Đồ án 3. Đa Chương & Đồng Bộ

Môi trường thực hành và hướng dẫn cài đặt:

Môi trường thực hành

- Máy ảo VM Ware xem hướng dẫn xử dụng trong file **[VMWare].GioiThieuVMWare**
- Hệ Điều Hành Ubuntu 10.10 (http://www.ubuntu.com/desktop/get-ubuntu/download)
- Source code nachos 3.4 **đã được chỉnh sửa** (sẽ cung cấp trên moodle).

Hướng dẫn cài đặt xem trong các file sau:

- **[1] Bien dich va Cai dat Nachos.**
- **[Film] Huong Dan Install Nachos**

Phần 1. Hiểu về đa chương

Chương trình người dùng trên HĐH thật sẽ chạy dưới dạng tiến trình (process), còn chương trình người dùng của Nachos chạy trong một tiểu trình (thread). Do đó, để thực thi đa chương ta cần biết cách tạo thread và hiểu về sự điều phối các thread trong Nachos.

Chúng ta cần viết **system call Exec** để thực thi một chương trình mới trong process hiện tại. Thực chất system call Exec làm các việc sau:

- Tạo ra một không gian địa chỉ mới
- Load chương trình vào khoảng bộ nhớ mới được cấp phát
- Sau đó tạo thread mới (bằng phương thức Thread::Fork()) để thực thi chương trình.

Lưu ý là sau khi thread mới chạy, giữa thread cha và thread con có thể xảy ra hiện tượng race condition do việc truy cập bộ nhớ, do đó chúng ta ***cần phải đồng bộ giữa 2 thread***.

Hiện tại, Nachos sử dụng thuật toán round-robin với time slice cố định để điều phối giữa các thread. Khi một thread đang thực thi trên máy ảo MIPS mà time slice dành cho nó đã hết thì một interrupt được phát sinh, tạo ra một trap vào system mode. Trong kernel space, bộ lập lịch của CPU (scheduler) ngừng thread hiện tại và lưu lại trạng thái (state) của nó. Sau đó scheduler tìm thread kế tiếp trên ready list, phục hồi trạng thái của nó và cho CPU tiếp tục chạy.

Các tập tin trong đồ án này:

- **progtest.cc** kiểm tra các thủ tục để chạy chương trình người dùng
- **syscall.h** system call interface: các thủ tục ở kernel mà chương trình người dùng có thể gọi
- **exception.cc** xử lý system call và các exception khác ở mức user, ví dụ như lỗi trang, trong phần mã chúng tôi cung cấp, chỉ có 'halt' system call được viết
- **bitmap.*** các hàm xử lý cho lớp bitmap (hữu ích cho việc lưu vết các ô nhớ vật lý) **filesys.h**
- **machine.*** mô phỏng các thành phần của máy tính khi thực thi chương trình người dùng: bộ nhớ chính, thanh ghi, v.v.
- **mipssim.cc** mô phỏng tập lệnh của MIPS R2/3000 processor

- **console.*** mô phỏng thiết bị đầu cuối sử dụng UNIX files. Một thiết bị có đặc tính (i) đơn vị dữ liệu theo byte, (ii) đọc và ghi các bytes cùng một thời điểm, (iii) các bytes đến bất đồng bộ
- **synchconsole.*** nhóm hàm cho việc quản lý nhập xuất I/O theo dòng trong Nachos.
- **../test/*** Các chương trình C sẽ được biên dịch theo MIPS và chạy trong Nachos.
- **thread.*:** Các hàm liên quan tới việc quản lý các thread bên trong hệ thống Nachos như: Cấp phát stack, đưa một thread vào trạng tháy sleep, thay đổi trạng thái hoạt động của một thread, lưu trữ trạng thái khi xuất hiện "context switching".
- **synch.*:** Gồm các lớp thực hiện việc đồng bộ như: Semaphore, Lock, …
- **scheduler.***: Gồm các hàm quản lý việc lập lịch và điều phối các tiểu trình.
- **list.*:** Lớp dùng để quản lý danh sách các đối tượng.
- **addrspace.*:** Lớp dùng để quản lý việc cấp phát và thu hồi bộ nhớ cho tiến trình.

Phần 2. [85%] Exceptions, System calls, Đa chương , lập lịch và đồng bộ.

Tham Khảo:

- **[5] Da Chuong & Dong Bo Hoa**
- **[3] Cach Viet Mot SystemCall**

Trong phần này chúng ta sẽ thiết kế và cài đặt để hỗ trợ đa chương trình trên Nachos. Các bạn phải viết thêm các system calls về quản lý tiến trình và giao tiếp giữa các tiến trình. Các bạn phải phát triển chương trình từ đồ án 1. Phải chắc rằng đồ án 1 của các bạn đã viết đúng và đầy đủ.

Nachos hiện tại chỉ là môi trường đơn chương. Chúng ta sẽ lập trình để cho mỗi tiến trình được duy trì trong system thread của nó. Chúng ta phải quản lý việc cấp phát và thu hồi bộ nhớ, quản lý phần dữ liệu và đồng bộ hóa các tiến trình/ tiểu trình. Lưu ý nên thiết kế giải pháp trước khi lập trình. Chi tiết như sau:

Thay đổi mã cho các exception khác (không phải system call exceptions) để tiến trình có thể hoàn tất, chứ không halt máy như trước đây. Một run time exception sẽ không gây ra việc HĐH phải shut down. Xử lý cho việc đồng bộ hóa khi tiến trình kết thúc.

Cài đặt đa tiến trình. Chương trình hiện tại giới hạn bạn chỉ thực thi 1 chương trình, bạn phải có vài thay đổi trong file addrspace.h và addrspace.cc để chuyển hệ thống từ đơn chương thành đa chương. Bạn sẽ cần phải:

a. Giải quyết vấn đề cấp phát các frames bộ nhớ vật lý, sao cho nhiều chương trình có thể nạp lên bộ nhớ cùng một lúc.
b. Phải xử lý giải phóng bộ nhớ khi user program kết thúc.
c. Phần quan trọng là thay đổi đoạn lệnh nạp user program lên bộ nhớ. Hiện tại, việc cấp phát không gian địa chỉ giả thiết rằng một tiến trình được nạp vào các đoạn liên tiếp nhau trong bộ nhớ. Một khi chúng ta hỗ trợ đa chương trình, bộ nhớ sẽ không còn biểu diễn liên tiếp nhau nữa. Nếu chúng ta không lập trình đúng đắn thì khi nạp một chương trình mới có thể làm phá hỏng HĐH của bạn.

Cài đặt system call **SpaceID Exec(char* name) hoặc SpaceID Exec(char* name, int priority (có khi cài đồ án cộng điểm)**. Exec gọi thực thi một chương trình mới trong một system thread

mới. Bạn cần phải đọc hiểu hàm "StartProcess" trong progtest.cc để biết cách khởi tạo một user space trong 1 system thread. Exec trả về -1 nếu bị lỗi và thành công thì trả về Process SpaceID của chương trình người dùng vừa được tạo. Đây là thông tin cần phải quản lý trong lớp Ptable.

Cài đặt system calls **int Join(SpaceID id)** và **void Exit(int exitCode)**. **Join** sẽ đợi và block dựa trên tham số "SpaceID id". **Exit** trả về exit code cho tiến trình nó đã join. Exit code là 0 nếu chương trình hoàn thành thành công, các trường hợp khác trả về mã lỗi. Mã lỗi được trả về thông qua biến exitcode. Join trả về exit code cho tiến trình nó đã đang block trong đó, -1 nếu join bị lỗi. Một user program chỉ có thể join vào những tiến trình mà đã được tạo bằng system call Exec. Bạn không thể join vào các tiến trình khác hoặc chính tiến trình mình. Bạn phải sử dụng semaphore để phối hợp hoạt động giữa Joining và Exiting của tiến trình người dùng.

Cài đặt system call **int CreateSemaphore(char* name, int semval)**. Như trong project 1 bạn phải cập nhật file start.s, start.c và syscall.h để thêm system call mới. Bạn tạo cấu trúc dữ liệu để lưu 10 semaphore. System call **CreateSemaphore** trả về 0 nếu thành công, ngược lại thì trả về -1.

Cài đặt system call **int Up(char* name),** và **int Down(char* name)**. Tham số name là tên của **semaphore**. Cả hai system call trả về 0 nếu thành công và -1 nếu lỗi. Lỗi có thể xảy ra nếu người dùng sử dụng sai tên semaphore hay semaphore đó chưa được tạo. Xem gợi ý khai báo lớp quản lý các **semphare** mà Nachos tạo ra để cung cấp cho chương trình người dùng ở phụ lục.

Cài đặt một chương trình **shell** đơn giản để kiểm tra các system call đã cài đặt ở trên. **Shell** nhận một lệnh tại một thời điểm và thực thi chương trình tương ứng. **Shell** nên **"join"** mỗi chương trình và đợi cho đến khi chương trình kết thúc. Khi trả về của hàm **Join**, hiển thị ra exitcode nếu nó khác 0 (thoát bình thường). **Shell** của bạn cũng phải cho phép chương trình chạy background. Bất kì lệnh nào bắt đầu bằng '&' thì chạy theo dạng background. (ví dụ: &create, thì chương trình create chạy theo mode background)