Trường Đại học Khoa học tự nhiên – Khoa Công nghệ thông tin.

Đồ án thực hành 02

Operating System – Hệ điều hành.

Nhóm 3T
Tháng 11, 2024.

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN THỰC HÀNH SỐ 02

Bộ môn: Hệ điều hành.

Tên đề tài:

"System Call".

Tên nhóm: 3T.

Thành viên:

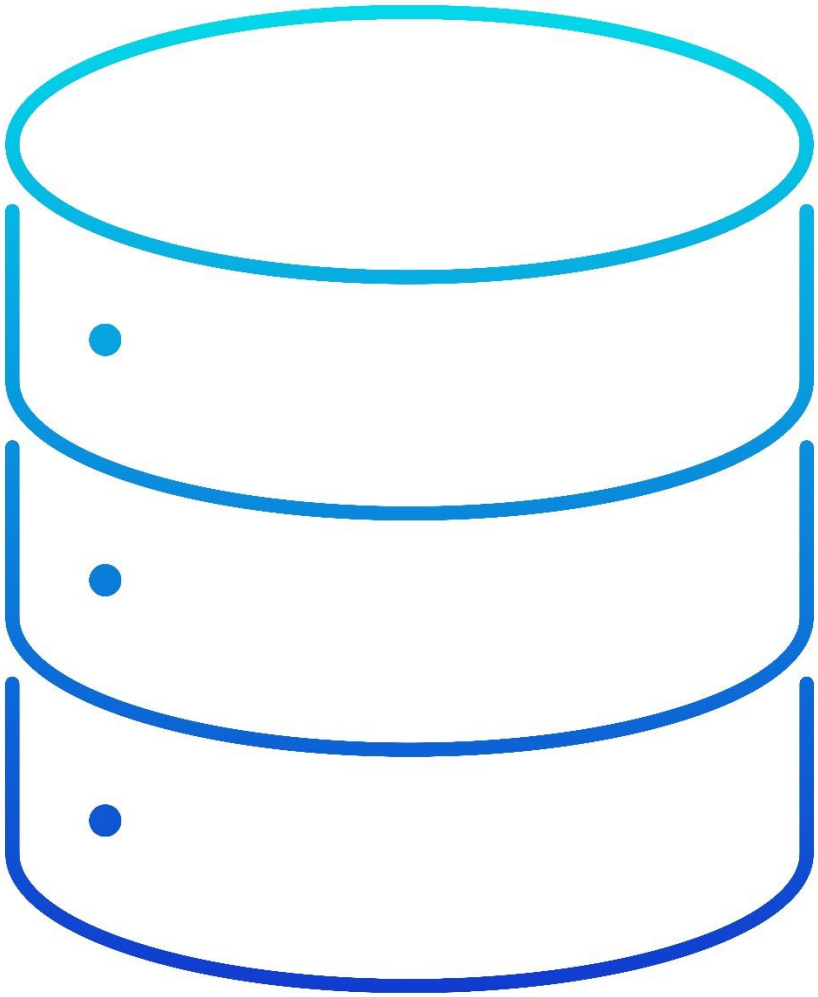
1. 22120384 – Nguyễn Đình Trí.
2. 22120398 – Vũ Hoàng Nhật Trường.
3. 22120412 – Nguyễn Anh Tường.

Thông tin chung:

1. **Bộ môn:** Nhập môn khoa học dữ liệu.
2. **Giảng viên lý thuyết:** Thầy Trần Trung Dũng.
3. **Giảng viên thực hành:** Thầy Lê Thanh Quân.
4. **Mã lớp:** 22_4.
5. **Tên nhóm:** 3T.
6. **Danh sách thành viên:**
    a. 22120384 – Nguyễn Đình Trí.
    b. 22120398 – Vũ Hoàng Nhật Trường.
    c. 22120412 – Nguyễn Anh Tường.
7. **Link github repository:** “Click here to go to our github repository.”

Contents

Mở đầu: *Giới thiệu chung.*



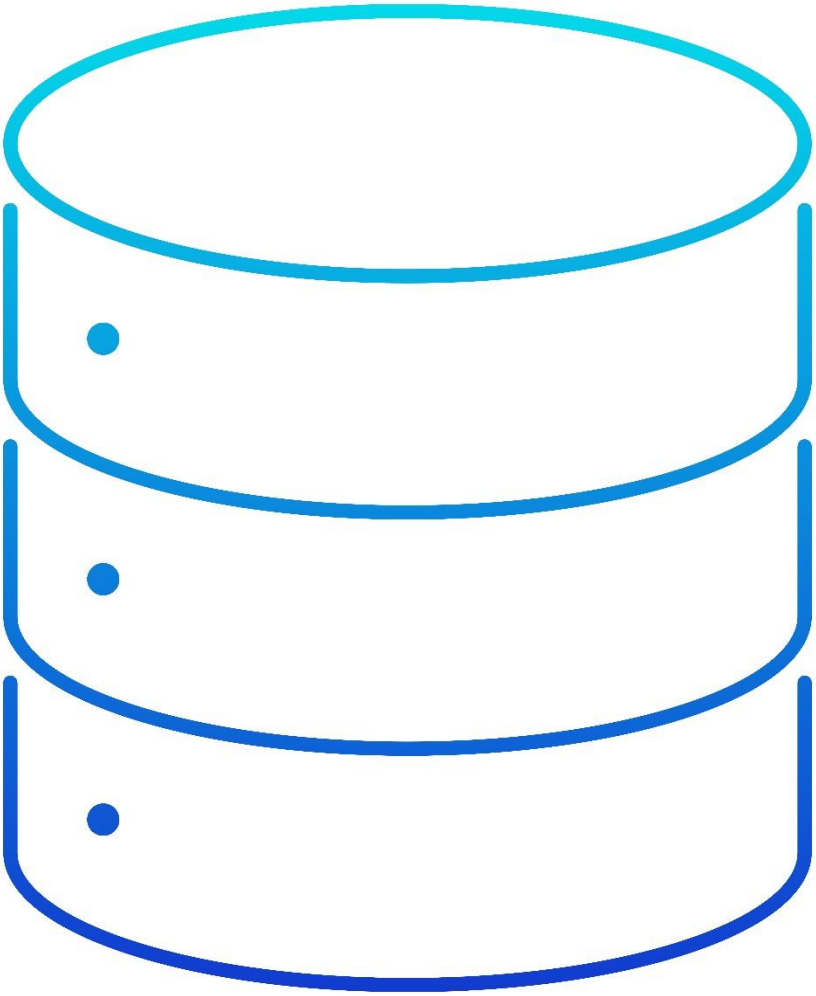
I. Bảng phân công công việc:

Họ tên	MSSV	Câu thực hiện	Tiến độ
Nguyễn Anh Tường	22120412	Sysinfo	100%
Nguyễn Đình Trí	22120384	GDB	
Vũ Hoàng Nhật Trường	22120398	Tracing	100%

II. Đánh giá tổng thể:

100%

Câu 1: *GDB*



I. Cách sử dụng GDB để debug:

Trong terminal, nhập **make qemu-gdb**. Điều này sẽ chạy QEMU và kích hoạt gỡ lỗi. Ở đây, cổng là 25000. Sau đó mở một terminal khác và chạy **gdb-multiarch -x .gdbinit**. Thao tác này sẽ chạy các lệnh trong .gdbinit, cho phép gỡ lỗi từ xa và đặt Arch thành riscv64.

II. Giải thích cách thức hoạt động và trả lời các câu hỏi:

1/Giải thích một số câu lệnh:

1. (gdb) b syscall

```
(gdb) b syscall
Breakpoint 1 at 0x80001cf0: file kernel/syscall.c, line 146.
```

- **Ý nghĩa**: Đặt một điểm dừng (breakpoint) tại hàm syscall.
- **Chi tiết:**
  - GDB sẽ dừng thực thi chương trình ngay khi hàm syscall được gọi.
  - "Breakpoint 1 at 0x80001cf0" cho biết đây là điểm dừng số 1 và địa chỉ bộ nhớ của hàm syscall là 0x80001cf0.
  - Thông tin "file kernel/syscall.c, line 146" chỉ rõ rằng hàm syscall được định nghĩa tại dòng 146 của tệp kernel/syscall.c.
- ---

2. (gdb) c

```
(gdb) c
Continuing.
[Switching to Thread 1.3]

Thread 3 hit Breakpoint 1, syscall () at kernel/syscall.c:146
146     {
```

- **Ý nghĩa**: Tiếp tục thực thi chương trình (continue).
- **Chi tiết**:
  - Sau khi đặt điểm dừng, chương trình đang tạm dừng. Lệnh c yêu cầu GDB tiếp tục thực thi cho đến khi gặp điểm dừng tiếp theo hoặc chương trình kết thúc.
  - Trong trường hợp này, chương trình tiếp tục chạy cho đến khi chạm tới điểm dừng tại hàm syscall.

Thông báo: [Switching to Thread 1.3]

- **Ý nghĩa**: Chuyển ngữ cảnh sang luồng (thread) 1.2.
- **Chi tiết**:
  - Khi chương trình đang chạy đa luồng, GDB chuyển sang luồng cụ thể có liên quan đến điểm dừng.
  - Luồng 1.2 là luồng kích hoạt điểm dừng.

Thông báo: Thread 3 hit Breakpoint 1, syscall () at kernel/syscall.c:146

- **Ý nghĩa**: Thông báo rằng luồng 2 đã chạm vào điểm dừng.
- **Chi tiết**:
    - "Thread 2" là luồng đã thực thi đến điểm dừng.
    - "Breakpoint 1" cho biết đây là điểm dừng số 1 đã được đặt trước đó.
    - "syscall () at kernel/syscall.c:146" cho biết chương trình đã dừng tại hàm syscall ở dòng 146 trong tệp kernel/syscall.c.

3. (gdb) layout src

- **Ý nghĩa**: Hiển thị giao diện dạng văn bản của mã nguồn.
- **Chi tiết**:
    - Lệnh này mở một bố cục (layout) hiển thị mã nguồn trực tiếp trong cửa sổ GDB.
    - Giúp bạn xem mã nguồn ngay trong GDB để dễ dàng theo dõi.

4. (gdb) backtrace

- **Ý nghĩa**: Hiển thị ngăn xếp lệnh (call stack).
- **Chi tiết**:
    - Call stack liệt kê các hàm đang được gọi, từ hàm hiện tại đến các hàm gọi trước đó.
    - Thông tin này giúp bạn xác định dòng chảy của chương trình dẫn đến điểm dừng.
    - Mỗi dòng trong backtrace cho biết hàm nào được gọi, các tham số được truyền vào, và dòng mã nơi cuộc gọi xảy ra.

III. Trả lời câu hỏi và giải thích

1. Looking at the backtrace output, which function called syscall?

Syscall được gọi bởi usertrap(), được thể hiện trong output của backtrace.

2. What is the value of p->trapframe->a7 and what does that value represent?

Ở đây, ta sẽ sử dụng 3 câu lệnh:

Lệnh 1: (gdb) n → thực hiện câu lệnh hiện tại và di chuyển đến dòng lệnh tiếp theo

Lệnh 2: (gdb) n

Lệnh 3: (gdb) p p->trapframe->a7 → lệnh in giá trị

```
kernel/syscall.c
    148   struct proc *p = myproc();
    149
 >  150   num = p->trapframe->a7;
    151   //num=* ( int *) 0;
    152   if(num > 0 && num < NELEM(syscalls) && syscalls[num])
    153   {
remote Thread 1.3 (src) In: syscall                    L150  PC: 0x80001d04
(gdb) n
(gdb) n
(gdb) p p->trapframe->a7
$1 = 7
```

Vậy giá trị của **p->trapframe->a7** là 7.

Mở file **user/initCode.S,** ta thấy number syscall được lưu trong thanh ghi a7.

initcode.S

```
user > initcode.S
  1  # Initial process that execs /init.
  2  # This code runs in user space.
  3
  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
 13
 14  # for(;;) exit();
 15  exit:
 16          li a7, SYS_exit
 17          ecall
 18          jal exit
 19
 20  # char init[] = "/init\0";
 21  init:
 22    .string "/init\0"
```

Mở file **kernal/syscall.h**. Số 7 đại dại cho **SYS_exec**.

syscall.h

```
kernel > C syscall.h > SYS_open
  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
  9  #define SYS_fstat   8
 10  #define SYS_chdir   9
 11  #define SYS_dup    10
 12  #define SYS_getpid 11
 13  #define SYS_sbrk   12
 14  #define SYS_sleep  13
 15  #define SYS_uptime 14
 16  #define SYS_open   15
 17  #define SYS_write  16
```

3. What was the previous mode that the CPU was in?

Lệnh: p/t $sstatus

```
(gdb) p/t $sstatus
$2 = 100000000000000000000000000100010
```

SPP (Supervisor Previous Privilege):

Theo tài liệu kiến trúc RISC-V, **bit thứ 5 (SPP)** từ phải sang là bit quyết định **privilege mode** trước khi trap xảy ra:

- SPP = 0: **User Mode** trước trap.
- SPP = 1: **Supervisor Mode** trước trap.

Ở đây SPP = 0, nên previous mode là **User mode.**

4. Write down the assembly instruction the kernel is panicing at. Which register corresponds to the varialable num?

Thay thế lệnh **num = p->trapframe->a7;** bởi **num = * (int *) 0;**

```
144  void
145  syscall(void)
146  {
147    int num;
148    struct proc *p = myproc();
149
150    //num = p->trapframe->a7;
151    num=* ( int *) 0;
152    if(num > 0 && num < NELEM(syscalls) && syscalls[num])
153    {
154      // Use num to lookup the system call function for num, call it,
155      // and store its return value in p->trapframe->a0
156      p->trapframe->a0 = syscalls[num]();
```

Sau khi thực hiện lệnh **make qemu**:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
scause=0xd sepc=0x80001d02 stval=0x0
panic: kerneltrap
```

Ở đây, **sepc** đề cập đến địa chỉ của mã trong kernel nơi xảy ra sự cố.

Sepc= 0x80001d02, tôi tìm kiếm địa chỉ **80001d02** trong file **kernel/kernel.asm.**

```
4384    //num = p->trapframe->a7;
4385    num=* ( int *) 0;
4386      80001d02: 00002903          lw  s2,0(zero) # 0 <_entry-0x80000000>
```

→ Câu lệnh lỗi : **lw s2,0(zero),** thanh ghi tương ứng với num là : **s2**

5. Why does the kernel crash?

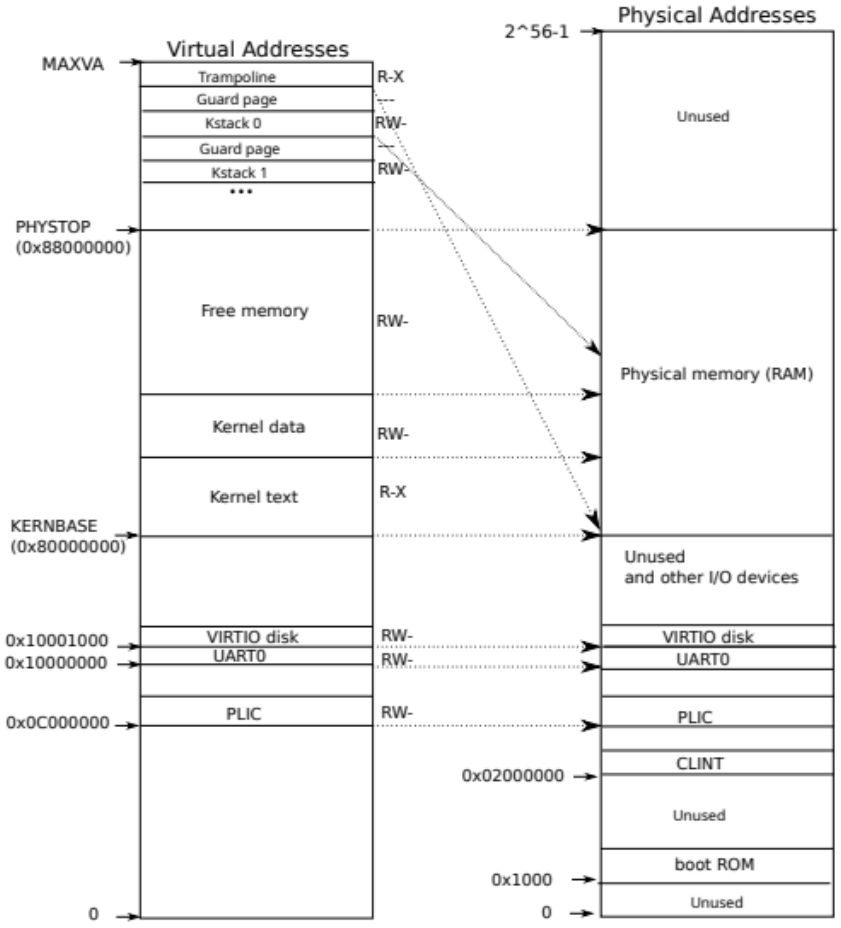


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

Để kiểm tra chúng ta gặp lỗi gì, thực hiện lệnh sau:

```
(gdb) p $scause
$1 = 13
```

In giá trị của thanh ghi **scause**, thanh ghi này lưu nguyên nhân gây ra **trap** (ngoại lệ hoặc ngắt) trên kiến trúc RISC-V.

Trên kiến trúc **RISC-V**, giá trị của scause cho biết nguyên nhân gây trap:

- **13** tương ứng với **Load Page Fault**.

Điều này có nghĩa là:

Kernel đã thực hiện một **lệnh load (đọc dữ liệu)** từ bộ nhớ. Lỗi đó xảy ra

trong khi tải dữ liệu từ địa chỉ bộ nhớ 0 vào thanh ghi s2, nhưng trong sách giáo trình, địa chỉ 0 không ánh xạ vào không gian kernel.

→Trap (ngoại lệ) xảy ra, dẫn đến lỗi.

6. What is the name of the binary that was running when the kernel paniced? What is its process id (pid)?

```
(gdb) n
(gdb) n
(gdb) p p->name
$1 = "initcode\000\000\000\000\000\000\000"
```

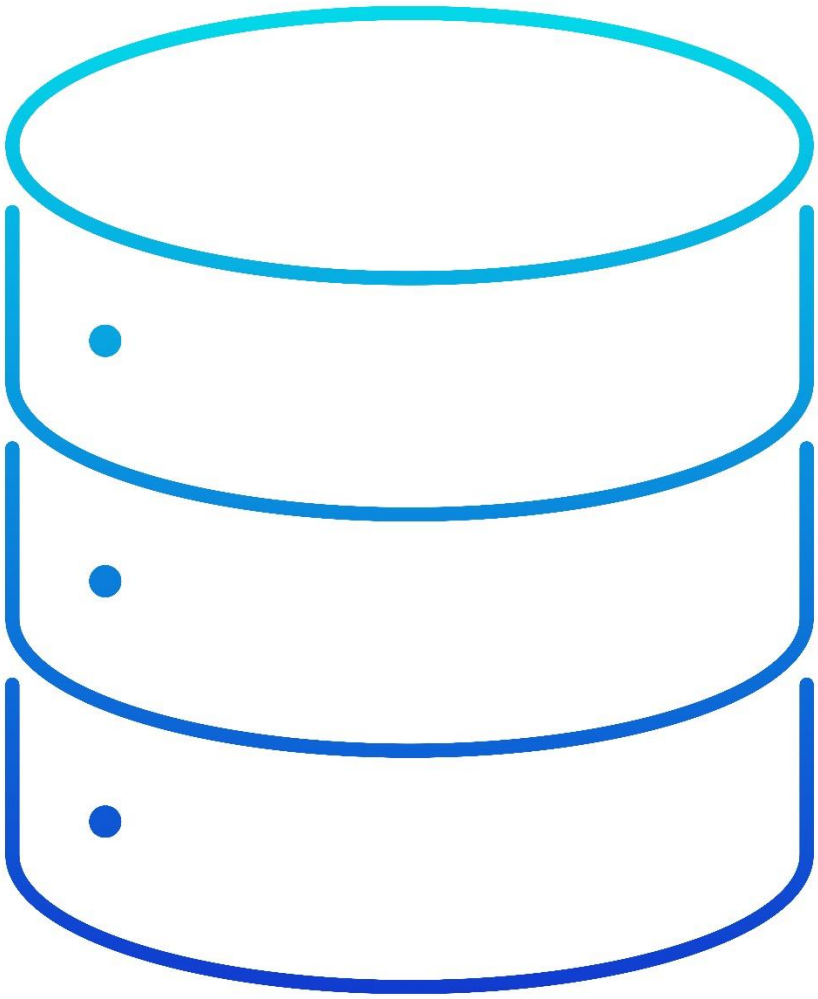
→ Tên của tệp nhị phân đang chạy khi kernal gặp lỗi: **initcode**

Để xem **pid** của tiến trình, ta thực hiện lệnh **(gdb) p *p** để in ra chi tiết nội dung tiến trình trên, chính là tiến trình mà con trỏ **p** đang trỏ đến.

```
(gdb) p *p
$2 = {lock = {locked = 0, name = 0x800071b8 "proc", cpu = 0x0}, state = RUNNING, chan = 0x0, killed = 0, xstate = 0, pid = 1, parent = 0x0,
  kstack = 274877894656, sz = 4096, pagetable = 0x87f55000, trapframe = 0x87f56000, context = {ra = 2147488508, sp = 274877898368,
    s0 = 274877898416, s1 = 2147526800, s2 = 2147525728, s3 = 1, s4 = 2147551512, s5 = 3, s6 = 2147596080, s7 = 1, s8 = 2147596376, s9 = 4,
    s10 = 0, s11 = 0}, ofile = {0x0 <repeats 16 times>}, cwd = 0x80018ba0 <itable+24>, name = "initcode\000\000\000\000\000\000\000",
```

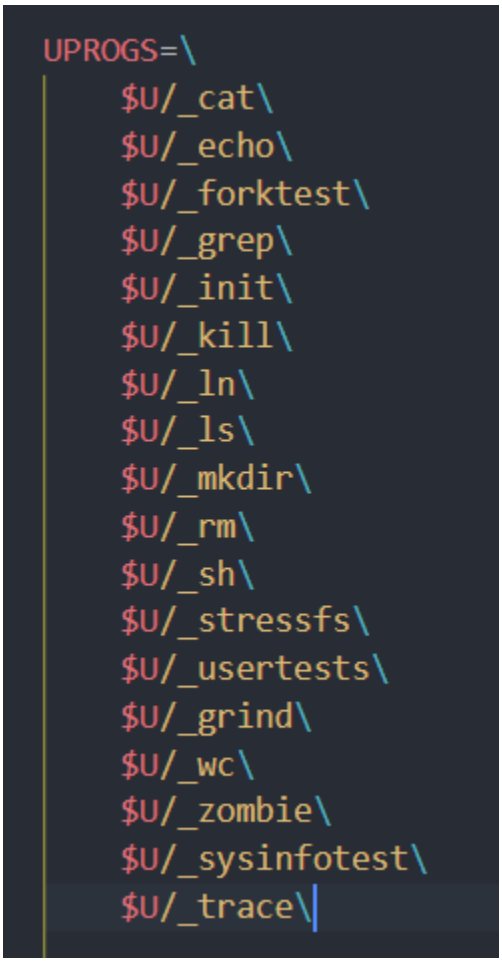
→ **pid = 1**

Câu 2: *Tracing.*



Phần khai báo:

1. **Makefile:**



Mục đích: khai báo dùng để biên dịch file trace.c trên make.

2. **user/user.h:**

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

Mục đích: khai báo cấu trúc của systemcall **trace**, với tham số **mask** là kiểu số nguyên.

3. **user/usys.pl:**

```
entry("trace");
```

Mục đích: Tạo liên kết trong file usys.S cho systemcall tương ứng trace (ánh xạ hàm trace trong user -> sys_trace trong kernel).

Ghi chú: usys.S là file đóng vai trò là lớp liên kết kernel và user mode. Giúp chuyển lời gọi từ user_mode sang kernel_mode. Nói cách khác thì đây là file dùng để giao tiếp với kernel.

4. **kernel/syscall.h**

```
#define SYS_trace  22
```

Mục đích: Tạo một ID cho systemcall trace với một mã định danh là 22.

Ghi chú: ID này dùng để ánh xạ hàm đến hàm xử lý tương ứng của nó trong kernel thông qua một hàm ánh xạ trong file syscall.c

5. **kernel/syscall.c:**

```
extern uint64 sys_trace(void);
```

Mục đích: khai báo một hàm có tên là sys_trace(void) đã được định nghĩa ở một nơi khác (thông qua từ khóa extern).

```
[SYS_trace]   sys_trace,
```

Mục đích: Thêm một khai báo ánh xạ ID trong hàm (*syscalls[])(void) đã được định nghĩa trong syscall.h (ID = 22).

```
char* syscallnames[] = {
  "",
  "fork","exit","wait","pipe","read",
  "kill","exec","fstat","chdir","dup",
  "getpid","sbrk","sleep","uptime","open",
  "write","mknod","unlink","link","mkdir",
  "close","trace","sysinfo",
};
```

Mục đích: Ánh xạ số ID của systemcall với tên của systemcall ở dạng string. Mảng được dùng để kernel in tên của systemcall cần theo dõi hoặc ghi log, dùng để truy vết hoạt động của các hàm.

Ví dụ: như đang thực thi thì kernel sẽ dùng mảng này để ghi tên systemcall ra log.

6. **kernel/proc.h**

```
int trace_mask;
```

Mục đích: Tạo 1 biến **trace_mask** trong struct **proc** để lưu giá trị số nguyên của mask.

Phần cài đặt:

1. **kernel/sysproc.c**

```
uint64
sys_trace(void)
{
  int mask;

  // Lấy tham số từ user space
  argint(0, &mask);

  // Lưu giá trị mask vào biến trace_mask của tiến trình hiện tại
  struct proc *p = myproc();
  p->trace_mask = mask;

  return 0; // Trả về thành công
}
```

Mục đích: Lấy giá trị mask trong tầng user và lưu vào trong biến **trace_mask** của struct **proc** trong tầng kernel. Hàm trả về 0 nếu thành công.

2. **kernel/proc.c**

```
fork(void)

  // Copy trace_mask from parent to child.
  np->trace_mask = p->trace_mask;
```

Mục đích: Khi 1 tiến trình **fork()** để tạo ra tiến trình con, tiến trình con đó cũng có nhu cầu được biết giá trị mask để trace, nên ta sẽ phải sao chép **trace_mask** của cha vào con (Vì ta không thể chắc chắn rằng khi trace một process, process đó có gọi **fork()** để sinh ra process con hay không, do đó ta phải liệt kê hết).

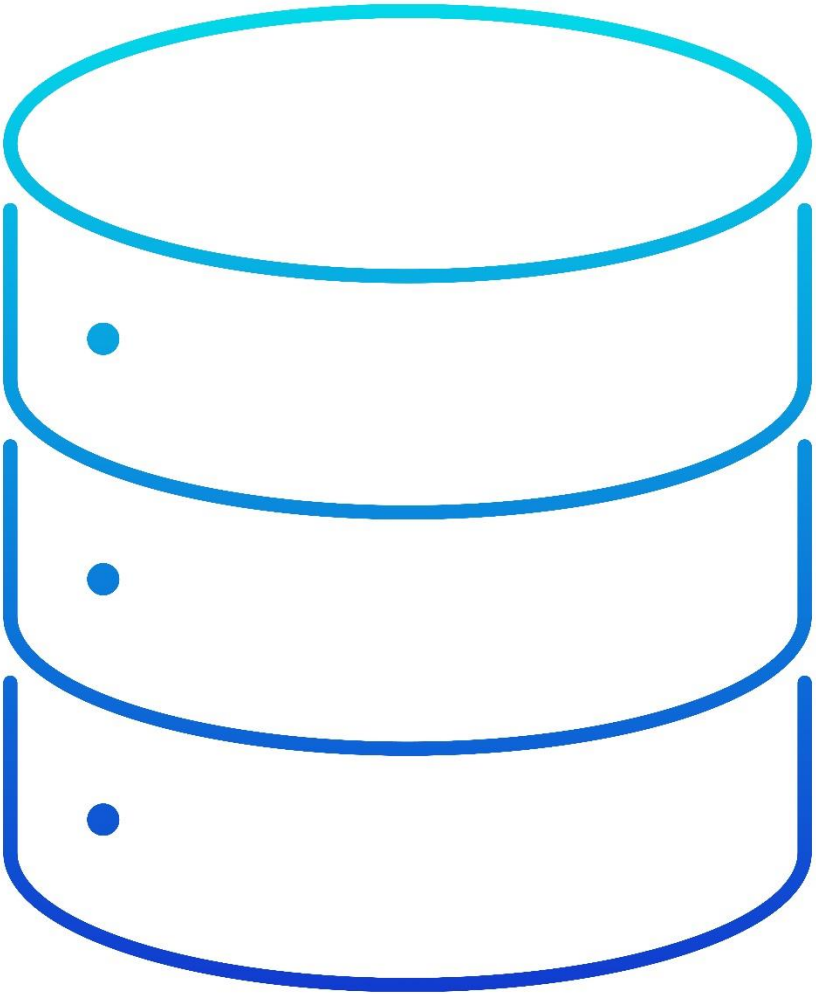
3. **kernel/syscall.c**

```
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->trace_mask & (1 << num))
    {  // Kiểm tra bit tương ứng trong trace_mask
      printf("%d: syscall %s -> %ld\n", p->pid, syscallnames[num], p->trapframe->a0);
    }
  }
```

Mục đích: Hiệu chỉnh hàm syscall để trả ra thông tin trace được, gồm **process ID**, **tên của syscall** và **giá trị trả ra của syscall** đó (trong thanh ghi a0).

Lưu ý: Ở đây ta so sánh **p->trace_mask & (1 << num)**, với **num** là số thứ tự của syscall đã được khai báo trong file syscall.h. Ta sẽ kiểm tra **bit thứ num** từ **phải sang trái** của **trace_mask** có được bật hay không, bằng cách **bitwise trace_mask** với **(1<<num)** (dịch chuyển số 1 trong hệ nhị phân 4 bytes sang trái **num** lần), nếu bật thì xuất ra thông tin, nếu tắt thì không cần xuất ra.

Câu 3: *Sysinfo.*



I. Phần khai báo:

1. Makefile:

```
UPROGS=\
...
  $U/_sysinfotest\
```

Mục đích: khai báo dùng để biên dịch file sysinfotest.c trên make.

2. User/User.h:

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

Mục đích: khai báo cấu trúc của systemcall có ý nghĩa là sysinfo.

3. User/Usys.pl:

```
entry("sysinfo");
```

Mục đích: Tạo liên kết trong file usys.S cho systemcall tương ứng sysinfo.

Ghi chú: usys.S là file đóng vai trò là lớp liên kết kernel và user mode. Giúp chuyển lời gọi từ user_mode sang kernel_mode. Nói cách khác thì đây là file dùng để giao tiếp với kernel.

4. Kernel/Syscall.h

```
#define SYS_sysinfo 23
```

Mục đích: Tạo một ID cho systemcall sysinfo với một mã định danh là 23.

Ghi chú: ID này dùng để ánh xạ hàm (ví dụ: sysinfo) đến hàm xử lý tương ứng của nó trong kernel thông qua một hàm ánh xạ trong file syscall.c

5. **Kernel/Syscall.c:**

```
extern uint64 sys_sysinfo(void);
```

Mục đích: khai báo một hàm có tên là sys_sysinfo(void) đã được định nghĩa ở một nơi khác (thông qua từ khóa extern).

```
[SYS_sysinfo] sys_sysinfo,
```

Mục đích: Thêm một khai báo ánh xạ ID trong hàm (*syscalls[])(void) đã được định nghĩa trong syscall.h (ID = 23).

```
char* syscallnames[] = {
  "",
  "fork","exit","wait","pipe","read",
  "kill","exec","fstat","chdir","dup",
  "getpid","sbrk","sleep","uptime","open",
  "write","mknod","unlink","link","mkdir",
  "close","sysinfo",
};
```

Mục đích: Ánh xạ số ID của systemcall với tên của systemcall ở dạng string. Mảng được dùng để kernel in tên của systemcall cần theo dõi hoặc ghi log, dùng để truy vết hoạt động của các hàm.

Ví dụ: như đang thực thi thì kernel sẽ dùng mảng này để ghi tên systemcall ra log.

7. **Kernel/defs.h:**

```
uint64 nproc(void);
uint64 nfree(void);
```

Mục đích: Khai báo hàm tính num of process và free memory.

Ghi chú: defs.h là file khai báo các hàm, macro và các cấu trúc dữ liệu được sử dụng trên toàn bộ hệ thống. Giúp liên kết các khai báo giữa các file trong kernel.

8. Kernel/sysproc.c:

```
#include "sysinfo.h"
```

Mục đích: Khai báo file header sysinfo vì trong file này có một số định nghĩa quan trọng.

Ví dụ: struct sysinfo, hàm systemcall int sysinfo(struct sysinfo *info), …

II. Phần cài đặt:

1. Kernel/Proc.c:

```
uint64
nproc(void){
  int procCount = 0;
  struct proc* p;
  for (p=proc; p<&proc[NPROC]; p++){
    if (p->state != UNUSED){
      procCount++;
    }
  }
  return procCount;
}
```

- **proc:** Là một mảng các cấu trúc tiến trình (struct proc) lưu trữ thông tin về tất cả các tiến trình trong hệ thống.
- **NPROC:** Là số lượng tối đa các tiến trình mà hệ điều hành có thể quản lý cùng một lúc.

Vì trong XV6, mỗi tiến trình được quản lý bởi một phần tử trong mảng proc.

- **Trạng thái UNUSED:**
  - Mỗi tiến trình trong hệ thống có một thuộc tính state để biểu thị trạng thái hiện tại (ví dụ: RUNNING, SLEEPING, UNUSED,...).
  - UNUSED là trạng thái của các slot tiến trình chưa được sử dụng (chưa được cấp phát cho bất kỳ tiến trình nào).
  - Chỉ những tiến trình không ở trạng thái UNUSED mới được tính vào procCount.

Hoạt động:

- Duyệt qua từng phần tử của mảng proc.
- Kiểm tra trạng thái của mỗi tiến trình.
- Tăng biến đếm nếu tiến trình không ở trạng thái UNUSED.

Kết quả:

- Trả về tổng số lượng tiến trình đang tồn tại hoặc hoạt động.

Ý nghĩa:

- Hỗ trợ giám sát hệ thống.
- Tích hợp trong system call hoặc dùng để log trạng thái kernel.

2. **Kernel/kalloc.c:**

```
uint64
nfree(void){
  struct run* r;
  int byteCount = 0;
  acquire(&kmem.lock);
  r = kmem.freelist;
  while(r){
    r = r->next;
    byteCount++;
  }
  release(&kmem.lock);
  return byteCount * PGSIZE;
}
```

Cách làm: đếm số lượng các khối bộ nhớ tự do trong danh sách kmem.freelist.

Hoạt động:

- Duyệt danh sách liên kết để đếm số khối bộ nhớ tự do.
- Tính tổng dung lượng trống bằng cách nhân số khối với kích thước của một trang (PGSIZE).
- Trả về kết quả dưới dạng số byte.

Ý nghĩa:

- Cung cấp thông tin về trạng thái bộ nhớ hệ thống.
- Được sử dụng trong các chức năng giám sát và báo cáo trạng thái kernel.

3. Kernel/sysproc.h:

```
uint64
sys_sysinfo(void)
{
  uint64 addr;
  struct sysinfo info;
  struct proc* p = myproc();
  argaddr(0, &addr);
  info.freemem = nfree();
  info.nproc = nproc();

  if(copyout(p->pagetable, addr, (char*)&info, sizeof(info)) < 0)
    return -1;
  return 0;
}
```

Cách hoạt động:

- Nhận địa chỉ từ chương trình người dùng.
- Thu thập thông tin trong kernel.
- Sao chép kết quả ra không gian người dùng.

Kết quả:

- Trả về 0 nếu thành công, -1 nếu xảy ra lỗi.

Mục đích:

- Cung cấp thông tin về trạng thái hệ thống (bộ nhớ trống và số tiến trình) cho chương trình người dùng.