ĐẠI HỌC QUỐC GIA TP. HCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN HỆ ĐIỀU HÀNH

GIẢI QUYẾT ĐA CHƯƠNG TRONG NACHOS

MSSV 1: 19120048 Họ tên: Hồ Nguyễn Trâm Anh

MSSV 2: 19120662 Họ tên: Đinh Trần Xuân Thi

MSSV 3: 19120682 Họ tên: Lê Hoàng Trọng Tín

MSSV 4: 19120694 Họ tên: Châu Lý Phương Trinh

MSSV 5: 19120695 Họ tên: Nguyễn Văn Trịnh

MỤC LỤC

I. Thông tin nhóm, công việc của từng thành viên và đánh giá

1. Thông tin nhóm

- Các thành viên trong nhóm:
  - MSSV: 19120662 Họ tên: Đinh Trần Xuân Thi
  - MSSV: 19120695 Họ tên: Nguyễn Văn Trịnh
  - MSSV: 19120694 Họ tên: Châu Lý Phương Trinh
  - MSSV: 19120682 Họ tên: Lê Hoàng Trọng Tín
  - MSSV: 19120048 Họ tên: Hồ Nguyễn Trâm Anh

2. Bảng phân chia công việc

Thành viên	**Phân chia công việc**
Đinh Trần Xuân Thi	Code System Call Exec, Join, Exit + Report
Nguyễn Văn Trịnh	Viết report class pcb và class ptable
Châu Lý Phương Trinh	Chỉnh sửa các file address.cc, address.h, system.h, system.cc, thread.h, thread.cc, system.h, system.cc.. + Report
Lê Hoàng Trọng Tín	Viết report class thread và class bitmap
Hồ Nguyễn Trâm Anh	Viết các chương trình ping.c, pong.c, scheduler.c, test lại toàn bộ chương trình + Report

3. Đánh giá tổng quan toàn bộ project

Đã hoàn thành	**Mức độ hoàn thành**
Tìm hiểu các lớp: Ptable, PCB, Bitmap, Thread	100%

Giải quyết đa chương trong Nachos	100%
Tổng kết: hoàn thành tất cả nội dung của đồ án	100%

II. Tìm hiểu các lớp Ptable, PCB, Bitmap, Thread

1. Lớp PCB

- **Lớp PCB** (Process Control Block) được sử dụng để lưu các thông tin để quản lý 1 process. Cụ thể như sau:

Thuộc tính		**Chức năng**
	Semaphore* joinsem;	semaphore cho quá trình join
	Semaphore* exitsem;	semaphore cho quá trình exit
	Semaphore* multex;	semaphore cho quá trình truy xuất độc quyền
	int exitcode;	Mã khi thoát chương trình
	int numwait;	số tiến trình đã join
	int parentID;	ID của tiến trình cha
	Thread* thread	Tiến trình của chương trình

Phương thức	PCB(int id);	- Khởi tạo một đối tượng để biểu diễn cho một tiến trình. + Khởi tạo các biến joinsem, exitsem, mutex + Gán các biến parentID, exitcode, numwait, thread
	~PBC();	Phương thức hủy đối tượng PCB, giải phóng vùng nhớ đã cấp cho các con trỏ khi tạo lập đối tượng
	int Exec(char *filename, int pID);	Nạp chương trình có tên lưu trong biến filename và process là pID
	void JoinWait() :	quản lý tiến trình Join (Tiến trình cha đợi tiến trình con kết thúc)
	void ExitWait():	quản lý tiến trình Join (Tiến trình con kết thúc)
	void JoinRelease()	quản lý tiến trình Release (Báo cho tiến trình cha thực thi tiếp)

	void ExitRelease():	quản lý tiến trình Release (Cho phép tiến trình con kết thúc.Tăng giảm số tiến trình đã Join – với việc sử dụng biến mutex)
	void IncNumWait()	Tăng số tiến trình chờ
	void DecNumWait():	Giảm số tiến trình chờ
	int GetExitCode():	Trả về exitcode
	char* GetNameThread()	Trả về tên của tiến trình

2. Lớp Ptable

- **Lớp Ptable:** dùng để quản lý các tiến trình được chạy trong hệ thống,

  PCB* pcb[MAXPROCESS] là một bảng mô tả tiến trình, có cấu trúc mảng một chiều có số phần tử tối đa là 10 theo yêu cầu của đồ án, mỗi phần tử của mảng thuộc kiểu dữ liệu PCB.

  Hàm constructor của lớp sẽ khởi tạo tiến trình cha (là tiến trình đầu tiên) ở vị trí thứ 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 systemcall Exec().

		Chức năng
Thuộc tính	BitMap *bm	Đánh dấu các vị trí đã được sử dụng trong pcb

	PCB *pcb[MAXPROCESS];	
	int psize;	
	Semaphore *bmsem;	Dùng để ngăn chặn 2 tiến trình xảy ra cùng 1 lúc
Phương thức	PTable(int size);	Khởi tạo size đối tượng pcb để lưu size process. Gán giá trị ban đầu là null. Khởi tạo *bm va *bmsem sử dụng
	~PTable();	Hủy các đối tượng đã tạo
	int ExecUpdate(char* filename);	Thực hiện cập nhật, Xử lý cho system call SC_Exec. Sau khi đã kiểm tra tính đúng đắn của tình trạng hiện tại như còn đủ chỗ chứa, file đã tồn tại hay chưa, chương trình được gọi có là chính nó không,.. ta cấp phát block mới và gọi thực thi phương thức Exec của lớp PCB. Cuối cùng trả về pid nếu thực hiện thành công.
	int ExitUpdate(int ec);	Thực hiện đưa ra khỏi mảng, Xử lý cho system call SC_Exit Sau khi kiểm tra tồn tại và có phải tiến trình chính, ta thực hiện "Remove()"

	int JoinUpdate(int pID);	Đưa vào mảng, Xử lý cho system call SC_Join Sau khi ngắt và lưu tình trạng hiện tại, kiểm tra bộ nhớ, có trùng vị trí hiện tại, ta tiến hành đưa vào
	int GetFreeSlot();	Tìm một slot trống để lưu thông tin cho tiến trình mới
	bool IsExist(int pid);	Kiểm tra xem id của tiến trình có tồn tại không
	void Remove(int pID);	Xóa một processID ra khỏi mảng quản lý nó, khi mà tiến trình này đã kết thúc
	char* GetName(int pID)	Trả về tên của tiến trình

3. Bitmap

Bitmap: Là một dãy bit dùng để quản lý các thành phần đã được khởi tạo hay chưa trong một mảng – đánh dấu các khung trang còn trống

Thuộc tính	int numBits;	số lượng bit trong bitmap
	int numWords;	số lượng từ lưu trữ bitmap (làm tròn nếu numBits

		không phải là bội số của số bit trong một từ)
	unsigned int* map;	lưu trữ bit
Phương thức	BitMap(int nitems);	Khởi tạo một bitmap, với số lượng bit "nitems". Ban đầu, tất cả các bit đều bị xóa.
	void Mark(int which);	Đánh dấu bit thứ “which”
	void Clear(int which);	Xóa bit thứ “which”
	bool Test(int which);	Kiểm tra bit thứ “which” đã được đánh dấu chưa
	int Find();	Trả về số bit đầu chưa đánh dấu và đặt nó đã sử dụng – tìm và cấp phát 1 bit. Nếu tất cả đề đã đánh dấu thì trả về -1.
	int NumClear();	Trả về số bit chưa đánh dấu
	void Print();	In dãy bit
	void FetchFrom(OpenFile *file);	Truy cập – đọc dãy bit từ một file
	void WriteBack(OpenFile *file)	Lưu trữ – viết dãy bit vào một file

4. Thread

Thread: tạo ra các tiểu trình bao gồm việc nạp và cấp phát vùng nhớ Stack, quản lý trạng thái của tiến trình.

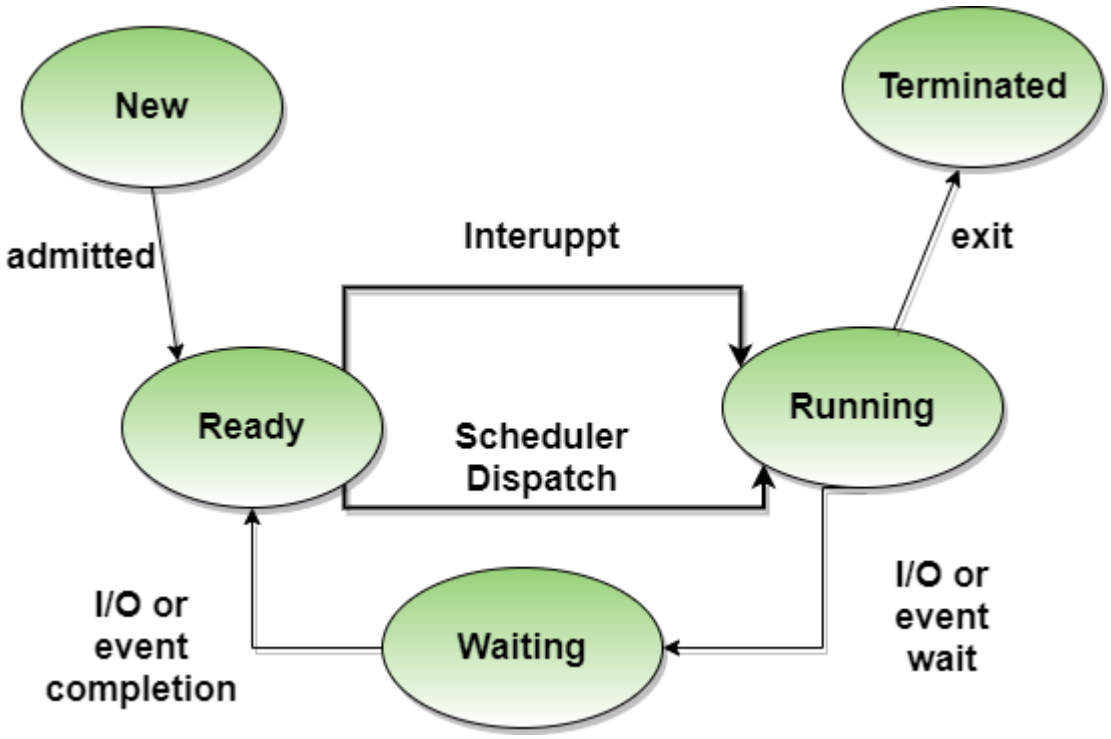
		Chức năng
Thuộc tính	int* stack;	Dưới cùng của ngăn xếp. NULL nếu đây là thread chính. Nếu NULL, không thu hồi vùng nhớ stack
	ThreadStatus status;	các trạng thái JUST_CREATED, RUNNING, READY, BLOCKED
	int* stackTop;	con trỏ ngăn xếp hiện tại
	int machineState[MachineStateSize];	tất cả các thanh ghi ngoại trừ stackTop
Phương thức	void StackAllocate(VoidFunctionPtr func, int arg);	Cấp phát một ngăn xếp cho thread. Được sử dụng private bên trong Fork()
	void Fork(VoidFunctionPtr func, int arg);	chuyển trạng thái từ JUST_CREATED sang READY
	void Yield();	chuyển trạng thái từ RUNNING sang READY nếu hàng đợi của ready không rỗng. Nó đặt thread hiện tại trở về cuối của hàng đợi ready và chuyển thread ở đầu ready queue sang running queue. Nếu hàng đợi ready rỗng, nó không ảnh

		hưởng gì và thread hiện tại sẽ tiếp tục chạy
	void Sleep();	Hàm Sleep() dùng để chuyển đổi trạn thái từ RUNNING sang BLOCKED. Hàm này thường được gọi khi một thread bắt đầu một yêu cầu nhập/ xuấ hoặc phải đợi một sự kiện nào đó và nó không thể tiếp tục nếu quá trình nhập xuất hoặc sự kiện không xảy ra
	void Finish();	chấm dứt một thread
	void CheckOverflow();	Kiểm tra xem thread có bị tràn ngăn xếp của nó hay không

III. THIẾT KẾ VÀ CÀI ĐẶT

1. Tìm hiểu mô hình quản lý tiến trình

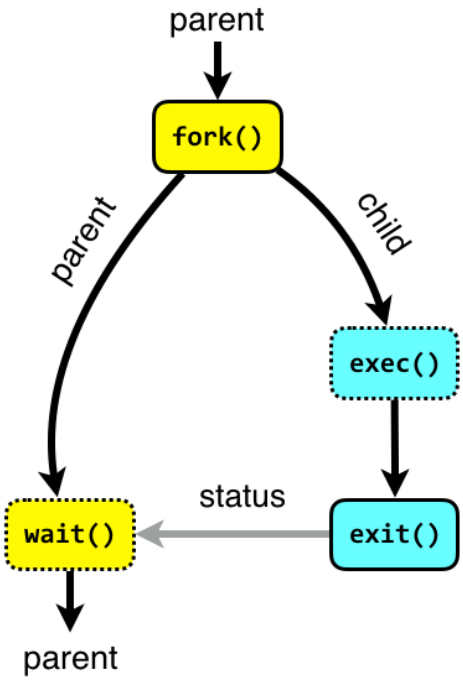
- Khi một tiến trình được thực thi, nó sẽ chuyển qua các trạng thái khác nhau. Ở một thời điểm, nó có thể ở một trong năm trạng thái dưới đây.



- **New**: Tiến trình đang được tạo lập
- **Ready**: Tiến trình đang chờ được cấp phát CPU để xử lý
- **Waiting**: Tiến trình phải đợi một sự kiện nào đó xảy ra (như nhập/xuất..)
- **Running**: Tiến trình được thực thi
- **Terminated**: Tiến trình hoàn tất xử lý

❖ Mô hình quản lý tiến trình

- Một trong những triết lý của unix là: **do one thing and do it well.** Trên tinh thần này, quản lý tiến trình cơ bản được thực hiện với một số system calls. Mỗi system call có một mục đích duy nhất (đơn giản). Các system calls này có thể được kết hợp để thực hiện các hành vi phức tạp hơn
- Hình bên dưới thể hiện một mô hình quản lý tiến trình đơn giản sử dụng các system calls như fork(), exec(), exit(), wait() sẽ được trình bày chi tiết phía dưới



- **fork**

  ➔ Tiến trình cha dùng fork để tạo ra tiến trình con mới. Tiến trình con là bản copy của cha. Sau khi fork, cả tiến trình cha và con đều thực thi cùng một chương trình nhưng ở hai tiến trình riêng biệt

- **exec**

  ➔ Tiến trình con có thể sử dụng exec sau fork để thay đổi vùng nhớ của tiến trình bằng một chương trình mới, làm cho tiến trình con thực thi một chương trình khác với chương trình cha

- **exit**

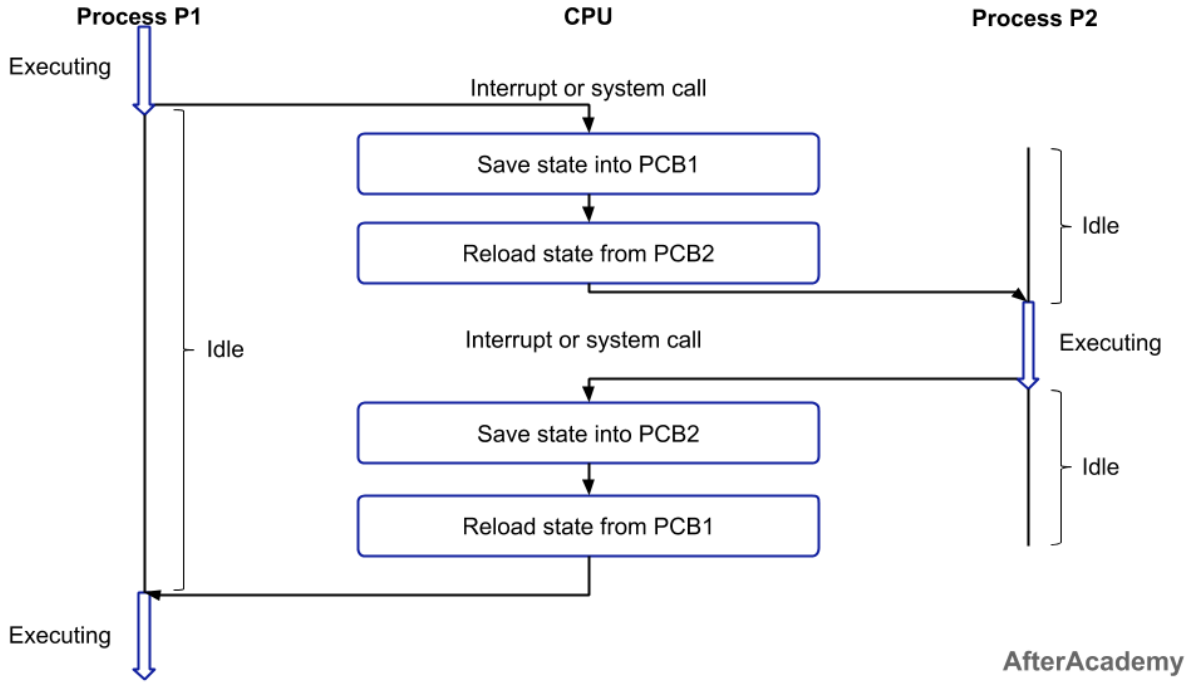
  ➔ Kết thúc một tiến trình với exit status

- **wait**

  ➔ Tiến trình cha dùng wait để tạm ngừng thực thi cho đến khi tiến trình con kết thúc. Tiến trình cha có thể nhận được exit status của tiến trình con khi kết thúc

❖ Context Switching

Việc điều phối các process sẽ bao gồm, việc ngừng process hiện tại lại, lưu lại trạng thái của process này, lựa chọn process tiếp theo sẽ được chạy, load trạng thái của process tiếp theo đó lên, rồi chạy tiếp process tiếp theo. Quá trình này được gọi là Context Switch. Đây là đặc tính tiêu biểu của hệ điều hành đa nhiệm, giúp cho các tiến trình có thể chia sẻ một CPU duy nhất.



2. Các lớp và các hàm quan trọng

Vậy hệ điều hành tổ chức một tiến trình như thế nào? Mỗi một tiến trình sẽ được biểu diễn bằng một PCB (Process Control Block) và tất cả các tiến trình sẽ được quản lý bằng lớp PTable

Trong đó có các thuộc tính và phương thức quan trọng như :

- Lớp PCB:
  + Thread* : Con trỏ trỏ đến thread mà đối tượng thuộc lớp pcb quản lý
  + parentID: id của tiến trình cha

+ 3 thuộc tính Semaphore: Để quản lý các quá trình Join, Exit và nạp thread
+ Exec(char* filename, int id) : Tạo một thread có tên là filename và process là id

- Lớp PTable:
  + PCB* pcb[MAX_PROCESS]: Mảng con trỏ trỏ đến các đối tượng lớp pcb mà đối tượng lớp ptable quản lý. Trong phạm vi đồ án này, MAX_PROCESS = 10, tức ptable quản lý tối đa 10 tiến trình.
  + ExecUpdate(char* name): Trả về kết quả thực thi của pcb->Exec

Chi tiết của hai lớp PCB và PTable đã được báo cáo phía trên.

3. Cài đặt chi tiết

- Bước 1
  + Thêm các lớp PCB và PTable vào thư mục code/threads/ của hệ điều hành nachos
  + Chỉnh sửa lại file Makefile.common
- Bước 2
  + Trong file system.h ta thêm các file pcb.h và ptable.h. Khai báo các biến con trỏ addrLock, gPhysPageBitMap, pTab
  + Trong file system.cc: Tạo lập đối tượng addrLock, gPhysPageBitMap, pTab và hủy đối tượng
- Bước 3
  + Chỉnh sửa file machine.h
    #define NumPhysPages 128
  + Chỉnh sửa file disk.h
    #define SectorSize 512
- Bước 4
  + Chỉnh sửa addrspace.cpp (/userprog): Chương trình hiện tại chỉ thực thi được 1 chương trình, cần phải chỉnh sửa trong file

addrspace.h và addrspace.cc để hệ điều hành thực hiện được đa chương

- Thay pageTable[i].physicalPage = i bằng hàm tìm 1 trang trống và đánh dấu đã sử dụng
- Thêm đoạn code xử lý nếu numPage > số trang còn trống

- Bước 5
  + Cài đặt system call Exec: gọi để thực thi một file
    - Đọc địa chỉ tên chương trình "name" từ thanh ghi r4.
    - Tên chương trình lúc này đang ở trong user space. Gọi hàm User2System đã được khai báo trong lớp machine để chuyển vùng nhớ user space tới vùng nhớ system space.
    - Nếu bị lỗi thì báo "Không mở được file" và gán -1 vào thanh ghi 2.
    - Nếu không có lỗi thì gọi pTab.ExecUpdate(name), trả về và lưu kết quả thực thi phương thức này vào thanh ghi r2.
  + Cài đặt các system call hỗ trợ là Join và Exit

IV. HƯỚNG DẪN SỬ DỤNG CHƯƠNG TRÌNH

❖ Chương trình Ping: In ra màn hình 1000 chữ A

```
#include "syscall.h"
void main()
{
int i;
for (i=0; i<1000; i++)
PrintChar('A');
}
```

❖ Chương trình Pong: In ra màn hình 1000 chữ B

```
#include "syscall.h"
void main()
```

```
{
int i;
for (i=0; i<1000; i++)
PrintChar('B');
}
```

❖ Chương trình scheduler: In xen kẽ A B để kiểm tra hệ điều hành có multitasks không

```
#include "syscall.h"
void main()
{    int pingPID, pongPID;          PrintString("Ping-
Pong test starting ...\n\n");
     pingPID = Exec("./test/ping", 4);
     pongPID = Exec("./test/pong", 4);
}
```

❖ Kết quả

V. TÀI LIỆU THAM KHẢO

https://github.com/nguyenthanhchungfit/Nachos-Programing-HCMUS/

http://www.it.uu.se/education/course/homepage/os/vt18/module-2/process-management/

https://kipalog.com/posts/He-dieu-hanh--Process

Cac-Lop-Trong-Project-3.docx