**TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**TRƯỜNG ĐIỆN – ĐIỆN TỬ**

**VIỆN ĐIỆN TỬ VIỄN THÔNG**



**BÁO CÁO BÀI TẬP LỚN**

**HỆ ĐIỀU HÀNH**

**Giảng viên hướng dẫn: TS. Phạm Văn Tiến**

**Nguyễn Duy Tuấn**

**20172895**

**DTVT10-K62**

***Hà Nội, tháng 6 năm 2022***

MỤC LỤC:

[**CHƯƠNG 1: HỆ ĐIỀU HÀNH PINTOS** 3](#_Toc106872126)

[**1.1 Tổng quan về Pintos:** 3](#_Toc106872127)

[**1.2 Hệ thống thư mục của Pintos:** 4](#_Toc106872128)

[**CHƯƠNG 2: USER PROGRAM** 5](#_Toc106872129)

[**2.1 Process:** 5](#_Toc106872130)

[**2.2 Thread** 6](#_Toc106872135)

[**2.3 Thao tác tệp Syscalls** 9](#_Toc106872140)

[**2.4 Kết quả chạy chương trình** 13](#_Toc106872145)

[**CHƯƠNG 3: VIRTUAL MEMORY** 14](#_Toc106872146)

[**3.1 Paging** 14](#_Toc106872147)

[**3.2 Framing.** 17](#_Toc106872151)

**3.3 Kết quả chạy chương trình………………………………………………..20**

[**4. Kết quả** 21](#_Toc106872156)

# **CHƯƠNG 1: HỆ ĐIỀU HÀNH PINTOS**

Ở chương này, ta sẽ tìm hiểu tổng quan về hệ điều hành Pintos cũng như cấu trúc thư mục bên trong.

## **1.1 Tổng quan về Pintos:**

Pintos là hệ điều hành đơn giản với kiến trúc 80x86 được thiết kế bởi đại học Stanford nhằm phục vụ cho mục đích học tập và nghiên cứu trong khóa học CS140 của sinh viên. Hệ điều hành này hỗ trợ các cơ chế cơ bản như kernel threads, khởi động và chạy user programs, hệ thống file, tuy nhiên chúng chỉ dừng ở mức cơ bản nhất. Do đó, qua dự án Pintos, sinh viên sẽ tiến hành lập trình và cải tiến các cơ chế trên sao cho chúng đạt được hiệu quả tốt nhất.

Các dự án nhỏ trong Pintos bao gồm:

* Project 1: Threads.
* Project 2: User Programs.
* Project 3: Virtual Memory.
* Project 4: File Systems.

Các dự án trên sẽ được chạy và kiểm tra bằng một hệ thống mô phỏng. Bản chất nó là một chương trình với chức năng mô phỏng lại một hệ vi xử lý kiến trúc 80x86. Báo cáo này sẽ sử dụng trình mô phỏng QEMU.

## **1.2 Hệ thống thư mục của Pintos:**

Bên trong mã nguồn pintos, chúng ta có thể thấy cấu trúc thư mục của dự án theo đường dẫn “…/pintos/src”.

* Thư mục “threads/”: là nơi chưa mã nguồn của kernel gốc. Ở dự án 1 ta sẽ chủ yếu triển khai tại thư mục này.
* Thư mục “userpro/”: chưa mã nguồn cho trình tải nạp ứng dụng người dùng (user programs). Tại đây triển khai dự án 2.
* Thư mục “vm/”: Thư mục này hoàn toàn rỗng, ta cần triển khai Virtual memory tại đây (Dự án 3).
* Thư mục “filesys/”: Chứa mã nguồn cho hệ thống file cơ bản. Thư mục này được sử dụng trong dự án 2 và dự án 4.
* Thư mục “devices/”: Đây là thư mục chứa mã nguồn cho việc giao tiếp của hệ thống với các thiết bị vào ra. Chúng ta sẽ cần thay đổi mã nguồn cho bộ định thời timer trong dự án số 1.
* Thư mục “lib/”: Chứa các thư viện chuẩn trong C phục vụ cho các dự án. Ta không cần thay đổi mã nguồn tại đây.
* Thư mục “tests/”: Chứa các kết quả chạy thử của các dự án. Ta có thể thay đổi mã nguồn nếu cần trong quá trình kiểm thử.

# **CHƯƠNG 2: USER PROGRAM**

## **2.1 Process:**

Trong tác vụ này, chúng ta chủ yếu sửa đổi `process.c` và xử lý các chuỗi. Tuy nhiên, để kiểm tra tính đúng đắn của thuật toán của chúng ta trong tác vụ này, chúng ta cũng phải triển khai syscall\_write trong syscall.c

### **2.1.1 Cấu trúc dữ liệu và chức năng**

Không có cấu trúc dữ liệu phức tạp, chỉ sử dụng string và int.

-<process.c>:

- sửa đổi process\_execute (const char \* file\_name)

- sửa đổi start\_process (void \* file\_name\_)

- thêm push\_argument (void \*\* esp, int argc, int argv [])

- sửa đổi process\_wait (tid\_t child\_tid)

-<syscall.c>

- thêm sys\_write (int fd, const void \* buffer, unsigned size, int \* ret)

**2.1.2 Các thuật toán**

Hàm process\_execute cung cấp file\_name, bao gồm chuỗi lệnh và đối số. Đầu tiên, tách tên tệp để tách lệnh và đối số. Chúng ta lấy lệnh này làm tên của luồng mới và sau đó, truyền các đối số cho hàm start\_process, load` và setup\_stack. Sau khi phân tích mã pintos ban đầu, chúng ta biết rằng sau khi process\_execute tạo luồng, chương trình người dùng không thực thi ngay lập tức, nó nhập vào start\_process và hàm này gọi hàm load, sẽ cấp phát bộ nhớ cho các chương trình người dùng. Vì vậy, thời gian để thiết lập ngăn xếp là sau khi tải, Để thiết lập ngăn xếp, trước tiên chúng ta ghi nhớ các đối số và tên lệnh và lưu địa chỉ của chúng để sử dụng trong tương lai. Sau đó, thêm căn chỉnh và địa chỉ của argv được lưu trữ trước đó cũng đảm bảo rằng argv [argc] là một con trỏ null. Tiếp theo, thêm địa chỉ của argv,argc và cuối cùng là địa chỉ return. Chúng ta tạo hàm push\_argument để thực hiện việc chia nhỏ đối số cho argv trong start\_process.

**2.1.3 Đồng bộ hóa**

Khi phân tích cú pháp dòng lệnh, chúng ta sử dụng strtok\_r do pintos đưa ra thay vì strtok. Sự khác biệt giữa strtok và strtok\_r là cái sau là threadsafe. save\_ptr trong strtok\_r được cung cấp bởi người gọi nhưng strtok dựa vào một con trỏ để nhớ nơi nó đang tìm kiếm. Luồng có thể bị gián đoạn bất cứ lúc nào, vì vậy điều quan trọng là phải an toàn cho luồng. Chúng ta sử dụng filesys\_lock để thực hiện thao tác với tệp bằng cách thêm phương thức trong thread.c và thread.h.

**2.1.4 Cơ sở lý luận**

Trong tác vụ này, chúng ta tách tên lệnh và các đối số khác và chuyển chúng vào hàm load và thêm chúng vào ngăn xếp với thứ tự chính xác. Mục đích chính của nhiệm vụ này là gán các tham số cho một ngăn xếp cụ thể theo một thứ tự cụ thể và logic rõ ràng.

**2.2 Thread**

**2.2.1 Cấu trúc dữ liệu và chức năng**

**Cấu trúc dữ liệu :**

Chúng ta tạo một cấu trúc mới có tên là child

- tid\_t tid; / \* tid của chủ đề \* /

- bool isrun; / \* liệu luồng con có được chạy thành công hay không \* /

- struct list\_elem child\_elem; / \* danh sách luồng con \* /

- struct semaphore sema; / \* semaphore để kiểm soát việc chờ đợi \* /

- int store\_exit; / \* trạng thái thoát của chuỗi con \* /

Chúng ta thêm một số thuộc tính mới vào struct thread

- struct list child; / \* Danh sách các luồng con \* /

- struct child \* thread\_child; / \* Lưu trữ các con của luồng này \* /

- int st\_exit; / \* Trạng thái thoát \* /

- struct semaphore sema; / \* Kiểm soát logic của tiến trình con\* /

- bool success; / \* Đánh giá khi luồng con thực thi thành công \* /

- struct thread \* parent; / \* Luồng cha \* /

Chúng ta tạo một cấu trúc mới có tên là syscalls

- tệp danh sách cấu trúc; / \* Danh sách các tệp đã mở \* /

- int file\_fd;  / \* Bộ mô tả của tệp \* /

- struct file \* file\_owned; / \* Tệp đã mở \* /

**Chức năng:**

- thêm get\_user (const uint8\_t \* uaddr)

- sửa đổi syscall\_handler (struct intr\_frame \* f)

- sửa đổi syscall\_init (void)

- thêm sys\_halt (intr\_frame \* f), sys\_exit (intr\_frame \* f), sys\_exec (intr\_frame \* f), sys\_write (intr\_frame \* f), int sys\_wait (intr\_frame \* f).

**2.2.2 Các thuật toán:**

Trong tác vụ này, chúng ta cần hoàn thành 4 syscall

- SYS\_HALT, / \* tắt hệ thống \* /

- SYS\_EXEC, / \* bắt đầu một chương trình mới với process\_execute () \* /

- SYS\_WAIT, / \* đợi một process con cụ thể thoát ra \* /

- SYS\_PRACTICE / \* thêm 1 vào đối số đầu tiên của nó và trả về kết quả \* /

<process.c>:

Trong hàm process\_execute, gọi sema\_down cho luồng hiện tại để đồng bộ hóa quá trình thực thi của nó. Nếu tiến trình được thực thi thành công, hãy trả về tid của tiến trình con khác trả về TID\_ERROR. Trong hàm process\_wait, chúng ta sẽ kiểm tra xem child\_tid có tồn tại trong thread\_current () -> childs hay không. Nếu không, hàm này sẽ trả về -1, hoặc chúng ta sẽ kiểm tra xem luồng con (child) đã được đợi trước đó chưa, vì tài liệu quy định rằng một luồng con (child) chỉ nên được đợi nhiều nhất một lần. Nếu nó đã được đợi trước đó, hãy trả về -1, nếu không hãy đặt cờ chờ là true (thành công). Sau khi kiểm tra, chúng ta xuống semaphore sema để chặn tiến trình cho đến khi luồng con kết thúc, Cuối cùng chúng ta xóa process con khỏi childs và trả về store\_exit của nó.

<syscall.c>:

Chúng ta có một hàm syscall\_handler với switch-case để thực thi mã tương ứng. Loại lệnh gọi hệ thống được đọc từ ngăn xếp của quy trình người dùng. Tuy nhiên, vì chúng nằm trong không gian địa chỉ ảo của quy trình người dùng, chúng ta nên kiểm tra xem địa chỉ đó có được trỏ đến địa chỉ hợp lệ hay không trước khi thực hiện các lệnh gọi hệ thống. Việc xác minh cụ thể bao gồm kiểm tra xem địa chỉ có bên dưới PHYS\_BASE không, địa chỉ có được ánh xạ hay không hoặc địa chỉ có giá trị trống hay không. Nếu địa chỉ không hợp lệ, thì chúng ta cần giải phóng trang bộ nhớ và giải phóng mọi ổ khóa (lock) hoặc bán tín hiệu (semaphore) khỏi quy trình này trước khi thoát.

\*\*halt\*\*

Được thực hiện bởi hàm sys\_halt (intr\_frame \* f), chỉ cần gọi hàm shutdown\_power\_off.

\*\*exec\*\*

Được triển khai bởi hàm sys\_exec (intr\_frame \* f), trước tiên để kiểm tra xem tệp được tham chiếu bởi file\_name có hợp lệ hay không (con trỏ đến địa chỉ memeory, trang và nội dung của trang có hợp lệ hay không). Nếu nó không hợp lệ, trả về -1, nếu không, chúng ta gọi hàm process\_execute (const char \* file\_name).

\*\*wait\*\*

Được triển khai bởi hàm sys\_wait (pid\_t pid), trước tiên hãy kiểm tra xem đối số mà nó truyền vào (pid) có hợp lệ hay không. Nếu nó không hợp lệ, trả về -1, nếu không, chúng ta gọi hàm process\_wait (tid\_t child\_tid) được thực hiện trong tác vụ 1.

\*\*practice\*\*

Syscall thực hành chỉ thêm 1 vào đối số đầu tiên của nó argv [0] và trả về kết quả.

**2.2.3 Đồng bộ hóa**

Trong quá trình thực thi process, execute sẽ trả về -1 nếu process bị lỗi thì không trả về được. Để giải quyết vấn đề này, chúng ta thêm thành công vào cấu trúc luồng để ghi lại xem liệu luồng có thành công hay không. Hơn nữa, chúng ta sử dụng chuỗi cấu trúc parent để lấy nguồn gốc của nó và đặt trạng thái của nó theo kết quả tải. Đó là một thiết kế tốt để ghi lại kết quả thực thi của luồng con ở dạng cha thay vì con. Cuối cùng nhưng không kém phần quan trọng, chúng ta sử dụng semaphore để nhận ra parent đang đợi child. Khi một tiến trình con được tạo, nó sẽ chặn sema để chặn luồng cha. Khi luồng con hoàn thành, nó sẽ sema để đánh thức luồng cha. Trong quá trình wating of process (chúng ta nhận ra quá trình chờ đợi bởi semaphore), khi một parent cần đợi child của mình, sema\_down sẽ được gọi để chặn luồng cha. Và khi child thực hiện xong, parent sẽ thức giấc.

**2.2.4 Cơ sở lý luận**

Trong nhiệm vụ này, chúng ta hoàn thành ba lệnh gọi hệ thống kernel và một lệnh cho thực hành. Để đạt được mục tiêu, chúng ta tạo một cấu trúc mới có tên là child và thêm một số thuộc tính vào cấu trúc thread. Semaphores được sử dụng trong nhiệm vụ này để ngăn chặn tình trạng đụng độ. Hơn nữa, chúng ta lưu trữ luồng cha và kiểu thiết kế như vậy giúp chúng ta tìm luồng cha thành thạo hơn (chúng ta sẽ thay đổi trạng thái của luồng cha nhanh hơn).

**2.3 Thao tác tệp Syscalls**

**2.3.1 Cấu trúc dữ liêu và chức năng**

**Cấu trúc:** Chúng ta thêm một số thuộc tính mới vào cấu trúc chuỗi

- struct list\_files; / \* danh sách các tệp đã mở \* /

- int file\_fd; / \* Số luồng tệp có \* /

- struct file \* file\_owned; / \* tệp đã mở \* /

Chúng ta tạo một cấu trúc mới có tên là process\_file

struct thread\_file {

int fd;

struct file\* file;

struct list\_elem file\_elem;

**Chức năng:**

- syscall.c>

- sửa đổi syscall\_handler (struct intr\_frame \*)

- sửa đổi syscall\_init (void)

- thêm các chức năng syscall

- void sys\_create (struct intr\_frame \* f); / \* tạo syscall \* /

- void sys\_remove (struct intr\_frame \* f); / \* loại bỏ syscall\* /

- void sys\_open (struct intr\_frame \* f); / \* mở syscall\* /

- void sys\_wait (struct intr\_frame \* f); / \* syscall chờ \* /

- void sys\_filesize (struct intr\_frame \* f); / \* kích thước syscall \* /

- void sys\_read (struct intr\_frame \* f); / \*đọc syscall \* /

- void sys\_write (struct intr\_frame \* f); / \* ghi syscall \* /

- void sys\_seek (struct intr\_frame \* f); / \* tìm syscall\* /

- void sys\_tell (struct intr\_frame \* f); / \* syscall tell\* /

- void sys\_close (struct intr\_frame \* f); / \* đóng syscall \* /

- thêm is\_valid\_pointer (void \* esp, uint8\_t argc)

- thêm find\_file\_id (int file\_id)

- thêm check\_ptr2 (const void \* vaddr)

- <thread.c>&& <thread.h>

- thêm khóa tệp chung để đảm bảo sự an toàn của chuỗi

- sửa đổi một số chức năng khởi tạo trong thread.c

**2.3.2 Các thuật toán**

Trong tác vụ này, chúng ta cần thực hiện 9 syscall khác: tạo, loại bỏ, mở, kích thước tệp, đọc, ghi, tìm kiếm, nói và đóng. Khi một chương trình người dùng đang chạy, chúng ta phải đảm bảo rằng không ai có thể sửa đổi tệp thực thi của nó trên đĩa. Đối với tác vụ này, chúng ta sử dụng khóa toàn cầu (global key) để đảm bảo tệp syscalls là chuỗi an toàn. Khi mọi cuộc gọi tổng hợp được gọi, nó phải nhận được khóa và sau đó, giải phóng khóa. Bên cạnh đó, hệ thống tệp của Pintos không an toàn theo luồng, các cuộc gọi hệ thống hoạt động tệp không thể gọi nhiều chức năng của hệ thống tệp cùng một lúc. Để đảm bảo điều này, chúng ta thêm một biến mới file\_fd vào struct thread để giữ cho bộ giải mã tệp lớn hơn STDIN\_FILENO và STDOUT\_FILENO, file\_owned là để giữ các tệp đã mở của một chuỗi. Hàm syscall\_handler sẽ xác định loại hàm syscall sẽ thực thi. Tất cả các đối số đã có trong ngăn xếp khi chương trình người dùng gọi một cuộc gọi tổng hợp. Vì vậy, chúng ta chỉ cần lấy tham số từ ngăn xếp. Mỗi tệp syscalls sẽ gọi các chức năng tương ứng trong thư viện hệ thống tệp trong filesys.c sau khi nó có được khóa hệ thống tệp toàn cầu, khóa này cuối cùng sẽ được giải phóng.

- Hàm syscall\_init (void)sẽ khởi tạo syscalls để lưu trữ chức năng của syscalls trong tác vụ này.

- Hàm is\_valid\_pointer (void \* esp, uint8\_t argc) được sử dụng để xử lý các lỗi không xác định trong kiểm tra bad-read. Hàm này chỉ được sử dụng để kiểm tra xem bộ nhớ có hợp lệ hay không và nó chỉ được sử dụng trong sys\_read.

- Hàm special\_eixt được sử dụng để vượt qua bài kiểm tra đặc biệt có bộ nhớ không hợp lệ, trang và nội dung trang sai. Chúng ta sẽ kết thúc quá trình và thoát với -1 bởi printf.

- Hàm check\_ptr2 được sử dụng để kiểm tra xem có một số lỗi như bộ nhớ không hợp lệ, trang và nội dung sai của trang cho mỗi cuộc gọi tổng hợp hay không.

\*\*create\*\*

Được triển khai bởi hàm sys\_create (struct intr\_frame \* f), chỉ cần gọi hàm filesys\_create và có được khóa bằng cách get\_lock\_f (), giải phóng nó sau khi kết thúc bằng release\_lock\_f (). Quá trình khóa này sẽ được sử dụng trong hoạt động hệ thống tệp khác trong các phương pháp sau.

\*\*remove\*\*

Được thực hiện bởi hàm sys\_remove (struct intr\_frame \* f), chỉ cần gọi hàm filesys\_remove (const char \* name).

\*\*open\*\*

Được thực hiện bởi hàm sys\_open (struct intr\_frame \* f), trước tiên, chúng ta cần mở tệp fisrt bằng hàm filesys\_open (const char \* name). Sau đó, chúng ta cần đẩy tệp có cấu trúc thread\_file vào danh sách tệp tệp đã mở của luồng.

\*\*write\*\*

Được thực hiện bởi hàm sys\_write (struct intr\_frame \* f), trước tiên chúng ta cần đánh giá xem chúng ta sẽ ghi vào stdout hay tệp. Nếu nó sẽ được ghi vào stdout, chúng ta sử dụng hàm putbuf (const char \* buffer, size\_t n) để hoàn thành nó. Nếu nó sẽ được ghi vào tệp, đầu tiên chúng ta tìm tệp bằng id find\_file\_id (int file\_id) trong luồng hiện tại, sau đó thực hiện thao tác ghi bằng file\_write (struct file \* file, const void \* buffer, off\_t size).

\*\*seek\*\*

Được thực hiện bởi hàm sys\_seek (struct intr\_frame \* f), chỉ cần gọi hàm file\_seek (struct file \* file, off\_t new\_pos).

\*\*Tell\*\*

Được triển khai bởi hàm sys\_tell (struct intr\_frame \* f), trước tiên hãy tìm tệp từ find\_file\_id (int file\_id). Sau đó gọi hàm `file\_tell (struct file \* file) để thực hiện lệnh gọi syscall.

\*\*Close\*\*

Được triển khai bởi hàm sys\_close (struct intr\_frame \* f), trước tiên hãy tìm tệp từ find\_file\_id (int file\_id). Sau đó gọi hàm `file\_close (struct file \* file) để thực hiện đóng cuộc gọi syscall. Cuối cùng, chúng ta xóa tệp trong danh sách của chuỗi và giải phóng tệp.

\*\*Filesize\*\*

Được triển khai bởi hàm sys\_filesize (struct intr\_frame \* f), trước tiên hãy tìm tệp từ find\_file\_id (int file\_id). Sau đó gọi hàm `file\_length (struct file \* file) để thực hiện kích thước tệp syscall.

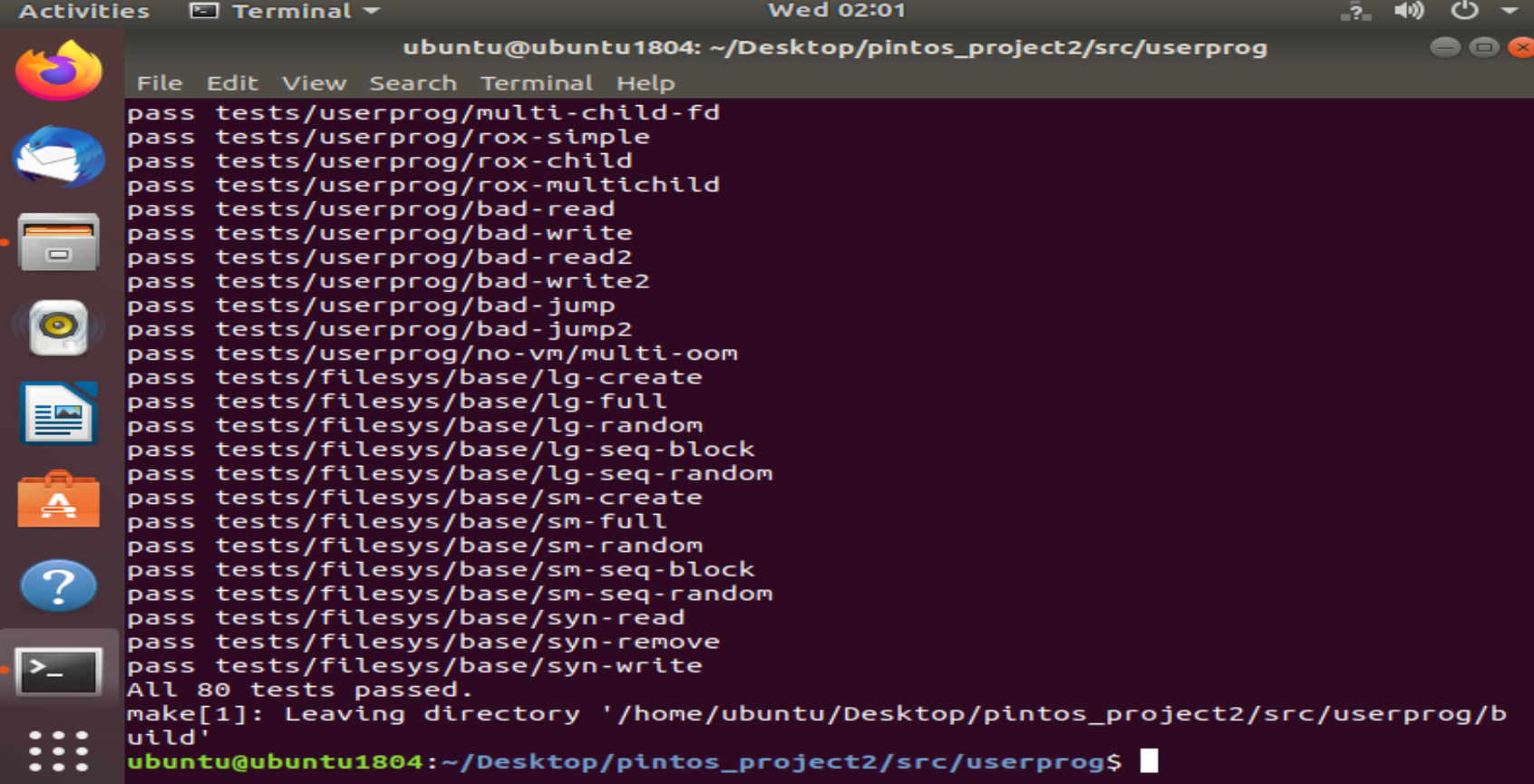
**2.3.3 Đồng bộ hóa**

Tất cả các hoạt động của tệp được bảo vệ bằng khóa hệ thống tệp toàn cầu (global file system lock), có thể ngăn chặn I / O trên cùng một fd cùng một lúc. Đầu tiên, chúng ta sẽ kiểm tra xem luồng hiện tại có đang giữ khóa toàn cục lock\_f hay không. Nếu vậy, chúng ta phát hành nó. Sau đó, chúng ta phải đóng tất cả tệp mà luồng hiện tại đang mở và giải phóng tất cả con của nó. Ngoài ra, chúng ta vô hiệu hóa sự gián đoạn, khi chúng ta đi qua thread\_current () -> parent-> childs hoặc thread\_current () -> files, để ngăn lỗi không thể đoán trước hoặc tình trạng chạy đua trong chuyển đổi ngữ cảnh. Vì vậy chúng sẽ không gây ra tình trạng chạy đua.

**2.3.4 Cở sở lý luận**

Trong tác vụ này, chúng ta hoàn thành 9 lệnh gọi hệ thống kernel cho các hệ thống tệp. Để đạt được mục tiêu, chúng ta tạo một cấu trúc mới có tên là thread\_file và thêm một số thuộc tính vào cấu trúc thread. Khóa được sử dụng trong nhiệm vụ này để ngăn chặn tình trạng đụng độ. Hơn nữa, một số tình huống đặc biệt như bộ nhớ, trang và tệp không hợp lệ đã được xem xét bằng cách kill quá trình bởi special\_exit.

**2.4 Kết quả chạy chương trình**



**CHƯƠNG 3: VIRTUAL MEMORY**

**3.1 Paging**

**3.1.1 Cấu trúc dữ liệu và chức năng**

- struct lock frametable - Khóa để truy cập frametable

- struct lock pagetable - Khóa để truy cập có thể phân trang

- struct hash frametable - Một bảng băm được sử dụng để lưu trữ dữ liệu khung

- struct frame - Một cấu trúc được sử dụng để lưu trữ các khung trong bảng băm

- struct frame\_elem - Một elem danh sách được sử dụng để lưu trữ các khung trong frametable

- struct void \* faddr - Địa chỉ của khung đang được tìm kiếm trong bảng khung (frametable)

- void \* paddr - Địa chỉ của trang trong bộ nhớ hiện đang chiếm khung

- int owner - Được sử dụng để lưu trữ chủ sở hữu của khung hiện tại

- unsigned frame\_hash (const struct hash\_elem \* h, void \* aux UNUSED) - Được sử dụng để lấy chỉ số băm (hash) cho khung

- bool frame\_less (const struct hash\_elem \* h, const struct hash\_elem \* y, void \*

aux UNUSED) – Trả về giá trị đúng/sai cho khung chỉ số băm (hash) ít hơn.

- struct frame \* frame\_lookup (const void \* address) - Được sử dụng để tra cứu một khung.

- trong bảng khung (frametable): struct frame \* frame\_delete (const void \* address) - Được sử dụng để xóa một khung

- từ bảng khung (frametable):

+ struct hash spt - Bảng băm cho bảng trang bổ sung

+ struct page - Một cấu trúc được sử dụng để lưu trữ các trang trong bảng trang (page table)

+ struct list\_elem page\_elem - Một danh sách được sử dụng để lưu trữ các trang trong bảng trang bổ sung.

- void \* paddr - Được sử dụng để lưu trữ địa chỉ ảo của trang đang cất giữ (stored)

- void \* data - Được sử dụng để lưu trữ vị trí của dữ liệu cần được cài đặt vào một trang hoặc bị xóa khỏi một trang khi được yêu cầu. Điều này sẽ được sử dụng để xử lý lỗi trang. Có thể sẽ được thay đổi thành một con trỏ int.

- void \* stack - Đây sẽ là địa chỉ của con trỏ ngăn xếp cho trang (nếu có). Điều này được khởi tạo là null, và cũng được sử dụng như một phép thử để biết liệu trang này có giữ một phần của ngăn xếp hay không.

- unsigned page\_hash (const struct hash\_elem \* h, void \* aux UNUSED) - Được sử dụng để lấy chỉ mục băm cho trang.

- bool page\_less (const struct hash\_elem \* h, const struct hash\_elem \* y, void \*

aux UNUSED) – Trả về đúng/sai khi so sánh giá trị nhỏ hơn của 2 hash\_elem.

- struct page \* page\_lookup (địa chỉ const void \*) - Dùng để tra cứu một trang

từ bảng trang bổ sung

- struct page \* page\_delete (const void \* address) - Được sử dụng để xóa một trang

từ bảng trang bổ sung

- static bool setup\_stack (const char \* cmd\_line, void \*\* esp) - Được sửa đổi để tăng khả năng mở rộng của ngăn xếp (stack). Cũng có thể sửa đổi để sử dụng bảng khung (frametable) và bảng trang (page table).

- static void page\_fault (struct intr\_frame \* f) - Được sửa đổi để cho phép các lỗi trang được sửa dựa trên SPT

- static bool load\_segment (struct file \*, off\_t ofs, uint8\_t \* upage, uint32\_t read\_bytes, uint32\_t zero\_bytes, bool writable)- Đã sửa đổi nó để bổ sung tính năng tải chậm và cũng để bổ sung chức năng bảng trang (page table) và bảng khung (frametable).

Hầu hết các hàm process.c sẽ được sửa đổi để đạt được kết quả cần thiết từ bảng trang (page table) và khung (frametable) được tạo ra.

**3.1.2 Thuật toán**

Vấn đề: Mô tả cách để truy cập dữ liệu được lưu trữ trong SPT về một trang nhất định. Quá trình này khá đơn giản. Đầu tiên, chúng ta sử dụng một bảng băm có chức năng SPT, cho phép truy cập khá nhanh với ít tốn tài nguyên bộ nhớ so với chèn (insertions ) và xóa (deletions). Khi chúng ta thực hiện truy cập vào SPT, điều đầu tiên chúng ta cần là địa chỉ ảo (virtual address) của bảng trang đang được tìm kiếm . Điều này đóng vai trò là chìa khóa cho hàm hash\_table chúng ta đã sử dụng cho SPT . Sau đó, chúng ta gọi hash\_lookup với giá trị đầu vào là địa chỉ ảo đề cập ở trên, sau đó sẽ hàm trả về cho chúng ta dữ liệu trang tương ứng mà chúng ta muốn từ SPT, trong một cấu trúc được gọi là trang (page). Đối với điều này, chúng ta phải triển khai các chức năng cho phép băm địa chỉ và một hàm so sánh hai trang địa chỉ và cho chúng ta biết cái nào nhỏ hơn trong hai cái.

**3.1.3 Tổng hợp**

Vấn đề: Khi hai process của người dùng đều cần một khung mới cùng một lúc, làm thế nào để tránh được đụng độ? Chúng ta có một global key cho bảng khung mà chúng ta sử dụng. Bất cứ khi nào một quy trình cần một khung mới, họ nhất thiết phải thêm mục nhập của mình vào bảng khung (frametable) và do đó, trước khi khung được cấp phát, chúng ta chỉ cần có được một khóa toàn cục (global key) (được khởi tạo thành 1 trong init.c).  Bằng cách này, bất cứ khi nào process khác muốn có được một khung mới, nó cũng phải có cùng một khóa (key) và do đó sẽ đợi cho đến khi khung khác đã hoàn thành phân bổ khung thì sau đó process có thể nhận được khung của riêng nó. Sử dụng đồng bộ hóa khóa đơn giản này, chúng ta có thể tránh hầu hết sự đụng độ giữa các process với nhau.

**3.2 Framing.**

**3.2.1 Cấu trúc dữ liệu và chức năng**

- int dirty - Chúng ta thêm một dirty bit vào cấu trúc khung trong bảng khung để xác định có lỗi hay không?

- struct semaphore sema\_page - Một semaphore được sử dụng để đảm bảo đồng bộ hóa trong bảng trang.

- int evicted - Một cấu trúc int trong khung để hiển thị nếu khung đã bị loại bỏ hay không?

- int paged\_out - Được sử dụng để xem trang này có còn trong bộ nhớ hay không?

**3.2.2 Thuật toán.**

B1: Khi một khung là bắt buộc nhưng không có khung nào là miễn phí, một số khung phải được giải phóng sau khi hoàn tất. Mô tả quy trình để chọn khung để giải phóng. Thuật toán thay thế khung chúng ta sử dụng là một thuật toán LIFO (Last In First Out) đơn giản.



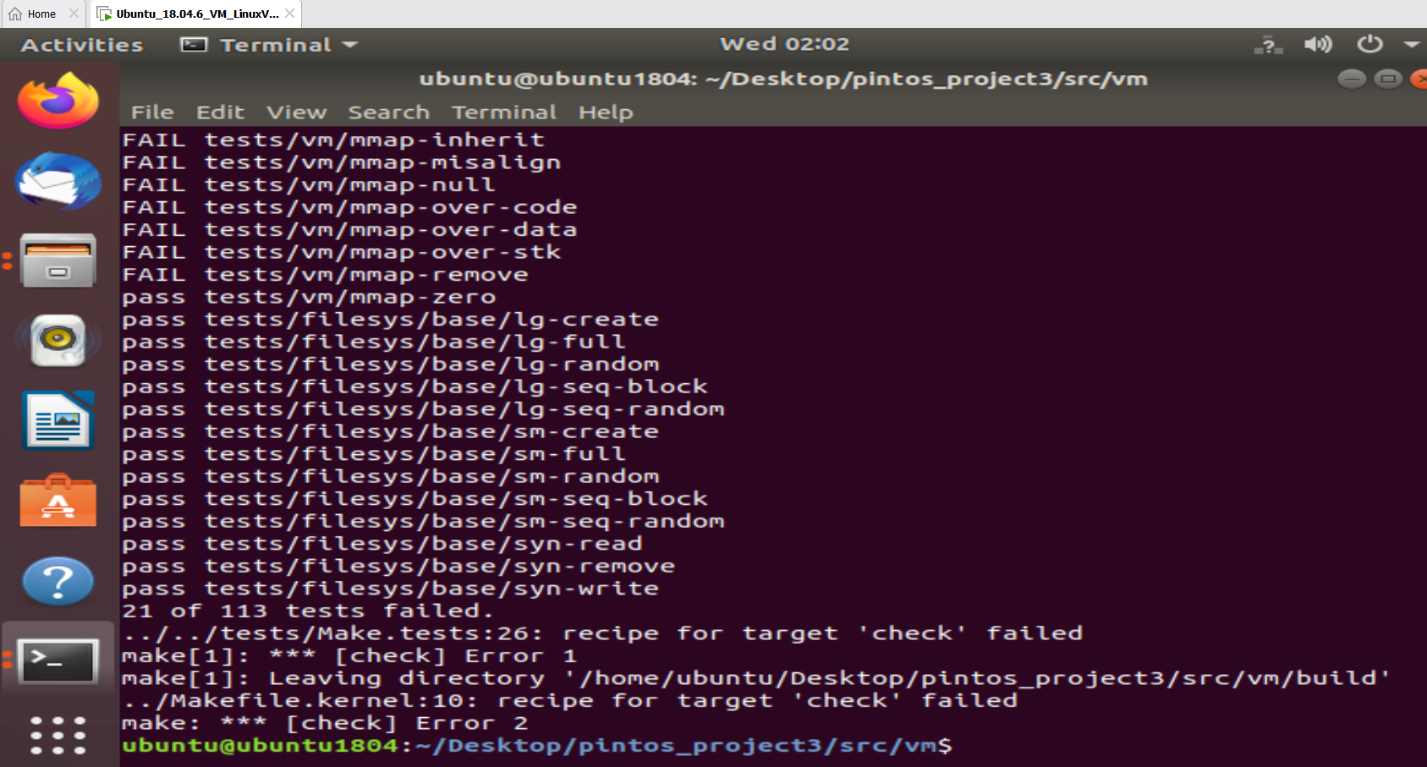
Điều đầu tiên chúng ta làm là thiết lập sự kết nối giữa các hash trên hashtable để tạo ra queue.  Khi đó, chúng ta chuyển trực tiếp đến phần tử cuối cùng của bảng khung ra khỏi và dịch các node khác lên và lặp lại quy trình. Sau đó, chúng ta kiểm tra xem có hay không bit truy cập cho trang đã được sử dụng. Nếu có, điều này làm cho khung trở thành đối tượng phải đẩy ra khỏi hàng chờ. Sau đó, chúng ta kiểm tra xem dirty bit cũng đã được thiết lập chưa. Nếu có, chúng ta chỉ cần loại bỏ trang ngay lập tức.  Nếu không, chúng ta sẽ kiểm tra một sốtrang (Để tiết kiệm thời gian ta giới hạn số lượng node trong LIFO queue là 5), phía trước trang đã chọn và xem liệu có trang nào trong số đó có cả dirty và bit truy cập của chúng được thiết lập. Nếu có, thì trang đó sẽ trục xuất ngay lập tức. Bản thân việc trục xuất khá đơn giản vì hầu hết quá trình có thể sẽ được thực hiện trong palloc\_get\_page. Chúng ta chỉ cần loại bỏ khung và sau đó, phân bổ một khung mới, và cuối cùng đặt khung mới cấp phát trở lại bảng khung. Chúng ta cũng có thể lưu trữ khung đã bị loại bỏ vào một hàng đợi khác để khôi phục trở lại bảng khung chính (nếu muốn)

B2: Khi một quá trình P thu được một khung đã được sử dụng trước đó bởi một xử lý Q, làm cách nào để bạn điều chỉnh bảng trang (và bất kỳ dữ liệu nào khác cấu trúc) để phản chiếu khung Q không còn? Khi một quá trình P nhận được một khung mà trước đó, quá trình thay đổi điều này khá đơn giản. Điều đầu tiên chúng ta làm là thay đổi trang được giữ trong khung đã cho bằng cách cài đặt khung riêng của process P vào khung đang được sử dụng. Sau đó, chúng ta cập nhật SPT để tạo một bản ánh xạ cho khung. Cuối cùng chúng ta cập nhật bảng khung để trỏ đến trang mới trong đó và cũng cập nhật PID chủ sở hữu trong cấu trúc khung cho bảng khung. Thông qua quy trình đơn giản, ngắn gọn này, chúng ta đã chuyển giao thành công khung từ Q đến P.

B3: Giải thích kinh nghiệm của bạn để quyết định xem một trang có lỗi đối với địa chỉ ảo không hợp lệ sẽ khiến ngăn xếp được mở rộng thành trang bị lỗi. Kịch bản mà chúng ta phát triển cho mục đích này là một con trỏ cho mọi phần tử trong struct hash SPT, là một bảng trang bổ sung. Khi giá trị này là null, điều đó cho biết rằng trang đang được truy cập không phải là quyền truy cập ngăn xếp, và do đó chúng ta nên thực hiện quy trình page\_fault bình thường. Nếu nó hoàn toàn không có trong SPT, chúng ta coi nó như một truy cập ảo không hợp lệ. Nếu con trỏ ngăn xếp không null, thì giá trị được lưu trữ trong con trỏ này sẽ là vị trí hiện tại của esp, cho biết rằng đây là một truy cập ngăn xếp. Trước khi thực hiện việc này, chúng ta chạy địa chỉ này lên hoặc xuống để đảm bảo rằng chúng ta có thể kiểm tra một trang. Sau đó, chúng ta có thể thêm một kiểm tra nơi chúng ta có thể xem nếu esp hiện không có trong trang hiện tại và chuyển hướng lỗi trang vào trang đó. Đối với phạm vi của câu hỏi này, chúng ta tin rằng chúng ta không cần phải giải thích điều này.

B4: Giải thích những điều cơ bản về thiết kế đồng bộ hóa máy ảo của bạn. Trong đặc biệt, giải thích cách nó ngăn chặn đụng độ. (Tham khảo đến sách giáo khoa để giải thích các điều kiện cần thiết để tránh đụng độ.) Cách chúng ta thực hiện đồng bộ hóa VM là thông qua các semaphores cho mọi trang chúng ta sử dụng. Điều này, cho phép mọi quyền truy cập trang (và thông qua các khóa cho các bảng, truy cập bảng) để được bảo vệ khỏi các đụng độ với chi phí cao hơn sự phức tạp và rủi ro của các đụng độ. Điều này mặc dù cho phép chúng ta hoạt động song song và tăng độ tin cậy ở phía kernel. Cách chúng ta ngăn chặn đụng độ trong thiết kế của mình là một ứng dụng khá đơn giản của cách phòng tránh đụng độ mà chúng ta đã học trên lớp. Khi một quy trình sở hữu một trang, nó không ngăn các quy trình khác cố gắng truy cập trang này thông qua trang lỗi hoặc các phương pháp khác. Do đó, các đụng độ hầu hết được ngăn chặn theo cách này vì điều khoản giữ và chờ bị ngăn chặn trong hầu hết các trường hợp. Chúng ta nhận ra mặc dù có thể có những cách khác mà sự đụng độ có thể diễn ra và do đó, cân nhắc khác mà chúng ta sẽ thực hiện là semaphore đơn để truy cập bảng trang.

**3.3 Kết quả chạy chương trình.**

****

**4. Kết quả**

Như vậy, ở báo cáo này em đã thực hiện triển khai giải Project 2 kèm Project 3. Em xin chân thành cảm ơn thầy Phạm Văn Tiến đã cung cấp rất nhiều kiến thức bổ ích qua các tiết học cũng như giới thiệu hệ điều hành Pintos để sinh viên chúng em có một trải nghiệm rất thực tiễn trong môn học Hệ Điều Hành.

Link Github: https://github.com/tuanhust2111/btlh-h