ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Tài liệu hướng dẫn thực hành

HỆ ĐIỀU HÀNH

Biên soạn: ThS Phan Đình Duy

ThS Nguyễn Thanh Thiện

KS Trần Đại Dương

KS Trần Hoàng Lộc

MỤC LỤC

NỘI QUY THỰC HÀNH

1. Sinh viên tham dự đầy đủ các buổi thực hành theo quy định của giảng viên hướng dẫn (GVHD) (6 buổi với lớp thực hành cách tuần hoặc 10 buổi với lớp thực hành liên tục).
2. Sinh viên phải chuẩn bị các nội dung trong phần "Sinh viên viên chuẩn bị" trước khi đến lớp. GVHD sẽ kiểm tra bài chuẩn bị của sinh viên trong 15 phút đầu của buổi học (nếu không có bài chuẩn bị thì sinh viên bị tính vắng buổi thực hành đó).
3. Sinh viên làm các bài tập ôn tập để được cộng điểm thực hành, bài tập ôn tập sẽ được GVHD kiểm tra khi sinh viên có yêu cầu trong buổi học liền sau bài thực hành đó. Điểm cộng tối đa không quá 2 điểm cho mỗi bài thực hành.

Bài 7. THỰC HÀNH TRÊN HỆ ĐIỀU HÀNH PINTOS

(Dành riêng cho sinh viên học thực hành Hệ điều hành theo hình thức 2)

7.1 PHẦN 1: TỔNG QUAN VỀ PINTOS

7.1.1 Giới thiệu Pintos

Pintos là một hệ điều hành đơn giản dành cho kiến trúc tập lệnh x86. Nó được phát triển tại Đại học Stanford bởi Ben Pfaff vào năm 2004. Pintos được xây dựng để thay thế cho Not Another Completely Heuristic Operating System (Nachos) - một hệ điều hành tương tự được phát triển trước đó tại Đại học UC Berkeley bởi Thomas E. Anderson. Cả Nachos và Pintos được sử dụng chủ yếu cho mục đích giảng dạy với một số bài tập lập trình đi kèm. Thông qua việc thực hiện các bài tập này, sinh viên được tiếp cận với những khái niệm căn bản trong thiết kế và cài đặt hệ điều hành, bao gồm tiểu trình, quản lý bộ nhớ và truy cập hệ thống tập tin. Pintos cũng đồng thời cũng giúp sinh viên phát triển khả năng debug khi viết chương trình. Pintos và các bài tập thực hành đi kèm với nó được viết bằng ngôn ngữ C.

Nội dung phần thực hành bên dưới được xây dựng dựa trên tài liệu hướng dẫn thực hành Pintos của Đại học Stanford [1] và học

phần CS162 của Đại học UC Berkeley [2] (chi tiết xem ở phần tài liệu tham khảo). Mã nguồn của Pintos dùng cho phần thực hành này được cung cấp tại website môn học.

7.1.2 Cấu trúc Pintos

Pintos được tổ chức thành các thư mục như sau:

threads/: Nhân (kernel) của Pintos. Nội dung thực hành 1 sẽ thực hiện với các mã nguồn trong thư mục này.

userprog/: Thư mục chứa các chương trình của người dùng (user program). Nội dung thực hành 2 sẽ thực hiện với các mã nguồn trong thư mục này.

vm/: Thư mục chứa các mã nguồn dành cho việc quản lý bộ nhớ. Mặc định đây là một thư mục trống, các mã nguồn cần thiết sẽ được cài đặt ở nội dung thực hành 3.

filesys/: Hệ thống quản lý tập tin của Pintos.

devices/: Thư mục chứa các thư viện dành cho các tác vụ nhập/xuất dữ liệu: bàn phím, đĩa cứng, bộ định thời, …

lib/: Thư viện chính của Pintos, là một tập con của thư viện C chuẩn. Các mã nguồn trong thư mục này được biên dịch và sử dụng cho nhân của Pintos cũng như chương trình của người dùng. Tất cả các header trong thư mục này đều có thể được sử dụng thông qua cú pháp #include <...>. Chú ý **không** chỉnh sửa các mã nguồn bên trong thư mục này.

lib/kernel/: Các thư viện được sử dụng cho nhân của Pintos (không được dùng cho user program). Một số kiểu dữ liệu được cài đặt trong thư viện này có thể được sử dụng khi làm việc với mã nguồn nhân của Pintos: bitmaps, doubly linked lists và hash tables.

lib/user/: Các thư viện được sử dụng cho chương trình của người dùng (không dùng cho nhân của Pintos).

tests/: Thư mục chứa các bộ test dành cho từng đồ án.

examples/: Thư mục này chứa một số chương trình mẫu dùng cho nội dung thực hành 2.

misc/, **utils**/: Hai thư mục này chứa một số thư viện hỗ trợ cần thiết để Pintos có thể thực thi được. Chú ý **không** thao tác với các tập tin bên trong hai thư mục này.

Makefile.build: Tập tin mô tả cách thức để build nhân của Pintos. Chỉ được chỉnh sửa tập tin này nếu muốn thêm mã nguồn mới.

7.1.3 Cài đặt, build, chạy và debug Pintos

7.1.3.1 Cài đặt Pintos

Pintos được thiết kế để hoạt động trên môi trường của các hệ thống tương tự Unix. Nó đã được kiểm tra và hoạt động tốt trên GNU/Linux, cụ thể là các bản phân phối Debian và Ubuntu, cũng như Solaris.

Để có thể cài đặt và sử dụng Pintos cho các nội dung thực hành, cần cài đặt các công cụ sau:

- GCC phiên bản 4.0 trở đi.

- GNU binutils. Pintos sử dụng các thư viện addr2line, ar, ld, objcopy và ranlib nằm trong gói thư viện này.

- Perl phiên bản 5.8.0 trở đi.

- GNU make phiên bản 3.80 trở đi.

Bên cạnh các công cụ bắt buộc ở trên, có thể cài thêm một số công cụ sau: QEMU, phiên bản 0.8.0 trở đi, GDB (sử dụng để debug), X server.

Sau khi cài đặt các công cụ trên, tiến hành cài đặt Pintos theo các bước sau:

- Cài đặt Bochs phiên bản 2.2.6 (tham khảo hướng dẫn tại: https://web.stanford.edu/class/cs140/projects/pintos/pintos_12.html#SEC167)

- Cài đặt scripts trong thư mục src/utils. Sao chép các tập tin backtrace, pintos, pintos-gdb, pintos-mkdisk, pintos-set-cmdline, và Pintos.pm vào trong đường dẫn PATH.

Cài đặt src/misc/gdb-macros vào một thư mục public. Kế tiếp, chỉnh sửa GDBMACROS trong tập tin pintos-gdb (tập tin vừa được sao chép, không phải tập tin gốc) trỏ đến thư mục cài đặt gdb-macros. Kiểm tra việc cài đặt bằng cách chạy pintos-gdb (không có tham số). Nếu không có bất cứ thông báo lỗi nào về việc thiếu gdb-macros thì chứng tỏ nó đã được cài đặt một cách chính xác.

Biên dịch các thành phần còn lại của Pintos bằng cách di chuyển vào thư mục *src/utils*, gõ lệnh *make*. Thêm thư viện squish-pty vào PATH. Quá trình cài đặt Pintos đã hoàn tất.

7.1.3.2 Build Pintos

Sau khi cài đặt xong, tiến hành build các mã nguồn dành cho nội dung thực hành 1. Di chuyển đến thư mục threads (bằng lệnh cd). Kế tiếp, thực thi lệnh make. Một thư mục con có tên build sẽ được tạo ra, gồm 1 tập tin Makefile và một vài thư mục con. Tiếp tục thực thi lệnh make để build mã nguồn kernel bên trong. Quá trình build sẽ diễn ra trong khoảng 30 giây.

Kết thúc quá trình build, trong thư mục build sẽ có các tập tin sau:

Makefile: Đây là bản sao của tập tin /src/Makefile.build. Nó mô tả cách build mã nguồn nhân của Pintos.

kernel.o: Tập tin object chứa toàn bộ kernel. Tập tin này là kết quả của việc link các tập tin object được biên dịch từ các tập tin mã nguồn kernel riêng lẻ. Nó cũng chứa các thông tin debug nên có thể dùng GDB (hoặc các công cụ debug khác) để debug nó.

kernel.bin: Tập tin image của kernel. Về bản chất, tập tin này chính là tập tin kernel.o nhưng phần thông tin debug được bỏ đi để tiết kiệm không gian lưu trữ, đồng thời cũng giữ cho kích thước của kernel ở mức 512 kB, không vượt quá giới hạn được đặt ra theo thiết kế của kernel loader.

loader.bin: Tập tin image của kernel loader – đây là một đoạn mã nguồn được viết bằng hợp ngữ, được dùng để đọc dữ liệu kernel từ trên đĩa đưa vào trong bộ nhớ và tiến hành khởi chạy kernel. Kích thước của tập tin này là 512 byte, kích thước này là cố định bởi PC BIOS.

Các thư mục con của thư mục build chứa các tập tin object và các tập tin dependency (phụ thuộc), tất cả chúng được tạo ra bởi trình biên dịch (compiler). Các tập tin dependency này cần phải được biên dịch lại khi bất kỳ mã nguồn hoặc tập tin header nào bị thay đổi.

7.1.3.3 Chạy Pintos

Để chạy Pintos, sử dụng lệnh pintos. Lệnh pintos có cú pháp như sau:

pintos argument

Các tham số này sẽ được truyền đến Pintos kernel để thực thi. Có thể chạy một ví dụ đơn giản với tham số run alarm-multiple, bằng cách thực hiện lệnh pintos run alarm-multiple (cần thực hiện lệnh này ở bên trong thư mục build). Lệnh này sẽ tạo ra một tập tin bochsrc.txt, sau đó khởi chạy Bochs. Bochs sẽ mở một cửa sổ mới, hiển thị các thông tin mô phỏng máy tính và một thông báo của BIOS sẽ xuất hiện. Sau đó Pintos khởi động và chạy chương trình alarm-multiple, chương trình này sẽ xuất ra màn hình một vài dòng chữ. Có thể xem danh sách các tham số được sử dụng bởi

lệnh pintos và ý nghĩa của chúng bằng cách thực thi lệnh pintos với tham số -h.

7.1.3.4 Debug Pintos

Pintos hỗ trợ nhiều công cụ dùng để debug. Danh sách các công cụ có thể xem đầy đủ trong hướng dẫn trên website của Pintos. Ở đây chỉ giới thiệu một số công cụ phổ biến, thường được sử dụng

a. printf()

Hàm printf() được cài đặt sẵn trong Pintos, người dùng có thể gọi nó từ bất cứ đâu trong kernel. Cách sử dụng hàm này tương tự như hàm printf() trong ngôn ngữ C.

b. Macro ASSERT

Pintos cung cấp macro ASSERT, được định nghĩa trong tập tin '<debug.h>', để đánh giá biểu thức. Cú pháp của macro này như sau:

ASSERT (expression)

Macro này sẽ kiểm tra giá trị của biểu thức. Nếu biểu thức này có giá trị 0 (false), kernel sẽ phát sinh lỗi thông qua một thông báo. Nội dung thông báo này sẽ bao gồm biểu thức gây ra lỗi, tập tin mã nguồn, vị trí và backtrace dùng để tìm nguyên nhân gây ra lỗi.

c. Hàm và các tham số hàm

Bên cạnh macro ASSERT, trong tập tin ‘<debug.h>’, Pintos cung cấp thêm một số macro khác để định nghĩa một số thuộc tính đặc biệt cho hàm hoặc các tham số của hàm. Có thể sử dụng các tham số này cho mục đích debug:

UNUSED: Sử dụng macro này để thông báo cho trình biên dịch biết là tham số này sẽ không được sử dụng bên trong hàm.

NO_RETURN: Sử dụng macro này để thông báo cho trình biên dịch biết là hàm sẽ không có kết quả trả về.

NO_INLINE: Sử dụng macro này để thông báo cho trình biên dịch biết là không sử dụng các hàm nội tuyến (inline function).

PRINTF_FORMAT (format, first): Sử dụng macro này để thông báo cho trình biên dịch biết hàm này sẽ hoạt động như lệnh printf() với danh sách tham số và định dạng được truyền vào.

d. Backtraces

Khi có một lỗi phát sinh trong kernel, nó sẽ in ra một backtrace – một thông báo ngắn diễn tả lỗi nào đã phát sinh, đi kèm với nó là một danh sách địa chỉ các hàm đã thực hiện tại thời điểm lỗi phát sinh.

Có thể thêm hàm debug_backtrace() – hàm này được định nghĩa trong ‘<debug.h>’ – để in ra backtrace tại bất cứ vị trí nào trong mã nguồn. Ngoài ra cũng có thể sử dụng hàm

debug_backtrace_all() để in ra backtrace của tất cả các tiểu trình (thread).

e. GDB

GDB có thể được sử dụng khi chạy lệnh pintos.

Đầu tiên, thực thi lệnh pintos với tham số gdb, ví dụ: pintos --gdb -- run mytest.

Kế tiếp, mở một terminal thứ hai, thực thi lệnh pintos-gdb để thực thi GDB trên 'kernel.o': pintos-gdb kernel.o

Sau đó, thực hiện lệnh GDB sau: target remote localhost:1234

Từ lúc này, GDB đã kết nối với simulator (đang chạy Pintos) thông qua localhost và người dùng đã có thể sử dụng các lệnh GDB thông thường để debug.

7.2 PHẦN 2: THỰC HÀNH

Trước khi bắt đầu thực hiện các nội dung thực hành bên dưới, cần đảm bảo Pintos đã được cài đặt và hoạt động ổn định.

7.2.1 NỘI DUNG 1: TIỂU TRÌNH

7.2.1.1 Mục tiêu

Phần thực hành này giúp sinh viên:

- Làm quen và hiểu khái niệm tiểu trình trong Pintos.
- Hiểu, sử dụng và cài đặt được các thao tác liên quan đến tiểu trình trên Pintos.
- Hiểu và cài đặt các thuật toán định thời CPU trên Pintos: priority và multilevel feedback queue.

7.2.1.2 Sinh viên chuẩn bị

- Đọc trước nội dung thực hành ở phần 7.2.1.3
- Đọc trước các khái niệm và các thao tác liên quan đến tiểu trình ở phần 7.2.1.4
- Đọc trước cơ chế cấp phát bộ nhớ của Pintos ở phần 7.2.1.6
- Tìm hiểu các cơ chế đồng bộ trên Pintos

- Ôn lại giải thuật định thời theo độ ưu tiên. Đọc trước và hiểu được các bước thực hiện giải thuật định thời multilevel feedback queue ở phần 7.2.1.7

7.2.1.3 Nội dung thực hành

a. Alarm clock

Trong Pintos, các tiểu trình có thể gọi hàm *timer_sleep()* để tự đưa chúng vào trạng thái sleep:

```
/**
* This function suspends execution of the calling thread until time has
* advanced by at least x timer ticks. Unless the system is otherwise idle, the
* thread need not wake up after exactly x ticks. Just put it on the ready queue
* after they have waited for the right number of ticks. The argument to
* timer_sleep() is expressed in timer ticks, not in milliseconds or any another
* unit. There are TIMER_FREQ timer ticks per second, where TIMER_FREQ is
* a constant defined in devices/timer.h (spoiler: it's 100 ticks per second).
*/
void timer_sleep (int64_t ticks);
```

Mã nguồn được cài đặt hiện tại của hàm *timer_sleep()* hoạt động chưa hiệu quả vì có khả năng xuất hiện busy waiting khi thực thi. Do đó, cần cài đặt lại hàm *timer_sleep()* để giải quyết vấn đề trên. Như vậy, yêu cầu ở phần này là **cài đặt lại hàm *timer_sleep()* trong tập tin devices/timer.c** để giải quyết vấn đề busy waiting.

b. Priority Scheduler

Trong Pintos, mỗi tiểu trình có giá trị độ ưu tiên từ 0 (nhỏ nhất) đến 63 (cao nhất). Tuy nhiên, thuật toán lập lịch hiện tại được cài đặt trong Pintos không sử dụng các giá trị này. Vì vậy cần chỉnh sửa thuật toán lập lịch để các tiểu trình có độ ưu tiên cao sẽ luôn được chạy trước các tiểu trình có độ ưu tiên thấp.

Cài đặt giải thuật định thời theo độ ưu tiên bằng cách cài đặt các hàm sau (nằm trong tập tin threads/thread.c):

- *void thread_set_priority (int new_priority):* Thiết lập độ ưu tiên mới cho tiểu trình. Nếu tiểu trình hiện tại không có độ ưu tiên cao nhất thì ngừng thực thi và đưa tiểu trình về trạng thái chờ (gọi là yields).
- *int thread_get_priority (void):* Trả về độ ưu tiên hiện tại của tiểu trình.

Khi cài đặt các hàm này, cần chú ý rằng một tiểu trình không thể thể truy xuất và thay đổi giá trị độ ưu tiên của tiểu trình khác. Bất cứ sự truy xuất hoặc thay đổi nào như vậy cũng đều là không hợp lệ và không được phép diễn ra.

c. Advanced Scheduler

Yêu cầu đặt ra đối với phần này là **cài đặt giải thuật định thời multilevel feedback queue** (chi tiết về giải thuật và các hàm cần cài đặt được trình bày ở phần 7.2.1.7). Sau khi cài đặt xong giải thuật này, cần thay đổi thiết lập của bộ lập lịch hiện tại của

Pintos để có khả năng lựa chọn giải thuật theo độ ưu tiên hoặc giải thuật multilevel feedback queue trong quá trình thực thi, thông qua việc sử dụng biến bool thread_mlfqs trong tập tin thread.h.

Các hàm cần cài đặt sẽ nằm ở các tập tin sau: threads/synch.c, threads/thread.c và threads/thread.h.

7.2.1.4 Tiểu trình

a. Tổng quan về tiểu trình

Mã nguồn hiện tại của Pintos đã cài đặt sẵn hàm tạo và kết thúc tiểu trình, một bộ lập lịch để chuyển đổi giữa các tiểu trình và một số giải pháp đồng bộ (semaphores, locks, condition variables và optimization barriers).

Các mã nguồn này trông có vẻ khó hiểu khi mới đọc qua. Có thể đặt các hàm printf() (hoặc sử dụng các công cụ debug khác đã giới thiệu ở phần debug) vào bất cứ đâu trong các mã nguồn này, sau đó biên dịch và chạy lại để có thể hiểu rõ cách thức hoạt động của chúng.

Khi một tiểu trình được tạo bằng hàm thread_create(), nó sẽ cần phải nhận vào một tham số là tên một hàm đã khai báo trước – hàm này chính là hàm mà tiểu trình sẽ thực thi. Khi tiểu trình được tạo ra và chạy, nó sẽ bắt đầu thực thi từ đầu hàm thực thi này. Khi hàm thực thi này kết thúc (trả về kết quả), tiểu trình cũng sẽ kết thúc.

Mỗi tiểu trình, về bản chất, hoạt động như một chương trình con nằm bên trong Pintos, trong đó hàm thực thi được truyền vào thread_create() có vai trò giống như hàm main().

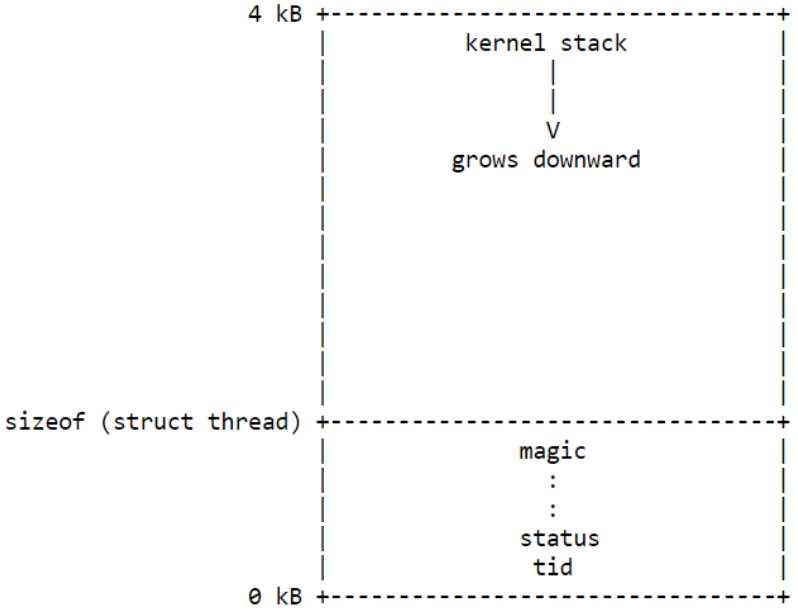
Tại một thời điểm bất kỳ, chỉ có một tiểu trình chạy, tất cả các tiểu trình khác đều không hoạt động. Bộ lập lịch sẽ quyết định tiểu trình nào sẽ được chạy tiếp theo. Nếu không có tiểu trình nào sẵn sàng để chạy, một tiểu trình idle (đang nghỉ) đặc biệt sẽ chạy.

Mã nguồn thực thi quá trình chuyển ngữ cảnh (context switch) được cài đặt bên trong threads/switch.S, đây là mã nguồn x86. Nó lưu trạng thái của tiểu trình đang chạy và khôi phục trạng thái của tiểu trình kế tiếp khi tiểu trình này nhận CPU.

Có thể sử dụng GDB để theo vết việc chuyển ngữ cảnh và hiểu rõ những gì xảy ra trong quá trình này.

b. Cấu trúc thread

Mỗi tiểu trình được thể hiện bằng một trang nhớ có kích thước 4KB. Các thông tin về cấu trúc của tiểu trình nằm ở phần đầu trang nhớ này. Phần còn lại được sử dụng cho stack của thread, stack sẽ được mở rộng lớn dần từ trên xuống (xem hình bên dưới).



Với thiết kế trang nhớ này, có 2 ràng buộc được đặt ra. Thứ nhất, kích thước cấu trúc của tiểu trình không được phép tăng quá nhiều, vì điều đó sẽ làm giảm kích thước của stack, stack sẽ không có đủ không gian để thực thi. Cấu trúc cơ bản của tiểu trình chỉ có kích thước vài byte và nó chỉ nên ở dưới mức 1 kB.

Thứ hai, stack của tiểu trình không được tăng lên quá lớn. Nếu stack bị tràn, nó sẽ làm gián đoạn quá trình thực thi của tiểu trình. Do đó, các hàm kernel cài đặt không nên cấp phát các cấu trúc có kích thước lớn hoặc sử dụng mảng như các biến tĩnh cục bộ.

Mỗi tiểu trình có các thành phần sau:

- tid_t tid: Định danh của tiểu trình – đây là giá trị đặc trưng của tiểu trình, mỗi tiểu trình có một tid duy nhất trong suốt

vòng đời của nó. tid có kiểu số nguyên (int) và mỗi tiểu trình sẽ nhận giá trị tid tăng dần, bắt đầu bằng 1 ở tiến trình khởi đầu.

- enum thread_status status: Trạng thái của tiểu trình, gồm các trạng thái sau: THREAD_RUNNING, THREAD_READY, THREAD_BLOCKED, THREAD_DYING.
- char name[16]: Tên của tiểu trình
- uint8_t *stack: Stack của tiểu trình
- int priority: Độ ưu tiên của tiểu trình, có giá trị từ PRI_MIN (0) đến PRI_MAX (63). Giá trị càng nhỏ ứng với độ ưu tiên càng thấp, 0 là độ ưu tiên thấp nhất, 63 là độ ưu tiên cao nhất. Pintos hiện tại không sử dụng các giá trị độ ưu tiên này, tuy nhiên, khi thực hiện nội dung thực hành 1, cần sử dụng các giá trị ưu tiên này để cài đặt thuật toán lập lịch theo độ ưu tiên.
- struct list_elem allelem: Thành phần này được sử dụng để liên kết tiểu trình với danh sách tất cả các tiểu trình. Mỗi tiểu trình sẽ được thêm vào danh sách khi nó được tạo ra và xóa khỏi danh sách khi nó kết thúc.
- struct list_elem elem: Thành phần này được sử dụng để đưa tiểu trình vào trong danh sách liên kết, đó có thể là ready_list (hàng đợi các tiểu trình sẵn sàng chạy các tiểu

trình) hoặc danh sách các tiểu trình đang đợi semaphore trong hàm sema_down().

- uint32_t *pagedir: Bảng trang của tiến trình (nếu là tiến trình người dùng). Chỉ sử dụng thành phần này ở nội dung thực hành 2 và 3.
- unsigned magic: Luôn được thiết lập giá trị là THREAD_MAGIC, giá trị này được định nghĩa trong threads/thread.c và được dùng để phát hiện lỗi tràn stack (khi stack bị tràn, giá trị này sẽ thay đổi).

Khi thực hiện các nội dung thực hành, có thể thêm các thành phần khác cũng như thay đổi hoặc xóa các thành phần hiện tại của cấu trúc tiểu trình.

c. Các hàm của tiểu trình

Mã nguồn tập tin threads/thread.c đã cài đặt sẵn một số hàm public để hỗ trợ việc thực thi tiểu trình:

- void thread_init (void): Khởi tạo tiểu trình hệ thống
- void thread_start (void): Khởi chạy tiểu trình
- void thread_tick (void): Timer interrupt sẽ gọi hàm này mỗi khi timer tick.
- tid_t thread_create (const char *name, int priority, thread_func *func, void *aux): Khởi tạo tiểu trình với các tham số gồm tên, độ ưu tiên, hàm thực thi của tiểu trình và tham số truyền cho hàm thực thi. Hàm thread_create sẽ trả

về tid của tiểu trình nếu được tạo thành công. Tiểu trình sẽ bắt đầu thực thi sau khi được tạo.

- void thread_block (void): Chuyển tiểu trình từ trạng thái running sang trạng thái block.
- void thread_unblock (struct thread *thread): Chuyển tiểu trình từ trạng thái block sang trạng thái ready.
- struct thread *thread_current (void): Trả về tiểu trình đang chạy.
- tid_t thread_tid (void): Trả về id của tiểu trình đang chạy, tương đương với gọi thread_current ()->tid.
- const char *thread_name (void): Trả về tên của tiểu trình đang chạy, tương đương với gọi thread_current ()->name.
- void thread_exit (void) NO_RETURN: Thoát khỏi tiểu trình hiện tại.
- void thread_yield (void): Trả quyền cấp phát CPU về cho bộ lập lịch, để chọn tiểu trình mới thực thi. Tiểu trình mới này vẫn có thể là tiểu trình đang chạy.
- void thread_foreach (thread_action_func *action, void *aux): Duyệt qua tất cả các tiểu trình và gọi thực thi hàm action(t, aux) cho từng tiểu trình. Hàm action là một hàm có nguyên mẫu hàm như sau: void thread_action_func (struct thread *thread, void *aux), hàm này sẽ được tiểu trình thực thi.

Các hàm sau đây đã được thiết kế sẵn, nhưng chưa được cài đặt. Chúng sẽ phải được cài đặt để hoàn thành bộ định thời theo độ ưu tiên và bộ định thời multilevel feedback queue:

- int thread_get_priority (void): Lấy độ ưu tiên của tiểu trình.
- void thread_set_priority (int new_priority): Gán độ ưu tiên mới cho tiểu trình.
- int thread_get_nice (void): Lấy giá trị nice của tiểu trình.
- void thread_set_nice (int new_nice): Gán giá trị nice mới cho tiểu trình.
- int thread_get_recent_cpu (void): Lấy giá trị recent CPU.
- int thread_get_load_avg (void): Lấy giá trị load average.

d. Chuyển đổi tiểu trình (thread switching)

Hàm schedule() được sử dụng cho việc chuyển đổi tiểu trình. Hàm này chỉ được sử dụng bên trong threads/thread.c và chỉ được gọi bởi 3 hàm cần thực thi chuyển đổi tiểu trình là: thread_block(), thread_exit() và thread_yield().

7.2.1.5 Đồng bộ

Pintos cung cấp nhiều cơ chế để hỗ trợ thực hiện đồng bộ như: disabling interrupts, semaphore, lock, … Chi tiết về các cơ chế khác có thể xem tại website của Pintos.

7.2.1.6 Cấp phát bộ nhớ

Pintos có 2 cách cấp phát bộ nhớ: cấp phát bộ nhớ theo trang và cấp phát bộ nhớ theo khối (bất kỳ kích cỡ nào).

a. Cấp phát theo trang

Cơ chế cấp phát theo trang được khai báo trong threads/palloc.h. Nó thường được sử dụng để cấp phát một trang nhớ tại một thời điểm, nhưng cũng có thể được sử dụng để cấp phát nhiều trang liên tiếp trong một lần. Quá trình cấp phát này chia bộ nhớ thành 2 phần (gọi là pool), gồm có kernel pool và user pool.

Thông thường mỗi pool sẽ được chia một nửa bộ nhớ, nhưng tỷ lệ này có thể thay đổi thông qua tham số -ul. User pool được dùng cho việc cấp phát các tiến trình của người dùng còn kernel pool sử dụng cho tất cả các yêu cầu cấp phát khác.

b. Cấp phát theo khối (bock allocator)

Cơ chế cấp phát theo khối được định nghĩa trong threads/malloc.h. Nó có thể cấp phát bất kỳ khối bộ nhớ nào. Việc cấp phát theo khối được dựa trên cấp phát theo trang. Mỗi khối được cấp phát đều được lấy ra từ kernel pool.

Việc cấp phát theo khối sử dụng 2 hướng tiếp cận khác nhau. Hướng tiếp cận thứ nhất chia thành các khối có kích thước 1KB hoặc nhỏ hơn (khoảng ¼ kích thước trang). Kích thước vùng nhớ

được cấp phát được làm tròn thành lũy thừa của 2 gần nhất, hoặc là 16 byte. Sau đó, các vùng nhớ này sẽ được gom nhóm thành 1 trang.

Hướng tiếp cận thứ hai chia các khối có kích thước lớn hơn 1 KB. Các khối này sẽ được gom nhóm thành trang gần nhất và tạo thành một nhóm các trang liên tục theo cơ chế cấp phát trang.

7.2.1.7 Bộ định thời multilevel feedback queue

a. Giới thiệu

Bộ định thời multilevel feedback queue gồm có nhiều hàng đợi khác nhau, mỗi hàng đợi sẽ chứa các tiểu trình có độ ưu tiên khác nhau. Bộ định thời này sẽ luôn chọn tiểu trình từ hàng đợi không trống có độ ưu tiên cao nhất. Nếu hàng đợi có độ ưu tiên cao nhất có nhiều tiểu trình, mỗi tiểu trình sẽ được định thời bằng giải thuật “round robin". Các phần của bộ định thời này đòi hỏi dữ liệu phải được cập nhật sau một khoảng thời gian nhất định (là bội số của timer ticks).

b. Tính toán số thực có dấu chấm động

Trong các phần sau, việc tính toán sẽ cần sử dụng các số thực, không phải số nguyên. Tuy nhiên, Pintos không hỗ trợ các phép toán với số thực có dấu chấm động. Trong mã nguồn Pintos đi kèm với tài liệu này, tập tin fixed-point.h nằm trong thư mục pintos/src/threads/ có chứa các cấu trúc và hàm cần thiết để biểu

diễn và tính toán số thực. Khi cài đặt, nên sử dụng kiểu fixed_point_t và các hàm có liên quan để biểu diễn các giá trị là số thực.

c. Niceness

Mỗi tiểu trình có một giá trị số nguyên gọi là nice, giá trị này xác định mức độ "mới" của tiểu trình so với các tiểu trình khác.

Giá trị nice bằng 0 không ảnh hưởng tới độ ưu tiên của tiểu trình. Giá trị nice dương (tối đa 20) sẽ làm giảm độ ưu tiên của tiểu trình và sẽ khiến nó từ bỏ giờ CPU mà nó nhận được. Ngược lại, giá trị nice âm (tối thiểu là -20) sẽ giúp tiểu trình có xu hướng nhận được CPU từ các tiểu trình khác.

Giá trị nice của tiểu trình đầu tiên là 0. Các tiểu trình khác sẽ có giá trị nice được kế thừa từ tiểu trình cha của nó. Các hàm liên quan đến nice sau đây cần được cài đặt (nguyên mẫu hàm của chúng được định nghĩa sẵn trong tập tin threads/thread.c):

• int thread_get_nice (void): Trả về giá trị nice của tiểu trình hiện tại.

• void thread_set_nice (int new_nice): Thiết lập giá trị nice mới cho tiểu trình hiện tại, đồng thời tính lại độ ưu tiên của tiểu trình dựa trên giá trị nice mới. Nếu tiểu trình đang chạy không có độ ưu tiên cao nhất, nó cần từ bỏ việc sử dụng CPU.

d. Xác định độ ưu tiên

Bộ định thời cần cài đặt có 64 độ ưu tiên, cũng đồng thời có 64 hàng đợi được đánh số từ 0 (PRI_MIN) đến 63 (PRI_MAX). Độ ưu tiên 0 là nhỏ nhất, 63 là cao nhất.

Độ ưu tiên của tiểu trình được xác định tại thời điểm khởi tạo. Sau đó, nó sẽ được xác định lại sau mỗi 4 chu kỳ xung clock. Độ ưu tiên mới được xác định bằng công thức:

priority = PRI_MAX − (recent_cpu/4) − (nice × 2)

Trong đó, recent_cpu là thời gian CPU ước tính mà tiểu trình đã sử dụng gần đây (cách xác định recent_cpu sẽ được trình bày ở mục 5.5 kế tiếp) còn nice là giá trị nice của tiểu trình. Kết quả của biểu thức trên trên sẽ được làm tròn xuống số nguyên gần nhất. Các giá trị ¼ và 2 trong biểu thức trên được xác định thông qua thực nghiệm. Kết quả độ ưu tiên tính được phải luôn nằm trong phạm vi từ PRI_MIN đến PRI_MAX.

Công thức trên được thiết kế để các tiểu trình đã nhận được CPU gần đây sẽ có độ ưu tiên thấp hơn ở lần lập lịch kế tiếp. Đây chính là chìa khóa để ngăn chặn tình trạng bỏ đói (starvation): một tiểu trình chưa được nhận CPU gần đây sẽ có recent_cpu bằng 0, sẽ luôn đảm bảo sẽ được nhận CPU sớm.

e. Xác định recent CPU

Giá trị recent_cpu được sử dụng để đo lường lượng giờ CPU mỗi tiểu trình đã nhận được gần đây. Để ước tính giá trị này, hàm

exponentially weighted moving average được sử dụng, hàm này có dạng tổng quát:

$x(0) = f(0)$

$x(t) = a \times x(t - 1) + f(t)$

$a = k/(k + 1)$

Trong biểu thức này, x(t) là moving average tại thời điểm $t \geq 0$, f(t) là hàm trung bình và k điều khiển tỷ lệ. Khai triển hàm này sẽ được các biểu thức sau:

$x(1) = f(1)$

$x(2) = a \times f(1) + f(2)$

$x(3) = a2 \times f(1) + a \times f(2) + f(3)$

$x(4) = a3 \times f(1) + a2 \times f(2) + a \times f(3) + f(4)$

Giá trị f(t) có hệ số 1 tại thời điểm t, hệ số a tại thời điểm t + 1, hệ số a2 tại thời điểm t + 2, tương tự cho các thời điểm khác.

Giá trị khởi tạo của recent_cpu là 0 khi tiểu trình đầu tiên được tạo. Nếu là các tiểu trình khác, giá trị này sẽ là giá trị recent_cpu của tiểu trình cha. Mỗi lần một ngắt timer diễn ra, recent_cpu sẽ tăng lên 1 đối với tiểu trình đang thực thi. Thêm vào đó, mỗi giây, giá trị recent_cpu của mỗi tiểu trình được xác định (bất kể tiểu trình đó đang chạy, ready hay blocked) qua công thức sau:

recent_cpu = (2 × load_avg)/(2 × load_avg + 1) × recent_cpu + nice

Trong công thức này, load_avg là moving average của số tiểu trình sẵn sàng được chạy (xem cách xác định ở phần 5.6).

Giá trị recent_cpu có thể âm nếu tiểu trình có nice âm, khi đó không được làm tròn recent_cpu về 0.

Khi thực hiện biểu thức này, cần tính giá trị (2 × load_avg)/(2 × load_avg + 1) trước, sau đó nhân với recent_cpu. Thực hiện khác thứ tự có thể dẫn đến lỗi tràn bộ nhớ.

Cần phải cài đặt hàm thread_get_recent_cpu() trong threads/thread.c:

• int thread_get_recent_cpu(void): Trả về giá trị recent_cpu của tiểu trình hiện tại (tính theo phần trăm), được làm tròn đến số nguyên gần nhất.

f. Xác định load average

Giá trị load_avg, thường được gọi là thời gian tải trung bình, dùng để ước tính số tiểu trình trung bình được thực thi trong vòng một phút trước đó. Giống như recent_cpu, giá trị này cũng là exponentially weighted moving average.

Tuy nhiên, khác với độ ưu tiên và recent_cpu, load_avg được xác định trên phạm vi toàn hệ thống, không phải đặc trưng theo từng tiểu trình. Tại thời điểm hệ thống bắt đầu hoạt động, giá trị này được khởi tạo bằng 0. Mỗi một giây trôi qua, nó được cập nhật lại theo công thức sau:

load_avg = (59/60) × load_avg + (1/60) × ready_threads

Trong công thức này, ready_threads là tổng số tiểu trình đang chạy hoặc đang ở trạng thái sẵn sàng (ready) tại thời điểm cập nhât.

Giá trị này được trả về thông qua hàm thread_get_load_avg(), hàm này đã được định nghĩa trong threads/thread.c nhưng chưa được cài đặt:

• int thread_get_load_avg(void): Trả về thời gian tải trung bình của hệ thống (tính theo phần trăm), làm tròn tới số nguyên gần nhất.

g. Những vấn đề cần lưu ý

- Khi bộ định thời multilevel feedback queue được sử dụng, không cho phép tích lũy độ ưu tiên.
- Khi bộ định thời multilevel feedback queue được sử dụng, tiểu trình sẽ không được trực tiếp tác động lên độ ưu tiên của chính nó. Tham số độ ưu tiên đưa vào hàm thread_create() sẽ được bỏ qua, cũng như việc gọi thread_set_priority() sẽ không có tác dụng nào, đồng thời gọi hàm thread_get_priority() sẽ trả về giá trị độ ưu tiên được xác định bởi bộ định thời.
- Do các biểu thức tính toán đều xuất hiện dấu thập phân, nên sử dụng kiểu và các hàm tính toán dấu chấm động (đã được định nghĩa trong tập tin fixed-point.h).

7.2.2 NỘI DUNG 2: CHƯƠNG TRÌNH NGƯỜI DÙNG (USER PROGRAM)

7.2.2.1 Mục tiêu

Phần thực hành này giúp sinh viên:

- Hiểu và thực hiện việc truyền tham số thực thi khi khởi tạo tiến trình trong Pintos.
- Hiểu, sử dụng và cài đặt được các system call quản lý tiến trình và quản lý tập tin trong Pintos.

7.2.2.2 Sinh viên chuẩn bị

- Đọc trước nội dung thực hành ở phần 7.2.2.3
- Đọc trước nội dung về hệ thống tập tin ở phần 7.2.2.4
- Ôn lại các khái niệm về system call. Đọc trước phần 7.2.2.5 để hiểu cách chương trình người dùng được thực thi trên Pintos.
- Đọc trước nội dung về cấu trúc bộ nhớ ảo ở phần 7.2.2.6

7.2.2.3 Nội dung thực hành

Yêu cầu dành cho phần này là cài đặt và mở rộng các chức năng liên quan đến chương trình người dùng đã có sẵn trên Pintos. Hiện tại Pintos có khả năng nạp và chạy các chương trình người

dùng nhưng các chương trình này không nhận tham số dòng lệnh cũng như không thể gọi các system call.

a. Truyền tham số thực thi

Trong phiên bản Pintos hiện tại, hàm *process_execute (char *file_name)* trong tập tin userprog/process.c được sử dụng để tạo mới tiến trình ở cấp độ người dùng. **Cần cài đặt truyền tham số** cho hàm này, ví dụ như cung cấp hai tham số cho hàm trên lần lượt là ["ls", "-ahl"] thì lúc thực thi sẽ gọi hàm dưới dạng "process_execute("ls -ahl")". Hai tham số này sẽ được chương trình người dùng sử dụng thông qua argc và argv.

Hiện tại, do việc truyền tham số chưa được cài đặt, cho nên tất cả các chương trình đều sẽ gặp lỗi khi truy xuất vào biến argv[0] (biến này chứa tên chương trình và được dùng để khởi chạy chương trình). Chỉ khi nào việc cài đặt chức năng này hoàn tất thì các chương trình ở cấp độ người dùng mới hoạt động được.

b. Các system call điều khiển tiến trình

Pintos chỉ hỗ trợ duy nhất một system call là exit. Yêu cầu được đặt ra ở phần này là **cài đặt thêm các system call sau**: halt, exec, wait và practice. Mỗi syscall được khai báo và cài đặt trong thư mục thư viện cấp độ người dùng, cụ thể là tập tin lib/user/syscall.c. Tập tin này chứa các thông tin về tham số của từng system call và cách chuyển xử lý về chế độ kernel. **Các hàm**

xử lý system call của kernel cần phải được cài đặt trong tập tin userprog/syscall.c.

Chức năng cụ thể của từng system call cần được cài đặt như sau:

- halt: Tắt (dừng) hệ thống.

- exec: Tạo một chương trình mới bằng *process_execute()* (Pintos không hỗ trợ fork. Về bản chất thì exec của Pintos tương tự như fork của Linux).

- wait: Chờ một tiến trình con thoát.

- practice: Thêm 1 vào tham số đầu tiên, sau đó trả về kết quả. System call này được sử dụng cho mục đích thử nghiệm giống như tên gọi của nó.

Để cài đặt các system call, trước hết cần tìm hiểu thao tác đọc và ghi một cách an toàn các dữ liệu đang có trong không gian bộ nhớ ảo của các tiến trình cấp độ người dùng. Các tham số của system call nằm trong stack của tiến trình người dùng, ngay bên trên con trỏ stack. Chú ý rằng, nếu con trỏ stack không hợp lệ khi một chương trình gọi một system call, nhân của hệ điều hành sẽ không thể bị crash. Một số tham số của system call là con trỏ chỉ đến buffer bên trong không gian địa chỉ của tiến trình người dùng, những buffer này có thể không hợp lệ bất cứ lúc nào nên cần phải cẩn thận khi thực hiện các thao tác với chúng.

Cần phải xử lý trường hợp một system call không được thực thi trọn vẹn vì lỗi truy cập bộ nhớ không hợp lệ. Các lỗi này bao gồm con trỏ null, con trỏ không hợp lệ (trỏ đến vùng nhớ chưa được ánh xạ) hoặc con trỏ của vùng nhớ nhân hệ điều hành. Đặc biệt lưu ý rằng, có thể xảy ra trường hợp một vùng nhớ 4 byte chứa 2 byte nhớ hợp lệ và 2 byte nhớ không hợp lệ, nếu vùng nhớ này nằm trên biên của bảng phân trang. Khi xảy ra trường hợp này, cần phải chấm dứt (terminate) tiến trình người dùng.

c. Các system call liên quan đến tập tin

Yêu cầu ở phần này là **cài đặt thêm các system call cần cho việc sử dụng tập tin**, cụ thể là các thao tác: create, remove, open, filesize, read, write, seek, tell và close. Hiện tại Pintos đã có sẵn một hệ thống quản lý tập tin đơn giản. Lưu ý rằng, khi cài đặt, chỉ **cần gọi các hàm thích hợp có sẵn** trong thư viện tập tin hệ thống, **không cần phải tự cài đặt** bất cứ thao tác với tập tin nào khác.

Hệ thống tập tin của Pintos không phải là thread-safe. Do đó, khi cài đặt các system call trên, cần đảm bảo là chúng sẽ không được gọi nhiều hàm xử lý tập tin cùng lúc. Đến nội dung thực hành 3, một số hàm đồng bộ phức tạp hơn sẽ được thêm vào hệ thống tập tin của Pintos, nhưng ở phần này, chỉ cần sử dụng một biến lock toàn cục để đảm bảo thread safety. Lưu ý là không được chỉnh sửa thư mục **/filesys** trong phần này.

Một điểm nữa cần lưu ý là khi một tiến trình đang chạy, cần phải đảm bảo rằng không có bất kỳ đối tượng nào khác có thể chỉnh sửa tập tin thực thi (đang lưu trên đĩa) của nó. Hai hàm *file_deny_write()* và *file_allow_write()* có thể được sử dụng để thực hiện điều này.

Chú ý: Mã nguồn đã thực hiện ở nội dung thực hành 2 sẽ được tiếp tục sử dụng để cài đặt và mở rộng ở nội dung thực hành 3. Do đó, kết quả kiểm thử hoạt động của các chức năng ở nội dung thực hành 3 sẽ phụ thuộc vào việc các system call được cài đặt như thế nào ở bài này.

7.2.2.4 Giới thiệu về hệ thống tập tin của Pintos

Hệ thống quản lý tập tin của Pintos có một số hạn chế như sau:

- Thực thi nội tại không đồng bộ: Các tiến trình thực thi đồng thời có thể tương tác với nhau. Cần phải sử dụng cơ chế đồng bộ để đảm bảo tại một thời điểm chỉ có 1 tiến trình được thực thi mã nguồn hệ thống.
- Kích thước tập tin là cố định tại thời điểm tạo. Số lượng tập tin có thể tạo cũng bị giới hạn.
- Dữ liệu tập tin là liên tục (nằm trên các sector liên tiếp trên ổ đĩa).
- Không có thư mục con.
- Tên tập tin bị giới hạn trong 14 ký tự.

- Nếu một thao tác hệ thống bị lỗi trong khi đang thực thi, nó có thể làm hỏng ổ đĩa và không có cách nào khôi phục được.

Bên cạnh các hạn chế trên cần chú ý một điểm quan trong sau: Lệnh filesys_remove() đã được cài đặt và thực thi giống như lệnh này trên môi trường Unix. Điều này có nghĩa, nếu một tập tin đang được mở, khi nó bị xóa đi, tất cả các khối được cấp phát cho nó sẽ không được tự động giải phóng, một số tiểu trình (đã mở nó) vẫn có thể truy cập nó, cho đến khi tiểu trình cuối cùng đóng nó lại.

7.2.2.5 Chương trình người dùng được thực thi như thế nào?

Pintos có thể chạy các chương trình viết bằng ngôn ngữ C thông thường, miễn là chúng có thể chạy được với lượng bộ nhớ mà Pintos có thể cung cấp được và chỉ sử dụng các system call đã được cài đặt.

Với phần thực hành này, hàm malloc() sẽ không thể được cài đặt bởi không có system call nào được phép cấp phát bộ nhớ. Pintos cũng không thể chạy các chương trình sử dụng phép tính dấu chấm động, bởi vì kernel sẽ không thể lưu và khôi phục các giá trị dấu chấm động khi chuyển đổi qua lại giữa các tiểu trình.

Thư mục src/examples chứa một số chương trình mẫu. Tập tin Makefile trong thư mục này biên dịch các ví dụ mẫu được cung cấp, chúng có thể được chỉnh sửa và biên dịch để sử dụng trong

các chương trình khác. Lưu ý là một vài ví dụ chỉ có thể chạy nếu nội dung thực hành 3 được thực hiện xong.

Pintos cũng có thể nạp các tập tin thực thi có định dạng ELF (thường được sử dụng trên Linux, Solaris và nhiều hệ điều hành khác) bằng chương trình loader được cung cấp sẵn trong userprog/process.c.

Cần chú ý rằng, nếu không sao chép chương trình test vào hệ thống tập tin, Pintos sẽ không thể thực thi bất cứ chương trình mang tính thực tế nào.

7.2.2.6 Cấu trúc của bộ nhớ ảo (Virtual memory layout)

Bộ nhớ ảo trong Pintos được chia thành 2 vùng: user virtual memory (bộ nhớ ảo của người dùng) và kernel virtual memory (bộ nhớ ảo của kernel). Bộ nhớ của ảo người dùng bắt đầu từ địa chỉ ảo 0 đến địa chỉ PHYS_BASE được định nghĩa trong threads/vaddr.h và có giá trị mặc định là 0xc0000000 (3 GB). Bộ nhớ ảo của kernel chiếm phần không gian còn lại của bộ nhớ ảo, từ địa chỉ PHYS_BASE cho đến 4 GB.

Bộ nhớ ảo của người dùng được xác định theo từng tiến trình. Khi nhân hệ điều hành thực hiện việc chuyển từ tiến trình này sang tiến trình khác, nó đồng thời cũng chuyển đổi không gian bộ nhớ ảo bằng cách thay đổi cấu trúc bảng trang của bộ xử lý (hàm pagedir_activate() trong userprog/pagedir.c).

Ngược lại, bộ nhớ ảo kernel có tính toàn cục. Nó cũng được ánh xạ theo cách thức tương tự, không có sự phân biệt giữa tiến trình người dùng hoặc tiến trình hệ thống. Trong Pintos, bộ nhớ ảo kernel được ánh xạ 1-1 với bộ nhớ vật lý, bắt đầu từ địa chỉ PHYS_BASE. Điều này có nghĩa là địa chỉ ảo PHYS_BASE sẽ truy cập vào địa chỉ vật lý 0, địa chỉ ảo PHYS_BASE + 0x1234 sẽ truy cập vào địa chỉ vật lý 0x1234 và cứ như vậy cho đến hết bộ nhớ vật lý.

Lưu ý rằng, một chương trình người dùng chỉ có thể truy cập vào bộ nhớ ảo của người dùng. Bất kỳ một truy cập nào vào bộ nhớ ảo của kernel đều sẽ gây ra một lỗi trang và phải được xử lý bởi hàm page_fault() trong userprog/exception.c đồng thời tiến trình sẽ phải kết thúc. Tiến trình hoặc tiểu trình tạo ra bởi nhân hệ điều hành có thể truy cập cả hai bộ nhớ ảo. Tuy nhiên, nếu có bất kỳ sự truy cập vào một vùng nhớ chưa được ánh xạ từ các đối tượng này thì cũng sẽ phát sinh một lỗi trang, tương tự như với chương trình người dùng.

7.2.3 NỘI DUNG 3: BỘ NHỚ ẢO (VIRTUAL MEMORY)

7.2.3.1 Mục tiêu

Phần thực hành này giúp sinh viên:

- Hiểu các khái niệm liên quan đến bộ nhớ ảo, cơ chế phân trang. Thực hiện một thuật toán phân trang trong Pintos.
- Hiểu và cài đặt được các system call quản lý việc ánh xạ bộ nhớ trong Pintos.
- Hiểu và cài đặt được các thao tác quản lý stack và truy xuất bộ nhớ trong Pintos.

7.2.3.2 Sinh viên chuẩn bị

- Đọc trước nội dung thực hành ở phần 7.2.3.3
- Đọc trước nội dung về bộ nhớ ở phần 7.2.3.4
- Ôn lại các khái niệm về phân trang, các thuật toán phân trang

7.2.3.3 Nội dung thực hành

a. Phân trang (Paging)

Yêu cầu đầu tiên là **cài đặt việc phân đoạn các trang được nạp từ tập tin thực thi**. Các trang phải được nạp sao cho kernel của hệ điều hành có thể "bắt" được các lỗi trang của chúng. Trong

quá trình thay thế, các trang có sự thay đổi từ thời điểm nạp cần phải được lưu xuống swap. Lưu ý là các trang không có sự thay đổi, bao gồm cả các trang chỉ đọc, sẽ không được lưu xuống swap, bởi vì chúng luôn có thể đọc từ trong tập tin thực thi.

Cài đặt một thuật toán phân trang toàn cục có hiệu quả xấp xỉ LRU là yêu cầu thứ hai cần thực hiện trong phần này.

Để thực hiện các yêu cầu này, cần phải thay đổi mã nguồn chức năng nạp chương trình, cụ thể là vòng lặp trong hàm *load_segment()* trong tập tin userprog/process.c

Mỗi lần vòng lặp được thực thi, biến *page_read_bytes* nhận giá trị là số byte đọc được từ tập tin thực thi và biến *page_zero_bytes* nhận giá trị là số byte được khởi tạo bằng 0. Tổng của hai biến này luôn bằng PGSIZE (4,096). Việc xử lý một trang phụ thuộc vào giá trị của các biến này, cụ thể như sau:

- Nếu *page_read_bytes* bằng với PGSIZE, trang này sẽ là trang được yêu cầu từ underlying file khi nó được truy xuất lần đầu.
- Nếu *page_zero_bytes* bằng PGSIZE, trang này sẽ không cần phải đọc từ đĩa bởi vì nó chỉ toàn 0. Với trường hợp này, cần tạo một trang mới chứa toàn 0 khi lỗi trang đầu tiên xảy ra.

- Ngược lại, khi cả *page_read_bytes* và *page_zero_bytes* đều khác PGSIZE, một phần của trang sẽ được đọc từ underlying file và phần còn lại sẽ là 0.

b. Stack

Trong nội dung thực hành 2, stack chỉ là một trang đơn nằm ở phía trên cùng không gian bộ nhớ ảo của người dùng và các chương trình bị giới hạn trong vùng stack này. Yêu cầu đặt ra cho phần này là **cấp phát thêm các trang mới cho stack khi cần thiết, cụ thể là khi stack tăng lên vượt quá kích thước hiện tại của nó.**

Bên trong một system call hoặc khi có một lỗi trang xảy ra do chương trình của người dùng, có thể sử dụng thuộc tính *esp* của cấu trúc *struct intr_frame* (được truyền vào *syscall_handler()* hoặc *page_fault()*) để lấy giá trị hiện tại của con trỏ stack.

Khi cài đặt stack, có thể áp đặt một số giới hạn về kích thước của stack như cách mà phần lớn các hệ điều hành hiện tại đang thực hiện. Một vài hệ điều hành cho phép người dùng có thể điều chỉnh giá trị này, chẳng hạn như lệnh ulimit trên một số hệ thống Unix. Trên nhiều hệ thống GNU/Linux, kích thước giới hạn này mặc định là 8 MB.

Trang đầu tiên của stack có thể được cấp phát và khởi tạo thông qua tham số dòng lệnh tại thời điểm nạp, điều này sẽ giúp nó không cần phải chờ khi có lỗi xảy ra.

Tất cả các trang stack đều được xem là có thể thay thế. Khi một trang bị thay thế, nó cần phải được lưu xuống vùng swap.

c. Ánh xạ tập tin và bộ nhớ

Trong phần này, cần **cài đặt các system call** sau:

*1. mapid_t mmap (int fd, void *addr):* Ánh xạ tập tin đang mở là *fd* vào trong không gian bộ nhớ ảo của tiến trình. Toàn bộ tập tin sẽ được ánh xạ liên tục vào các trang ảo bắt đầu từ địa chỉ *addr*.

Nếu kích thước của tập tin không phải là bội số của PGSIZE thì cần phải thêm một số byte ở cuối trang được ánh xạ vào cuối tập tin. Các byte này sẽ được gán giá trị là 0 khi trang gặp lỗi và được bỏ qua khi trang được lưu trở lại xuống đĩa.

Nếu system call này được thực thi thành công, nó sẽ trả về một giá trị gọi là mapping ID – giá trị duy nhất được dùng để xác định việc ánh xạ bên trong tiến trình. Ngược lại, nếu thất bại, nó sẽ trả về -1, có nghĩa là sẽ không có mapping id hợp lệ và ánh xạ đang có bên trong tiến trình sẽ được giữ nguyên.

Trong các trường hợp sau, một lời gọi mmap sẽ thất bại: tập tin fd có độ dài 0 byte; addr không được page-aligned; addr bằng 0 (do Pintos coi trang ảo 0 là không thể được ánh xạ); phạm vi các

trang sắp được ánh xạ chồng lên các trang đã được ánh xạ, bao gồm stack hoặc các trang được ánh xạ tại thời điểm nạp. Các đặc tả tập tin 0 và 1, đại diện cho các thao tác nhập xuất từ dòng lệnh, cũng sẽ không được ánh xạ.

2. void munmap (mapid_t mapping): Hủy thao tác ánh xạ được biểu diễn bởi giá trị *mapping*, đây là mapping ID được trả về bởi system call *mmap* ở trên.

Tất cả các ánh xạ đều ngầm định bị hủy khi tiến trình kết thúc (bằng system call exit hay do bất cứ nguyên nhân nào). Khi một ánh xạ bị hủy, chỉ những trang được lưu bởi tiến trình sẽ được lưu lại xuống tập tin, các trang khác không cần phải lưu xuống, chúng sẽ được xóa khỏi danh sách các trang ảo của tiến trình.

Việc đóng hoặc xóa một tập tin sẽ không tự động hủy đi bất cứ ánh xạ nào của nó. Khi một ánh xạ được tạo ra, nó sẽ có hiệu lực cho đến khi *unmap* được gọi hoặc tiến trình kết thúc. Có thể sử dụng hàm *file_reopen* để lấy giá trị tham chiếu đến tập tin ứng với mỗi ánh xạ của nó.

Nếu hai hoặc nhiều tiến trình ánh xạ cùng một tập tin, không cần phải đảm bảo chúng toàn vẹn dữ liệu. Hệ điều hành Unix xử lý tình huống này bằng cách thực hiện hai ánh xạ cùng chia sẻ chung trang vật lý, nhưng system call *mmap* sẽ cần có thêm một tham số để cho phép người dùng thiết lập trang ánh xạ là chung hay riêng.

d. Truy xuất bộ nhớ người dùng

Một tiến trình người dùng có thể truy xuất các trang mà nội dung của chúng đang nằm trong tập tin hoặc không gian swap bằng cách truyền địa chỉ vào các system call.

Khi truy xuất bộ nhớ người dùng, các mã nguồn ở phần nhân được cài đặt phải có khả năng xử lý lỗi trang hoặc ngăn chặn việc xảy ra lỗi trang. Có thể **cài đặt phần này bằng cách thực hiện một trong hai hướng** chỉnh sửa mã nguồn sau: thay đổi các hàm ở các tập tin userprog/pagedir.c và threads/vaddr.h hoặc chỉ thay đổi hàm page_fault() trong userprog/exception.c.

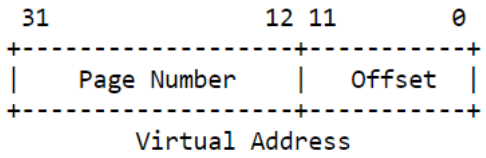
Việc ngăn chặn lỗi trang xảy ra đòi hỏi sự kết hợp giữa các mã nguồn xử lý truy cập và chọn trang "nạn nhân". Để thực hiện điều này, có thể mở rộng bảng khung trang để lưu lại thông tin trang nào chứa khung không được chọn làm "nạn nhân".

7.2.3.4 Các vấn đề liên quan đến bộ nhớ

a. Trang

Một trang, hay còn gọi là trang ảo, là một vùng nhớ liên tục có kích thước 4096 byte (giá trị này cũng được gọi là kích thước trang). Một trang phải luôn được page-aligned, tức là luôn bắt đầu từ địa chỉ ảo là một số chia hết cho kích thước trang. Do đó, một địa chỉ ảo 32 bit có thể chia thành 2 phần địa chỉ: 20 bit biểu diễn

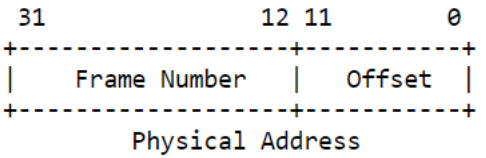
số trang và 12 bit biểu diễn độ dời trang (offset), như hình minh họa sau:



Mỗi tiến trình có một một tập các trang dành cho riêng nó, thường nằm từ địa chỉ ảo 0xc0000000. Các trang của kernel thì ngược lại, mang tính toàn cục. Kernel có thể truy cập cả các trang người dùng và trang của kernel, nhưng tiến trình người dùng chỉ có thể truy cập các trang của nó mà thôi.

b. Khung trang

Một khung trang, hay còn gọi là một khung vật lý, là một vùng nhớ vật lý liên tục. Giống như trang ảo, khung trang có kích thước khung (bằng với kích thước trang) và địa chỉ cũng phải được page-aligned. Và do đó, một địa chỉ vật lý 32 bit cũng được chia thành 2 phần: 20 bit biểu diễn số khung và 12 bit biểu diễn độ dời khung (frame offset), như minh họa ở hình sau:



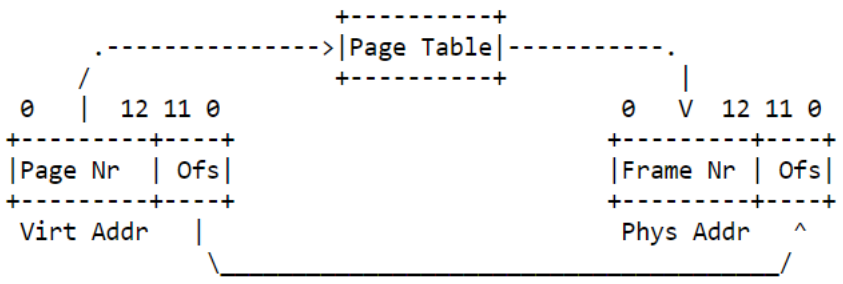
Kiến trúc x86 không cung cấp bất cứ cách nào để có thể trực tiếp truy cập vùng nhớ tại một địa chỉ nhớ vật lý. Pintos xử lý việc

này bằng cách ánh xạ bộ nhớ ảo kernel trực tiếp lên bộ nhớ vật lý: trang đầu tiên của bộ nhớ ảo kernel được ánh xạ với khung đầu tiên của bộ nhớ vật lý, trang thứ hai ứng với khung thứ hai, tương tự như vậy cho các trang và khung tiếp theo.

c. Bảng trang

Trong Pintos, bảng trang là một cấu trúc dữ liệu mà CPU dùng để chuyển đổi một địa chỉ ảo sang một địa chỉ vật lý, tức là từ một trang sang một khung trang. Cấu trúc của một bảng trang được quyết định bởi kiến trúc x86. Pintos đã cài đặt sẵn các mã nguồn quản lý bảng trang trong Pintos tập tin pagedir.c.

Sơ đồ bên dưới minh họa mối liên hệ giữa trang và khung trang. Địa chỉ ảo nằm ở bên trái, bao gồm số trang và độ dời. Bảng trang sẽ chuyển đổi số trang này thành một khung trang, sau đó kết hợp với độ dời để xác định địa chỉ vật lý (nằm ở bên phải).



d. Swap Slots

Swap slot là một vùng không gian liên tục có kích thước của một trang nằm trên ổ đĩa của phân vùng swap.

e. Tổng quan về quản lý tài nguyên

Để quản lý và đảm bảo quá trình truy xuất bộ nhớ diễn ra ổn định và an toàn, cần cài đặt một số cấu trúc dữ liệu sau:

- Supplemental page table: Cho phép việc xử lý lỗi trang bằng cách sử dụng supplementing page table.
- Frame table (bảng khung trang): Sử dụng cấu trúc này để tăng hiệu quả của việc lựa chọn trang thay thế.
- Swap table: Lưu vết việc sử dụng swap slot.
- Table of file mappings (bảng ánh xạ tập tin): Sử dụng để lưu vết và đánh dấu trang nào đã được tập tin nào ánh xạ vào trong bộ nhớ ảo.

Có thể kết hợp các cấu trúc này lại với nhau, miễn là thuận tiện cho việc cài đặt. Đối với mỗi cấu trúc, cần xác định rõ các thông tin nào sẽ được lưu trữ, mức độ truy cập và phạm vi lưu trữ của từng cấu trúc.

Quản lý Supplemental Page Table

Supplemental page table bổ sung cho bảng trang các thông tin về mỗi trang. Điều này cần thiết do mỗi trang bị giới hạn bởi định dạng của bảng trang. Supplemental page table được sử dụng cho ít nhất 2 mục đích. Thứ nhất, khi có lỗi trang, kernel sẽ tìm trang ảo xuất hiện lỗi ở trên supplemental page table để xác định đâu là nguyên nhân. Thứ hai, kernel sẽ tham khảo supplemental page

table khi có một tiến trình kết thúc để quyết định là tài nguyên nào sẽ được giải phóng.

Có hai hướng cơ bản để thiết kế supplemental page table: theo đoạn hoặc theo trang. Cũng có thể sử dụng bảng trang để lưu vết các thành phần của supplemental page table. Để làm được điều này, cần phải thay đổi mã nguồn bảng trang của Pintos (trong tập tin pagedir.c).

Phần quan trọng nhất của supplemental page table là xử lý lỗi trang. Trong phần thực hành này, một lỗi trang chỉ thể hiện rằng có một trang cần phải lấy ra từ tập tin hoặc phân vùng swap. Do đó việc xử lý lỗi trang cần phải được điều chỉnh để giải quyết các vấn đề này. Việc xử lý lỗi trang nên được cài đặt bằng cách chỉnh sửa hàm page_fault() trong userprog/exception.c.

- Quản lý bảng khung trang

Bảng khung trang gồm nhiều phần tử, mỗi phần tử ứng với một khung trang chứa một trang của người dùng. Mỗi phần tử chứa một con trỏ chỉ đến trang. Bảng khung trang cho phép Pintos xử lý hiệu quả việc chọn trang thay thế (khi không có khung nào trống).

Khung được sử dụng cho trang người dùng được lấy từ "user pool" thông qua việc gọi palloc_get_page(PAL_USER). PAL_USER là tham số bắt buộc phải dùng, để tránh việc cấp phát từ "kernel pool".

Chức năng quan trọng nhất của bảng khung trang là chọn một khung chưa được sử dụng. Điều này rất đơn giản khi có một khung trống. Khi không có khung nào trống, một khung phải được giải phóng bằng cách thay thế các trang của khung đó.

- Quản lý Swap Table

Swap table quản lý và lưu thông tin các swap slot đang sử dụng cũng như đang trống. Nó phải cho phép chọn một swap slot chưa được sử dụng vào việc thay thế một trang. Nó cũng phải cho phép giải phóng một swap slot khi dữ liệu từ trang của nó được đọc trở lại hoặc khi tiến trình chứa trang đó kết thúc.

TÀI LIỆU THAM KHẢO

[1] Project Pintos: https://web.stanford.edu/class/cs140/projects/pintos/pintos.html, truy cập ngày 10/1/2019.

[2] CS162 - Operating Systems and Systems Programming: inst.eecs.berkeley.edu/~cs162/, truy cập ngày 10/1/2019.