/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
  add-auto-load-safe-path /home/huytienlam/xv6-labs-2023/.gdbinit
line to your configuration file "/home/huytienlam/.config/gdb/gdbinit".
To completely disable this security protection add
  set auto-load safe-path /
line to your configuration file "/home/huytienlam/.config/gdb/gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
  info "(gdb)Auto-loading safe path"
The target architecture is set to "riscv:rv64".
warning: No executable has been specified and target does not support
```

Hình 3. Khởi động GDB

Tiếp theo, chúng ta lần lượt gõ các lệnh **b syscall** để đặt breakpoint tại vị trí syscall ở trong **kernel/syscall.c**, line 133, và **c** để tiếp tục quá trình xử lý tới khi gặp breakpoint vừa đặt.

```
(gdb) b syscall
Breakpoint 1 at 0x8000203a: file kernel/syscall.c, line 133.
(gdb) c
Continuing.

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

Hình 4. b syscall và c

Gõ câu lệnh **layout src** để xem trình bày source của file. Cuối cùng, gõ **backtrace** để tạo một bản báo cáo hiển thị chuỗi các cuộc gọi hàm dẫn đến breakpoint tại thời điểm.

```
kernel/syscall.c
B+> 133 {
134     int num;
135     struct proc *p = myproc();
136
137     num = p->trapframe->a7;
138     //num = * (int *) 0;
139     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
140         // Use num to lookup the system call function for num, call it,
141         // and store its return value in p->trapframe->a0
142         p->trapframe->a0 = syscalls[num]();
143     } else {
144         printf("%d %s: unknown sys call %d\n",
145             p->pid, p->name, num);
146         p->trapframe->a0 = -1;
147     }
148 }
149
150
151
152
153
154

remote Thread 1.1 In: syscall
(gdb) backtrace
#0  syscall () at kernel/syscall.c:133
#1  0x0000000080001d6e in usertrap () at kernel/trap.c:67
#2  0x0505050505050505 in ?? ()
(gdb)
```

Hình 5. Kết quả của layout src và backtrace

Theo output của backtrace, chúng ta đang gọi **syscall()** từ **kernel/syscall.c** ở dòng 133, với **syscall()** gọi từ **usertrap()** tại dòng 67 của **kernel/trap.c**.

2.1.2 Giá trị của p->trapframe->a7

Từ vị trí cũ ở câu trên, chúng ta gõ **n** hai lần để di chuyển xuống dòng **num = p->trapframe->a7** rồi gõ tiếp **p/x *p** để in ra **struct proc** ở hệ hexa. Sau khi có kết quả, chúng ta gõ tiếp **p p->trapframe->a7** và thu được kết quả là 7, giá trị của **p->trapframe->a7**.

```

kernel/syscall.c
133 {
134     int num;
135     struct proc *p = myproc();
136
137     num = p->trapframe->a7;
138     //num = * (int *) 0;
139     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
140         // Use num to lookup the system call function for num, call it,
141         // and store its return value in p->trapframe->a0
142         p->trapframe->a0 = syscalls[num]();
143     } else {
144         printf("%d %s: unknown sys call %d\n",
145             p->pid, p->name, num);
146         p->trapframe->a0 = -1;
147     }
148 }
149
150
151
152
153
154

remote Thread 1.3 In: syscall L137 PC: 0x80002050
#2 0x0505050505050505 in ?? ()
(gdb) n
(gdb) n
(gdb) p/x *p
$1 = {lock = {locked = 0x0, name = 0x800081b8, cpu = 0x0}, state = 0x4, chan = 0x0, killed = 0x0, xstate = 0x0, pid = 0x1, parent = 0x0,
    kstack = 0x3fffffd000, sz = 0x1000, pagetable = 0x87f73000, trapframe = 0x87f74000, context = {ra = 0x800014c0, sp = 0x3fffffd0,
    s0 = 0x3fffffdea0, s1 = 0x80008d50, s2 = 0x80008920, s3 = 0x1, s4 = 0x3fffffded0, s5 = 0x8000ebd8, s6 = 0x3, s7 = 0x800199f0, s8 = 0x1,
    s9 = 0x80019b18, s10 = 0x4, s11 = 0x0}, ofile = {0x0 <repeats 16 times>}, cwd = 0x80016e60, name = {0x69, 0x6e, 0x69, 0x74, 0x63, 0x6f,
    0x64, 0x65, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}}
(gdb) p p->trapframe->a7
$2 = 7

```

Hình 7. Giá trị của `p->trapframe->a7`

Từ mã nguồn trong `user/initcode.S`, chúng ta thấy rằng mã syscall sẽ được thực thi được lưu trữ trong thanh ghi `a7`.

```

4  #include "syscall.h"
5
6  # exec(init, argv)
7  .globl start
8  start:
9      la a0, init
10     la a1, argv
11     li a7, SYS_exec
12     ecall

```

Hình 6. Code của `initcode.S`

Trong trường hợp này, như chúng ta đã đọc trước đó thì trong thanh ghi `a7` đang chứa số 7. Đối chiếu số 7 với `kernel/syscall.h`, nó tương trưng cho `SYS_exec`.

```

1  // System call numbers
2  #define SYS_fork    1
3  #define SYS_exit    2
4  #define SYS_wait    3
5  #define SYS_pipe    4
6  #define SYS_read    5
7  #define SYS_kill    6
8  #define SYS_exec    7

```

Hình 8. Mã lệnh 7

2.1.3 Trạng thái của CPU

Chúng ta tiếp tục sử dụng câu lệnh **p /x \$sstatus** với **p** sử dụng để hiển thị thông tin, **/x** để chọn dạng hexa và **\$sstatus** là thanh ghi chứa trạng thái trước đó của CPU. Ở đây, chúng ta nhận thấy trạng thái trước đó của CPU đó là **0x22**.

```
(gdb) p /x $sstatus
$3 = 0x22
```

Hình 9. Trạng thái trước đó của CPU

2.1.4 Kernel Panic

```
kernel > C syscall.c
130
131 void
132 syscall(void)
133 {
134     int num;
135     struct proc *p = myproc();
136
137     //num = p->trapframe->a7;
138     num = * (int *) 0;
139     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
140         // Use num to lookup the system call function for num, call it,
141         // and store its return value in p->trapframe->a0
142         p->trapframe->a0 = syscalls[num]();
```

Hình 10. Thay câu lệnh

Trong file **kernel/syscall.c**, thay câu lệnh **num = p->trapframe->a7** bằng **num = * (int *) 0;** và chạy **make qemu**. Chúng ta sẽ thấy hiện lỗi **panic: kerneltrap**.

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
scause=0x000000000000000d
sepc=0x000000008000204e stval=0x0000000000000000
panic: kerneltrap
```

Hình 11. Kernel Panic

sepc là nơi lưu trữ địa chỉ chỉ ra vị trí của lệnh gây ra **kernel panic**, hữu ích trong việc xử lý lỗi. Khi này, chúng ta sẽ thu được **sepc = 0x000000008000204e**. Chúng ta quan tâm tới 8 chữ số cuối cùng là **8000204e** và tìm mã lỗi này trong **kernel/kernel.asm**.

```
4510 //num = p->trapframe->a7;
4511 num = * (int *) 0;
4512 | 8000204e: 00002683          lw  a3,0(zero) # 0 <_entry-0x80000000>
```

Hình 12. Mã lỗi từ sepc

Chúng ta thấy rằng ngay tại đây, **num = * (int *) 0;** chính là câu lệnh gây ra mã lỗi. Mã hợp ngữ liên quan chính là **lw a3,0(zero)** và **num** ở đây đang liên quan đến thanh ghi **a3**.

2.1.5 Vì sao Kernel lại Crash?

Để kiểm tra trạng thái của **processor** và **kernel** tại lệnh gây ra lỗi, khởi động gdb như ban nãy và đặt một **breakpoint** tại **epc** (bộ đếm chương trình ngoại lệ) gây ra lỗi, tức mã

sepc của chúng ta lúc này. Các câu lệnh sẽ gõ lần lượt là **b *0x000000008000204e** để đặt breakpoint, **layout asm** và gõ **c** để tiếp tục. Khi này thread sẽ **đụng breakpoint** chúng ta vừa đặt.

```
(gdb) b *0x000000008000204e
Breakpoint 1 at 0x8000204e: file kernel/syscall.c, line 138.

0x8000203a <syscall> addi    sp,sp,-32
0x8000203c <syscall+2> sd      ra,24(sp)
0x8000203e <syscall+4> sd      s0,16(sp)
0x80002040 <syscall+6> sd      s1,8(sp)
0x80002042 <syscall+8> addi    s0,sp,32
0x80002044 <syscall+10> auipc   ra,0xfffff
0x80002048 <syscall+14> jalr    -410(ra)
0x8000204c <syscall+18> mv      s1,a0
B+> 0x8000204e <syscall+20> lw      a3,0(zero) # 0x0
0x80002052 <syscall+24> addiw   a4,a3,-1
0x80002056 <syscall+28> li      a5,20
0x80002058 <syscall+30> bltu    a5,a4,0x80002076 <syscall+60>
0x8000205c <syscall+34> slli    a4,a3,0x3
0x80002060 <syscall+38> auipc   a5,0x6
0x80002064 <syscall+42> addi    a5,a5,944
0x80002068 <syscall+46> add     a5,a5,a4
0x8000206a <syscall+48> ld       a5,0(a5)
0x8000206c <syscall+50> beqz    a5,0x80002076 <syscall+60>
0x8000206e <syscall+52> ld       s1,88(a0)
0x80002070 <syscall+54> jalr    a5
0x80002072 <syscall+56> sd      a0,112(s1)
0x80002074 <syscall+58> j       0x80002092 <syscall+88>
0x80002076 <syscall+60> addi    a2,s1,344
0x8000207a <syscall+64> lw      a1,48(s1)

remote Thread 1.1 In: syscall
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at kernel/syscall.c:138
```

Hình 13. Đặt breakpoint ở mã kernel panic

Chúng ta gõ lệnh **n** và kernel lại **panic** nên không thể tiếp tục được mà sẽ bị treo, khi này chúng ta nhấn **Ctrl + C** để dừng thực thi lệnh. Sau đó, dùng lệnh **p \$scause**, hay supervisor trap cause, chứa mã nguyên nhân gây lỗi.

```
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, syscall () at kernel/syscall.c:138
(gdb) n

Thread 2 received signal SIGINT, Interrupt.
panic (s=s@entry=0x800083c0 "kerneltrap") at kernel/printf.c:126
(gdb) p $scause
$1 = 13
```

Hình 14. Tìm ra mã lỗi

Ở đây mã chúng ta đọc được là **13**, khi đối chiếu với cuốn sách **RISC-V Privileged Instructions**, chúng ta biết được đây là lỗi **Load Page Fault**. Còn gọi là **lỗi nạp trang**, **Load Page Fault** là một loại ngoại lệ xảy ra khi chương trình cố gắng đọc dữ liệu từ một trang bộ nhớ **không tồn tại** trong bộ nhớ vật lý, hoặc nếu trang đó **chưa được tải**. Lí do chính cho việc **kernel crash** là do đã xảy ra lỗi khi **nạp dữ liệu** từ địa chỉ bộ nhớ **0** vào thanh ghi **a3**.

Dựa vào giáo trình **xv6: a simple, Unix-like teaching operating system**, hình 3.3, có thể thấy rằng địa chỉ **0** không được ánh xạ vào không gian của **kernel**, mà **kernel** bắt đầu từ địa chỉ ảo tại **KERNBASE** là **0x80000000**.

2.1.6 Tên nhị phân khi Kernel Panic

Chúng ta sẽ khởi động lại hai terminal mới như ban đầu. Bản chất cần làm đó chính là in ra phần tên **name** của kiểu dữ liệu tên **proc** được cài sẵn. Chúng ta sẽ quay về **b syscall** để đặt breakpoint như trước đó, **c** để tiếp tục, **layout src** và dùng hai câu lệnh **n** để di chuyển xuống, sau đó gõ **p p->name** để xem tên nhị phân.

```
(gdb) b syscall
Breakpoint 1 at 0x8000203a: file kernel/syscall.c, line 133.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at kernel/syscall.c:133
133 _ {
kernel/syscall.c
B+ 133 {
134 int num;
135 struct proc *p = myproc();
136 //num = p->trapframe->a7;
> 137 num = *(int *) 0;
139 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
140 // Use num to lookup the system call function for num, call it,
141 // and store its return value in p->trapframe->a0
142 p->trapframe->a0 = syscalls[num]();
143 } else {
144 printf("%d %s: unknown sys call %d\n",
145 p->pid, p->name, num);
146 p->trapframe->a0 = -1;
147 }
148 }
149
150
151
152
153
154

remote Thread 1.3 In: syscall
(gdb) n
(gdb) n
(gdb) p p->name
$1 = "initcode\000\000\000\000\000\000\000"
(gdb)
```

Hình 15. Tên nhị phân của mã lỗi

Phần tên được in ra là **initcode\000\000\000\000\000\000\000**. Có thể thấy rằng ngay tại vị trí này là **initcode**, cũng chính là process đầu tiên trong **xv6**. In ra cấu trúc **proc** sẽ cho phép người dùng xem thông tin về quá trình này, tiếp tục sử dụng lệnh **p *p**.

```
(gdb) p *p
$2 = {lock = {locked = 0, name = 0x800081b8 "proc", cpu = 0x0}, state = RUNNING, chan = 0x0, killed = 0, xstate = 0, pid = 1, parent = 0x0,
kstack = 274877894656, sz = 4096, pagetable = 0x87f73000, trapframe = 0x87f74000, context = {ra = 2147488960, sp = 274877898352,
s0 = 274877898400, s1 = 2147519824, s2 = 2147518752, s3 = 1, s4 = 274877898448, s5 = 2147544024, s6 = 3, s7 = 2147588592, s8 = 1,
s9 = 2147588888, s10 = 4, s11 = 0}, ofile = {0x0 <repeats 16 times>}, cwd = 0x80016e60 <itable+24>,
name = "initcode\000\000\000\000\000\000\000"}
(gdb)
```

Hình 16. Kiểm tra thông tin của quá trình

Quan sát kỹ, chúng ta sẽ thấy **pid = 1**. Vậy Process ID của **initcode** sẽ là **1**.

2.2 System Call Tracing

2.2.1 Quá trình thực hiện

Tương tự như Lab 1, chúng ta sẽ mở **Makefile**. Từ đó, kéo xuống tới dòng **UPROGS=** và thêm vào phía dưới cụm **\$U/_trace** và lưu lại.

Đầu tiên, ta sẽ khai báo một **prototype** cho **system call** này vào **user/user.h**.

```
24 int uptime(void);
25 int trace(int);
```

Hình 17. Khai báo prototype cho trace

Tiếp theo, ta sẽ thêm một đoạn mã tạm thời (**stub**) vào **user/usys.pl**.

```
38 entry("uptime");
39 entry("trace");
```

Hình 18. Thêm stub cho trace

Chúng ta sẽ định nghĩa một **system call** mới trong **kernel/syscall.h** với mã là **22**.

```
22 #define SYS_close 21
23 // thêm vào lời gọi hệ thống
24 #define SYS_trace 22
```

Hình 19. Định nghĩa system call cho trace

Vào **kernel/proc.h**, thêm một biến số nguyên **trace_mask** vào **struct proc** với vai trò theo dõi và gỡ lỗi. Nếu **trace_mask** bằng **0**, tắt gỡ lỗi đi.

```
108 //chức năng là theo dõi và gỡ lỗi
109 int trace_mask; // tracing
110 //nếu trace_mask = 0 tắt gỡ lỗi
111 //kết thúc
```

Hình 20. Thêm biến trace_mask

Thêm một hàm **uint64 sys_trace(void)** trong **kernel/sysproc.c** để tạo ra một **system call** mới từ việc ghi nhớ **argument** vào một biến mới trong cấu trúc **proc**. Hàm này lấy **argument** của **system call** thông qua **argint**, lấy giá trị tại **argument** thứ nhất (0-indexed) và gán vào biến **mask**. Tiếp theo, dùng **myproc()** lấy **pointer** chỉ đến **proc** rồi gán giá trị của **mask** vào trường **trace_mask** của cấu trúc **proc**, để lưu trữ các cấu hình cho việc ghi nhật ký hoặc theo dõi trong quá trình thực thi của tiến trình.

```
kernel > C sysproc.c
96 // tracing
97 uint64
98 sys_trace(void)
99 {
100     int mask;
101     argint(0, &mask);
102     struct proc *p = myproc();
103     p->trace_mask = mask;
104     return 0;
105 }
```

Hình 21. Hàm sys_trace

Tiến hành chỉnh sửa hàm **fork** trong **kernel/proc.c** để sao chép **trace_mask** từ tiến trình cha sang tiến trình con, đồng thời thêm **p->trace_mask = 0** vào hàm **static freeproc** để tắt chức năng theo dõi.

```

317     safestrcpy(np->name, p->name, sizeof(p->name));
318
319     // đảm bảo chức năng theo dõi dc đồng nhất giữa tiến trình cha và con
320     np->trace_mask = p->trace_mask;
321     pid = np->pid;
174 //Kết thúc chương trình, trace_mask = 0
175     p->trace_mask = 0;
176 }

```

Hình 22. Chỉnh sửa hàm **fork** và **freeproc**

Cuối cùng, vào trong **kernel/syscall.c**, thêm những lệnh cho hàm **trace** với cú pháp tương tự thành phần trong những phần **prototype**, **array mapping**, **syscall_names**. Thay đổi hàm **void syscall(void)** để in đầu ra của **trace**.

```

102 extern uint64 sys_mkdir(void);
103 extern uint64 sys_close(void);
104 extern uint64 sys_trace(void);
129 [SYS_mkdir] sys_mkdir,
130 [SYS_close] sys_close,
131 [SYS_trace] sys_trace,
135 static char* syscall_names[] = {
136     "fork", "exit", "wait", "pipe", "read", "kill", "exec", "fstat", "chdir", "dup", "getpid",
137     "sbrk", "sleep", "uptime", "open", "write", "mknod", "unlink", "link",
138     "mkdir", "close", "trace", "sysinfo"
139 };
141 void
142 syscall(void)
143 {
144     int num;
145     struct proc *p = myproc();
146     //num = * (int *) 0;
147     num = p->trapframe->a7;
148     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
149         // Use num to lookup the system call function for num, call it,
150         // and store its return value in p->trapframe->a0
151         p->trapframe->a0 = syscalls[num]();
152     }
153     if(p->trace_mask & (1 << num)){
154         printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num - 1], p->trapframe->a0);
155     }
156 } else {
157     printf("%d %s: unknown sys call %d\n",
158         p->pid, p->name, num);
159     p->trapframe->a0 = -1;
160 }
161 }

```

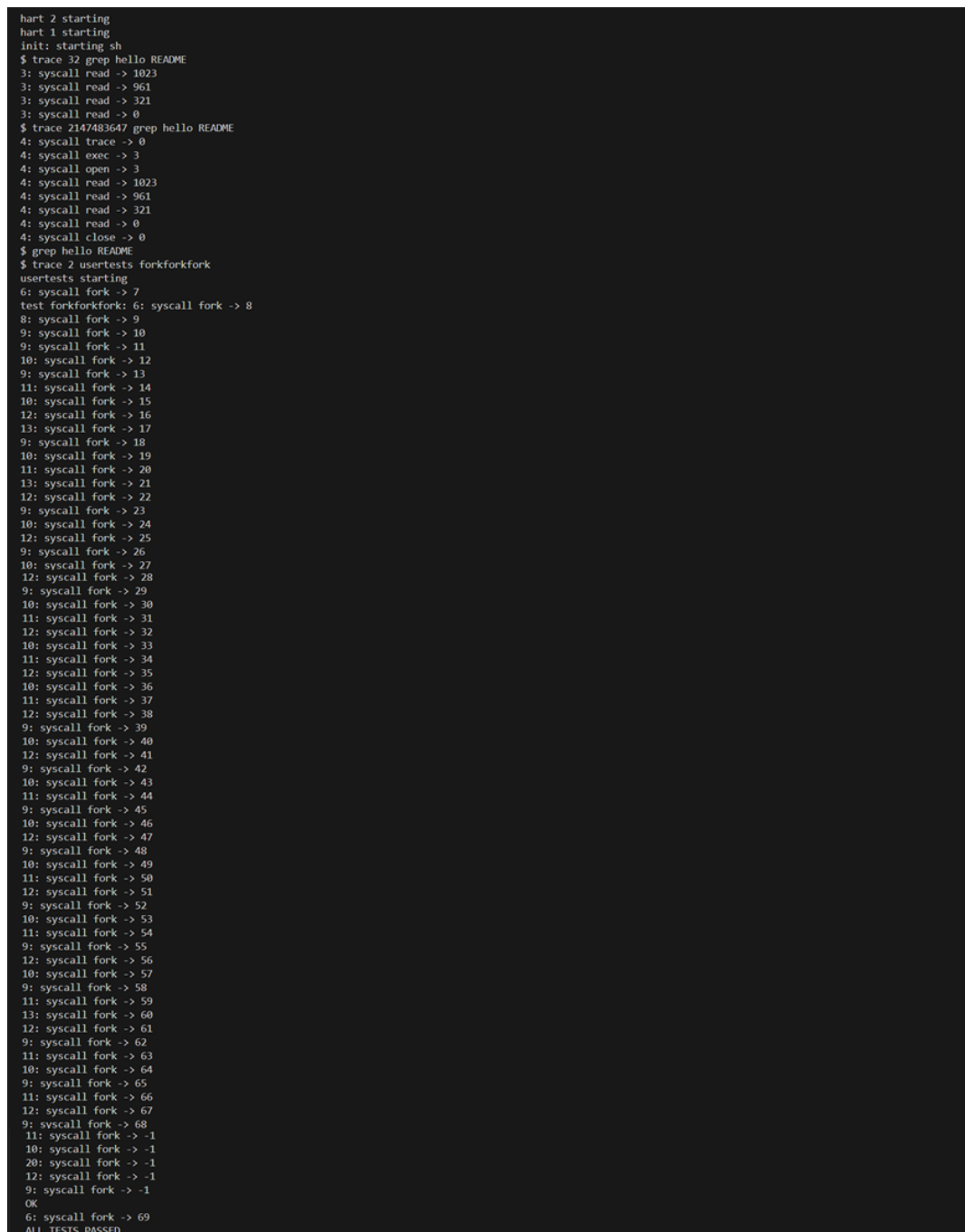
Hình 23. Thay đổi **syscall** cho **trace**

Sự thay đổi của chúng ta bắt đầu từ việc gán **giá trị** của thanh ghi **a7** trong **trapframe** của tiến trình hiện tại vào biến **num**. Sau đó, kiểm tra sự tồn tại của con trỏ tới hàm xử lý **system call** cho **num**, đồng thời xem **giá trị** của **num** có nằm trong phạm vi **hợp lệ** của các **system call** không, với **NELEM(syscalls)** đại diện cho số lượng **system call**. Nếu hợp lệ, lưu **giá trị** trả về của nó vào thanh ghi **a0** của **trapframe** trong tiến trình. Tiếp theo, chúng ta kiểm tra xem **trace_mask** có được thiết lập cho **num** không. Nếu được thiết lập, thì tiến trình sẽ tiến hành in ra đầu ra của **trace**.

2.2.2 Kết quả

Chúng ta sẽ tiến hành **make qemu** và lần lượt gõ những câu lệnh, chờ câu lệnh chạy hoàn tất rồi gõ tiếp tục. Các câu lệnh bao gồm:

```
$ trace 32 grep hello README
$ trace 2147483647 grep hello README
$ grep hello README
$ trace 2 usertests forkforkfork
```



```
hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 961
3: syscall read -> 321
3: syscall read -> 0
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 961
4: syscall read -> 321
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
usertests starting
6: syscall fork -> 7
test forkforkfork: 6: syscall fork -> 8
8: syscall fork -> 9
9: syscall fork -> 10
9: syscall fork -> 11
10: syscall fork -> 12
9: syscall fork -> 13
11: syscall fork -> 14
10: syscall fork -> 15
12: syscall fork -> 16
13: syscall fork -> 17
9: syscall fork -> 18
10: syscall fork -> 19
11: syscall fork -> 20
13: syscall fork -> 21
12: syscall fork -> 22
9: syscall fork -> 23
10: syscall fork -> 24
12: syscall fork -> 25
9: syscall fork -> 26
10: syscall fork -> 27
12: syscall fork -> 28
9: syscall fork -> 29
10: syscall fork -> 30
11: syscall fork -> 31
12: syscall fork -> 32
10: syscall fork -> 33
11: syscall fork -> 34
12: syscall fork -> 35
10: syscall fork -> 36
11: syscall fork -> 37
12: syscall fork -> 38
9: syscall fork -> 39
10: syscall fork -> 40
12: syscall fork -> 41
9: syscall fork -> 42
10: syscall fork -> 43
11: syscall fork -> 44
9: syscall fork -> 45
10: syscall fork -> 46
12: syscall fork -> 47
9: syscall fork -> 48
10: syscall fork -> 49
11: syscall fork -> 50
12: syscall fork -> 51
9: syscall fork -> 52
10: syscall fork -> 53
11: syscall fork -> 54
9: syscall fork -> 55
12: syscall fork -> 56
10: syscall fork -> 57
9: syscall fork -> 58
11: syscall fork -> 59
13: syscall fork -> 60
12: syscall fork -> 61
9: syscall fork -> 62
11: syscall fork -> 63
10: syscall fork -> 64
9: syscall fork -> 65
11: syscall fork -> 66
12: syscall fork -> 67
9: syscall fork -> 68
11: syscall fork -> -1
10: syscall fork -> -1
20: syscall fork -> -1
12: syscall fork -> -1
9: syscall fork -> -1
OK
6: syscall fork -> 69
ALL TESTS PASSED
```

Hình 24. Kết quả system call tracing

Kết quả ra đúng nếu chương trình chạy tất cả các test case đều ổn, tuy Process ID có thể sẽ khác nhau.

2.3 Sysinfo

2.3.1 Quá trình thực hiện

Tương tự như trên, chúng ta sẽ mở **Makefile**. Từ đó, kéo xuống tới dòng **UPROGS=** và thêm vào phía dưới cụm **\$U/_ sysinfotest** và lưu lại.

Đầu tiên, ta sẽ khai báo một **prototype** cho **system call** này vào **user/user.h**.

```
25 int trace(int);
26 int sysinfo(struct sysinfo *);
```

Hình 25. Khai báo prototype cho sysinfo

Tiếp theo, ta sẽ thêm một đoạn mã tạm thời (**stub**) vào **user/usys.pl**.

```
39 entry("trace");
40 entry("sysinfo");
```

Hình 26. Thêm stub cho sysinfo

Chúng ta sẽ định nghĩa một **system call** mới trong **kernel/syscall.h** với mã là **22**.

```
23 // thêm vào lời gọi hệ thống
24 #define SYS_trace 22
25 #define SYS_sysinfo 23
```

Hình 27. Định nghĩa system call cho sysinfo

Tạo file **kernel/sysinfo.h** chứa structure **sysinfo** với các thuộc tính số nguyên 64-bit **freemem** dùng để chứa lượng free memory theo bytes, và **nproc** chứa số lượng tiến trình.

```
C sysproc.c M x C sysinfo.h M x
kernel > C sysinfo.h
1 struct sysinfo {
2     uint64 freemem; // amount of free memory (bytes)
3     uint64 nproc; // number of process
4 };
```

Hình 28. Tạo struct sysinfo trong sysinfo.h

Thêm một hàm **uint64 sys_sysinfo(void)** trong **kernel/sysproc.c** để tạo ra một **system call** mới, sao chép một **sysinfo** về **user space**. Tạo một biến cấu trúc **sysinfo** có tên là **info** để lưu trữ thông tin hệ thống, và một biến **uint64 addr** để lưu địa chỉ trong **user space** nơi dữ liệu **sysinfo** sẽ được sao chép đến.

Với **info**, sử dụng các hàm có sẵn, chúng ta gán giá trị của lượng bộ nhớ trống trên hệ thống vào trường **freemem**, rồi gán số lượng tiến trình đang chạy trên hệ thống vào **nproc**. Tiếp theo, dùng hàm **argaddr** lấy địa chỉ của **argument** đầu tiên (0-indexed) và gán vào biến **addr**. Dùng tiếp hàm **copyout** sao chép dữ liệu từ **info** vào **user space** tại địa chỉ được chỉ định bởi **addr**, với **p->pagetable** là bảng trang của tiến trình. Nếu sao chép không thành công và trả về giá trị nhỏ hơn 0, hàm trả về -1 để báo lỗi. Nếu sao chép thành công, hàm trả về 0 để báo hiệu rằng **system call** này đã được thực hiện thành công.


```

106 //info
107 uint64
108 sys_sysinfo(void){
109     struct sysinfo info;
110     uint64 addr;
111     struct proc *p = myproc();
112     info.freemem = freemem();
113     info.nproc = proctnum();
114
115     argaddr(0, &addr);
116
117     if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
118         return -1;
119     return 0;
120 }

```

Hình 30. Hàm `sys_sysinfo`

Cuối cùng, vào trong **kernel/syscall.c**, thêm những lệnh cho **sysinfo** với cú pháp tương tự thành phần trong những phần **prototype**, **array mapping**, **syscall_names**.

```

103 extern uint64 sys_close(void);
104 extern uint64 sys_trace(void);
105 extern uint64 sys_sysinfo(void);
130 [SYS_close] sys_close,
131 [SYS_trace] sys_trace,
132 [SYS_sysinfo] sys_sysinfo,
133 static char* syscall_names[] = {
134     "fork", "exit", "wait", "pipe", "read", "kill", "exec", "fstat", "chdir", "dup", "getpid",
135     "sbrk", "sleep", "uptime", "open", "write", "mknod", "unlink", "link",
136     "mkdir", "close", "trace", "sysinfo"
137 };

```

Hình 29. Thêm system call cho `sysinfo`

Thêm hàm **int freemem (void)** vào **kernel/kalloc.c** để thu thập lượng bộ nhớ trống trong hệ thống. Hàm **acquire(&kmem.lock)** sử dụng để khóa bộ nhớ, ngăn chặn các race condition khi truy cập vào cấu trúc dữ liệu bộ nhớ, đảm bảo không có tiến trình khác có thể thay đổi dữ liệu bộ nhớ trong khi chúng ta đang tính toán kích thước bộ nhớ trống. Tiếp theo, khởi tạo biến **size** để lưu trữ số lượng, con trỏ **r** đến danh sách bộ nhớ trống và bắt đầu từ **kmem.freelist** – danh sách chứa các phần tử bộ nhớ trống. Vòng lặp while duyệt qua từng block bộ nhớ trống trong danh sách và tăng giá trị của biến **size** lên mỗi lần duyệt. Sau đó, **release(&kmem.lock)** để mở khóa bộ nhớ, cho phép các tiến trình khác tiếp tục truy cập vào cấu trúc dữ liệu bộ nhớ. Kết thúc, trả về kích thước tổng của bộ nhớ trống, tính bằng cách nhân số lượng block bộ nhớ trống với kích thước của mỗi block.

```

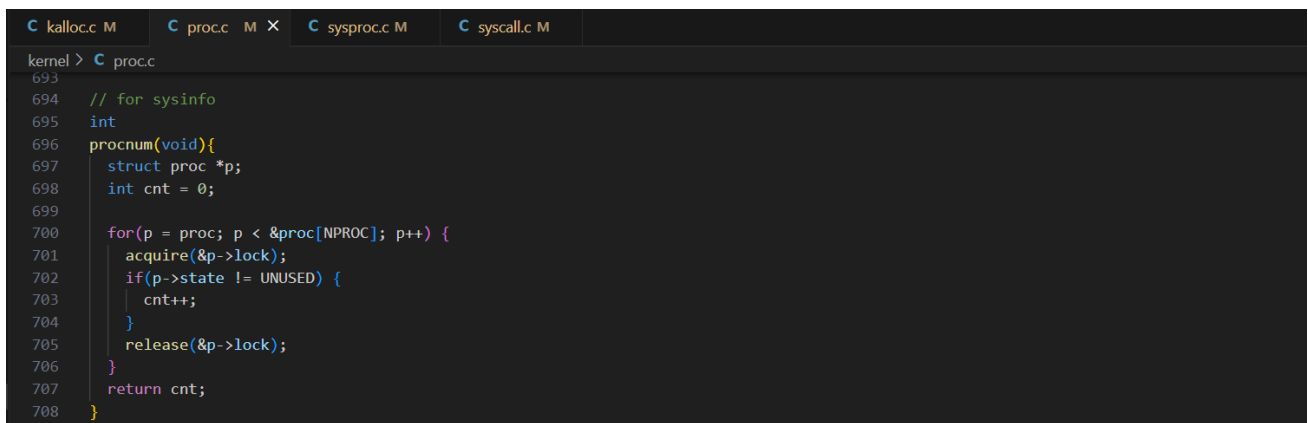
C kalloc.c M X C sysproc.c M C syscall.c M
kernel > C kalloc.c
84 int freemem(void)
85 {
86     acquire(&kmem.lock);
87     struct run *p = kmem.freelist;
88     uint num = 0;
89     while(p)
90     {
91         p = p->next;
92         num++;
93     }
94     release(&kmem.lock);
95     return num * PGSIZE;
96 }

```

Hình 31. Hàm `freemem` trong `kalloc.c`

Chúng ta sẽ thêm hàm **int procnum(void)** trong file **kernel/proc.c** để đếm số lượng tiến trình đang hoạt động trong hệ thống. Khai báo một con trỏ **p** kiểu **proc** để duyệt qua danh sách các tiến trình, và biến đếm **cnt** với giá trị **0**. Vòng lặp **for** được sử dụng để duyệt qua mảng các tiến trình từ **proc** đến **proc[NPROC]**, với **proc** là mảng chứa thông tin của các tiến trình, và **NPROC** là số lượng tối đa các tiến trình có thể có trong hệ thống.

Hệ thống cần phải đảm bảo rằng không có tiến trình nào khác đang thực hiện thay đổi trên cùng một tiến trình dùng **acquire(&p->lock)**. Nếu trạng thái của tiến trình không phải là **UNUSED**, thì tăng biến đếm **cnt**. Sau khi kiểm tra, dùng **release(&p->lock)** để mở khoá. Sau khi duyệt qua và đếm xong, hàm trả về số lượng tiến trình đang không ở trạng thái **UNUSED**, tức số lượng tiến trình đang hoạt động trong hệ thống.



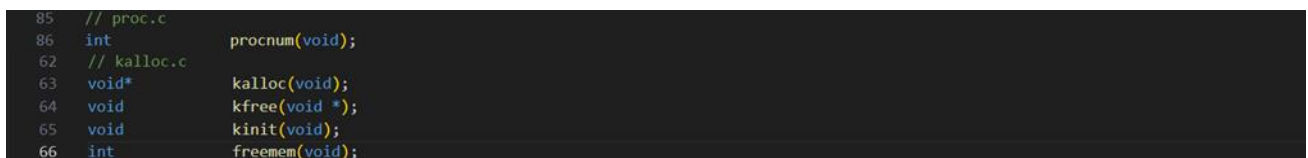
```

kernel > C proc.c
693
694 // for sysinfo
695 int
696 procnum(void){
697     struct proc *p;
698     int cnt = 0;
699
700     for(p = proc; p < &proc[NPROC]; p++) {
701         acquire(&p->lock);
702         if(p->state != UNUSED) {
703             cnt++;
704         }
705         release(&p->lock);
706     }
707     return cnt;
708 }

```

Hình 32. Hàm procnum trong proc.c

Bước cuối cùng là thêm các **prototype** cho các hàm mới tạo trong các file **kernel/proc.c** và **kernel/kalloc.c** vào **kernel/defs.h**.



```

85 // proc.c
86 int procnum(void);
87
88 // kalloc.c
89 void* kalloc(void);
90 void kfree(void *);
91 void kinit(void);
92 int freemem(void);

```

Hình 33. Thêm prototype vào defs.h

2.3.3 Kết quả

Sau khi **make qemu**, gõ **sysinfotest** và test ra đúng khi kết quả như hình.



```

hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK

```

Hình 34. Kết quả của sysinfo

Hoàn Thành

[illegible]

Systemcall xv6

Chương 4

Tổng Kết

Qua đồ án **Systemcall xv6**, chúng ta sẽ hiểu rõ hơn về việc sử dụng GDB cũng như code thêm những lệnh system call mới cho xv6 bằng ngôn ngữ C. Quá trình làm việc của nhóm chúng em khá rõ ràng, rành mạch, và có logic, phân chia công việc đều đặn. Chúng em xin chân thành cảm ơn!

MSSV	Họ và tên	Công việc	Hoàn thành
22127151	Lâm Tiến Huy	Thực hiện Using GDB. Làm báo cáo chính.	100%
22127290	Nguyễn Thị Thu Ngân	Thực hiện Sysinfo.	100%
22127408	Kha Vĩnh Thuận	Thực hiện System Call Tracing.	100%

Hình 36. Bảng phân công công việc

Tài liệu tham khảo

Lab: System calls. (n.d.). <https://pdos.csail.mit.edu/6.1810/2023/labs/syscall.html>

Cox, R., Kaashoek, F., & Morris, R. (2022). *xv6: a simple, Unix-like teaching operating system.*

<https://pdos.csail.mit.edu/6.1810/2023/xv6/book-riscv-rev3.pdf>

Waterman, A., Asanovic, K., Hauser, J., & RISC-V International. (2020). *The RISC-V Instruction Set*

Manual: Vol. Volume II: Privileged Architecture (Document Version 20211203).

<https://drive.google.com/file/d/1EMip5dZlnypTk7pt4WWUKmtjUKTOkBqh/view>

[Lab Report] MIT 6.S081 Lab: system calls (v2022). (2023, February 6).

https://jinzhec2.github.io/blog/post/6.s081_2022_lab2/