ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

UNIVERSITY OF SCIENCES

BÁO CÁO ĐỒ ÁN 3 - NACHOS

Môn học: Hệ điều hành

Học kỳ I (2021 – 2022)

Sinh viên: 19120338 - Trần Hoàng Quân

19120383 - Huỳnh Tấn Thọ

19120407 - Lâm Hải Triều

19120426 - Phan Đặng Diễm Uyên

19120469 - Sử Nhật Đăng

Trường: Đại học Khoa học Tự Nhiên, ĐHQG - HCM

Giáo viên: Thầy Lê Viết Long

Thành phố Hồ Chí Minh, tháng 12 năm 2021

MỤC LỤC

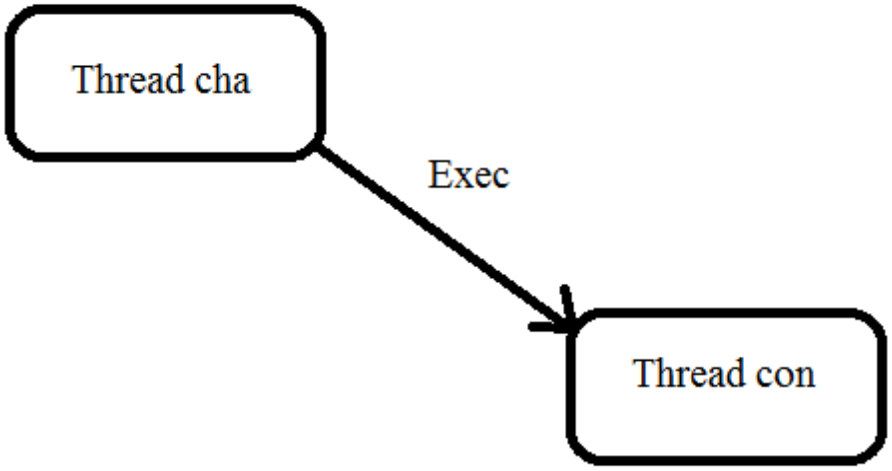
Thông tin nhóm và công việc phân công

STT	MSSV	Họ và tên	Công việc	Tỉ lệ hoàn thành
01	19120338	Trần Hoàng Quân	- Viết system call Exec	100%
02	19120383	Huỳnh Tấn Thọ	- Viết system call Join	100%
03	19120407	Lâm Hải Triều	- Viết báo cáo	100%
04	19120426	Phan Đặng Diễm Uyên	- Viết system call Exit	100%
05	19120469	Sử Nhật Đăng	- Sửa bug từ source code được cung cấp	100%

Mô hình quản lý tiến trình

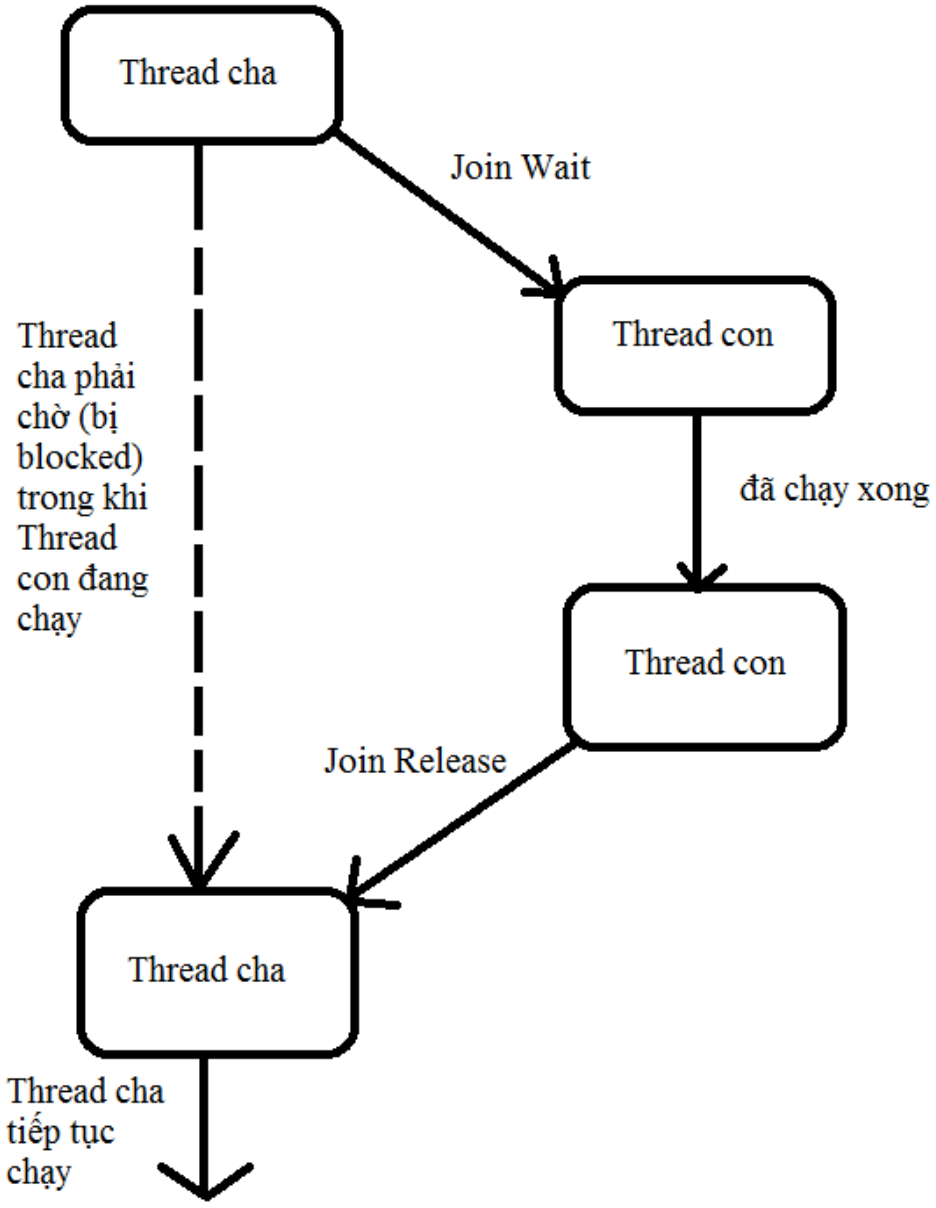
1. Exec

Trên Thread cha sẽ có 1 đoạn code để thực hiện việc khởi chạy tiến trình cho những Thread con. Với mỗi Thread con được khởi chạy, sẽ có 1 biến đếm ở lớp Thread của Thread cha tăng lên 1



2. Join

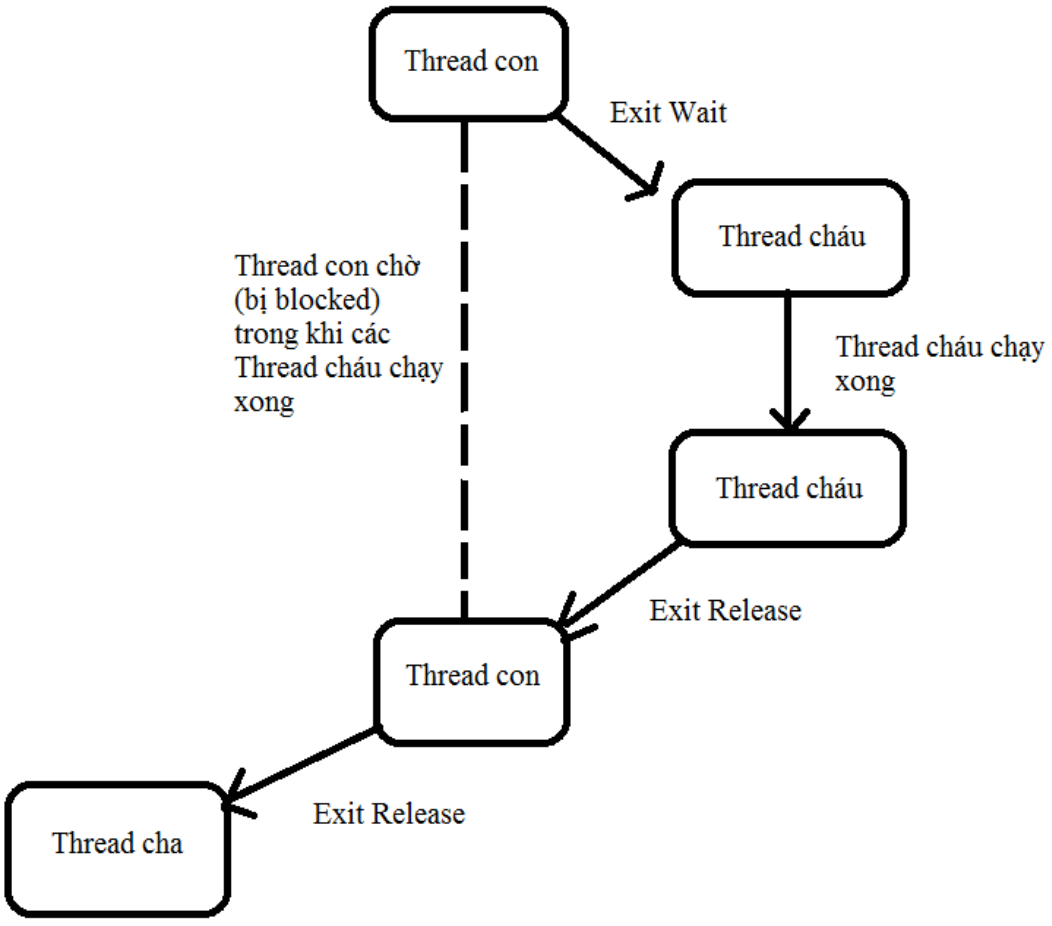
Khi một Thread cha Exec một Thread con, đôi khi ta lại phải cần Thread cha chờ đợi Thread con chạy xong hết, rồi Thread cha mới tiếp tục chạy. Lúc này, Thread cha sẽ chờ bằng cách Join Wait (bên trong thực chất chỉ là Semaphore Up()), và khi Thread con chạy xong, Thread con sẽ thực hiện Join Release (bên trong thực chất là Semaphore Down()). Sau khi Thread con chạy xong, Thread cha sẽ ngưng chờ và tiếp tục chạy



3. Exit

Khi một Thread đã hoàn thành công việc của mình và muốn kết thúc, nó cần phải chờ tất cả các Thread con của nó hoàn thành hết, thì nó mới được Exit. Việc chờ các thread con này được phối hợp dựa trên một Semaphore, và sử dụng ExitWait() để gọi. Mỗi khi 1 Thread hoàn thành xong công việc, và tất cả các thread con của nó cũng đã hoàn thành xong, thì Thread này sẽ cho phép Thread cha của nó (nếu có) tiếp tục thực hiện công viêrc của mình bằng cách ExitRelease()

Giả sử, ta không sử dụng Exit mà chỉ sử dụng mỗi Join, thì Nachos sẽ ngay lập tức kết thúc chương trình ngay khi Thread con đầu tiên hoàn thành công việc.



Các lớp cần tìm hiểu

1. PTable

Lớp PTable được sinh ra để quản lý toàn bộ tiến trình trong trong Nachos. Trong cài đặt của nhóm, chỉ có 1 thể hiện duy nhất của lớp PTable được sử dụng toàn cục trong toàn bộ Nachos, đấy chính là *PTable* pTab* được khai báo và khởi tạo toàn cục ở file *system.h* và *system.cc*.

Lớp PTable được khai báo và cài đặt lần lượt ở *PTable.h* và *PTable.cc*

Dưới đây là thông tin của lớp PTable

Tên	**Loại**	**Kiểu dữ liệu**	**Nội dung**	**Ghi chú**
MAXPROCESS	Hằng số		Số lượng process tối đa mà nhóm cho phép Nachos được chạy khi chạy đa chương	Được cung cấp bởi giáo viên
bm	Thuộc tính	BitMap*	Quản lý các đối tượng PCB đã/đang/sẽ/chưa được dùng	Được cung cấp bởi giáo viên
pcb	Thuộc tính	PCB*	Mảng lưu trữ các PCB được quản lý gián tiếp bởi bm Số lượng phần tử tối đa của mảng:	Được cung cấp bởi giáo viên

			MAXPROCESS	
psize	Thuộc tính	int	Kích thước khởi tạo của BitMap = Số lượng phần tử sử dụng của pcb <= MAXPROCESS	Được cung cấp bởi giáo viên
bmsem	Thuộc tính	Semaphore*	Semaphore dùng để xử lí độc quyền truy xuất trên các phương thức của PTable	Được cung cấp bởi giáo viên
PTable(int size)	Constructor	Ptable	Phương thức khởi tạo cho PTable, với tham số size chính là psize cần nạp vào để khởi tạo	Được cung cấp bởi giáo viên
~PTable()	Destructor		Phương thức huỷ cho PTable	Được cung cấp bởi giáo viên

ExecUpdate(char* filename)	Phương thức	int	Input: tên của file chương trình cần chạy Output: process id của chương trình đấy khi chạy đa chương Chức năng: Khởi chạy 1 tiến trình có tên là filename trong nachos	Được cung cấp bởi giáo viên Được cung cấp bởi giáo viên
ExitUpdate(inec)	Phương thức	int	Input: exit code của tiến trình đang chạy Output: exit code cuối cùng của tiến trình đấy Chức năng: Cho phép kết thúc quá trình Join của 1 tiến trình khi nó đã chạy xong	Được cung cấp bởi giáo viên
JoinUpdate(int pID)	Phương thức	int	Input: process ID của một tiến	Được cung cấp bởi giáo viên

			trình cần thực hiện Join Output: exit code khi chạy tiến trình đấy Chức năng: Cho phép thực hiện quá trình Join một tiến trình	
GetFreeSlot()	Phương thức	int	Input: không có Output: trả về một vị trí còn trống của bm Chức năng: tìm 1 vị trí còn trống trên bm để thực hiện chạy đa chương cho 1 tiến trình mới	Được cung cấp bởi giáo viên
IsExist(int pID)	Phương thức	bool	Input: process id của một tiến trình Output: Yes nếu tiến trình đó có tồn tại (tức là đang chạy, chưa	Được cung cấp bởi giáo viên

			exit), No nếu ngược lại Chức năng: Kiểm tra xem liệu 1 tiến trình có đang chạy và đã nạp vào trong PTable hay chưa	
Remove(int pID)	Phương thức	void	Input: process id của một tiến trình cần loại bỏ ra khỏi PTable Output: không có Chức năng: loại bỏ một tiến trình đã được nạp vào trong PTable	Được cung cấp bởi giáo viên
GetName(int pID)	Phương thức	char*	Input: process id của tiến trình cần lấy tên trong PTable Output: tên tương ứng của process id đấy	Được cung cấp bởi giáo viên

			Chức năng: lấy tên của một chương trình có process id tương ứng trong PTable	

2. PCB

Tương ứng với mỗi tiến trình mà ta muốn chạy đa chương khi được nạp vào PTable, sẽ có 1 PCB (Process control block) tương ứng để quản lý 1 tiến trình đấy. PCB trong đồ án lần này được thể hiện trong chính *PCB* bm[MAXPROCESS]* của chính *pTab* mà ta đã đề cập ở phần "1. PTable"

Lớp PCB được khai báo và cài đặt lần lượt ở *PCB.h* và *PCB.cc*

Dưới đây là thông tin của lớp PCB

Tên	**Loại**	**Kiểu dữ liệu**	**Nội dung**	**Ghi chú**
joinsem	Thuộc tính	Semaphore*	Semaphore để xử lí truy xuất độc quyền khi ta cần Join một tiến trình	Được cung cấp bởi giáo viên
exitsem	Thuộc tính	Semaphore*	Semaphore để xử lí truy xuất độc quyền khi ta cần exit một tiến trình sau khi nó chạy xong	Được cung cấp bởi giáo viên

mutex	Thuộc tính	Semaphore*	Semaphore để xử lí truy xuất độc quyền khi ta cần chạy (execute) 1 process mà PCB đang nắm quyền quản lý	Được cung cấp bởi giáo viên
exitCode	Thuộc tính	int	Exit code của một tiến trình trong toàn bộ vòng đời của nó, exit code có thể thay đổi liên tục	Được cung cấp bởi giáo viên
thread	Thuộc tính	Thread*	Thread sẽ tiến hành chạy chương trình do PCB quản lý	Được cung cấp bởi giáo viên
pid	Thuộc tính	int	Process id của tiến trình được PCB quản lý	Được cung cấp bởi giáo viên
numwait	Thuộc tính	int	Số tiến trình hiện tại đang được Join	Được cung cấp bởi giáo viên Được cung cấp bởi giáo viên Được cung cấp bởi giáo viên

parentID	Thuộc tính	int	Process id của tiến trình cha (tiến trình đã call PTable quản lý PCB này để khởi chạy)	Được cung cấp bởi giáo viên
JoinStatus	Thuộc tính	int	Trạng thái join của tiến trình hiện tại, nếu tiến trình hiện tại đang không Join, sẽ có giá trị -1, ngược lại, sẽ có giá trị bằng parentID	Được cung cấp bởi giáo viên
PCB(int id)	Constructor	PCB	Phương thức khởi tạo cho PCB, với id chính là process id của tiến trình được PCB quản lý	Được cung cấp bởi giáo viên
~PCB()	Destructor		Phương thức huỷ cho PCB	Được cung cấp bởi giáo viên

Exec(char* filename, int pID)	Phương thức	int	Input: tên của tiến trình cần chạy, process id của chính tiến trình đấy Output: process id của bản thân tiến trình đấy Chức năng: thực hiện việc chạy (execute) tiến trình được có filename và pID được nạp vào	Được cung cấp bởi giáo viên
GetID()	Phương thức	int	Input: không có Output: trả về process ID của tiến trình đang được PCB quản lý Chức năng: lấy process id của tiến trình đang được PCB quản lý	Được cung cấp bởi giáo viên
GetNumWait()	Phương thức	int	Input: không có	Được cung cấp bởi giáo viên

			Output: trả về số tiến trình hiện tại đang được Join (getter cho numwait)	
JoinWait()	Phương thức	void	Input: không có Output: không có Chức năng: cho tiến trình được PCB quản lý chờ, bằng cách thực hiện việc Join cho tiến trình hiện tại với tiến trình cha	Được cung cấp bởi giáo viên
ExitWait()	Phương thức	void	Input: không có Output: không có Chức năng: cho tiến trình được PCB quản lý chờ, bằng cách thực hiện việc Exit cho tiến	Được cung cấp bởi giáo viên

			trình hiện tại được PCB quản lý	
JoinRelease()	Phương thức	void	Input: không có Output: không có Chức năng: Giải phóng tiến trình hiện tại đang chờ nếu nó đang Join	Được cung cấp bởi giáo viên
ExitRelease()	Phương thức	void	Input: không có Output: không có Chức năng: Giải phóng tiến trình hiện tại đang chờ nếu nó đang Exit (tức là đã chạy xong)	Được cung cấp bởi giáo viên
IncNumWait()	Phương thức	void	Input: không có Output: không có Chức năng: Tăng số lượng tiến trình đang chờ	Được cung cấp bởi giáo viên

DecNumWait()	Phương thức	void	Input: không có Output: không có Chức năng: Giảm số lượng tiến trình đang chờ	Được cung cấp bởi giáo viên
SetExitCode(int ec)	Phương thức	void	Input: giá trị exit code muốn gán Output: không có Chức năng: Gán giá trị của exitCode bằng ec (setter của exitCode)	Được cung cấp bởi giáo viên
GetExitCode()	Phương thức	int	Input: không có Output: giá trị của exitCode hiện tại Chức năng: Trả về giá trị hiện tại của exitCode (getter của exitCode)	Được cung cấp bởi giáo viên

GetNameThread	Phương thức	char*	Input: không có Output: tên của tiến trình hiện tại mà PCB đang nắm giữ Chức năng: trả ra tên của tiến trình hiện tại đang được PCB quản lý. Tên tiến trình được lưu trự trong thread	Được cung cấp bởi giáo viên

3. BitMap

Để có thể lưu vết của các tiến trình hiện hành, ta sẽ sử dụng BitMap để quản lý việc cấp phát/huỷ cấp phát và một số hành động khác như tìm kiếm, thêm tiến trình mới vào,… cho đa chương trong Nachos. BitMap trong đồ án lần này có 2 thể hiện:

- Thuộc tính *BitMap* bm* của *PTable pTab* (đã đề cập ở PTable)
- Biến cục bộ *BitMap* physicalPage* được khai báo ở file *system.h* và *system.cc*. Biến này dùng để tương tác với các AddrSpace trong class Thread, dùng để quản lý trang vật lý trong Nachos

Lớp BitMap được khai báo và cài đặt lần lượt ở *BitMap.h* và *BitMap.cc*

Dưới đây là thông tin của lớp BitMap:

Tên	Loại	Kiểu dữ liệu	Nội dung	Ghi chú
BitsInByte	Hằng số		Số bit trong một byte của Nachos	Được cung cấp bởi giáo viên
BitsInWord	Hằng số		Số bit trong một word của Nachos	Được cung cấp bởi giáo viên
BitMap(int nitems)	Constructor	BitMap	Phương thức khởi tạo cho BitMap, với nitems chính là số bit mà BitMap phải quản lý	Được cung cấp bởi giáo viên
~Bitmap()	Destructor		Phương thức huỷ của BitMap	Được cung cấp bởi giáo viên
numBits	Thuộc tính	int	Số lượng bit mà BitMap phải quản lý	Được cung cấp bởi giáo viên
numWords	Thuộc tính	int	Số lượng word đổi ra từ numBits Công thức: NumWords = ceil(numBits/BitsInWord)	Được cung cấp bởi giáo viên

map	Thuộc tính	unsigned int*	Để quản lý các bit hiệu quả nhất, thì BitMap sẽ quản lý các Word đổi ra từ các Bit. Map chính là danh sách các Word mà BitMap quản lý	Được cung cấp bởi giáo viên
Mark(int which)	Phương thức	void	Input: vị trí bit ta muốn đánh dấu Output: không có Chức năng: đánh dấu 1 bit ở vị trí which, có ý nghĩa rằng bit này đã bị sử dụng	Được cung cấp bởi giáo viên
Clear(int which)	Phương thức	void	Input: vị trí bit ta muốn bỏ đánh dấu Output: không có Chức năng: bỏ đánh dấu 1 bit	Được cung cấp bởi giáo viên

			ở vị trí which, có ý nghĩa ngược lại với Mark(int which)	
Test(int which)	Phương thức	bool	Input: vị trí mà bit mà ta muốn kiểm tra Output: - True nếubit ở vị trí này đã bị đánh dấu - False nếu ngược lại Chức năng: Kiểm tra 1 bit ở vị trí bất kì, để biết rằng nó có bị đánh dấu hay chưa	Được cung cấp bởi giáo viên
Find()	Phương thức	int	Input: không có	Được cung cấp bởi giáo viên

			Output: vị trí của 1 bit chưa bị đánh dấu Chức năng: tìm kiếm 1 vị trí bit trống, chưa bị đánh dấu của BitMap	
NumClear()	Phương thức	int	Input: không có Output: số lượng bit chưa bị đánh dấu trong BitMap Chức năng: đếm số lượng các bit còn trống (chưa bị đánh dấu) trong BitMap	Được cung cấp bởi giáo viên
Print()	Phương thức	void	Input: không có Output: không có Chức năng: in ra màn hình các vị trí bit đã đánh dấu	Được cung cấp bởi giáo viên

FetchFrom(OpenFile* file)	Phương thức	void	Input: file muốn đọc dữ liệu từ BitMap Output: không có Chức năng: nạp dữ liệu từ BitMap vào 1 file	Được cung cấp bởi giáo viên
WriteBack(OpenFile* file)	Phương thức	void	Input: file muốn ghi dữ liệu vào BitMap Output: không có Chức năng: ghi dữ liệu từ file vào BitMap	Được cung cấp bởi giáo viên

4. Thread

Mỗi tiến trình được gọi trong nachos, khi chạy đều được quản lý bởi 1 thread. Các thread có chức năng quản lý việc khởi chạy, kết thúc các tiến trình. Đồng thời, các thread cũng có khả năng quản lý thứ tự làm việc của các tiến trình (bắt 1 tiến trình A chờ 1 tiến trình B,..). Thread trong đồ án này có 4 thể hiện

- Thuộc tính *Thread* thread* của thuộc tính *BitMap* bm* nằm trong *PTable pTab* (đã đề cập ở PTable)
- Thuộc tính *Thread* thread* của biến cục bộ *BitMap* physicalPage* được khai báo ở file *system.h* và *system.cc*.

- Biến cục bộ *Thread* currentThread* được khai báo ở file *system.h* và *system.cc.*
- Biến cục bộ *Thread* threadToBeDestroyed* được khai báo ở file *system.h* và *system.cc.*

Lớp Thread được khai báo và cài đặt lần lượt ở *thread.h* và *thread.cc*

Dưới đây là thông tin của lớp Thread

Tên	**Loại**	**Kiểu dữ liệu**	**Nội dung**	**Ghi chú**
MachineStateSize	Hằng số		Kích thước của mảng để lưu trữ trạng thái thanh ghi CPU	Được cung cấp bởi giáo viên
StackSize	Hằng số		Kích thước của stack của mỗi thread được tạo ra (stack dùng để execute tiến trình mà thread quản lý)	Được cung cấp bởi giáo viên
ThreadStatus	Enum	ThreadStatus	Các trạng thái của 1 thread. Có 4 trạng thái: JUST_CREATED, RUNNING, READY, BLOCKED	Được cung cấp bởi giáo viên
stackTop	Thuộc tính	int*	Stack pointer của Thread hiện tại	Được cung cấp bởi giáo viên

machineState[MachineStateSize]	Thuộc tính	int	Mảng lưu trữ các trạng thái thanh ghi của CPU cho mỗi thread	Được cung cấp bởi giáo viên
Thread(char* threadName)	Constructor	Thread	Phương thức khởi tạo cho Thread, trong đó threadName chính là tên của Thread mà người dùng muốn đặt.	Được cung cấp bởi giáo viên
~Thread()	Destructor		Phương thức huỷ của Thread	Được cung cấp bởi giáo viên
processID	Thuộc tính	int	ID của tiến trình mà thread đang quản lý	Được cung cấp bởi giáo viên
Fork(VoidFunctionPtr func, int arg)	Phương thức	void	Input: hàm mà người dùng muốn khởi chạy và các tham số của hàm đó Output: không có	Được cung cấp bởi giáo viên

			Chức năng: Khởi tạo các thông tin cần thiết cho mộ tiểu trình, sau đó khởi chạy nó.	
Yield()	Phương thức	void	Input: không có Output: không có Chức năng: Nhường CPU cho 1 thread muốn được chạy	Được cung cấp bởi giáo viên
Sleep()	Phương thức	void	Input: không có Output: không có Chức năng: Nhường CPU để đưa thread hiện tại vào trạng thái ngủ vì thread này hiện đang bị blocked.	Được cung cấp bởi giáo viên

Finish()	Phương thức	void	Input: không có Output: không có Chức năng: kết thúc tiến trình của thread hiện hành.	Được cung cấp bởi giáo viên
CheckOverflow()	Phương thức	void	Input: không có Output: không có Chức năng: Kiểm tra xem stack của thread có bị tràn hay không	Được cung cấp bởi giáo viên
setStatus(ThreadStatus st)	Phương thức	void	Input: không có Output: không có Chức năng: setter cho thuộc tính status	Được cung cấp bởi giáo viên
getName()	Phương thức	char*	Input: không có Output: không có	Được cung cấp bởi giáo viên

			Chức năng: getter cho thuộc tính name	
Print()	Phương thức	void	Input: không có Output: không có Chức năng: in ra màn hình tên của thread	Được cung cấp bởi giáo viên
stack	Thuộc tính	int*	Stack để khởi chạy tiến trình mà thread đang quản lý	Được cung cấp bởi giáo viên
status	Thuộc tính	ThreadStatus	Trạng thái hiện tại của thread	Được cung cấp bởi giáo viên
name	Thuộc tính	char*	Tên của thread	Được cung cấp bởi giáo viên
userRegisters[NumTotalRegs]	Phương thức	int	Các thanh ghi dành cho thread khi phải thực thi user code	Được cung cấp bởi giáo viên

ThreadFinish()	Phương thức tĩnh	void	Input: không có Output: không có Chức năng: dọn rác cho biến toàn cục currentThread	**Do nhóm thêm vào**
SaveUserState()	Phương thức	void	Input: không có Output: không có Chức năng: lưu lại trạng thái các thanh ghi của machine vào userRegisters	Được cung cấp bởi giáo viên
RestoreUserState()	Phương thức	void	Input: không có Output: không có Chức năng: ghi lại trạng thái các thanh ghi của machine bằng với các thanh ghi userRegisters	Được cung cấp bởi giáo viên

space	Thuộc tính	AddrSpace*	Vùng nhớ user cho thread	Được cung cấp bởi giáo viên

Các system call mà nhóm cần cài đặt thêm vào

4. Exec

Exec là system call cho phép Nachos có thể gọi và thực thi bất kì 1 chương trình nào.

Dưới đây là các bước để cài đặt Exec

Đầu tiên, ta cần setup để cho nachos có thể hiểu được cách gọi bằng system call này.

Bước này bọn em sẽ không đề cập đến vì đã miêu tả quá rõ ở đồ án số 2.

Tiếp theo, bọn em sẽ tiến hành cài đặt system call này, đây là code:

```
int doSC_Exec() {

	//Get the address of process name
	int virtAddr = machine->ReadRegister(R4);
	char* processName = NULL;

	//Get the process name
	processName = machine->User2System(virtAddr, MAX_BUFFER_LENGTH + 1);

	//invalid process
	if (NULL == processName) {
		printf("\n Error opening process");
		machine->WriteRegister(R2, -1);
		return -1;
	}

	//ExecUpdate(processName) return process id,
	int pid = pTab->ExecUpdate(processName);

	//return this process id as result
	machine->WriteRegister(R2, pid);

	//increase program counter
	machine->IncreasePC();

	return pid;
}
```

Exec nhận vào tham số là tên của tiến trình muốn khởi chạy => Ta lấy tên tiến trình bằng cách đọc địa chỉ của mảng tên tại thanh ghi R4, sau đó đọc vùng nhớ một lượng là MAX_BUFFER_LENGTH + 1 (255) và chuyển vùng nhớ này từ User sang System. Lúc này, processName sẽ chứa tên của tiến trình cần khởi chạy

Nếu không đọc được tên của tiến trình => Exec lập tức báo lỗi, trả về -1 (bằng cách ghi vào thanh ghi R2 giá trị -1), và kết thúc systemcall

Ngược lại, ta nạp tên của tiến trình cần chạy (processID) vào phương thức ExecUpdate(char*) (phương thức này đã được giải thích ở bảng trên, thuộc lớp PTable) của pTab (đây là biến toàn cục PTable* pTab được định nghĩa ở system.h). ExecUpdate sẽ tiến hành khởi chạy cho tiến trình có tên được nạp vào. Kết thúc hàm, ExecUpdate sẽ trả về giá trị là process ID của tiến trình này. Process ID này sẽ được lưu vào biến pid

Biến pid chính là giá trị trả về của system call Exec. Ta trả về kết quả là pid bằng cách ghi vào thanh ghi R2 giá trị pid.

Trước khi kết thúc hoàn toàn system call, ta tăng biến đếm program counter của nachos để tiếp tục thực hiện lệnh kế tiếp. Sau đó kết thúc phần xử lý của Exec.

5. Join

Join là system call cho phép 1 tiến trình phải chờ 1 tiến trình khác chạy xong, thì mới có thể chạy tiếp. Join được sử dụng phối hợp với một system call khác gọi là system call Exit (sẽ được đề cập ở dưới)

Đầu tiên, ta cần setup để cho nachos có thể hiểu được cách gọi bằng system call này. Bước này bọn em sẽ không đề cập đến vì đã miêu tả quá rõ ở đồ án số 2.

Tiếp theo, bọn em sẽ tiến hành cài đặt system call này, đây là code:

```
int doSC_Join() {

        //get process id
        int pid = machine->ReadRegister(R4);

        //JoinUpdate return exit code, if there is no error, exit code = 0;
        int exitCode = pTab->JoinUpdate(pid);

        //return exit code
        machine->WriteRegister(R2, exitCode);

        //increase program counter
        machine->IncreasePC();

        return exitCode;
}
```

System call Join nhận tham số đầu vào chính là process id của tiến trình con, mà tiến trình cha phải chờ đợi tiến trình con này. Để lấy được process id này, ta chỉ cần đơn giản đọc giá trị của thanh ghi R4, và gán giá trị nào vào biến pid.

Tiếp theo, ta tiến hành Join bằng cách truyền biến pid này vào phương thực JoinUpdate(int) (phương thức này đã được giải thích ở bảng trên, thuộc lớp PTable) của pTab (đây là biến toàn cục PTable* pTab được định nghĩa ở system.h). JoinUpdate(int) trả về exit code sau khi xử lí xong. Ta gán exit code này vào biến exitCode.

Biến exitCode chính là giá trị trả về của system call Join. Ta trả về kết quả của exitCode bằng cách ghi giá trị của biến này vào thanh ghi R2

Trước khi kết thúc hoàn toàn system call, ta tăng biến đếm program counter của nachos để tiếp tục thực hiện lệnh kế tiếp. Sau đó kết thúc phần xử lý của Join.

6. Exit

Khi một tiến trình cha join với 1 tiến trình con, khi tiến trình con chạy xong, nếu không có system call Exit, Nachos sẽ không nhường CPU lại cho tiến trình cha mà thực hiện việc dừng Nachos (do hiểu nhầm rằng chạy kết thúc tiến trình con <=> hoàn thành hết công việc). Để giải quyết điều này, ta cần phải cài đặt System call Exit.

Exit là system call cho phép tiến trình con có thể kết thúc và nhường lại CPU cho tiến trình cha tiếp tục chạy.

Đầu tiên, ta cần setup để cho nachos có thể hiểu được cách gọi bằng system call này. Bước này bọn em sẽ không đề cập đến vì đã miêu tả quá rõ ở đồ án số 2.

Tiếp theo, bọn em sẽ tiến hành cài đặt system call này, đây là code:

```
void doSC_Exit() {

        //Get exit code from join process
        int joinExitCode = machine->ReadRegister(R4);

        //if there are any erros => stop process
        if (0 != joinExitCode) {
                machine->IncreasePC();
                return;
        }

        //Exit process for the current threads
        int exitCode = pTab->ExitUpdate(joinExitCode);

        //Release the current thread
        Thread::ThreadFinish();

        //increase program counter
        machine->IncreasePC();
}
```

Exit nhận tham số đầu vào chính là exit code của tiến trình được trả về bởi system call Join. Ta tiến hành đọc giá trị exit code này bằng cách đọc dữ liệu ở thanh ghi R4, và gán giá trị vào biến joinExitCode.

Khi mà việc Join không có lỗi xảy ra, system call Join luôn trả về exit code là 0. Giả sử giá trị joinExitCode ta nhận được là khác 0, tức là có lỗi đã xảy ra. Lúc này, ta tăng biến đếm program counter và kết thúc chương trình.

Nếu không có lỗi xảy ra, ta truyền giá trị joinExitCode này vào phương thức ExitUpdate(int) (phương thức này đã được giải thích ở bảng trên, thuộc lớp PTable) của pTab (đây là biến toàn cục PTable* pTab được định nghĩa ở system.h).

Tiếp theo, ta tiến hành dọn rác cho biến toàn cục currentThread bằng phương thức tĩnh ThreadFinish() do nhóm đã cài đặt.

Vì system call Exit không có giá trị trả về, ta chỉ đơn giản là kết thúc bằng cách tăng biến đếm program counter lên và kết thúc.

Các chỉnh sửa mà nhóm đã thực hiện ở source code

1. Khai báo các biến toàn cục mới ở system.h/system.cc

Ta có tổng cộng thêm 3 biến toàn cục mới so với đồ án 2:

- *Semaphore* addrLock*: semaphore cho việc truy cập vào thao tác với các lớp AddrSpace.
- *PTable* pTab*: ptable cho việc quản lý toàn bộ các tiến trình đa chương trong đồ án 3
- *BitMap* physicalPage*: cho việc quản lý các frame bộ nhớ trên nachos (phối hợp với lớp AddrSpace)

Ta tiến hành khai báo các biến toàn cục mới này trong Region #USER_PROGRAM nằm trong system.h/system.cc

Đây là vị trí khai báo ở system.h

```
#ifdef USER_PROGRAM
#include "machine.h"
#include "synchcons.h"
#include "ptable.h"
#include "../userprog/bitmap.h"

extern Semaphore* addrLock;
extern Machine* machine;        // user program memory and registers
extern SynchConsole* synchConsole;
extern PTable* pTab;
extern BitMap* physicalPage;
#endif
```

Đây là vị trí khai báo ở system.cc

```
#ifdef USER_PROGRAM     // requires either FILESYS or FILESYS_STUB
Machine *machine;       // user program memory and registers
SynchConsole *synchConsole;
PTable* pTab;
BitMap* physicalPage;
Semaphore* addrLock;
#endif
```

Và

```
#ifdef USER_PROGRAM
    machine = new Machine(debugUserProg);       // this must come first
        synchConsole = new SynchConsole();
        pTab = new PTable(10);
        physicalPage = new BitMap(256);
        addrLock = new Semaphore("addrLock", 1);
#endif
```

Giải thích:

- Ta khởi tạo pTab với khả năng quản lý tối đa 10 tiến trình
- Ta khởi tạo physicalPage với kích thước bitmap quản lý là 256 bit
- Ta khởi tạo addrLock là semaphore với ten là addrLock, giá trị khởi tạo ban đầu là 1

2. PTable

a. Phương thức GetFreeSlot()

Do lớp BitMap đã có sẵn phương thức Find() cho phép tìm kiếm và đánh dấu những vị trí bit còn trống, ta dễ dàng tận dụng lại phương thức này như sau:

```
int PTable::GetFreeSlot()
{
        return bm->Find();
}
```

b. Phương thức ExecUpdate(char* filename)

Do sau khi dùng phương thức Find() của lớp BitMap, việc đánh dấu Mark() của BitMap đã thừa thải (vì bản thân Find() đã thực hiện cả việc Mark() luôn rồi, nên ta có thể bỏ đi câu lênh Mark() sẵn có này bên phương thức ExecUpdate()

```
//bm->Mark(ID);
```

Mặt khác, theo yêu cầu của đề bài, ta tiến hành thêm thông tin process id thuộc process cha của tiến trình hiện hành bằng cách thêm dòng lệnh:

```
pcb[ID]->parentID = currentThread->processID;
```

Như vậy, ta chỉ thay duy nhất đoạn cuối cùng của phương thức ExecUpdate(char* filename), và nó sẽ trở thành:

```
        pcb[ID]= new PCB(ID);
        pcb[ID]->parentID = currentThread->processID;
        //bm->Mark(ID);
        int pID= pcb[ID]->Exec(filename,ID);

        bmsem->V();
        return pID;
}
```

c. Phương thức ExitUpdate(int ec)

Chỉ đơn giản, ở đoạn cuối của phương thức này, ta **luôn luôn** exit join, vì phương thức này chỉ được call từ system call Exit, mà để Exit chạy, thì phải có 1 tiến trình nào đấy đã được Join => việc kiểm tra tiến trình cha như code cũ là vô nghĩa, vì miễn vào được đoạn code này thì hiển nhiên là bắt buộc phải exit (vì tiến trình đang join), mà nếu không exit, nachos sẽ ngừng lại ngay lập tức, trong khi tiến trình cha còn chưa chạy xong

Ta thay đổi đoạn code cuối như sau:

```
        pcb[pID]->SetExitCode(ec);

        //Exit join
        pcb[pID]->JoinRelease();
        pcb[pID]->ExitWait();
        Remove(pID);

        return ec;
}
```

3. AddrSpace

a. Constructor

Chuyển từ *AddrSpace::AddrSpace(OpenFile* executable)* thành *AddrSpace::AddrSpace(char* filename)*. Vì có sự bất đồng bộ ở cách gọi constructor trong đoạn code có sẵn được giáo viên gửi. Nên nhóm quyết định chọn constructor là *AddrSpace::AddrSpace(char* filename)*

Mặt khác, ta cần phải thiết lập Semaphore* addrLock để có thể truy xuất độc quyền trên constructor này, do đó cần phải thêm đoạn code addrLock->P() và addrLock->V() ở cuối constructor

Dưới đây là cách tiền xử lí cho constructor mới giống constructor cũ (tức là sau khi tiền xử lí xong, logic code ở phía sai của cả 2 constructor sẽ **y hệt** nhau)

```
AddrSpace::AddrSpace(char* filename) {

        addrLock->P();
        //printf("%s\n", filename);
        OpenFile* executable = fileSystem->Open(filename);
        if (NULL == executable){
                printf("Unable to open file %s\n", filename);
                //printf("test\n");
                return;
        }
```

Tiếp theo, ta cần kiểm tra xem liệu numPages có lớn hơn số trang còn trống hay không (theo yêu cầu đề bài). Để có được số trang còn trống, ta chỉ đơn giản dùng phương thức NumClear() (được đề cập ở bảng trên, lớp BitMap) của BitMap* physicalPage đã được khai báo toàn cục ở system.h/system.cc. Như vậy, đoạn code kiểm tra này sẽ có dạng:

```
if (numPages > physicalPage->NumClear()) {
        printf("\nAddrSpace::Load : not enough memory for new process");
        numPages = 0;
        delete executable;
        addrLock->V();
        return;
}
```

Mặt khác, ở đoạn code khởi tạo không gian địa chỉ cho tiến trình, theo yêu cầu đề bài, ta cần phải thay pageTable[i].physicalPage = i bằng hàm tìm 1 trang trống và đánh dấu đã sử dụng. Cực kì đơn giản, hàm tìm trang trống này chính là phương thức Find() của lớp BitMap, cụ thể là Find() của physicalPage. Đoạn code sẽ trở thành:

```
    pageTable = new TranslationEntry[numPages];
    for (i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;   // for now, virtual page # = phys page #
        pageTable[i].physicalPage = physicalPage->Find();
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;  // if the code segment was entirely on
                                                        // a separate page, we could set its
                                                        // pages to be read-only
    }
```

Cuối cùng, ta thay đổi cách nachos chuyển data segment về memory như sau:

```
if (noffH.code.size > 0) {
        DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
                noffH.code.virtualAddr, noffH.code.size);

        for (int i = 0; i < numPages; ++i) {
                executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]) + (pageTable[i].physicalPage*PageSize),
                        PageSize,
                        noffH.code.inFileAddr + (i*PageSize));
        }
}
if (noffH.initData.size > 0) {
        DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
                noffH.initData.virtualAddr, noffH.initData.size);


        for (int i = 0; i < numPages; ++i) {
                executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]) + (pageTable[i].physicalPage*PageSize),
                        PageSize,
                        noffH.initData.inFileAddr + (i*PageSize));
        }
}
```

b. Destructor

Do ta đã thêm vào sự xuất hiện của physicalPage khi phối hợp làm việc với AddrSpace => ta cần phải thu hồi bộ nhớ của cả physicalPage

Đoạn code cho destructor sẽ như sau:

```
AddrSpace::~AddrSpace()
{
        for (int i = 0; i < numPages; i++)
        {
                physicalPage->Clear(pageTable[i].physicalPage);
        }
        delete pageTable;
}
```

4. PCB

a. Phương thức Exec(char *filename, int pID)

Ta cần thêm vào đoạn code để gán được giá trị parentID cho PCB hiện tại. Đoạn code này được chúng em thêm vào cuối phương thức Exec(char *filename, int pID)

```
        thread->processID= pID;
        parentID = currentThread->processID;
        thread->Fork(MyStartProcess,pID);
        mutex->V();
        return pID;
}
```

b. Phương thức MyStartProcess(int pID)

Do ta đã thay đổi contructor của AddrSpace, nên ta phải thay đổi cách ta construct AddrSpace trong phương thức này, cụ thể:

```
void MyStartProcess(int pID)
{
        char *filename= pTab->GetName(pID);
        AddrSpace *space;
        space = new AddrSpace(filename);
        if(space == NULL)
        {
```

5. Thread

a. Viết thêm phương thức tĩnh ThreadFinish() để dọn rác cho currentThread

Cụ thể, ta huỷ đi space của currentThread và ép currentThread dừng

```
void Thread::ThreadFinish()    {
        if (currentThread->space) {
                delete currentThread->space;
                //currentThread->space = NULL;
        }
        currentThread->Finish();
}
```

6. Machine

a. Thay đổi giá trị hằng số NumPhysPages

Ta thay đổi giá trị cũ thành 128

```
#define NumPhysPages    128
```

7. Disk

a. Thay đổi giá trị hằng số SectorSize

Ta thay đổi giá trị cũ thành 512

```
#define SectorSize              512     // number of bytes per disk sector
```

Chạy thử chương trình Scheduler

1. Chương trình scheduler

Đầu tiên, nhóm xin phép được viết lại chương trình scheduler như sau (không còn giống trong slide)

```
#include "syscall.h"

int main() {
        int pingPID, pongPID;
        int pingExitCode, pongExitCode;
        PrintString("Ping-Pong test starting...\n\n");
        pingPID = Exec("./test/ping");
        pongPID = Exec("./test/pong");

        pingExitCode = Join(pingPID);
        pongExitCode = Join(pongPID);

        //Exit(pingExitCode);
        //Exit(pongExitCode);

        PrintString("\nFinish Ping-Pong test\n");
}
```

Nhóm thêm vào việc call system call Join để chương trình scheduler có thể chờ cả 2 chương trình Ping và Pong chạy xong hết, và scheduler sẽ tiếp tục chạy

Giả sử ta không dùng Join, thì nếu như một trong 2 chương trình Ping hoặc Pong chạy xong trước, ngay lập tức, Nachos sẽ ngừng, và scheduler vẫn sẽ chưa chạy xong hết

Ta không cần phải call system call Exit, vì ngay sau khi Join xong, Nachos sẽ tự động call system call Exit cho ta

Cực kì dễ nhận ra, giả sử chương trình Scheduler của chúng ta chạy thành công (tức là cả tiểu trình Ping và Pong đều chạy xong), thì **chắc chắn,** dòng chữ "Finish Ping-Pong test" sẽ được xuất hiện ở cuối console trước khi Nachos ngừng

2. Chương trình Ping/Pong

Không có gì thay đổi so với yêu cầu đề bài

```c
#include "syscall.h"

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

}
```

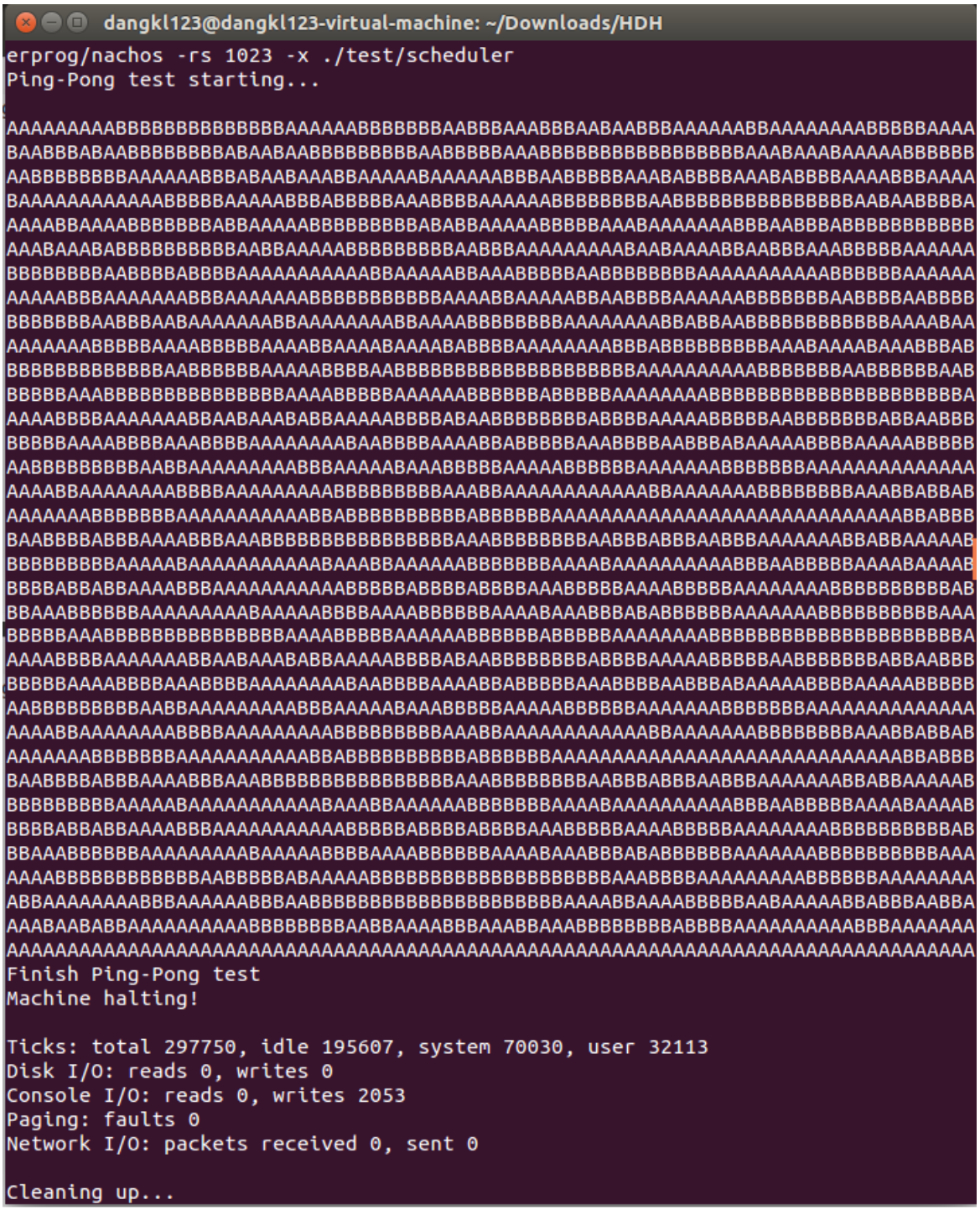
```c
#include "syscall.h"

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

}
```

3. Chạy thử

Trên màn hình console, ta browse đến folder nachos/nachos-3.4/code/test (folder chứa scheduler), sau đấy nhấn lệnh sau:

./userprog/nachos -rs 1023 –x ./test/scheduler

Ta sẽ ra được kết quả như dưới đây



Kết luận: Do Console xuất ra các chữ A và B xen kẽ lẫn nhau, cuối cùng, trước khi kết thúc chương trình, console lại tiếp tục xuất ra dòng chữ "Finish Ping-Pong test" rồi mới tiến hành halt nachos lại. Do đó, chúng ta đã cài đặt thành công đa chương cho Nachos

Tổng kết

1. Những điều đã làm được

STT	Nội dung
1	Viết thành công system call Exec
2	Viết thành công system call Join
3	Viết thành công system call Exit
4	Chạy thử thành công chương trình Scheduler

2. Những điều chưa làm được

Không có

Hướng dẫn sử dụng chương trình

- B1: Compile Nachos bằng cách browser đến thư mục ./nachos/nachos-3.4/code/, sau đó chạy lệnh:

  make

- B2: Vẫn ở vị trí thư mục ./nachos/nachos-3.4/code/, ta chạy lệnh:

  ./userprog/nachos -rs 1023 –x ./test/scheduler

- B3: Tận hưởng kết quả của chương trình scheduler trên console

Tài liệu tham khảo

[1] File được giảng viên cung cấp: Cac Lop Trong Project 3.docx

[2] File được giảng viên cung cấp: DoAn3_CQ_2021.docx

[3] File được giảng viên cung cấp: Cach Viet Mot SystemCall.pdf

[4] File được giảng viên cung cấp: Seminar_HDH_Project 3.pptx

[5] https://users.cs.duke.edu/~chase/cps110-archive/nachos-guide/nachos-labs-18.html

[6] https://mystudyhcmus.files.wordpress.com/2017/09/nachos_canban.doc

[7] https://users.cs.duke.edu/~chase/cps110-archive/nachos-guide/nachos-labs-14.html#21590

[8] https://users.cs.duke.edu/~chase/cps110-archive/nachos-guide/