Đồ án 2: Đa chương trình

Hệ điều hành. Ngày ra đề 27-4-2007 Ngày hết hạn17h00 9-6-2007

Trong đồ án 2 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 ly 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, nếu thiếu phần nào thì các bạn có thể tham khảo đáp án để thêm vào cho 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 ly việc cấp phát và thu hồi bộ nhớ. Chúng ta phải quản ly phần dữ liệu và đồng bộ hóa các tiến trình/tiểu trình. Các bạn phải thiết kế giải pháp trước khi lập trình. Chi tiết như sau:

1. 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 ly cho việc đồng bộ hóa khi tiến trình kết thúc.

2. Cài đặt đa tiến trình. Chương trình hiện tại chỉ 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 ly, sao cho nhiều chương trình có thể nạp lên bộ nhớ cùng một lúc.
    b. Phải xử ly 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 các bạn

3. Cài đặt system call **SpaceID Exec(char* name). 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 dung vừa được tạo (*chú ằng SpaceID cũng cần được lưu trữ giống như OpenFileID trong đồ án 1, chỉ một điều khác biệt là các bạn phải quản ly chúng bên ngoài lớp Thread.*)

4. 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.

5. 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. Nếu không còn entry trống để tạo semaphore mới, thì chúng ta trả về NULL, hoặc khởi tạo semaphore với giá trị âm (<0).

6. Cài đặt system call **int wait(char* name)**, và **int signal(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 y khai báo lớp quản ly các semaphore mà Nachos tạo ra để cung cấp cho chương trình người dùng ở phụ lục.

7. 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)

8. [Tanenbaum] Một sinh viên chuyên ngành nhân chủng học và biết khái niệm cơ bản về máy tính đã làm một nghiên cứu về loài khỉ Châu Phi, thử xem chúng có biết về deadlock. Anh ta giăng một sợi dây thừng ngang qua 1 vực sâu, giúp cho đàn khỉ có thể vượt qua vực bằng cách chuyền bằng tay. Nhiều con khỉ có thể chuyền qua cùng một lúc, nếu chúng đang di chuyển cùng chiều. Nếu đàn khỉ di chuyển từ chiều Đông sang Tây và chiều Tây sang Đông cùng một lúc, thì kết quả là deadlock (hai đàn khỉ sẽ bị kẹt ở giữa đoạn dây thừng), bởi vì bọn khỉ không thể trèo trên người lẫn nhau khi đang đu trên một sợi dây treo ngang vực. Nếu một con khỉ muốn vượt qua vực thì nó phải kiểm tra chắc chắn rằng không có con khỉ nào cũng đang băng qua vực theo chiều ngược lại. Viết chương trình sử dụng semaphore để tránh deadlock. Không cần quan tâm trường hợp khi mà đoàn khỉ đi từ Đông sang Tây quá đông làm cho sự di chyển của đoàn khỉ từ Tây sang Đông phải đợi vô thời hạn. *Chú y: vì nachos không hỗ trợ truyền tham số vào hàm*

main, nên chúng ta có thể dùng 1 file chung để lưu các biến dùng chung giữa các tiến trình.

9. Tài liệu (10%) bao gồm tài liệu bên trong chương trình (comment) và tài liệu bên ngoài. Tạo file README và đặt bên trong thư mục code. Tar và gzip đồ án của các bạn và nộp lại chỉ 1 file.tar.gz như trong đồ án 1.

*Gợi y lớp **Sem** và lớp **Stable** để cài đặt cho phần quản ly các semaphore mà Nachos cung cấp cho chương trình người dùng*

```
//stable.h
#ifndef STABLE_H
#define STABLE_H

#include "synch.h"
#include "bitmap.h"

#define MAX_SEMAPHORE 10

class Sem{
  char name[50];
  Semaphore *sem;

 public:
  Sem(char* na,int i){
    strcpy(this->name,na);
    sem = new Semaphore(name,i);
  }
  ~Sem(){delete sem;}
  void wait(){sem->P();}
  void signal(){sem->V();}
  char* GetName(){return name;}
};

class STable {
 private:
  BitMap* bm;
  Sem* semTab[MAX_SEMAPHORE];

 public:
  STable();
  ~STable();
  int create(char* name, int init);
  int wait(char *name);
  int signal(char *name);
};
#endif
```