TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG-HCM

KHOA CÔNG NGHỆ THÔNG TIN

–o–

fit@hcmus

BÁO CÁO ĐỒ ÁN

LAB 1: SYSTEM CALL

Giảng viên: ThS. Phạm Tuấn Sơn

Môn học: Hệ Điều Hành

Sinh viên: 23120255 - Lê Tấn Hiệp

23120262 - Tống Dương Thái Hoà

23120264 - Nguyễn Phúc Hoàng

Lớp 23/4

Thành phố Hồ Chí Minh

MỤC LỤC

1. Thông tin sinh viên và mức độ hoàn thiện

1.1. Thông tin sinh viên

STT	Họ và tên	MSSV	EMAIL
1	Lê Tấn Hiệp	23120255	23120255@student.hcmus.edu.vn
2	Tống Dương Thái Hoà	23120262	23120262@student.hcmus.edu.vn
3	Nguyễn Phúc Hoàng	23120264	23120264@student.hcmus.edu.vn

1.2. Mức độ hoàn thiện

STT	Yêu cầu	Điểm	Tỉ lệ hoàn thành
1	Tracing	5	100%
2	Sysinfo	5	100%

1.3. Tự đánh giá

- Mức độ hoàn thành: **2**/2 yêu cầu (chiếm 100%).
- Số điểm tự đánh giá: **10**/10 **điểm**.

2. Giới thiệu

2.1. Trace: Theo dõi System Call của tiến trình

trace là một system call mới được thêm vào kernel xv6, cho phép người dùng giám sát các system call khác được thực thi bởi một tiến trình và các tiến trình con của nó. Khi được kích hoạt, trace sẽ in ra thông tin chi tiết cho mỗi system call được theo dõi, bao gồm:

- **PID (Process ID):** Mã định danh của tiến trình thực hiện system call.
- **Tên system call:** Tên của system call (ví dụ: read, fork, exec).
- **Giá trị trả về:** Kết quả trả về của system call đó.

Chức năng này cực kỳ hữu ích cho việc gỡ lỗi (debug), phân tích hành vi của một chương trình, và kiểm tra hoạt động của hệ thống ở mức độ thấp.

2.2. Sysinfo: Lấy thông tin hệ thống

sysinfo là một system call cung cấp một giao diện cho các chương trình ở không gian người dùng (user-space) để truy vấn các thông tin quan trọng về trạng thái hiện tại của kernel. Cụ thể, sysinfo trả về các thông số:

- **Bộ nhớ trống (Free memory):** Tổng số byte bộ nhớ vật lý chưa được cấp phát.
- **Số tiến trình đang chạy (Running processes):** Tổng số tiến trình đang tồn tại trong hệ thống (không ở trạng thái UNUSED).
- **Số file đang mở (Open files):** Tổng số file descriptor đang được sử dụng trên toàn hệ thống.

Chức năng này giúp người dùng và các nhà phát triển giám sát tài nguyên hệ thống, tối ưu hóa hiệu suất và chẩn đoán các vấn đề liên quan đến tài nguyên.

3. Trace

3.1. Khái niệm

trace là một công cụ gỡ lỗi mạnh mẽ. Nó hoạt động bằng cách sử dụng một bitmask. Mỗi bit trong mask tương ứng với một số hiệu system call. Nếu bit thứ i được bật, system call có số hiệu i sẽ được theo dõi. Khi một tiến trình đã bật chế độ trace gọi một system call nằm trong mask, kernel sẽ in một dòng log ra console trước khi trả quyền điều khiển về cho tiến trình.

3.2. Chi tiết bước cài đặt

Quá trình cài đặt trace bao gồm các thay đổi ở cả user-space và kernel-space.

Bước 1: Định nghĩa System Call

- Định nghĩa số hiệu cho system call mới
- Thêm trace vào danh sách để script tự động tạo mã assembly stub cho lời gọi từ user-space.

Bước 2: Mở rộng cấu trúc tiến trình

- Thêm một trường trace_mask vào struct proc để lưu trữ bitmask theo dõi của mỗi tiến trình.

Bước 3: Cài đặt hàm xử lý System Call

- Viết hàm sys_trace(). Hàm này nhận một tham số nguyên (mask) từ người dùng và gán nó vào trường trace_mask của tiến trình hiện tại (myproc()).

Bước 4: Kế thừa trace_mask cho tiến trình con

- Trong hàm kfork(), đảm bảo rằng tiến trình con được tạo ra sẽ kế thừa trace_mask từ tiến trình cha. Điều này cho phép trace có thể theo dõi cả các tiến trình con.

Bước 5: Tích hợp logic theo dõi vào syscall()

- Tạo một mảng syscall_names[] để ánh xạ số hiệu system call sang tên chuỗi tương ứng.
- Sửa đổi hàm syscall(): Sau khi một system call được thực thi, hàm sẽ kiểm tra trace_mask của tiến trình hiện tại. Nếu bit tương ứng với system call vừa gọi được bật, nó sẽ in thông tin (PID, tên syscall, giá trị trả về) ra màn hình.

Bước 6: Tạo chương trình người dùng trace

- Thêm _trace vào danh sách UPROGS để biên dịch.
- Chương trình user/trace.c (không có trong patch nhưng được ngụ ý) sẽ nhận các đối số dòng lệnh, gọi trace(mask) để thiết lập mask, sau đó dùng exec() để thực thi chương trình cần theo dõi.

3.3. Ví dụ

Để theo dõi system call read (có số hiệu là 5, tương ứng với bitmask 1 << 5 = 32) của lệnh grep, ta chạy lệnh sau:

$ trace 32 grep hello README

Kết quả đầu ra:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 985
3: syscall read -> 465
3: syscall read -> 0
$
```

Phân tích lệnh:

- trace: Chương trình người dùng.
- 32: Bitmask, cho phép theo dõi system call read.
- grep hello README: Lệnh và các tham số sẽ được thực thi và theo dõi.

Phân tích kết quả:

- 3: PID của tiến trình grep.
- syscall read: Tên system call được theo dõi.
- -> 1023: Giá trị trả về. Trong trường hợp của read, đây là số byte đã đọc được. Giá trị 0 báo hiệu kết thúc file.
- kernel sẽ đọc dữ liệu từ ổ cứng theo các block (1 block là 1024 bytes được định nghĩa trong kernel/fs.h).
- grep là lệnh đọc theo buffer tối đa 1024 bytes (được viết trong file user/grep.c). Cụ thể, grep sẽ gọi read(fd, buf, 1024 - 1) (1024 - 1 ở đây là max, khi không còn buffer dư của lần đọc trước hoặc là đọc lần đầu tới 1023 kí tự. Đọc 1023 kí tự vì kí tự cuối cùng để báo hiệu kết thúc mảng char '\0').
- Trong grep.c, vòng lặp "while((n = read(fd, buf+m, sizeof(buf)-m-1)) > 0)" sẽ khiến đọc liên tục đến khi EOF (hoặc sẽ kết thúc ở lượt sau khi gặp 1 hàng có nhiều hơn hoặc bằng 1023 ký tự mà trong 1023 ký tự đầu không có ký tự \n, dẫn tới sau khi đọc hàng đó sẽ trả về n = 0, sai điều kiện dẫn đến dừng vòng lặp), nên có nhiều lần read, vậy nên có nhiều dòng. Dẫn đến không in được

pattern nếu không nhận được ký tự ký tự \n để vào đọc buffer và in pattern đó ra màn hình.

- kernel tìm block tương ứng trên disk, đọc 1024 bytes vào buffer cache (block size cố định).
- kernel xác định file còn bao nhiêu byte tại vị trí hiện tại.
- Phần trống trong block (tức là nếu số bytes dữ liệu hợp lệ không bằng 1024 bytes) không được gửi lên user.
- kernel chỉ copy đúng số bytes file có (như trong hình là 1023, 985, 465) vào buffer user-space.
- kernel trả về đúng số byte đó. Nếu hết file thì read() trả về 0.
- Vậy tại sao lại là 1023, 985, 465 mà không phải là 1023, 1023, x?
- Bởi vì mấu chốt nằm ở logic xử lý dòng và quản lý bộ đệm thông minh của grep. Vòng lặp chính trong grep.c: while((n = read(fd, buf+m, sizeof(buf)-m-1)) > 0):
- Lần đọc 1: read -> 1023
    + Lần đầu tiên, buffer trống nên biến m (số byte sót lại) bằng 0.
    + grep yêu cầu đọc 1024 - 0 - 1 = 1023 bytes.
    + Kernel đọc và trả về 1023 bytes, lấp đầy buffer.
- Xử lý và "Dồn" buffer:
    + grep quét 1023 bytes này để tìm các dòng hoàn chỉnh (kết thúc bằng \n).
    + Sau khi xử lý hết các dòng hoàn chỉnh, giả sử còn lại 38 bytes trong buffer (đây là phần đầu của một dòng chưa kết thúc).
    + grep sẽ dùng memmove để dồn 38 bytes này lên đầu buffer.
    + Biến m bây giờ có giá trị là 38.
- Lần đọc 2: read -> 985
    + Chương trình quay lại vòng lặp.
    + Buffer giờ không trống, nó đang chứa 38 bytes.
    + grep yêu cầu đọc 1024 - 38 - 1 = 985 bytes để lấp đầy phần còn lại.
    + Kernel đọc và trả về 985 bytes.
    + Dữ liệu mới được đặt ngay sau 38 bytes cũ, tạo thành một khối liền mạch để xử lý.
- Lần đọc 3: read -> 465
    + Quá trình lặp lại.
    + Giả sử sau lần xử lý thứ hai, còn lại một "mẩu" 558 bytes.
    + grep yêu cầu đọc 1024 - 558 - 1 = 465 bytes.

    + Kernel đọc và trả về 465 bytes.
- Lần đọc cuối: read -> 0
    + Sau khi đọc 465 bytes và xử lý, grep cố đọc tiếp nhưng file đã hết.
    + read trả về 0, vòng lặp kết thúc.
- Cuối cùng, kích thước file README đúng bằng những gì grep đọc được: 1023 + 985 + 465 = 2473 bytes, gần bằng 2.4KB. Vậy nên chính xác.

4. Sysinfo

4.1. Khái niệm

sysinfo là system call cung cấp một cơ chế tập trung để lấy thông tin thống kê về hệ thống. Thay vì tạo nhiều system call riêng lẻ (ví dụ: getfreemem(), getprocnum()), sysinfo trả về một cấu trúc dữ liệu (struct sysinfo) chứa nhiều thông tin cùng lúc, giúp giảm thiểu chi phí chuyển đổi ngữ cảnh giữa user-space và kernel-space.

4.2. Chi tiết bước cài đặt

Quá trình cài đặt sysinfo bao gồm các thay đổi ở cả user-space và kernel-space.

Bước 1: Định nghĩa cấu trúc sysinfo và System Call

- Định nghĩa cấu trúc sysinfo.
- Định nghĩa số hiệu cho system call mới:
- Thêm sysinfo vào danh sách user.

Bước 2: Cài đặt các hàm trợ giúp trong Kernel

Để thu thập thông tin, ba hàm trợ giúp đã được viết:

1. **kfreemem()**
    + **Mục đích**: Đếm tổng số byte bộ nhớ còn trống.
    + **Cách hoạt động**: Duyệt qua danh sách liên kết các trang bộ nhớ trống (kmem.freelist). Với mỗi nút trong danh sách (đại diện cho một trang), hàm cộng PGSIZE (kích thước của một trang, thường là 4096 bytes) vào tổng số.
    + **Đồng bộ hóa**: Hàm sử dụng acquire(&kmem.lock) và release(&kmem.lock) để đảm bảo an toàn khi truy cập kmem.freelist từ nhiều CPU cùng lúc.
2. **count_process()**
    + **Mục đích**: Đếm số lượng tiến trình đang hoạt động.

+ **Cách hoạt động**: Duyệt qua mảng các tiến trình proc[]. Với mỗi tiến trình, hàm kiểm tra xem trạng thái của nó có phải là UNUSED hay không. Nếu không, bộ đếm sẽ tăng lên.
+ **Đồng bộ hóa**: Hàm khóa từng tiến trình (acquire(&p->lock)) trước khi kiểm tra trạng thái và nhả khóa (release(&p->lock)) ngay sau đó để tránh race condition và deadlock.

3. **count_files()**
   + **Mục đích**: Đếm tổng số file đang được mở trên toàn hệ thống.
   + **Cách hoạt động**: Duyệt qua bảng file toàn cục (ftable.file[]). Một file được coi là đang mở nếu trường ref (tham chiếu) của nó lớn hơn 0.
   + **Đồng bộ hóa**: Hàm sử dụng acquire(&ftable.lock) và release(&ftable.lock) để bảo vệ bảng file.

Bước 3: Cài đặt hàm xử lý System Call sys_sysinfo

- **kernel/sysproc.c**:
  + Hàm sys_sysinfo() nhận một tham số là địa chỉ con trỏ trong user-space.
  + Nó tạo một biến struct sysinfo cục bộ trong kernel-space.
  + Gọi các hàm trợ giúp kfreemem(), count_process(), count_files() để điền dữ liệu vào struct này.
  + Sử dụng hàm copyout() để sao chép an toàn dữ liệu từ struct sysinfo trong kernel-space ra địa chỉ con trỏ ở user-space. copyout đảm bảo rằng địa chỉ do người dùng cung cấp là hợp lệ và thuộc về không gian địa chỉ của tiến trình. Nếu sao chép thất bại, nó trả về -1.

Bước 4: Tạo chương trình kiểm thử sysinfotest

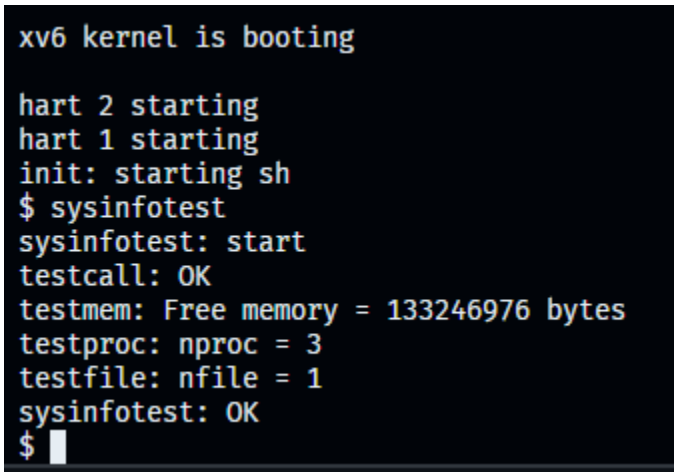
- **Makefile**: Thêm _sysinfotest vào danh sách UPROGS.
- Chương trình sysinfotest sẽ khai báo một biến struct sysinfo, gọi sysinfo() với địa chỉ của biến này, và sau đó in ra các giá trị đã nhận được.

4.3. Ví dụ

Khi chạy chương trình sysinfotest trong shell của xv6, kết quả thu được sẽ tương tự như sau:

$ sysinfotest

Kết quả đầu ra:



Phân tích lệnh:

- sysinfo: chương trình người dùng.
- Lệnh này không nhận tham số đầu vào. Lệnh chạy một loạt các bài kiểm tra con (testcall, testmem, testproc, testfile) và báo cáo kết quả.

Phân tích kết quả:

- sysinfotest: start: Thông báo cho biết chương trình sysinfotest đã bắt đầu thực thi.
- testcall: OK: Bài test con đầu tiên. Gọi system call sysinfo với một con trỏ hợp lệ để kiểm tra xem kernel có thể gọi hàm thành công mà không bị crash không. OK tức là bài test đã pass.
- testmem: Free memory = 133246976 bytes: Bài test "testmem".Cho thấy system call sysinfo đã báo cáo rằng có 133.246.976 bytes bộ nhớ vật lý trống (free memory) hiện có trong hệ thống của máy được test.
- testproc: nproc = 3: Bài test "testproc". nproc = 3 báo cáo rằng có tổng cộng 3 tiến trình (process) đang tồn tại trong hệ thống tại thời điểm gọi system call. 3 tiến trình đó là: init: Tiến trình đầu tiên được kernel khởi tạo. sh: Tiến trình shell được init fork ra, nơi đang được gõ lệnh. sysinfotest: Là tiến trình đang chạy bài test hiện tại được sh fork ra.
- testfile: nfile = 1: Bài test "testfile". nfile = 1 báo cáo rằng có 1 tệp đang được mở trong bảng tệp (file table) của toàn hệ thống. Tệp này rất có thể là tệp console (thiết bị console) mà tiến trình init đã mở khi khởi động (và được sh kế thừa) để dùng cho việc đọc (stdin) và ghi (stdout, stderr).

- sysinfotest: OK: Thông báo tổng kết, cho biết tất cả các bài test con bên trong sysinfotest đã chạy xong và đều trả về kết quả OK.

5. Cách dùng file 23120255_23120262_23120264.patch

Dùng để lưu trữ và chia sẻ các thay đổi giữa các phiên bản mã nguồn. Có thể sử dụng lệnh git apply để áp dụng tệp patch này vào kho lưu trữ, giúp tích hợp các thay đổi mà không cần sử dụng kho lưu trữ trung tâm.

Step 1 — Clone repo của pdos-mit

```
git clone git://g.csail.mit.edu/xv6-labs-2025
cd xv6-labs-2025
git checkout syscall
```

Step 2 — Copy file patch vào folder xv6-labs-2025/ (cùng level với Makefile, README.md,...)

Step 3 — Apply

```
git apply 23120255_23120262_23120264.patch
```

6. Cài đặt test tự động

Để đảm bảo chất lượng và tự động hóa kiểm thử, nhóm em đã chủ động mở rộng test bằng cách bổ sung một system call chuyên dụng. Lời gọi hệ thống này cho phép các kịch bản test ở user-space có thể can thiệp vào và kiểm chứng trực tiếp trạng thái nội bộ của kernel, đảm bảo hệ thống thực thi đúng như thiết kế.

6.1. Hướng dẫn test

Chạy lệnh lab1test trong shell của xv6, kết quả sẽ thực thi một số bài test tự động do chúng em tự thêm để có thể dễ dàng test hết 2 system call:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ lab1test
═══ START COMPREHENSIVE TEST SUITE FOR LAB 1 ═══

--- Part 1: Automated sysinfo tests ---


Test: sysinfo() with invalid pointer...
  → Check: sysinfo(NULL) should return -1
     - Expected: -1
     - Actual:   -1
     - Result:  [OK]
  → Check: sysinfo(kernel_addr) should return -1
     - Expected: -1
     - Actual:   -1
     - Result:  [OK]
... End of invalid pointer test: [SUCCESS]
```

```
Test: sysinfo.freemem with sbrk()...
  - Initial free memory: 133242880
  → Check: sbrk(PGSIZE) must succeed
     - Expected: true (non-zero)
     - Actual:   true
     - Result:  [OK]
  - Free after alloc: 133238784 (delta: -4096)
  → Check: Free memory must decrease after allocation
     - Expected: Actual value < Threshold value
     - Actual:   133238784 < 133242880
     - Result:  [OK]
  - Final free memory: 133242880 (delta from initial: 0)
  → Check: Free memory must return close to initial value
     - Expected:  Within range [133238784, 133246976]
     - Actual:   133242880
     - Result:  [OK]
... End of freemem test: [SUCCESS]
```

```
Test: sysinfo.nopenfiles with open()/close()...
  - Initial open files count: 1
  → Check: Opening README file must succeed
     - Expected: true (non-zero)
     - Actual:   true
     - Result:  [OK]
  → Check: nopenfiles must increase by 1 after open()
     - Expected: 2
     - Actual:   2
     - Result:  [OK]
  → Check: nopenfiles must return to initial value after close()
     - Expected: 1
     - Actual:   1
     - Result:  [OK]
... End of nfile test: [SUCCESS]
```

```
Test: sysinfo.nproc with fork()...
  - Initial process count: 3
  → Check: wait() returned child pid
     - Expected: 4
     - Actual:   4
     - Result:  [OK]
  → Check: fork() successfully created new process.
     - Result: [OK]
  → Check: Child process must exit with code 0 (indicating correct nproc)
     - Expected: 0
     - Actual:   0
     - Result:  [OK]
  → Check: nproc returns to initial value after child exits
     - Expected: 3
     - Actual:   3
     - Result:  [OK]
... End of nproc test: [SUCCESS]
```

```
--- Part 2: Visual trace tests ---

== Start TRACE Test (Visual Inspection) ==
Note: The following tests require you to manually check the output.

Test 1: Trace syscall 'getpid' (syscall 11).
  → YOU SHOULD SEE 1 trace line for 'getpid':
3: syscall getpid → 3

Test 2: Trace 'read' (5) & 'write' (16). DO NOT trace 'open' (15).
  → YOU SHOULD SEE 'write' and 'read', BUT NOT 'open':
3: syscall read → 10
  (Test syscall write)
3: syscall write → 23

Test 3: Trace 'fork' (1), 'exit' (2), 'wait' (3).
  → YOU SHOULD SEE 'fork', 'exit' (from child), and 'wait' (from parent):
3: syscall fork → 5
5: syscall getpid → 5
  [Child PID 5] Running and will exit...
5: syscall exit → exit
3: syscall wait → 5
  [Parent] Child process has exited.

== End TRACE Test ==

--- SUMMARY ---
Total automated tests run: 4
Total automated tests failed: 0

RESULT: ALL AUTOMATED TESTS PASSED.
(Please carefully check the trace test output above!)
$
```

6.2. Bảng test cases

Phần 1: Kiểm thử Tự động cho sysinfo (Automated sysinfo Tests)

Tên Test Case	Mục tiêu Kiểm thử	Mô tả Hoạt động	Kết quả Mong đợi
sysinfo(bad_ptr)	Kiểm tra tính an toàn (Safety)	Gọi sysinfo() với một con trỏ không hợp lệ (ví dụ: NULL hoặc trỏ ra ngoài không gian địa chỉ user).	Kernel không bị "crash". sysinfo() phải trả về lỗi (thường là -1) và test case báo [OK], chứng tỏ kernel đã xử lý được con trỏ xấu.
sysinfo.freemem (with sbrk)	Kiểm tra freemem	Kiểm tra xem sysinfo.freemem có phản ánh đúng lượng bộ nhớ trống hay không, đặc biệt là sau khi thay đổi bộ nhớ.	Test case này có thể gọi sbrk() để cấp phát thêm bộ nhớ. Lượng freemem sau khi gọi sbrk() phải giảm đi. Test báo [OK].
sysinfo.nproc (with fork)	Kiểm tra nproc	Kiểm tra xem sysinfo.nproc có đếm đúng số tiến trình đang hoạt động hay không.	Test case gọi fork() để tạo một tiến trình con. Số nproc phải tăng lên 1, sau đó giảm 1 khi tiến trình con exit(). Test báo [OK].

sysinfo.nopenfiles (with open/close)	Kiểm tra nopenfiles	Kiểm tra xem sysinfo.nopenfiles có đếm đúng tổng số file đang mở trên toàn hệ thống hay không.	Test case gọi open() một file. Số nopenfiles phải tăng 1. Sau đó gọi close(), số nopenfiles phải giảm 1. Test báo [OK].

Phần 2: Kiểm thử Trực quan cho trace (Visual trace Tests)

Tên Test Case	Mục tiêu Kiểm thử	Mô tả Hoạt động	Kết quả Trực quan Mong đợi
Test 1: Tracing getpid	Theo dõi 1 system call	Đặt trace mask chỉ để theo dõi getpid (syscall 11) và sau đó gọi getpid().	Chỉ 1 dòng trace cho getpid xuất hiện, ví dụ: 5: syscall getpid -> 5.
Test 2: Tracing read/write, NOT open	Theo dõi theo bitmask (Lọc)	Đặt trace mask để theo dõi read (5) và write (16), nhưng không theo dõi open (15). Sau đó chạy một lệnh (ví dụ: echo) mà có dùng open, read, write.	Các dòng trace cho read và write phải xuất hiện, nhưng không được có dòng trace nào cho open.
Test 3: Tracing fork/exit/wait	Tính kế thừa (Inheritance)	Đặt trace mask để theo dõi fork (1), exit (2), wait (3). Sau đó, gọi fork().	1. Dòng trace fork xuất hiện ở tiến trình cha. 2. Dòng trace exit xuất hiện từ tiến trình con (ví dụ: PID 7), chứng tỏ con đã kế thừa mask. 3. Dòng trace wait xuất hiện ở tiến trình cha.

7. Tài liệu tham khảo

[1] *Massachusetts Institute of Technology, "Lab: System calls," 6.1810 /2025 - Operating System Engineering [Online]. Available: pdos.csail.mit.edu/6.1810/2025/labs/syscall.html. Accessed: 15-Nov-2025.*

[2] *CS with Kas, "A user mode program in XV6 operating system," YouTube [Video] (CS with Kas channel). Available: youtube.com/watch?v=5zlfqDsCiZw. Accessed: 15-Nov-2025.*