ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO PROJECT 1

ĐA CHƯƠNG, LẬP LỊCH VÀ ĐỒNG BỘ HÓA TRONG NACHOS

Môn học: Hệ điều hành
Giảng viên: ThS. Lê Viết Long

Thành phố Hồ Chí Minh – 2024

Mục lục

I. Thông tin nhóm

Họ và tên	MSSV	email	Đóng góp
Nguyễn Hữu Gia Hiếu	*22127113*	*nhghieu22@fitus.clc.edu.vn*	*100%*
Trần Nguyên Đăng Khoa	*22127199*	*tndkhoa22@fitus.clc.edu.vn*	*100%*
Nguyễn Bá An	*22127472*	*nban22@fitus.clc.edu.vn*	*100%*
Trịnh Phạm Bảo Tín	*22127487*	*ptbtin22@fitus.clc.edu.vn*	*100%*

II. Đánh giá tổng quan

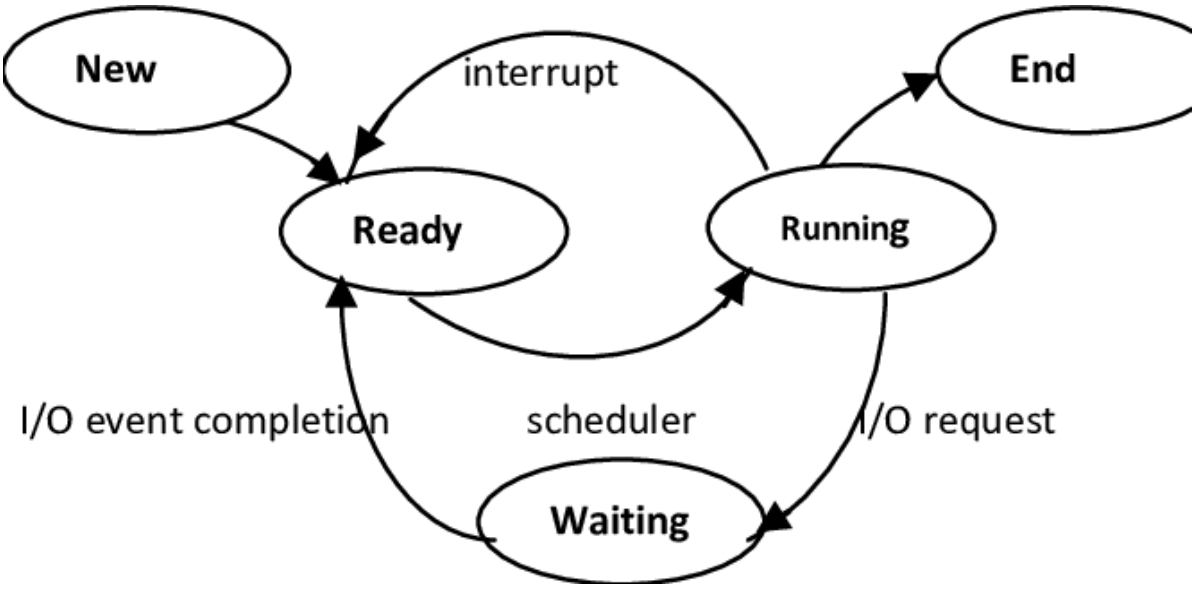
Yêu cầu	Hoàn thành
Các system call nhập/xuất file	*100%*
Đa chương, lập lịch và đồng bộ hóa trong Nachos	*100%*
Chương trình minh họa	*100%*

III. Lập trình đa chương

1) Một số khái niệm quan trọng

***) Process States:**

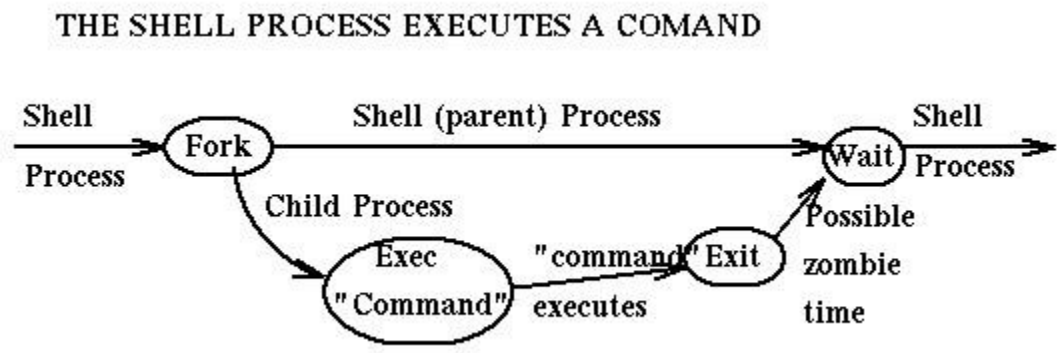
- Mỗi tiến trình trải qua nhiều trạng thái khác nhau từ khi được tạo ra cho đến khi kết thúc, và quá trình này được điều khiển bởi hệ điều hành.



- Trạng thái của tiến trình bao gồm:
  - **New**: Quá trình được khởi tạo.
  - **Running**: Quá trình đang thực thi sau khi được nạp vào bộ xử lý.
  - **Waiting**: Quá trình đợi để được nạp từ bộ nhớ ngoại vào bộ nhớ chính, chẳng hạn như ổ đĩa cứng, ổ flash hoặc CD-ROM.
  - **Ready**: Quá trình sẵn sàng để được lên lịch xử lý.
  - **Terminated**: Khi quá trình đã hoàn thành thực thi hoặc bị hệ điều hành kết thúc.

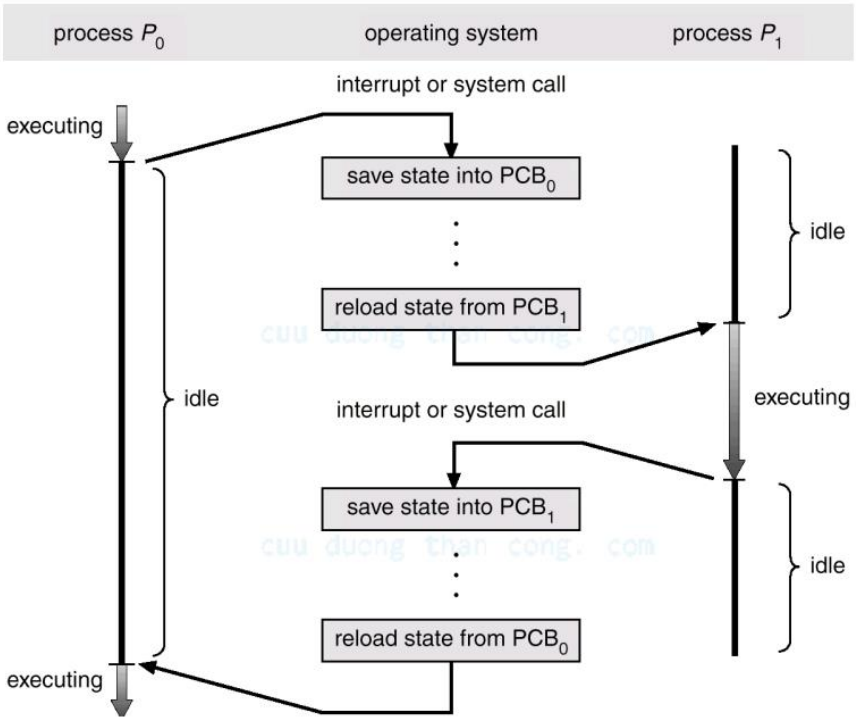
***) Process creation use folk:**

- Trong Unix, một tiến trình được tạo ra thông qua System Call fork(). Sau khi gọi fork(), quá trình sẽ tiếp tục gọi System Call Exec() để thay đổi vùng nhớ của nó và thực hiện một chương trình mới. Hệ thống sẽ quản lý và điều chỉnh các tiến trình này theo trạng thái của chúng.



***) Context Switching:**

- Trong Nachos đa chương, Context Switching là quá trình chuyển đổi giữa việc thực thi các tiến trình hoặc các chương trình con.
- Khi một tiến trình đang thực thi và cần phải dừng lại để cho phép tiến trình khác thực thi, Nachos phải thực hiện Context Switching.
- Quá trình này bao gồm việc lưu trữ trạng thái hiện tại của tiến trình đang thực thi và khôi phục trạng thái của tiến trình tiếp theo cần thực thi.
- Context Switching trong Nachos đa chương đóng vai trò quan trọng trong việc đảm bảo rằng tất cả các tiến trình được cung cấp thời gian CPU hợp lý và tối ưu hóa việc sử dụng tài nguyên hệ thống.



Nachos hiện chỉ là một môi trường đơn chương trình, tuy nhiên, cần phát triển để mỗi tiến trình được duy trì trong system thread của riêng nó. Điều này đòi hỏi tự quản lý cấp phát và thu hồi bộ nhớ, quản lý phần dữ liệu và đồng bộ hóa các tiến trình/tiểu trình. Điều này sẽ làm tăng khả năng mở rộng của Nachos và cải thiện tính linh hoạt và hiệu suất của hệ thống.

2) Một số class quan trọng

***)** Cơ sở của đồng bộ hóa chính là đối tượng Semaphore, được định nghĩa trong **./threads/synch.h**.

- Semaphore(char * debugName, int initialValue): Phương thức khởi tạo mặc định có tham số truyền vào là initialValue với ý nghĩa là Semaphore này sẽ có tối đa initialValue tiến trình được phép thực thi cùng lúc.

- void P(): Giảm biến đếm semaphore xuống, block tiến trình nếu như biến đếm này bằng 0.

- void V(): Tăng biến đếm semaphore lên và gọi một tiến trình thực thi nếu tiến trình này đang chờ thực thi từ hàng đợi queue.

a) Lớp PCB (Process Control Block)

Lớp PCB lưu thông tin để quản lý process và bao gồm các thuộc tính quan trọng như:

- int pid: Định danh của tiến trình để phân biệt các tiến trình.
- Thread* thread: Lưu tiến trình được nạp.
- int parentID: id của tiến trình cha.
- FileName: Lưu tên của tiến trình.
- 3 thuộc tính Semaphore: Để quản lý quá trình Join, Exit và nạp chương trình.

- Dưới đây là mô tả ngắn gọn về nhiệm vụ của class và từng method, thuộc tính:

*) Thuộc tính:

- **joinsem:** Semaphore để đợi khi tiểu trình tham gia (join) một tiểu trình khác.
- **exitsem:** Semaphore để đợi khi tiểu trình kết thúc (exit).
- **mutex:** Semaphore để đảm bảo đồng bộ hóa truy cập vào các biến thành viên của PCB.
- **exitcode:** Mã thoát của tiểu trình.
- **thread:** Con trỏ đến tiểu trình liên kết với PCB.
- **pid:** ID của tiểu trình.
- **numwait:** Số lượng tiểu trình đang chờ đợi kết thúc của tiểu trình hiện tại.
- **parentID:** ID của tiểu trình cha.
- **JoinStatus:** Trạng thái của việc tham gia với tiểu trình khác.x

*) Phương thức

- **PCB(int id):** Constructor, khởi tạo một PCB mới với ID đã cho.
- **~PCB():** Destructor, giải phóng bộ nhớ và các tài nguyên được cấp phát.
- **Exec(char filename, int pID):* Tạo một tiểu trình mới từ tập tin có tên filename và ID quy trình pID.
- **GetID():** Trả về ID của tiểu trình.
- **GetNumWait():** Trả về số lượng tiểu trình đang chờ đợi.
- **GetExitCode():** Trả về mã thoát của tiểu trình.
- **SetExitCode(int ec):** Thiết lập mã thoát của tiểu trình.
- **IncNumWait():** Tăng số lượng tiểu trình đang chờ đợi.

- **DecNumWait():** Giảm số lượng tiểu trình đang chờ đợi.
- **GetNameThread():** Trả về tên của tiểu trình.
- **JoinWait():** Chờ tiểu trình cha tham gia với tiểu trình hiện tại.
- **JoinRelease():** Phát hành tiểu trình cha.
- **ExitWait():** Chờ tiểu trình kết thúc.
- **ExitRelease():** Phát hành tiểu trình kết thúc.

*) Hàm phụ hỗ trợ cho class

- **MyStartProcess(int pID):** Method này được gọi khi tiểu trình mới được tạo và sẽ thực hiện chuẩn bị và chạy chương trình người dùng.

==> Class PCB trong Nachos có nhiệm vụ quan trọng trong việc quản lý tiểu trình và các tác vụ liên quan đến tiểu trình trong môi trường đa chương.

b) Lớp Thread

- Lớp Thread tạo ra các tiểu trình và bao gồm một số thuộc tính và hàm quan trọng như:

- char* name: Lưu tên của tiến trình, đường dẫn tương đối tới chương trình thực thi.
- AddrSpace * space: Vùng nhớ của tiến trình trên ram ảo.
- void Thread(char * debugName): Khởi tạo một tiến trình với debugName là đường dẫn tương đối tới file chương trình thực thi.
- void Fork(VoidFunctionPtr func, int arg): Cấp phát vùng nhớ Stack cho tiến trình và gán con trỏ hàm VoidFunctionPtr (func) và số nguyên int (arg).

- Class Thread trong Nachos là một cấu trúc dữ liệu để quản lý luồng thực thi. Dưới đây là mô tả ngắn gọn về nhiệm vụ của class và từng method, thuộc tính:

*) Thuộc tính

- **name:** Tên của luồng.
- **stackTop:** Con trỏ đến đỉnh của stack.
- **machineState:** Trạng thái của máy, bao gồm các thanh ghi CPU.
- **stack:** Con trỏ đến đáy của stack.
- **status:** Trạng thái của luồng (JUST_CREATED, RUNNING, READY, BLOCKED).

- **processID:** ID của tiến trình.

*) Phương thức

- *Thread(char debugName):** Constructor, khởi tạo một luồng với tên là "debugName".
- **~Thread():** Destructor, giải phóng bộ nhớ và các tài nguyên được cấp phát cho luồng.
- **Fork(VoidFunctionPtr func, int arg):** Tạo một luồng mới chạy hàm "func" với đối số "arg".
- **Finish():** Kết thúc việc thực thi của luồng.
- **Yield():** Nhường CPU cho luồng khác nếu có.
- **Sleep():** Cho luồng vào trạng thái ngủ và nhường CPU cho luồng khác.
- **CheckOverflow():** Kiểm tra xem stack của luồng có bị tràn không.
- **setStatus(ThreadStatus st):** Thiết lập trạng thái của luồng.
- **getName():** Trả về tên của luồng.
- **Print():** In thông tin về luồng.
- **StackAllocate(VoidFunctionPtr func, int arg):** Cấp phát và khởi tạo stack cho luồng.
- **SaveUserState():** Lưu trạng thái của CPU của chương trình người dùng trong quá trình chuyển đổi ngữ cảnh.
- **RestoreUserState():** Khôi phục trạng thái của CPU của chương trình người dùng trong quá trình chuyển đổi ngữ cảnh.

===> Class Thread trong Nachos có nhiệm vụ quản lý các luồng thực thi, bao gồm việc tạo, kết thúc, nhường CPU và kiểm tra stack overflow

c) Lớp BitMap

Lớp này để lưu vết các tiến trình hiện hành và bao gồm các hàm quan trọng như:

- void Mark(int which): Đánh dấu khung trang này được sử dụng.
- int Find(): Tìm một khung trang trống và đánh dấu nó đã được sử dụng.
- int NumClear(): Trả về tổng số khung trang còn trống trên bộ nhớ.

- Dưới đây là mô tả ngắn gọn về nhiệm vụ của class và từng method, thuộc tính:

*) Thuộc tính

- **numBits:** Số lượng bit trong Bitmap.
- **numWords:** Số lượng từ (words) được sử dụng để lưu trữ Bitmap.
- **map:** Mảng unsigned integers để lưu trữ các bit.

*) Phương thức

- **BitMap(int nitems):** Constructor, khởi tạo một Bitmap với "nitems" bit, tất cả các bit đều được xóa ban đầu.
- **~BitMap():** Destructor, giải phóng bộ nhớ và các tài nguyên được cấp phát.
- **Mark(int which):** Thiết lập (set) bit thứ "which".
- **Clear(int which):** Xóa (clear) bit thứ "which".
- **Test(int which):** Kiểm tra xem bit thứ "which" có được thiết lập hay không.
- **Find():** Trả về số của bit đầu tiên chưa được thiết lập và đồng thời thiết lập nó. Nếu không có bit nào chưa được thiết lập, trả về -1.
- **NumClear():** Trả về số lượng bit chưa được thiết lập trong Bitmap.
- **Print():** In các bit được thiết lập trong Bitmap (dùng cho mục đích debug).
- **FetchFrom(OpenFile file):* Đọc nội dung của Bitmap từ một file Nachos.
- **WriteBack(OpenFile file):* Ghi nội dung của Bitmap vào một file Nachos.

===> Class BitMap trong Nachos có nhiệm vụ quản lý và thực hiện các thao tác trên Bitmap, như xác định bit chưa được thiết lập, thiết lập bit, xóa bit và lưu trữ nội dung của Bitmap vào/và từ một file Nachos.

d) Lớp PTable

- Dùng để quản lý các tiến trình được chạy trong hệ thống.
- PCB* pcb[MAX_PROCESSES] là một bảng mô tả tiến trình có cấu trúc mảng một chiều có số phần tử tối đa MAX_PROCESSES = 10 theo yêu cầu của đồ án.
- Hàm constructor của lớp sẽ khởi tạo tiến trình cha (là tiến trình đầu tiên) ở vị trí 0 tương đương với phần tử đầu tiên của mảng. Từ tiến trình này, chúng ta sẽ tạo ra các tiến trình con thông qua system call Exec().

- Class PTable (Process Table) trong Nachos là một cấu trúc dữ liệu để quản lý các tiểu trình và các tác vụ liên quan đến việc tạo, kết thúc và tham gia các tiểu trình. Dưới đây là mô tả ngắn gọn về nhiệm vụ của class và từng method, thuộc tính:

*) Thuộc tính

- **bm:** BitMap để theo dõi trạng thái của các slot trong bảng tiến trình.
- **pcb[MAXPROCESS]:** Mảng các con trỏ đến các PCB (Process Control Block) để lưu thông tin của các tiểu trình.
- **psize:** Kích thước của bảng tiến trình.
- **bmsem:** Semaphore để đảm bảo chỉ một tiểu trình được tạo vào một thời điểm.

*) Phương thức

- **PTable(int size):** Constructor, khởi tạo một bảng tiến trình với kích thước size.
- **~PTable():** Destructor, giải phóng bộ nhớ và các tài nguyên được cấp phát.
- *ExecUpdate(char filename):** Thực hiện cập nhật sau khi một tiểu trình được tạo từ tệp có tên filename.
- **ExitUpdate(int ec):** Thực hiện cập nhật sau khi một tiểu trình kết thúc với mã thoát là ec.
- **JoinUpdate(int pID):** Thực hiện cập nhật sau khi một tiểu trình tham gia vào một tiểu trình khác có ID là pID.
- **GetFreeSlot():** Tìm một slot trống trong bảng tiến trình để lưu thông tin của một tiểu trình mới.
- **IsExist(int pID):** Kiểm tra xem một tiểu trình có tồn tại trong bảng tiến trình hay không.
- **Remove(int pID):** Xóa thông tin của một tiểu trình khỏi bảng tiến trình sau khi nó kết thúc.
- **GetName(int pID):** Lấy tên của một tiểu trình có ID là pID.

*) Hàm phụ hỗ trợ cho class

- **MyStartProcess(int pID):** Method này được gọi khi một tiểu trình mới được tạo và sẽ thực hiện chuẩn bị và chạy chương trình người dùng.

===> Class PTable trong Nachos có nhiệm vụ quản lý các tiểu trình và các tác vụ liên quan đến chúng trong môi trường đa chương.

3) Cài đặt các system call

Bước 1: Khai báo biến toàn cục và tạo đối tượng

- Trong bước này, chúng ta đã khai báo các biến toàn cục trong file **./threads/system.h** và tạo đối tượng tương ứng trong file **system.cc**. Các biến toàn cục bao gồm:

- **Semaphore *addrLock**: Đối tượng Semaphore để đồng bộ hóa việc truy cập vào địa chỉ.
- **BitMap *gPhysPageBitMap**: Đối tượng BitMap để quản lý các frame của bộ nhớ vật lý.
- **PTable *pTab**: Đối tượng PTable để quản lý bảng tiến trình.

- Việc tạo đối tượng được thực hiện bằng cách khởi tạo các biến toàn cục này với các giá trị khởi đầu tương ứng.

Bước 2: Cài đặt lớp PCB và PTable

- Trong bước này, chúng ta đã cài đặt hai lớp mới là PCB và PTable để quản lý tiến trình. Sau đó, chúng ta đã tiến hành khai báo các lớp này trong file "Makefile.common" để đảm bảo rằng chúng được bao gồm và sử dụng trong dự án.

Bước 3: Thay đổi kích thước của khung trang và sector

- Chúng ta đã thực hiện việc điều chỉnh kích thước của khung trang và sector trong các file **./machine/machine.h** và **./machine/disk.h** bằng cách sử dụng các hằng số **NumPhysPage** và **SectorSize**. Việc này đảm bảo rằng hệ thống sẽ hoạt động ổn định và hiệu quả với kích thước bộ nhớ và ổ đĩa mới.

Bước 4: Chỉnh sửa lớp Thread

- Trong bước này, chúng ta đã thực hiện các chỉnh sửa trong lớp Thread trong file **./threads/thread.h**. Các chỉnh sửa bao gồm thêm hai thuộc tính mới là **int processID** để quản lý ID của tiến trình và **int exitStatus** để kiểm tra mã lỗi khi tiến trình kết thúc. Đồng thời, chúng ta đã cài đặt phương thức **void FreeSpace()** để giải phóng vùng nhớ mà tiến trình đang sử dụng.

Bước 5: Cài đặt hàm StartProcess

- Chúng ta đã cài đặt hàm **StartProcess(char *filename)** trong file **./userprog/progtest.cc**. Hàm này được sử dụng để Fork và trỏ vào vùng nhớ của tiến trình con.

Bước 6: Cài đặt lớp AddressSpace

- Trong bước này, chúng ta đã cài đặt lớp AddressSpace trong các file **./userprog/addrspace.cc** và **addrespace.h**. Các thay đổi được thực hiện để chuyển từ hệ thống đơn chương thành hệ thống đa chương, bao gồm:

- Giải quyết vấn đề cấp phát các frame bộ nhớ vật lý cho nhiều chương trình.
- Xử lý việc giải phóng bộ nhớ khi chương trình kết thúc.
- Thay đổi đoạn lệnh nạp chương trình lên bộ nhớ để phù hợp với hệ thống đa chương.

Bước 7: Cài đặt system call

- Cuối cùng, chúng ta đã cài đặt các system call như Exec, Join và Exit để quản lý các tiến trình trong hệ thống. Các hàm này đã được khai báo trong file **./userprog/syscall.h** và được cài đặt tương ứng trong các lớp PCB và PTable.
- Qua các bước này, chúng ta đã thực hiện một loạt các thay đổi và cài đặt để mở rộng và cải thiện Nachos, tạo điều kiện thuận lợi cho việc phát triển và nghiên cứu các tính năng mới trong hệ thống hệ điều hành.

a) System call Exec

- Khai báo trong **./userprog/syscall.h**: *SpaceId Exec(char *name);*
- Cài đặt hàm "*int Exec(char *filename, int pID);*" ở **class PCB**, trong **./threads/pcb.h**
- Cài đặt hàm "*int ExecUpdate(char* filename)*; //return PID" ở **class PTable**, trong **./threads/ptable.h**
- Chức năng: Thực thi một chương trình mới từ một tệp tin đã được mở.
- Tham số:

- **virtAddr**: Địa chỉ ảo của chuỗi ký tự chứa tên của chương trình cần thực thi.

- Thực hiện:

- Đọc tên của chương trình từ vùng nhớ người dùng và chuyển đổi từ địa chỉ ảo sang địa chỉ hệ thống.
- Kiểm tra xem tên chương trình có hợp lệ hay không.
- Mở tệp tin chứa chương trình bằng phương thức **Open** của **fileSystem**.
- Gọi phương thức **ExecUpdate** của **PTable** để tạo một tiến trình mới và gán ID cho nó.
- Trả về ID của tiến trình mới hoặc -1 nếu thất bại.

b) System call Join

- Khai báo ở **./userprog/syscall.h**: int Join(SpaceId id);

- Cài đặt: *void JoinWait()* và *void ExitRelease()* ở **class PCB**, trong **./threads/pcb.h**

- Cài đặt: *int JoinUpdate(int pID)* ở **class Ptable**

- Chức năng: Chờ đợi cho một tiến trình khác kết thúc và lấy exit code của nó.

- Tham số:

  - **id**: ID của tiến trình mà tiến trình hiện tại muốn chờ đợi.

- Thực hiện:

  - Đọc ID của tiến trình cần chờ từ thanh ghi.
  - Gọi phương thức **JoinUpdate** của **PTable** để chờ đợi tiến trình với ID tương ứng kết thúc.
  - Trả về exit code của tiến trình đã kết thúc hoặc -1 nếu không thành công.

c) System call Exit

- Khai báo ở **./userprog/syscall.h**: *void Exit(int exitCode)*.

- Cài đặt hàm: *void JoinRelease()* và *void ExitWait()* ở **class PCB**, trong **./threads/pcb.h**

- Cài đặt: *int ExitUpdate(int ec)* ở **class Ptable**

- Chức năng: Kết thúc tiến trình hiện tại với một mã kết thúc nhất định.

- Tham số:

  - **exitCode**: Mã kết thúc của tiến trình.

- Thực hiện:

  - Đọc mã kết thúc từ thanh ghi.

- Nếu mã kết thúc là 0 (không có lỗi), tiến trình tiếp tục thực hiện.
- Gọi phương thức ExitUpdate của PTable để thông báo về việc kết thúc của tiến trình và cập nhật mã kết thúc.
- Nếu có lỗi, tiến trình sẽ tiếp tục thực hiện mà không cần thông báo kết thúc.

IV. Lập trình đồng bộ

1) Một số class quan trọng

a) Class SEM

- Trong **.\code\threads\sem.h**

- Class Sem (Semaphore) trong Nachos là một cấu trúc dữ liệu đơn giản để thực hiện đồng bộ hóa trong môi trường đa chương. Dưới đây là mô tả ngắn gọn về nhiệm vụ của class và từng method, thuộc tính:

*) Thuộc tính

- **name:** Tên của Semaphore.
- **sem:** Con trỏ đến một Semaphore (Semaphore được cung cấp bởi thư viện synch.h).

*) Phương thức

- *Sem(char na, int i):** Constructor, khởi tạo một Semaphore mới với tên na và giá trị khởi tạo i.
- **~Sem():** Destructor, giải phóng bộ nhớ và các tài nguyên được cấp phát.
- **wait():** Thực hiện chờ đợi trên Semaphore.
- **signal():** Thực hiện phát tín hiệu trên Semaphore.
- **GetName():** Trả về tên của Semaphore.

===> Class Sem trong Nachos có nhiệm vụ đơn giản là thực hiện các tác vụ liên quan đến Semaphore như chờ đợi và phát tín hiệu để đảm bảo đồng bộ hóa trong môi trường đa chương.

b) Class STable

- Trong **\code\threads\stable.h**

- Class STable (Semaphore Table) trong Nachos là một cấu trúc dữ liệu để quản lý các Semaphore và các tác vụ liên quan đến việc tạo, chờ đợi và phát tín hiệu các Semaphore. Dưới đây là mô tả ngắn gọn về nhiệm vụ của class và từng method, thuộc tính:

*) Thuộc tính

- **bm:** BitMap để theo dõi trạng thái của các slot trong bảng Semaphore.
- **semTab[MAX_SEMAPHORE]:** Mảng các con trỏ đến các Semaphore để lưu thông tin của các Semaphore.

*) Phương thức

- **STable():** Constructor, khởi tạo một bảng Semaphore với số lượng Semaphore tối đa là MAX_SEMAPHORE.
- **~STable():** Destructor, giải phóng bộ nhớ và các tài nguyên được cấp phát.
- *Create(char name, int init):** Tạo một Semaphore mới với tên name và giá trị khởi tạo init.
- *Wait(char name):** Thực hiện chờ đợi trên Semaphore có tên name.
- *Signal(char name):** Thực hiện phát tín hiệu trên Semaphore có tên name.
- **FindFreeSlot():** Tìm một slot trống trong bảng Semaphore để tạo Semaphore mới.

==> Class STable trong Nachos có nhiệm vụ quản lý các Semaphore và các tác vụ liên quan đến chúng trong môi trường đa chương.

2) Một số system call quan trọng

a) Syscall CreateSemaphore

- Khai báo ở **./userprog/syscall.h**: *int CreateSemaphore(char* name, int semval);*
- Cài đặt hàm *int Create(char* name, int init)* ở **class Stable**, trong **./threads/stable.h**
- Chức năng: Tạo một Semaphore mới hoặc cập nhật giá trị ban đầu của một Semaphore đã tồn tại.
- Tham số:
  - **virtAddr**: Địa chỉ ảo của chuỗi ký tự chứa tên của Semaphore.
  - **semval**: Giá trị ban đầu của Semaphore.
- Thực hiện:

- Đọc tên của Semaphore từ vùng nhớ người dùng và chuyển đổi từ địa chỉ ảo sang địa chỉ hệ thống.
- Kiểm tra xem tên Semaphore có hợp lệ hay không.
- Gọi phương thức **Create** của SemaphoreTable để tạo Semaphore mới hoặc cập nhật giá trị của Semaphore đã tồn tại.
- Trả về ID của Semaphore đã tạo hoặc -1 nếu không thể tạo được.

b) Syscall Up

- Khai báo ở **./userprog/syscall.h**: *int Up(char* name);*

- Chức năng: Tăng giá trị của Semaphore, cho phép một hoặc nhiều tiến trình khác sử dụng tài nguyên chia sẻ được bảo vệ bởi Semaphore.

- Tham số:

- **virtAddr**: Địa chỉ ảo của chuỗi ký tự chứa tên của Semaphore.

- Thực hiện:

- Đọc tên của Semaphore từ vùng nhớ người dùng và chuyển đổi từ địa chỉ ảo sang địa chỉ hệ thống.
- Kiểm tra xem tên Semaphore có hợp lệ hay không.
- Gọi phương thức **Signal** của SemaphoreTable để tăng giá trị của Semaphore.
- Trả về 0 nếu thành công hoặc -1 nếu không thể tìm thấy Semaphore.

c) Syscall Down

- Khai báo ở **./userprog/syscall.h**: *int Down(char* name);*

- Chức năng: Giảm giá trị của Semaphore, chặn tiến trình hiện tại nếu giá trị Semaphore là 0, cho đến khi Semaphore trở thành dương.

- Tham số:

- **virtAddr**: Địa chỉ ảo của chuỗi ký tự chứa tên của Semaphore.

- Thực hiện:

- Đọc tên của Semaphore từ vùng nhớ người dùng và chuyển đổi từ địa chỉ ảo sang địa chỉ hệ thống.
- Kiểm tra xem tên Semaphore có hợp lệ hay không.

- Gọi phương thức **Wait** của SemaphoreTable để giảm giá trị của Semaphore và chặn tiến trình hiện tại nếu cần.
- Trả về 0 nếu thành công hoặc -1 nếu không thể tìm thấy Semaphore.

V. Chương trình minh họa – PING PONG

- Trong demo này, chúng ta sử dụng ba chương trình: Ping, Pong và Main, để minh họa việc điều phối hoạt động của hai tiến trình sử dụng Semaphore trong hệ điều hành Nachos. Mục tiêu là đảm bảo rằng các tiến trình Ping và Pong sẽ chạy xen kẽ nhau, xuất ra màn hình các ký tự A và B theo thứ tự đúng.

1) Chương Trình Ping:

- Nằm ở **./code/test/programA.c**

- Chương trình này sẽ thực hiện việc in ra màn hình 1000 ký tự 'A'.
- Trước khi in mỗi ký tự 'A', tiến trình Ping sẽ giảm giá trị của Semaphore "A" (Down("A")).
- Sau khi in xong một ký tự 'A', tiến trình Ping sẽ tăng giá trị của Semaphore "B" (Up("B")) để kích hoạt tiến trình Pong.

```c
#include "syscall.h"

int main()
{
    int i;
    for(i = 0; i < 1000; i++)
    {
        // print A once
        Down("A");
        PrintChar('A');
        // Wake programB up
        Up("B");
    }

    return 0;
}
```

2) Chương Trình Pong:

- Nằm ở **./code/test/programB.c**

- Chương trình này sẽ thực hiện việc in ra màn hình 1000 ký tự 'B'.

- Trước khi in mỗi ký tự 'B', tiến trình Pong sẽ giảm giá trị của Semaphore "B" (Down("B")).
- Sau khi in xong một ký tự 'B', tiến trình Pong sẽ tăng giá trị của Semaphore "A" (Up("A")) để kích hoạt tiến trình Ping.

```c
#include "syscall.h"

int main()
{
    int i;
    for(i = 0; i < 1000; i++)
    {
        // print B once
        Down("B");
        PrintChar('B');
        // Wake programA up
        Up("A");
    }

    return 0;
}
```

3) Chương Trình Main:

- Nằm ở **./code/test/ping_pong.c**

- Chương trình chính này sẽ khởi tạo hai Semaphore "A" và "B", với giá trị khởi tạo lần lượt là 1 và 0.
- Sau đó, nó sẽ tạo ra hai tiến trình Ping và Pong và đợi chúng kết thúc.

```c
#include "syscall.h"

int main()
{
    int i, ping, pong;

    // Create Semaphore named A to allow print A
    //A print first -> initValue = 1
    if(CreateSemaphore("A", 1) == -1)
        return -1;

    // Create Semaphore named B to allow print B
    // B print second -> initValue = 0 -> if programB run first, make
    // it sleep
    if(CreateSemaphore("B", 0) == -1)
        return -1;

    //Tell OS to run program A
    ping = Exec("./test/programA");

    //Tell OS to run program B
    pong = Exec("./test/programB");

    Join(ping);
    Join(pong);

    return 0;
}
```

4) Nguyên lý

- Ban đầu, Semaphore "A" có giá trị là 1, Semaphore "B" có giá trị là 0.
- Chương trình Main sẽ tạo ra tiến trình Ping đầu tiên.
- Tiến trình Ping sẽ giảm giá trị của Semaphore "A" xuống 0 và in ra ký tự 'A'.
- Tiến trình Ping tăng giá trị của Semaphore "B" lên 1 và kích hoạt tiến trình Pong.
- Tiến trình Pong giảm giá trị của Semaphore "B" xuống 0 và in ra ký tự 'B'.
- Tiến trình Pong tăng giá trị của Semaphore "A" lên 1 và kích hoạt tiến trình Ping.
- Quá trình lặp lại từ bước 3 đến bước 6 cho đến khi cả hai tiến trình đã in ra đủ 1000 ký tự 'A' và 'B'.

5) Kết luận

- Demo này minh họa cách sử dụng Semaphore trong Nachos để điều phối hoạt động của các tiến trình, đảm bảo rằng chúng chạy xen kẽ nhau và xuất ra màn hình các ký tự 'A' và 'B' theo thứ tự đúng. Bằng cách này, chúng ta có thể giải quyết vấn đề về đồng bộ hóa trong hệ thống điều hành Nachos.

VI. Tài liệu tham khảo

- [List video youtube - Lập trình Nachos HCMUS]
https://www.youtube.com/playlist?list=PLRgTVtca98hUgCN2_2vzsAAXPiTFbvHpO

- [Slide cuuduongthancong – Hệ Điều Hành] https://cuuduongthancong.com/sjdt/he-dieu-hanh/le-viet-long/dh-khoa-hoc-tu-nhien-hcm?src=subject

- [nachos-canban.doc]
https://mystudyhcmus.files.wordpress.com/2017/09/nachos_canban.doc

- [Lớp stable – thầy Lê Viết Long cung cấp]
https://courses.ctda.hcmus.edu.vn/mod/resource/view.php?id=81357

- [Lớp pcb, lớp ptable – thầy Lê Viết Long cung cấp]
https://courses.ctda.hcmus.edu.vn/mod/resource/view.php?id=81326

- [Tài liệu hướng dẫn cho project – thầy Lê Viết Long cung cấp]
https://courses.ctda.hcmus.edu.vn/mod/folder/download_folder.php?id=81270