TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

PROJECT 3

ĐA CHƯƠNG TRONG HỆ ĐIỀU HÀNH NACHOS

Lớp: 19_1
Môn: Hệ điều hành
GVHD: ThS. Lê Viết Long

Niên khóa: 2021-2022

Mục lục

I. Thông tin nhóm, phân công công việc và mức độ hoàn thành

1. Thông tin nhóm:

MSSV	Họ và tên	Email
19120189	Lê Tiến Đạt	19120189@student.hcmus.edu.vn
19120190	Nguyễn Văn Đạt	19120190@student.hcmus.edu.vn
19120399	Nguyễn Tiến Toàn	19120399@student.hcmus.edu.vn
19120492	Đỗ Thái Duy	19120492@student.hcmus.edu.vn
19120603	Nguyễn Bá Ngọc	19120603@student.hcmus.edu.vn

2. Bảng phân công công việc:

MSSV	Học và tên	Nhiệm vụ	Tiến độ
19120189	Lê Tiến Đạt	+ Viết báo cáo: Tìm hiểu và mô tả lớp PCB. + Viết code: Thay đổi 2 file addrspace.h và addrspace.cc để chuyển hệ thống từ đơn chương thành đa chương, chỉnh sửa thông số trong machine.h, disk.h	100%

19120190	Nguyễn Văn Đạt	+ Viết báo cáo: Tìm hiểu và mô tả lớp Ptable. + Viết code: Viết hàm thay đổi lệnh nạp user program lên bộ nhớ, sửa các lỗi của các hàm có sẵn trong 2 file ptable.c, pcb.c	100%
19120399	Nguyễn Tiến Toàn	+ Viết báo cáo: Viết cài đặt và thiết kế của chương trình Ping Pong. + Viết code: Viết system call SC_Exec, và các system call liên quan. Viết các hàm xử lý chính và các khai báo cần thiết trong các lớp liên quan đến đồ án.	100%
19120492	Đỗ Thái Duy	+ Viết báo cáo: Tìm hiểu và mô tả lớp Bitmap. Format bài báo cáo. + Viết code: Viết 3 chương trình ping.c, pong.c, scheduler.c trong thư mục test. Định nghĩa các chương trình trong test/Makefile.	100%

19120603	Nguyễn Bá Ngọc	+ Viết báo cáo: Tìm hiểu và mô tả lớp Thread. + Viết code: Format lại code và bổ sung các ghi chú (comment) hàm còn thiếu.	100%

3. Đánh giá mức độ hoàn thành:

STT	Tên công việc	Mức độ hoàn thành
1	Tìm hiểu và mô tả lớp Ptable	100%
2	Tìm hiểu và mô tả lớp PCB	100%
3	Tìm hiểu và mô tả lớp Bitmap	100%
4	Tìm hiểu và mô tả lớp Thread	100%
5	Viết chương trình Ping Pong để minh họa đa chương	100%
	Toàn bộ project	100%

II. Nội dung báo cáo:

Phần 1: Tìm hiểu và mô tả lớp Ptable

- **Lớp Ptable**: dùng để quản lý các tiến trình được chạy, gồm thuộc tính pcb là một bảng mô tả tiến trình, có cấu trúc mảng một chiều có số phần tử tối đa là 10, mỗi phần tử của mảng thuộc kiểu dữ liệu PCB.

1. Các thuộc tính:

- **BitMap *bm**;

Quản lý các đối tượng đã dùng và còn trống chưa dùng trong đối tượng lớp Ptable.

- **PCB *pcb[MAXPROCESS]**;

Mảng con trỏ trỏ đến các đối tượng lớp PCB mà đối tượng lớp PTable quản lý.

MAXPROCESS: Số lượng thread tối đa mà đối tượng PTable quản lý.

- **int psize**;

Lưu kích thước của pcb cần khởi tạo.

- **Semaphore *bmsem**;

Quản lý việc nạp thread. Dùng để ngăn chặn trường hợp nạp 2 tiến trình cùng một lúc.

2. Các phương thức:

- **PTable(int size)**;

+ **Chức năng**: Khởi tạo size đối tượng **PCB** để lưu size process. Gán giá trị ban đầu là null.

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 system call **Exec()**.

+ **Các bước cài đặt**:

a. Lưu giá trị **size** vào **psize** (size là số đối tượng pcb cần tạo).

b. Khởi tạo **bm** lưu bảng bitmap có thể lưu được "size" bit.

c. Khởi tạo **bmsem** là một Semaphore với tên là BMSem, dùng để quản lý việc nạp

Thread.

d. Chạy vòng for để gán từng giá trị của pcb là null.
e. Đánh dấu bm[0] đã sử dụng bởi tiến trình đầu tiên, khởi tạo và lưu tiến trình đó vào pcb[0].

- **~PTable();**

+ **Chức năng**: Hàm hủy Ptable.

+ **Các bước cài đặt**:

a. Kiểm tra và xóa **bm**.
b. Kiểm tra và xóa **bmsem**.
c. Dùng vòng lập for, kiểm tra và xóa từng **pcb[i]**.

- **int ExecUpdate(char* filename)**;

+ **Chức năng**: Hàm tạo **thread** mới với đường dẫn được truyền vào. Kết quả trả ra 0 nếu thành công và -1 nếu thất bại.

+ **Các bước cài đặt**:

a. Gọi **mutex->P()**: để giúp tránh tình trạng nạp 2 tiến trình cùng một lúc.
b. Kiểm tra sự tồn tại của chương trình "filename" bằng cách gọi phương thức Open của lớp fileSystem.
c. So sánh tên chương trình và tên của **currentThread** để chắc chắn rằng chương trình này không gọi thực thi chính nó bằng cách so sánh filename và tên **currentThread** bằng hàm **strcmp**.
d. Dùng hàm **GetFreeSlot()** để tìm slot trống trong bảng **PTable**, nếu trả về -1 thì không còn tiến trình trống -> lỗi.
e. Nếu còn ô trống thì khởi tạo một PCB mới với **processID** là ID tìm được với **GetFreeSlot()**, gán tên tiến trình cho PCB vừa khởi tạo.
f. Gán giá trị **parentID** của pcb vừa khởi tạo là **processID** của **currentThread.**
g. Gọi thực thi phương thức Exec của lớp PCB, sau đó gọi **bmsem->V()** và trả về kết quả thực thi của **PCB->Exec.**

- **int ExitUpdate(int ec)**;

+ **Chức năng**: Hàm **exit thread** hiện tại với exit code được truyền vào. Kết quả trả ra 0 nếu thành công và -1 nếu thất bại.

+ **Các bước cài đặt**:

a. Gán **processID** của Thread hiện tại đang chạy vào **pID**, kiểm tra **pID** có tồn tại hay không bằng hàm **isExit()**, nếu không trả về -1.
b. Nếu pID = 0, có nghĩa là Thread hiện tại là main process, chạy hàm **Halt()** và trả về 0.
c. Còn lại, gọi **setExitCode** của lớp PCB để đặt exitcode cho tiến trình đang gọi.
d. Kiểm tra trạng thái của pcb đang xét có **Join** với tiến trình nào không, nếu không thì tiến hành **JoinRelease()** để giải phóng tiến trình cha đang đợi, **ExitWait()** để xin tiến trình cha cho phép thoát và **Remove pID**.
e. Nếu trạng thái hợp lệ chạy hàm **Remove pID**.
f. Trả về **ec**.

- **int JoinUpdate(int pID)**;

+ **Chức năng**: Hàm join thread hiện tại với thread con và chờ thread con chạy xong. Id của thread con được truyền vào. Kết quả trả ra 0 nếu thành công và -1 nếu thất bại.

+ **Các bước cài đặt**:

a. Kiểm tra tính hợp lệ của **processID** là **pID** và kiểm tra **pcb[pID]** có phải là null hay không.
b. Nếu không có lỗi gì ta tiếp tục kiểm tra xem tiến trình gọi **Join** có phải là cha của tiến trình có **processID** là **pID** hay không.
c. Gọi hàm **JointWait()** của lớp PCB để đợi tiến trình con kết thúc.
d. Khi tiến trình con thực hiện xong hết, ta tiến hành hàm **GetExitCode()**, gán giá trị trả về vào int **ec**, kiểm tra **ec** có hợp lệ không, nếu không thì trả về 1.
e. Kết thúc các tiến trình con bằng cách gọi hàm **ExitRelease** của lớp PCB.
f. Trả về 0.

- **int GetFreeSlot()**;

+ **Chức năng**: Tìm vị trí trống để lưu thông tin của tiến trình mới.

+ **Các bước cài đặt**:

a. Gọi hàm **Find()** của lớp Bitmap để trả về bit đầu tiên trong **numBits** chưa được đánh dấu, sau đó đánh dấu bit đó. Nếu tất cả bit đều được đánh dấu thì trả về -1.

b. Return kết quả của hàm **Find()** trên.

- **bool IsExist(int pID)**;

+ **Chức năng**: Kiểm tra tiến trình có processID là pID đã Exit hay chưa.

+ **Các bước cài đặt**:

a. Kiểm tra **pID** có hợp lệ hay không, nếu không return 0.

b. Gọi hàm **Test()** của lớp Bitmap để kiểm tra bit thứ pID đã được đánh dấu hay chưa, trả về kết quả của hàm **Test()**.

- **void Remove(int pID)**;

+ **Chức năng**: Xóa ID của tiến trình đang chọn ra khỏi table khi tiến trình kết thúc.

+ **Các bước cài đặt**:

a. Kiểm tra **pID** có hợp lệ hay không, nếu không thì return hàm.

b. Kiểm tra bit thứ **pID** đã được đánh dấu hay chưa bằng hàm **Test()** của lớp Bitmap, nếu được đánh dấu rồi, ta xóa bit **pID** trong bm bằng hàm **Clear()** rồi delete **pcb[pID].**

- **char* GetName(int pID)**;

+ **Chức năng**: Trả về tên của tiến trình **pcb[pID]**.

+ **Các bước cài đặt**:

a. Kiểm tra tính hợp lệ của **pID**.

b. Kiểm tra bit **pID** đã được đánh dấu hay chưa bằng hàm **Test()** của lớp Bitmap.

c. Nếu thỏa những điều kiện là **pID** hợp lệ và bit **pID** đã được đánh dấu, trả về tên của **pcb[pID]** bằng hàm **GetNameThread()** của lớp PCB.

Phần 2: Tìm hiểu và mô tả lớp PCB

- **Lớp PCB**: hay tên đầy đủ là **Process Control Block** là 1 khối điều khiển tiến trình dùng để lưu trữ dữ liệu cần thiết để mô tả một tiến trình và các hành động của nó đồng thời quản lý tiến trình đó 1 cách hiệu quả. Trong một số hệ điều hành, PCB được đặt ở đầu của ngăn xếp nhân của tiến trình.

1. Các thuộc tính:

- **Semaphore *joinsem**;

Semaphore cho quá trình join thread.

- **Semaphore *exitsem**;

Semaphore cho quá trình exit thread.

- **Semaphore *mutex**;

Semaphore cho quá trình truy xuất độc quyền.

- **int exitcode**;

Là exitcode của tiến trình hiện tại có giá trị ban đầu là 0.

- **Thread *thread**;

Con trỏ thread trỏ đến thread mà PCB đang quản lý.

- **int pid**;

Là id của thread hiện tại mà PCB đang quản lý.

- **int numwait**;

Là số thread con join được execute.

- **int parentID**;

Là ID cua thread cha.

- **int JoinStatus**;

Là trạng thái hiện tại của PCB có Join với tiến trình nào không? Nếu không thì là giá trị mặc định -1, nếu có thì giá trị chính là ID của tiến trình mà nó Join.

2. Các phương thức:

- **PCB(int id)**;

+ **Chức năng**: Hàm khởi tạo mặc định của lớp PCB.

+ **Các bước cài đặt**:

a. Khởi tạo **joinsem** là 1 Semaphore tên là "JoinSem" có giá trị là 0.
b. Khởi tạo **exitsem** là 1 Semaphore tên là "ExitSem" có giá trị là 0.
c. Khởi tạo **mutex** là 1 Semaphore tên là "Mutex" có giá trị là 1.
d. Gán **pid thread** đang quản lý là **id** được truyền vào.
e. **exitcode** và **numwait** đều gán giá trị bằng 0.
f. Nếu như id khác 0 thì gán id thread cha bằng **processID** của thread hiện tại. Còn không thì id thread cha bằng 0 (không có).
g. Con trỏ **thread** bằng null và trạng thái join = -1.

- **~PCB()**;

+ **Chức năng**: Hàm phá huỷ của lớp PCB.

+ **Cách cài đặt**:

Nếu có tồn tại các con trỏ **joinsem**, **exitsem** hay **mutex** thì delete (xóa) các con trỏ đó.

- **int Exec(char*filename, int pID);**

+ **Chức năng**: Nạp vào chương trình có tên lưu trong biến filename và processID sẽ là pID.

+ **Các bước cài đặt**:

a. Đặt **semaphore mutex** đầu hàm để xem giá trị có lớn hơn 0 thì mới trừ đi 1 rồi cho thực thi tiếp.
b. Khởi tạo tiến trình cho chương trình có tên lưu trong **filename**.
c. Nếu không tạo được tiến trình thì in ra màn hình thông báo tăng **semaphore mutex** lên 1 đơn vị rồi trả về giá trị là -1.
d. Nếu tạo tiến trình thành công thì cho id tiến trình là **pID** đã lưu.
e. Tiến hành chạy tiến trình bằng cách **FORK** hàm **MyStartProcess()** với pID.

f. Tăng **semaphore mutex** lên 1 đơn vị rồi trả về pID.

- **Int GetID();**

+ **Chức năng**: Trả về ID của tiến trình được gọi thực hiện.

+ **Cách cài đặt**: trả về **pid**.

- **Int GetNumWait();**

+ **Chức năng**: Trả về số lượng tiến trình chờ.

+ **Cách cài đặt**: trả về **numwait**.

- **Int GetExitCode();**

+ **Chức năng**: Trả về **exitcode** của tiến trình được gọi.

+ **Cách cài đặt**: trả về **exitcode**.

- **Void SetExitCode(int ec);**

+ **Chức năng**: Gán **exitcode** của tiến trình được gọi bằng giá trị trong biến **ec.**

+ **Cách cài đặt**: Gán **exitcode** bằng **ec**.

- **Void IncNumWait();**

+ **Chức năng**: Tăng số lượng tiến trình chờ lên 1.

+ **Cách cài đặt**: **numwait**++ ;

- **Void DecNumWait();**

+ **Chức năng**: Giảm số lượng tiến trình chờ đi 1.

+ **Cách cài đặt**: Nếu như số lượng tiến trình đang chờ bé hơn 1 thì **numwait**-- ;

- **char* GetNameThread();**

+ **Chức năng**: Trả về tên của tiến trình được gọi.

+ **Cách cài đặt**: dùng hàm **getName()** trong **thread.h**;

- **void JoinWait();**

+ **Chức năng**: Yêu cầu thread hiện tại sẽ vào chế độ chờ thread con mới sắp join vào.

+ **Các bước cài đặt**:

a. Gán **JoinStatus** là **parentID.**

b. Tăng số **numwait** lên 1 đơn vị vì sắp join thêm 1 tiến trình.

c. Kiểm tra **semaphore joinsem** có lớn hơn 0 hay không nếu có thì giảm đi 1 đơn vị và tiếp tục nếu không thì rơi vào trạng thái chờ.

- **void JoinRelease();**

+ **Chức năng**: Giải phóng Thread đang chờ.

+ **Các bước cài đặt**:

a. Giảm số **numwait** đi 1 đơn vị vì giải phóng đi 1 tiến trình.

b. Tăng **semaphore joinsem** lên 1 đơn vị rồi gọi đến thread có **joinsem** đang trong trạng thái chờ.

- **void ExitWait();**

+ **Chức năng**: Thread hiện tại chờ thread con kết thúc.

+ **Các bước cài đặt**:

Kiểm tra **semaphore exitsem** có lớn hơn 0 hay không nếu có thì giảm đi 1 đơn vị và tiếp tục nếu không thì rơi vào trạng thái chờ.

- **void ExitRelease()**

+ **Chức năng**: Giải phóng thread cha nếu thread cha đang trong trạng thái exit và chờ các thread con exit.

+ **Các bước cài đặt**:

Tăng **semaphore exitsem** lên 1 đơn vị rồi gọi đến thread có **exitsem** đang trong trạng thái chờ.

- **void MyStartProcess(int pID)**;

+ **Chức năng**: Tiến hành khởi động tiến trình có ID trong pID để thực hiện **multithreading**.

+ **Các bước cài đặt**:

a. Tạo con trỏ **filename** trỏ tới tên của tiến trình sử dụng pID thông qua bảng ptable.
b. Tạo chỗ trong bộ nhớ cho tiến trình bằng cách khởi tạo biến **AddSpace**.
c. Nếu như bộ nhớ không đủ cấp phát cho tiến trình thì thông báo lên màn hình rồi thoát.
d. Nếu như đủ bộ nhớ thì cấp phát cho tiến trình .
e. Đăng ký vị trí bộ nhớ cho tiến trình.
f. Đi đến **page table** trong bộ nhớ.
g. Cuối cùng thực hiện lệnh **machine->Run()** vì **machine->Run()** không trả về gì cả nên thêm **ASSERT(FALSE)** đằng sau nó.

Phần 3: Tìm hiểu và mô tả lớp Bitmap

- **Lớp bitmap** là một dãy bit dùng để quản lý các thành phần đã được khởi tạo hay chưa trong một mảng. Lớp này thường được biểu diễn dưới dạng một mảng các số nguyên không dấu, trên đó ta dùng thuật toán modulo số học (chia lấy dư) để tìm bit mà mình quan tâm.

1. Các thuộc tính:

- **int numBits;**

Cho biết số lượng bit trong bitmap.

- **int numWords;**

Cho biết số lượng từ mà bitmap lưu trữ.

- **unsigned int *map;**

Mảng số nguyên không dấu được dùng để lưu trữ bit.

2. Các phương thức:

- **BitMap(int nitems);**

+ **Chức năng**: Khởi tạo một **bitmap** với số bit là "**nitems**", các bit trong mảng được xóa sạch và rỗng để có thể được thêm vào bất cứ đâu trong 1 danh sách.

+ **Các bước cài đặt**:

a. Gán thuộc tính **numBits** bằng tham số nitems được truyền vào
b. Thực hiện tính toán để tìm và làm tròn **numWords** bằng hàm divRoundUp nếu **numBits** không phải là bội số của 32 (số bit trong 1 từ).
c. Cấp phát mảng map với số lượng phần tử bằng **numWords.**
d. Xóa các bit từ 0 đến **numBits** để đảm bảo bit rỗng.

- **~BitMap();**

+ **Chức năng**: Được dùng để xóa mảng map và trả quyền quản lý vùng nhớ về lại cho hệ điều hành.

+ **Các bước cài đặt**:

Hủy mảng bằng **delete map**;

- **void Mark(int which);**

+ **Chức năng**: Đánh dấu bit thứ **which** trong mảng **bitmap**.

+ **Các bước cài đặt**:

a. Gọi hàm **ASSERT()** để kiểm tra điều kiện đầu vào của which. Nếu **which >= 0** và **< numBits** thì tiếp tục. Ngược lại sẽ báo lỗi và kết thúc chương trình.
b. Đánh dấu phần tử thứ ***which/32*** trong mảng **map** (bitmap) bằng 1, thông qua phép **OR** giữa giá trị của phần tử này với giá trị của phép dịch trái ***which%32*** bit của số 1. (phần dư của which khi chia cho 32)

- **void Clear(int which);**

+ **Chức năng**: Xóa bit thứ **which**.

+ **Các bước cài đặt**:

a. Gọi hàm **ASSERT()** để kiểm tra điều kiện đầu vào của which. Nếu **which >= 0** và **< numBits** thì tiếp tục. Ngược lại sẽ báo lỗi và kết thúc chương trình.
b. Ta thay đổi giá trị của phần tử thứ ***which/32*** trong mảng **map** (bitmap) bằng 0, thông qua phép **AND** giữa giá trị của phần tử này với giá trị của phủ định phép dịch trái ***which%32*** bit của số 1.

- **bool Test(int which);**

+ **Chức năng**: Kiểm tra bit thứ **which** đã được đánh dấu hay chưa?

+ **Các bước cài đặt**:

a. Gọi hàm **ASSERT()** để kiểm tra điều kiện đầu vào của which. Nếu **which >= 0** và **< numBits** thì tiếp tục. Ngược lại sẽ báo lỗi và kết thúc chương trình.
b. Kiểm tra bit thứ which đã được đánh dấu nếu tồn tại giá trị 1 của phép **AND** giữa giá trị của phần tử thứ ***which/32*** trong mảng **map** (bitmap) và giá trị của phép dịch trái ***which%32*** bit của số 1. Còn ngược lại giá trị 0 thì trả về FALSE (chưa được đánh dấu).

- **int Find();**

+ **Chức năng**: Trả về bit đầu tiên trong **numBits** chưa được đánh dấu, và sau đó đánh dấu nó. Nếu tất cả bit đều đã đánh dấu thì trả về -1.

+ **Các bước cài đặt**:

a. Chạy vòng lặp từ 0 đến **numBits-1**.
b. Kiểm tra nếu bit thứ i đầu tiên trong vòng lặp chưa được đánh dấu thì đánh dấu nó bằng hàm **Mark()** và trả về số **i** này.
c. Nếu chạy hết vòng lặp mà không tìm được bit thứ **i** nào thỏa thì trả về -1.

- **int NumClear();**

+ **Chức năng**: Trả về số lượng các bit chưa được đánh dấu.

+ **Các bước cài đặt**:

a. Khởi tạo biến số nguyên count để lưu trữ số lượng các bit chưa được đánh dấu.

b. Chạy vòng lặp từ 0 đến **numBits-1**.

c. Kiểm tra nếu bit thứ **i** trong vòng lặp chưa được đánh dấu thì tăng biến count lên 1.

d. Hoàn thành vòng lặp và trả về **count**.

- **void Print()**;

+ **Chức năng**: Xuất ra màn hình giá trị các bit trong mảng **map** (bitmap). Được dùng chủ yếu để debug.

+ **Các bước cài đặt**:

a. Chạy vòng lặp từ 0 đến **numBits-1**.

b. Nếu bit thứ **i** được đánh dấu thì xuất ra màn hình.

- **void FetchFrom(OpenFile *file);**

+ **Chức năng**: Truy cập vào để đọc dãy bit từ một file **Nachos.**

+ **Các bước cài đặt**:

Gọi hàm **ReadAt()** trong biến file để đọc ***numWords * kích cỡ của kiểu dữ liệu unsigned int*** phần tử bắt đầu từ chỉ số 0 vào mảng **map** (ép kiểu thành **char***).

- **void WriteBack(OpenFile *file);**

+ **Chức năng**: Lưu trữ dãy bit bằng cách viết vào 1 file **Nachos**.

+ **Các bước cài đặt**:

Gọi hàm **WriteAt()** trong biến file để ghi vào file ***numWords * kích cỡ của kiểu dữ liệu unsigned int*** phần tử bắt đầu từ chỉ số 0 từ mảng **map** (ép kiểu thành **char***).

Phần 4: Tìm hiểu và mô tả lớp Thread

- **Lớp thread** là một lớp hữu dụng để phục vụ quản lý đa chương với từng đơn tiến trình. Nó sẽ chia sẻ với các luồng khác về thông tin data, các dữ liệu của mình. Do đó các thread có thể truy cập lẫn nhau với các biến toàn cục. Để dễ dàng quản lý, chúng được gán các trạng thái: **JUST_CREATED**, **RUNNING**, **READY**, **BLOCKED**.

+ **JUST_CREATED**: thread đã được tạo nhưng stack của nó vẫn còn trống. (nằm trong hàm khởi tạo)

+ **RUNNING**: trong trạng thái đang được sử dụng

+ **READY**: trong trạng thái sẵn sàng dùng cho CPU. Khi được chọn sẽ chuyển sang trạng thái running.

+ **BLOCKED**: trạng thái bị khóa (nằm trong hàm Sleep()). Khi đó tiến trình sẽ đợi một sự báo hiệu nào đó. Sẽ không nằm trong danh sách sẵn sàng chạy khi đang trong trạng thái này.

1. Các thuộc tính:

- **int* stacktop**;

Con trỏ stack hiện tại.

- **int* stack;**

Khởi tạo một stack, stack này nhận giá trị NULL nếu đây là thread chính.

- **ThreadStatus status;**

Trạng thái của thread hiện tại.

- **char* name;**

Tên thread, đường dẫn tương đối tới chương trình thực thi.

- **int machineState[MachineStateSize];**

Danh sách tất cả các thanh ghi ngoại trừ stacktop.

- **void stackAllocate(VoidFunctionPtr func, int arg);**

Dùng để phân bổ một stack cho thread, biến này được sử dụng nội bộ bởi hàm Fork()

- **int userRegisters[NumtotalRegs];**

Tạo ra một mảng để lưu giữ cấp độ trạng thái đăng ký CPU của người dùng.

2. Các phương thức:

- **Thread(char* debugName);**

+ Chức năng: Khởi tạo một thread.

+ Các bước cài đặt:

a. Định danh cho **thread**.

b. Khởi tạo các giá trị ban đầu cho stacktop, stack và space(vùng nhớ của tiến trình trên ram ảo) là **NULL**.

c. Khởi tạo trạng thái ban đầu cho thread vừa tạo là **JUST_CREATED**.

- **~Thread();**

+ Chức năng: Hủy(xóa) một thread.

+ Các bước cài đặt:

a. Xác định thread cần xóa hoặc hủy có phải là thread hiện tại không.

b. Kiểm tra xem thead cần xóa hoặc hủy có phải là thread chính hay không.

c. Tiến hành xóa hoặc hủy thread được yêu cầu.

- **void Fork(VoidFunctionPtr func, int arg)**;

+ Chức năng: Khởi tạo các thông tin cần thiết cho một tiểu trình và đưa nó vào CPU để thực hiện.

+ Các bước cài đặt:

a. Gọi lại biến **void stackAllocate(VoidFunctionPtr func, int arg)** để phân bổ một stack cho thread (trong lúc cấp phát vùng nhớ này, **Fork** gán con trỏ hàm **voidFunctionPtr** và số nguyên int vào các thanh ghi đặc biệt).

b. Chuyển tiến trình vào **readylist**.

- **void Yield();**

+ Chức năng: Nhường **CPU** cho một thread trong **readylist**.

+ Các bước cài đặt:

a. Khai báo một biến **thread** kế tiếp để chạy.

b. Kiểm tra xem chương trình đang gọi hàm có phải là **thread** hiện tại không. Nếu không phải thì báo lỗi.

c. Thêm thread vừa khai báo vào hàng đợi. Kiểm tra nếu thread vừa khai báo có giá trị **NULL** thì tiếp tục chạy thread hiện tại và ngược lại.

- **void Sleep();**

+ Chức năng: đưa tiến trình vào trạng thái **BLOCKED**.

+ Các bước cài đặt:

a. Khai báo một biến thread kế tiếp.

b. Kiểm tra xem chương trình đang gọi hàm có phải là thread hiện tại hay không. Điều chỉnh status của thread hiện tại về trạng thái **BLOCKED** và xóa bỏ thread đó ra khỏi danh sách sẵn sàng.

c. Tạo một vòng lặp để kiểm tra danh sách đó, nếu nó không trống thì gọi hàm **interrupt->Idle()** để đợi cho đến interrupt kế tiếp.

d. Thread sẽ được đưa vào danh sách sẵn sàng và gỡ trạng thái block nếu như có trường hợp cần thiết.

- **void Finish();**

+ Chức năng: Kết thúc thread đang chạy.

+ Các bước cài đặt:

a. Kiểm tra xem thread gọi hàm có phải là thread hiện tại hay không.

b. Trỏ biến toàn cục **threadToBeDestroy** đến thread hiện tại.

c. Gọi lại hàm **Sleep()**.

- **void StackAllocate(voidFunctionPtr func, int arg)**;

+ Chức năng: cấp phát stack và tạo nên các bản ghi việc kích hoạt ở đầu để phục vụ cho func.

+ Các bước cài đặt:

a. Cấp phát bộ nhớ cho stack với **StackSize**.

b. Đặt giá trị kiểm tra lên đầu stack. Bất cứ khi nào chuyển sang một thread mới, nếu một thread bị tràn stack trong quá trình hoạt động, thì **scheduler** sẽ kiểm tra giá trị có thay đổi hay không.

c. Đưa chương trình đếm PC trỏ đến **TheadRoot**. Các công việc bắt đầu từ đây thay vì tại **user-supplied**, giúp cho việc đóng một thread nhanh chóng khi cần kết thúc.

- **void SaveUserState();**

+ Chức năng: lưu trạng thái các thanh ghi-trạng thái CPU ở mức độ người dùng-khi đang thực hiện phần việc ở mức người dùng.

+ Các bước cài đặt:

a. Chạy vòng lặp với số lần dựa trên biến **NumTotalRegs**.

b. Lưu các trạng thái CPU của người dùng vào từng phần tử của biến **userRegisters**.

- **void RestoreUserState()**;

+ Chức năng: khôi phục trạng thái các thanh ghi - trạng thái CPU ở mức độ người dùng hay khi đang thực hiện phần việc ở mức người dùng.

+ Các bước cài đặt:

a. Chạy vòng lặp với số lần dựa trên biến **NumTotalRegs**.

b. Lưu trữ từng giá trị vào thanh ghi CPU thông qua biến toàn cục machine và hàm **WriteRegister(int, int)**.

- **static void ThreadFinish();**

+ Chức năng: Kết thúc 1 thread đang chạy.

+ Các bước cài đặt:

Cho biến **currentThread** trỏ đến hàm **Finish()** để thực hiện.

- **static void InterruptEnable();**

+ Chức năng: bật biến interrupt.

+ Các bước cài đặt:

Cho biến toàn cục interrupt trỏ đến hàm Enable() để thực hiện hàm.

- **void FreeSpace();**

+ Chức năng: giải phóng vùng nhớ của tiến trình trên RAM ảo.

+ Các bước cài đặt:

a. Kiểm tra biến **space** có phải là giá trị **NULL**.
b. Nếu khác **NULL** thì ta gọi **delete space**, ngược lại thì kết thúc hàm.

- **void ThreadPrint(int arg);**

+ Chức năng: Xuất tên của thread ra màn hình.

+ Các bước cài đặt:

Gọi hàm **Print()** thông qua biến thread.

- **void CheckOverFlow()**;

+ Chức năng: Kiểm tra stack của một thread để xem nó có vượt quá không gian đã được cấp phát cho nó hay không.

+ Các bước cài đặt:

a. Kiểm tra nếu **stack** != **NULL** thì tiếp tục kiểm tra ký tự ở cuối stack có phải là biến đã được define để phát hiện tràn stack hay không.
b. Nếu **stack** == **NULL**, kiểm tra xem stack có là ký tự ở cuối stack có phải là biến đã được define để phát hiện tràn stack hay không. Kết thúc hàm.

Phần 5: Viết chương trình Ping Pong để minh họa đa chương

Ghi chú: Mã nguồn của đồ án 3 kế thừa mã nguồn của đồ án 2 tụi em đã thực hiện. Nghĩa là mã nguồn của đồ án này, trước tiên, đã được cài đặt để hoạt động đầy đủ và chính xác các chức năng yêu cầu trong đồ án trước đó. Một số lớp, hàm và chức năng quan trọng trong số đó là lớp **SynchConsole** (được định nghĩa và cài đặt trong hai file **synchcons.h** và **synchcons.cc**), system call **PrintChar**, **PrintString**, các hàm **IncreaseProgram-Counter()**, **User2System()**, **System2User()**.

Các bước cài đặt:

1. Thêm lớp PTable và PCB vào NachOS và khai báo các biến toàn cục:

- Thêm bốn file **ptable.h**, **ptable.cc**, **pcb.h** và **pcb.cc** vào thư mục **threads**.
- Vào **code/Makefile.common**, khai báo **ptable.h**, **pcb.h** tại **USERPROG_H**, khai báo **ptable.cc**, **pcb.cc** tại **USERPROG_C** và khai báo **ptable.o**, **pcb.o** tại **USERPROG_O**.
- Khai báo thêm thuộc tính là **processID** kiểu int trong **threads/thread.h** để phân biệt chỉ số của các tiến trình khác nhau và để tương thích với hai lớp **PTable** và **PCB**.
- Chỉnh sửa **threads/system.h**, include lần lượt **synch.h** (để sử dụng lớp **Lock**), **bitmap.h** (để sử dụng lớp **BitMap**) và **ptable.h** (để sử dụng lớp **PTable** và **PCB**, file này đã include **pcb.h**). Khai báo các biến toàn cục **PTable *ptable**, **Lock *addrLock** và **BitMap *gPhysPageBitMap**.
- Chỉnh sửa **threads/system.cc**, cấp phát và giải phóng các biến toàn cục vừa được khai báo trong **system.h**.

2. Chỉnh sửa các thông số

- Chỉnh sửa **NumPhysPages** là **128** trong **machine/machine.h**.
- Chỉnh sửa **SectorSize** là **512** trong **machine/disk.h**.
- Trong **userprog/addrspace.h**, khai báo thêm một constructor mới của lớp **AddrSpace** là **AddrSpace(char* filename)** dùng để cấp phát bộ nhớ cho tiến trình mới thông qua đường dẫn đến file thực thi và một hàm mới dùng để tính số trang còn trống là **numFreeSlot()**.

- Trong **userprog/addrspace.cc**, cài đặt hai thay đổi mới được khai báo trong **addrspace.h**. Thao tác chính của hàm **numFreeSlot()** là gọi đến phương thức **NumClear()** của lớp **BitMap** thông qua biến **gPhysPageBitMap**. Phương thức **AddrSpace(char* filename)** gọi lại hàm **numFreeSlot()** thông qua câu lệnh **if (numPages > numFreeSlot())** để kiểm tra số trang còn trống có đủ để cấp phát cho tiến trình mới hay không, nếu không đủ thì giải phóng tiến trình này. Toàn bộ đoạn code này là:

```
if (numPages > numFreeSlot()){ // goi ham tinh so trang trong
    printf("\nAddrSpace:Load: not enough memory for new process..!");
    numPages = 0;
    delete executable;
    addrLock->Release();
    return;
}
```

Thay **pageTable[i].physicalPage = gPhysPageBitMap->Find();** cho **pageTable[i].physicalPage = i;** để tìm một trang trống và đánh dấu trang đó đã sử dụng, phục vụ đa chương.

3. Cài đặt các system call

Ta cài đặt ba system call là **Exec**, **Join** và **Exit**. **Exec** dùng để gọi thực thi một chương trình người dùng, **Join** dùng để ngưng tiến trình cha đến khi tiến trình con hoàn tất và **Exit** dùng để hủy tiến trình thực hiện chương trình người dùng. Các system call này đã được khai báo từ trước trong **userprog/syscall.h** và các thao tác cần thiết để biên dịch system call thành mã thực thi đã được cài đặt trong **test/start.c** và **test/start.s**, vì vậy ta chỉ cần chỉnh sửa phần cài đặt các system call này trong **userprog/exception.cc**.

Các cài đặt này cụ thể như sau:

a. **System call Exec**

- Đọc địa chỉ tên chương trình từ thanh ghi r4.
- Gọi hàm **User2System** để chuyển từ vùng nhớ user space tới vùng nhớ system space.
- Nếu bị lỗi thì báo “Khong mo duoc file” và gán -1 vào thanh ghi r2.

- Nếu không có lỗi thì gọi **ptable.ExecUpdate(name);**, trả về và lưu kết quả thực thi phương thức này vào thanh ghi r2.

b. **System call Join**

- Đọc id của tiến trình cần join từ thanh ghi r4.
- Gọi thực hiện phương thức **JoinUpdate()** và trả kết quả về biến **exitCode**.
- Ghi **exitCode** vào thanh ghi r2.

c. **System call Exit**

- Đọc exit status của tiến trình đã join trước đó từ thanh ghi r2, lưu vào biến **exitStatus**.
- Gọi thực hiện phương thức **ExitUpdate()** và trả kết quả về biến **exitCode**.
- Ghi giá trị **exitCode** vào thanh ghi r2.

4. Cài đặt các chương trình người dùng để kiểm tra kết quả

Ta cài đặt ba chương trình mới nằm trong các file **test/ping.c**, **test/pong.c** và **test/scheduler.c**.

Cụ thể:

ping.c

```
#include "syscall.h"

void main()
{
        int i;
        for (i = 0; i < 1000; i++){
                PrintChar('A');
        }
        Exit(0);
}
```

pong.c

```
#include "syscall.h"

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

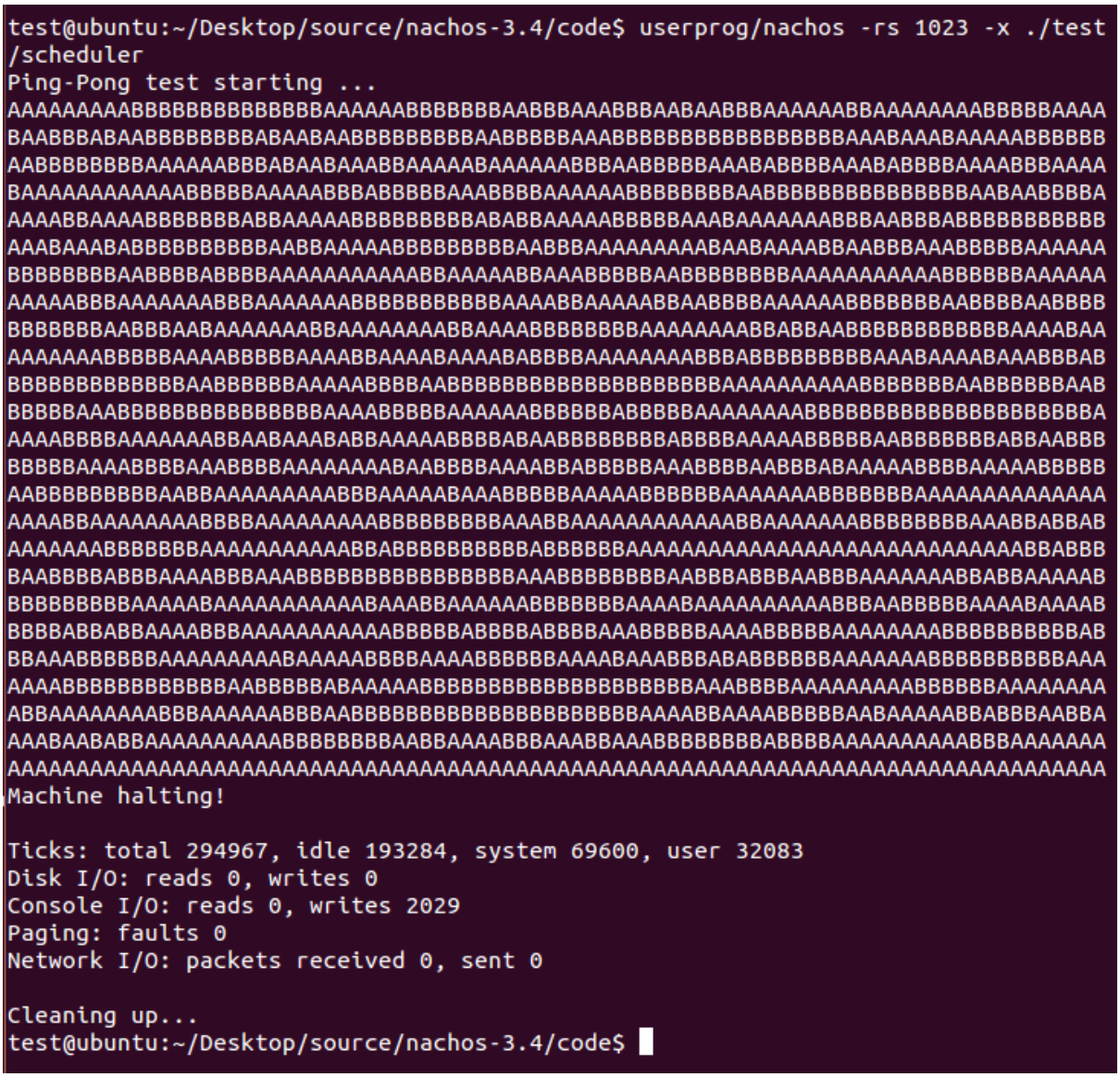
scheduler.c

```
#include "syscall.h"
void main()
{
        int pingPID, pongPID;
        PrintString("Ping-Pong test starting ...\n");

        pingPID = Exec("./test/ping");
        pongPID = Exec("./test/pong");
        Join(pingPID);
        Join(pongPID);
        Exit(0);

}
```

Ta khai báo các thủ tục cần thiết cho việc biên dịch các chương trình này trong **test/Makefile**.

III. Hình ảnh Demo

Ta kiểm tra kết quả qua việc gọi command **./userprog/nachos -rs 1023 -x ./test/scheduler**. sau khi đã biên dịch thành công bằng lệnh **make**.



IV. Tài liệu tham khảo

1. Nguồn tham khảo từ tài liệu của thầy Long:

[1] Cac Lop Trong Project3.docx

[2] DoAn3_CQ_2021.docx

[3] Seminar_HDH_Project 3.pptx

2. Nguồn tham khảo trên Internet:

[4] https://github.com/thoaihuynh2509/HeDieuHanh/tree/master/DoAn3

[5] https://github.com/nguyenthanhchungfit/Nachos-Programing-HCMUS/blob/master/report/P2.pdf

[6] https://www.youtube.com/watch?v=t0jtY1C129s&list=PLRgTVtca98hUgCN2_2vzsAAXPiTFbvHpO&ab_channel=Th%C3%A0nhChungNguy%E1%BB%85n

[7] https://drive.google.com/drive/folders/1aQcLxNqebf2DrMBBMD6ArnAvYk0iqUto

[8] https://text.123docz.net/document/4020870-can-ban-va-rat-can-ban-ve-he-dieu-hanh-nachos.htm?fbclid=IwAR1sU1pdcYAbmKOz2geYk2XyRDt6TXewZLTEw9mv2bHgOWNIUq8EI6YoeA4