HỆ ĐIỀU HÀNH

BÁO CÁO
NACHOS

Bộ môn Hệ điều hành
Khoa Công nghệ thông tin
Đại học Khoa học tự nhiên TP HCM

MỤC LỤC

1 Thông tin nhóm 2

2 Cài đặt System Call 3

- Cách xây dựng một System Call 3
- Xây dựng các System Call còn lại 5
- Demo 8

3 Cài đặt đa chương 9

- Mô tả yêu cầu 9
- Phương án giải quyết bài toán 9
- Demo 15

4 Cài đặt đồng bộ hóa 16

- Mô tả yêu cầu 16
- Phương án giải quyết bài toán 16
- Demo 20

5 Tài liệu tham khảo 21

1 Thông tin nhóm

MSSV	Họ Tên	Email	Điện thoại
1512003	Lê Tuấn Anh	ltanh035@gmail.com	0163 819 7063
1512004	Lê Việt Anh	levietanh.k15.it@gmail.com	0168 406 4269
1512029	Trần Quốc Bảo	tranquocbao3897@gmail.com	093 420 9840

2 Cài đặt System Call

❖ Cách xây dựng một System Call

Xây dựng system call **SC_Create**. System call này sẽ cho phép người dùng tạo một tập tin với tham số là tên tập tin. Trả về 0 nếu thành công, -1 nếu thất bại.

- **Bước 1:** Trong tập tin /code/userprog/syscall.h thêm dòng khai báo syscall mới (đã được thực hiện sẵn, nhưng cần sửa lại kiểu trả về của hàm Create từ void thành int)

```
#define SC_Create 4
int Create(char *name);
```

- **Bước 2:** Trong tập tin /code/test/start.c và /code/test/start.s thêm dòng (đã được thực hiện sẵn):

```
        .globl Create
        .ent Create
Create:
        addiu $2,$0,SC_Create
        syscall
        j $31
        .end Create
```

- **Bước 3:** Trong tập tin code/userprog/exception.cc sửa điều kiện **if** … thành **switch** … **case**. Với giá trị switch là **type** và các case chính là các system call ở đây case là **SC_Create**. Lưu ý trong phần xử lý System call này có xử lý liên quan đến tên tập tin là một chuỗi kí tự cho nên ta cần phải có hàm **User2System**, mục đích của hàm này là sao chép bộ nhớ đệm từ vùng nhớ người dùng (User Memory) vào vùng nhớ hệ thống (System Memory), và ngược lại ta có hàm **System2User**.
  - Lưu ý là hai hàm xử lý này được đặt bên trong lớp Machine trong file /code/machine.h và phần định nghĩa được đặt trong file /code/translate.cc bởi vì hai hàm này liên quan đến hệ thống và translate có nghĩa là chuyển đổi cho nên cài đặt như vậy là thích hợp nhất.

- Prototype của hai hàm như sau:

```
// Input: - User space address (int)
// - Limit of buffer (int)
// Output:- Buffer (char*)
// Purpose: Copy buffer from User memory space to System memory space
char* User2System(int virtAddr,int limit);
```

```
// Input: - User space address (int)
// - Limit of buffer (int)
// - Buffer (char[])
// Output:- Number of bytes copied (int)
// Purpose: Copy buffer from System memory space to User memory space
int System2User(int virtAddr,int len,char* buffer);
```

- **Bước 4:** Viết chương trình ở mức người dùng để kiểm tra system call **SC_Create** có hoạt động hay không? Chương trình được viết ở file /code/test/createfile.c
- **Bước 5:** Phải sửa lại file Makefile trong /code/test để có thể biên dịch chương trình vừa được viết để kiểm tra.
  - Thêm **createfile** vào dòng all:

```
all: halt shell matmult sort createfile
```

  - Thêm đoạn sau phía sau matmult:

```
createfile.o: createfile.c
$(CC) $(CFLAGS) -c createfile.c
createfile: createfile.o start.o
$(LD) $(LDFLAGS) start.o createfile.o -o createfile.coff
../bin/coff2noff createfile.coff createfile
```

- **Bước 6:** Biên dịch lại nachos.
- **Bước 7:** Thực thi chương trình người dùng createfile. Nếu không có thông báo lỗi nào hiện ra thì xem như chương trình đã thực thi thành công. Tập tin vừa được tạo sẽ nằm trong thư mục /code

❖ Xây dựng các System Call còn lại

a. **System call SC_CreateFile:** int CreateFile(char *filename)

- Sử dụng hàm tạo file của hệ thống (FileSystem).
- Trả về: 0 nếu thành công, hoặc -1 nếu có lỗi xảy ra. *(Các lỗi được liệt kê bên dưới)*
- Xử lý các exception:
  - Hệ thống không đủ bộ nhớ để xử lý mảng kí tự tên file.
  - Không thể tạo file do không có quyền hoặc các lỗi khác (nếu có).

b. **System call SC_Open:** OpenFileId Open(char *name, int type)

- Xây dựng một cấu trúc để lưu các file đang được mở và có thể trả về ID của chúng. Ở đây dùng con trỏ cấp hai và đặt nó trong lớp FileSystem, bởi vì nếu đặt nó trong file exception.cc thì mỗi lần gọi syscall thì file exception.cc sẽ bị xóa và tạo lại -> làm mất bảng mô tả file.
- Để an toàn bảng mô tả file được đặt trong trường private của lớp FileSystem. Xây dựng thêm các hàm Getter – Setter cho bảng mô tả file này.
- Ở đây dùng con trỏ cấp 2 cho dễ cấp phát và thu hồi vùng nhớ - dễ dàng quản lý các con trỏ cấp 1 là OpenFile*. Ở đây để các biến trong public cho dễ quản lý. Có viết thêm hàm *int EmptyEntryInTable()* để tìm các vị trí trống trong bảng file. Nếu trả về -1 thì bảng file đã đầy, cho nên không thể mở thêm file nữa. (Bỏ qua 2 ô đầu tiên 0, 1 trong bảng filde để dành cho Console In và Console Out)
- Đồng thời phải lưu lại được loại file đang mở là chỉ đọc hay đọc và ghi. Biến trạng thái này

```
48 class FileSystem {
49   public:
50     OpenFile** fileTable_;
51     int amount_;
52
```

lưu trong đối tượng OpenFile là hợp lý nhất.

- Sử dụng hàm Open của hệ thống thêm dữ liệu trường xử cho type, đồng thời cũng cập nhật bảng file và trả ra vị trí của OpenFile* đã mở trong bảng file. Thuộc tính type (int) được

đặt trong scope private của lớp đối tượng OpenFile cho nên có xây dựng thêm các hàm Getter và Setter.

- Cập nhật các hàm của lớp đối tượng FileSystem và OpenFile cho phù hợp với bảng file và thuộc tính type.
- Trả về: 0 nếu thành công, hoặc -1 nếu có lỗi xảy ra. *(Các lỗi được liệt kê bên dưới)*
- Xử lý các exception:
  - Bảng file đã đầy.
  - Hệ điều hành không còn đủ bộ nhớ để nạp tên file.
  - Lỗi không thể mở file. (Trả về từ FileSystem)

c. **System call SC_Close:** int CloseFile(OpenFileId id)

- **Lưu ý:** Vì nếu sửa void Close sẽ bị lỗi và ảnh hưởng đến nhiều file khác nên ta đặt lại tên là CloseFile để có thể trả về int.

❖ *Về cơ bản thì CloseFile không thể đóng được OpenFileID là 0, 1 vì chúng là console. Cho nên người dùng gọi lệnh đóng hai OpenFileID đặc biệt này chương trình sẽ báo lỗi.*

- Trả về: 0 nếu thành công, hoặc -1 nếu có lỗi xảy ra. *(Các lỗi được liệt kê bên dưới)*
- Xử lý các exception:
  - Người dùng truyền sai OpenFileID hoặc không hợp lệ.
  - Người dùng cố tình gọi (❖) ở trên.

d. **System call SC_Write:** int Write(char *buffer, int charcount, OpenFileId id)

- Nếu là file thông thường có OpenFileID từ 2-9 thì ta dùng hàm Write của lớp đối tượng OpenFile để ghi. Chỉ cần truyền đúng tham số.
- Nếu OpenFileID là ConsoleOutput thì ta dùng hàm Write của lớp SynchConsole với với biến toàn cục gSynchConsole (biến này được tạo cùng cách với biến toàn cục machine hay là fileSystem).
- Trả về: số byte thực sự được đọc, hoặc -1 nếu có lỗi xảy ra. *(Các lỗi được liệt kê bên dưới)*
- Xử lý các exception:
  - Người dùng truyền sai OpenFileID hoặc không hợp lệ. Bao gồm có truyền OpenFileID của ConsoleInput.
  - Lỗi cấp phát chuỗi chứa kết quả đọc được.

e. **System call SC_Read:** int Read(char *buffer, int charcount, OpenFileId id)

- Nếu là file thông thường có OpenFileID từ 2-9 thì ta dùng hàm Read của lớp đối tượng OpenFile để đọc. Chỉ cần truyền đúng tham số.
- Nếu OpenFileID là ConsoleInput thì ta dùng hàm Read của lớp SynchConsole với với biến toàn cục gSynchConsole (biến này được tạo cùng cách với biến toàn cục machine hay là fileSystem). **Lưu ý:** Thêm kí tự NULL vào cuối chuỗi đọc được.
- Trả về: số byte thực sự được đọc, hoặc -1 nếu có lỗi xảy ra, hoặc -2 nếu đã là cuối file. *(Các lỗi được liệt kê bên dưới)*
- Xử lý các exception:
  - Người dùng truyền sai OpenFileID hoặc không hợp lệ. Bao gồm có truyền OpenFileID của ConsoleOutput.
  - Đã đến cuối file. Không còn dữ liệu để đọc tiếp.
  - Lỗi cấp phát chuỗi nhận chuỗi từ user.

f. **System call SC_Seek:** int Seek(int pos, OpenFileId id)

- Sử dụng hàm Seek của lớp đối tượng OpenFile. Nó được gọi thông qua con trỏ được quản lý trong bảng file nằm trong lớp đối tượng FileSystem.
- Trả về: vị trí đã seek thành công, hoặc -1 nếu có lỗi xảy ra. *(Các lỗi được liệt kê bên dưới)*
- Xử lý các exception:
  - Người dùng truyền sai OpenFileID hoặc không hợp lệ.
  - Người dùng yêu cầu seek trên console.
  - Người dùng truyền vị trí pos cần seek không đúng. (Nhỏ hơn 0 hoặc vượt quá độ dài của file)

g. **Các system calls hỗ trợ khác:**

- SC_Print – void print(char *str): Xuất chuỗi str ra màn hình console.
- SC_ScanChar – char ScanChar(): đọc một kí tự từ console.
- SC_ScanLine – int ScanLine(char *buf, int limit): đọc một dòng từ console.

❖ Demo

- Link youtube: https://youtu.be/HZtmIbFNwzc

3 Cài đặt đa chương

❖ Mô tả yêu cầu

Nachos hiện tại giới hạn chỉ thực thi một chương trình, cần phải có vài thay đổi trong quản lý vùng nhớ **addrspace.h** và **addrspace.cc** để chuyển từ hệ thống đơn chương thành hệ thống đa chương. Phải giải quyết các vấn đề về cấp phát các frames bộ nhớ vật lý cho các tiến trình, để có thể cùng lúc nạp nhiều tiến trình vào bộ nhớ. Đồng thời thiết kế thêm các lớp bổ trợ để quản lý nhiều tiến trình.

❖ Phương án giải quyết bài toán

- Các lớp quan trọng về quản lý tiến trình cần phải tìm hiểu, và cài đặt: **PTable** và **PCB** là quan trọng nhất giúp quản lý các tiến trình thực thi. Cài đặt hai lớp này trong thư mục /code/threads.
- Mô hình vòng đời của tiến trình:

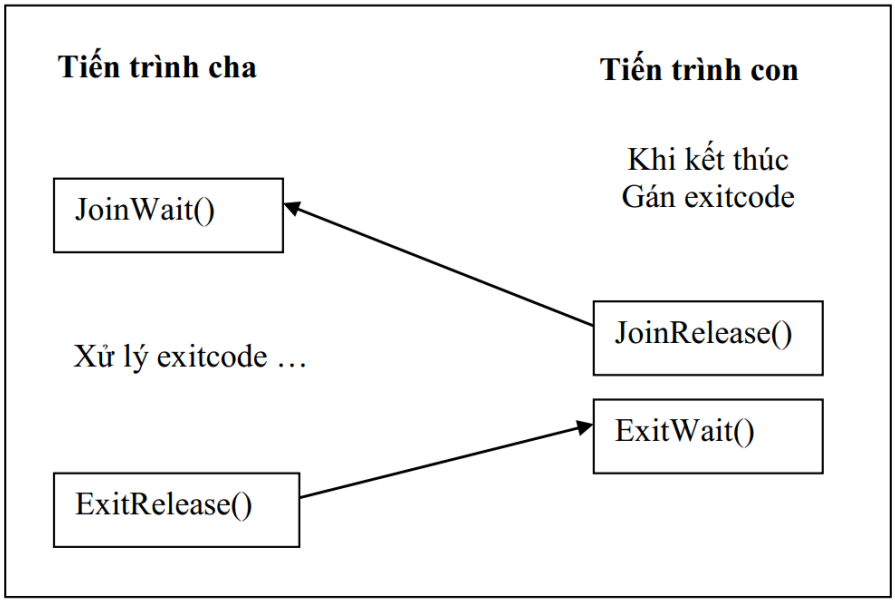

Figure 1. Circle Life Process

- Tiến trình con sau lời gọi system call **SC_Exec** thì được nạp vào bộ nhớ, sau đó ta Join nó vào tiến trình cha để tiến trình cha chờ tiến trình con kết thúc thì tiến trình cha mới được thực thi tiếp.
- Như vậy khi gọi Join thì HDH phải gọi **JoinWait** của tiến trình con, khi mà tiến trình con thực thi xong hoặc gọi system call **SC_Exit** thì lúc này HDH sẽ giải phóng tiến trình cha khỏi chờ bằng **JoinRelease**, gọi luôn **ExitWait** để xin phép tiến trình cha được kết thúc. Lúc này HDH sẽ gọi **ExitRelease** để kết thúc tiến trình con.

- Cài đặt cụ thể các lớp:

a. **Lớp Thread:** Ta thêm biến int processID để định danh các tiến trình đang chạy. Để ở public để cho dễ xử lý. processID của tiến trình được tạo ra là 0 để chắc chắn rằng tiến trình đầu tiên có processID = 0. (Tiến trình cha)

b. **Lớp PCB:** Lưu thông tin và phương thức để quản lý tiến trình.

```
class PCB {
private:
        Semaphore   *joinsem;     //semaphore cho qua trinh join
        Semaphore   *exitsem;     //semaphore cho qua trinh exit
        Semaphore   *mutex;
        int                 exitcode;
        Thread              *thread;
        int                 pid;
        int                 numwait;      //so tien trinh da join
public:
        int parentID;                   //ID cua tien trinh cha
        int             JoinStatus;   //Trang thai co Join voi tien trinh nao khong? neu co thi gia tri chinh la ID cua tien trinh ma no Join
        PCB(int id);
        ~PCB();
        int Exec(char *filename, int pID); //nap chuong trinh co ten luu trong bien filename va processID se la pID
        int GetID();                    // return pID của tiến trình gọi thực hiện
        int GetNumWait();               // return số lượng tiến trình chờ
        void JoinWait();                // 1. Tiến trình cha đợi tiến trình con kết thúc
        void ExitWait();                // 4. Tiến trình con kết thúc
        void JoinRelease();             // 2. Báo cho tiến trình cha thực thi tiếp
        void ExitRelease();             // 3. Cho phép tiến trình con kết thúc
```

```
        void IncNumWait();          // Tăng số tiến trình chờ
        void DecNumWait();          // Giảm số tiến trình chờ
        void SetExitCode(int ec);   // Đặt exitCode cho tiến trình
        int GetExitCode();          // Trả về exitCode
        char* GetNameThread();      // Trả về tên của tiến trình
};
```

- Constructor: Khởi các giá trị ban đầu. Quan trọng là các biến Semaphore, với joinsem và exitsem thì khởi tạo giá trị ban đầu là 0, còn với mutex thì là 1. Vì joinsem và exitsem sẽ được sử dụng trong các thao tác chờ (JoinWait, ExitWait).

```
joinsem = new Semaphore("joinsem", 0);
exitsem = new Semaphore("exitsem", 0);
mutex = new Semaphore("mutex", 1);
```

- Hàm int Exec(char *filename, int pID): Hàm dùng để nạp tiến trình mới.
  - Tạo tiến trình mới.
  - Gán các tham số pid, processID của thread vừa tạo, parentID.
  - Gọi phương thức Fork của tiến trình vừa tạo.

```
int PCB::Exec(char *filename, int pID) {
        mutex->P(); // Ngan khong cho nap 2 tien tinh cung luc
        thread = new Thread(filename); // Khoi tao thread
        if (thread == NULL) {
                ASSERT(FALSE);
                printf("Error: not enough memory.\n");
                mutex->V();
                return -1;
        }
        id = pID;
        hread->processID = pID;
        parentID = currentThread->processID;
        thread->Fork(StartProcess_2, pID);
        mutex->V();
        return pID;
}
```

c. **Lớp PTable:** Quản lý các tiến trình đang chạy. Mảng lưu các PCB. Hiện tại xử lý tối đa mảng PCB có 10 phần tử tức là chỉ có thể chạy 10 tiến trình cùng một lúc.

```
class PTable {
private:
	BitMap	*bm;
	PCB	*pcb[MAXPROCESS];
	int	psize;
	Semaphore	*bmsem;		//Dung de ngan chan truong hop nap 2 tien trinh cung luc
public:
	PTable(int size);		//Khoi tao size doi tuong pcb de luu size process. Gan gia tri ban dau la null. Nho khoi tao *bm va *bmsem de su dung
	~PTable();			//Huy doi tuong da tao
	int ExecUpdate(char* filename);		// return PID – SC_Exec
	int ExitUpdate(int ec);			// SC_Exit
	int JoinUpdate(int pID);		// SC_Join
	int GetFreeSlot();		//Tim slot trong de luu thong tin cho tien trinh moi
	bool IsExist(int pID);		//Kiem tra co ton tai process ID nay khong
	void Remove(int pID);		//Xoa mot processID ra khoi mang quan ly no, khi ma tien trinh nay da ket thuc
	char* GetName(int pID);	//Lay ten cua tien trinh co processID la pID
};
```

- Constructor: Quan trọng là phải dành phần tử đầu mảng cho tiến trình chính (main thread có processID = 0). Đồng thời khởi tạo các thuộc tính để sử dụng sau này.

```
// processID = 0 for main process
bm->Mark(0);
pcb[0] = new PCB(0);
pcb[0]->parentID = -1;
```

- Hàm int ExecUpdate(char* filename): đây là hàm dùng để nạp một tiến trình dựa trên đường dẫn tập tin thực thi. Làm theo hướng dẫn của giảng viên hướng dẫn thực hành. Có bốn bước quan trọng:
  - Xác thực đường dẫn tập tin thực thi và tập tin thực thi có hợp lệ hay không?
  - Kiểm tra xem nó có tự gọi chính nó hay không thông qua tên của tiến trình (cũng chính là đường dẫn tập tin thực thi).
  - Kiểm tra xem còn chỗ trống để thêm một tiến trình vào mảng để quản lý hay không?

- Tạo mới một PCB (lớp PCB), gán parentID của lớp mới tạo bằng với processID của currentThread (chính là tiến trình đang gọi ExecUpdate). Sau đó gọi phương thức Exec của PCB.
  - Lưu ý: Phải dùng bmsem để ngăn không cho tạo hai tiến trình cùng một lúc.

- Hàm int ExitUpdate(int ec): đầy là hàm dùng để kết thúc tiến trình hiện tại (currentThread).
  - Nếu currentThread là main thread thì gọi interrupt->Halt(); sau đó return.
  - Nếu không phải, kiểm tra xem có tồn tại processID của currentThread có tồn tại trong mảng quản lý hay không? Nếu không thì return;
  - Nếu tồn tại thì SetExitCode cho PCB. Sau đó gọi phương thức JoinRelease để giải phóng tiến trình cha đang đợi nó kết thúc, xong gọi ExitWait() để xin tiến trình cha cho phép kết thúc.
  - Sau đó giảm số tiến trình chờ của tiến trình cha. Rồi mới Remove tiến trình khỏi mảng quản lý.
- Hàm int JoinUpdate(int pID): để thực hiện đợi và block các tiến tình dựa trên định danh tiến trình pID.
  - Kiểm tra tính hợp lệ của pID. Không hợp lệ return.
  - Kiểm tra tiến trình gọi Join có phải là cha của tiến trình có processID là pID hay không. Không hợp lệ return.
  - Tăng số tiến trình chờ của tiến trình cha.
  - Gọi JoinWait để tiến trình cha chờ tiến trình con thực hiện xong.
  - Sau khi tiến trình con thực hiện xong, lấy exitcode của tiến trình con.
  - Gọi ExitRelease để cho phép tiến trình con kết thúc. Trả về exitcode.

- Sau khi có hai lớp PTable, PCB ta khai báo biến toàn cục PTable là pTab để sử dụng.
- Chỉnh sửa, thêm các hàm để xử lý việc cấp phát bộ nhớ cho các tiến trình:

a. **Khai báo các biến sau:** BitMap *gPhysPageBitMap – dùng để quản lý các trang vật lý. Semaphore *addrLock – dùng để ngăn không cho hai tiến trình cùng thực hiện thao tác nạp trang vật lý cùng lúc (vì có thể gây lỗi trong việc cấp phát trang).

b. **Ở lớp AddrSpace:** Ta thêm constructor AddrSpace(char *filename), đồng thời chỉnh sửa lại constructor cũ AddrSpace (OpenFile *executable) để phù hợp trong việc cấp phát đa chương.

c. **Thêm hàm:** void StartProcess_2(int id) ở file progtest.cc để sử dụng kết hợp với Fork của thread.

- Hàm này sẽ lấy id của tiến trình gọi Fork trong PTable.
- Sau đó lấy đường dẫn tập tin thực thi thông qua phương thức GetName của PTable thông qua id.
- Khởi tạo không gian địa chỉ vật lý thông qua OpenFile *executable sau khi gọi fileSystem->Open(filename);.
- Gán không gian địa chỉ vật lý vừa tạo cho tiến trình đang chạy hiện tại.

d. **Thêm các system call** trong exception.cc:

- SC_Exec – dùng để gọi thực thi một chương trình mới trong một system thread mới.
- SC_Exit – dùng để thực hiện thoát tiến trình đã join.
- SC_Join – dùng để thực hiện đợi và block dựa trên SpaceID đã tạo ra lúc gọi SC_Exec.

- Viết chương trình để test hoạt động của phần đa chương vừa mới cài đặt xong. Như hướng dẫn của giảng viên hướng dẫn thực hành. Lưu ý: Trong scheduler.c phải thêm:

```
#include "syscall.h"
void main()
{
    int pingPID, pongPID;

    print("Ping-Pong test starting ...\n\n");
    pingPID = Exec("./test/ping");
    pongPID = Exec("./test/pong");

    Join(pingPID);
    Join(pongPID);
}
```

- Như vậy, đã hoàn thành các nội dung đồ án 02, bao gồm:
- Giải quyết các vấn đề quản lý, tạo và xóa các tiến trình.
- Giải quyết các vấn đề cấp phát các frames bộ nhớ vật lý để cho nhiều chương trình có thể nạp lên bộ nhớ cùng lúc.
- Biến hệ điều hành nachos từ đơn chương trở thành đa chương.

❖ Demo

Figure 2. Result of scheduler

Link youtube: https://youtu.be/UMVMJBWoyJc

4 Cài đặt đồng bộ hóa

❖ Mô tả yêu cầu

Nachos hiện tại điều phối các tiến trình chạy ngẫu nhiêu không theo quy luật, vì vậy muốn chương trình chạy và điều phối theo yêu cầu của từng bài toán thì phải cài đặt các lớp và system call giải quyết. Nachos đã cung cấp sẵn các lớp để quản lý, cụ thể là lớp Semaphore dùng để khóa / mở cho tiến trình có thể chạy vào một vùng được chỉ định. Như vậy để giải quyết việc quản lý các Semaphore sao cho tiện lợi thì phải cài đặt lớp để quản lý, và các system call tương ứng.

Như ở bài toán trước, giải quyết vấn đề đa chương khi xuất A, B ra màn hình ta chỉ xuất được A, B xen kẽ ngẫu nhiên. Ở bài toán này chúng ta sẽ xây dựng sao cho giải quyết vấn đề này, tức là có thể xuất A rồi mới đến B xen kẽ nhau không còn ngẫu nhiên nữa.

❖ Phương án giải quyết bài toán

- Các lớp quan trọng về đồng bộ cần phải tìm hiểu, và cài đặt: Semaphore (đã được cài đặt sẵn), STable là quan trọng nhất giúp quản lý các Semaphore của chương trình. Như vậy ta chỉ cần cài đặt lớp STable trong /code/threads

```
class STable {
private:
    BitMap* bm;
    Semaphore* semTab[MAX_SEMAPHORE];
    // quản lý slot trống
    // quản lý tối đa 10 đối tượng Sem
public:
    // khởi tạo size đối tượng Sem để quản lý 10 Semaphore. Gán giá trị ban đầu là null
    // nhớ khởi tạo bm để sử dụng
    STable();
    ~STable();                          // hủy các đối tượng đã tạo
    int Create(char* name, int init);   // Kiểm tra Semaphore "name" chưa tồn tại thì tạo Semaphore mới. Ngược lại, báo lỗi.
```

```
    int Wait(char* name);          // Nếu tồn tại Semaphore "name" thì gọi this->P() để
thực thi. Ngược lại, báo lỗi.
    int Signal(char* name);        // Nếu tồn tại Semaphore "name" thì gọi this->V() để
thực thi. Ngược lại, báo lỗi.
    int FindFreeSlot(int id);      // Tìm slot trống.
};
```

- Cài đặt các System Call để sử dụng lớp Stable:

  a. **SC_CreateSemaphore**: dùng để tạo một Semaphore

➢ Khai báo hàm: int CreateSemaphore(char* name, int valInit) trong syscall.h và #define giá trị SC_CreateSemaphore.

➢ Quá trình xử lý:

- Đọc địa chỉ con trỏ name từ thanh ghi r4.
- Đọc giá trị valInit từ thanh ghi r5.
- Copy name từ User Memory sang System Memory bằng hàm User2System trong lớp Machine.
- Gọi thực hiện hàm semTab->Create(name, valInit) để tạo Semaphore, nếu có lỗi thì báo lỗi.
- Lưu kết quả chính là vị trí của Semaphore vừa tạo ra trong lưu trong mảng quản lý của lớp Stable.
- Trả kết quả vào thanh ghi r2.

  b. **SC_Up**: Tăng giá trị của Semaphore lên 1. (Giải phóng tiến trình đang chờ)

➢ Khai báo hàm: int Up(char* name) trong syscall.h và #define giá trị SC_Up.

➢ Quá trình xử lý:

- Đọc địa chỉ con trỏ name từ thanh ghi r4.
- Copy name từ User Memory sang System Memory bằng hàm User2System trong lớp Machine.
- Kiểm tra Semaphore “name” này có nằm trong mảng quản lý chưa, nếu chưa có thì báo lỗi và trả về -1.
- Gọi phương thức semTab->Signal(name).

- Lưu kết quả chính là vị trí của Semaphore vừa tạo ra trong lưu trong mảng quản lý của lớp STable.
- Trả kết quả vào thanh ghi r2.

c. **SC_Down**: Giảm giá trị của Semaphore lên 1. (Buộc tiến trình phải chờ)

➢ Khai báo hàm: int Down(char* name) trong syscall.h và #define giá trị SC_ Down.

➢ Quá trình xử lý:

- Đọc địa chỉ con trỏ name từ thanh ghi r4.
- Copy name từ User Memory sang System Memory bằng hàm User2System trong lớp Machine.
- Kiểm tra Semaphore "name" này có nằm trong mảng quản lý chưa, nếu chưa có thì báo lỗi và trả về -1.
- Gọi phương thức semTab->Wait(name).
- Lưu kết quả chính là vị trí của Semaphore vừa tạo ra trong lưu trong mảng quản lý của lớp STable.
- Trả kết quả vào thanh ghi r2.

- Khai báo biến toàn cục STable *semTab;
- Viết chương trình để test hoạt động của phần đồng bộ hóa vừa mới cài đặt xong, giải quyết bài toán xuất A, B không xen kẽ ngẫu nhiên mà A trước B sau.
- Viết chương trình scheduler_sync.c trong /code/test:

```
#include "syscall.h"
int main()
{
    int pingPID, pongPID;
    CreateSemaphore("A", 1); // Tạo ra Semaphore "A" với giá trị ban đầu là 1
    CreateSemaphore("B", 0); // Tạo ra Semaphore "B" với giá trị ban đầu là 0

    print("Ping-Pong test starting ...\n\n");
    pingPID = Exec("./test/ping_sync");
    pongPID = Exec("./test/pong_sync");

    Join(pingPID);
```

```
        Join(pongPID);
    }
```

- Viết chương trình ping_sync.c trong /code/test:

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

- Viết chương trình pong_sync.c trong /code/test:

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

- HDH nếu điều phối cho B chạy trước thì vướng phải Down(“B”) hiện tại Sem “B” đang là 0 nên không thể in ra ‘B’, điều phối qua cho A, A sẽ chạy được Down(“A”) do hiện tại Sem “A” đang là 1, nếu lúc này HDH điều phối qua cho B thì B vẫn vướng tại Down(“B”), bắt buộc phải in ‘A’ trước để Up(“B”). Lúc này Sem “A” là 0 còn Sem “B” là một. Tương tự qua lại như vậy. Cuối cùng chỉ có thể xuất A rồi mới xuất B cứ thế cho đến hết.

Như vậy, đã hoàn thành các nội dung đồ án 03, bao gồm:

- Giải quyết các vấn đề đồng bộ, quản lý đồng bộ tạo và sử dụng các Semaphore.
- Làm cho hệ điều hành nachos có khả năng đồng bộ hóa.

❖ Demo

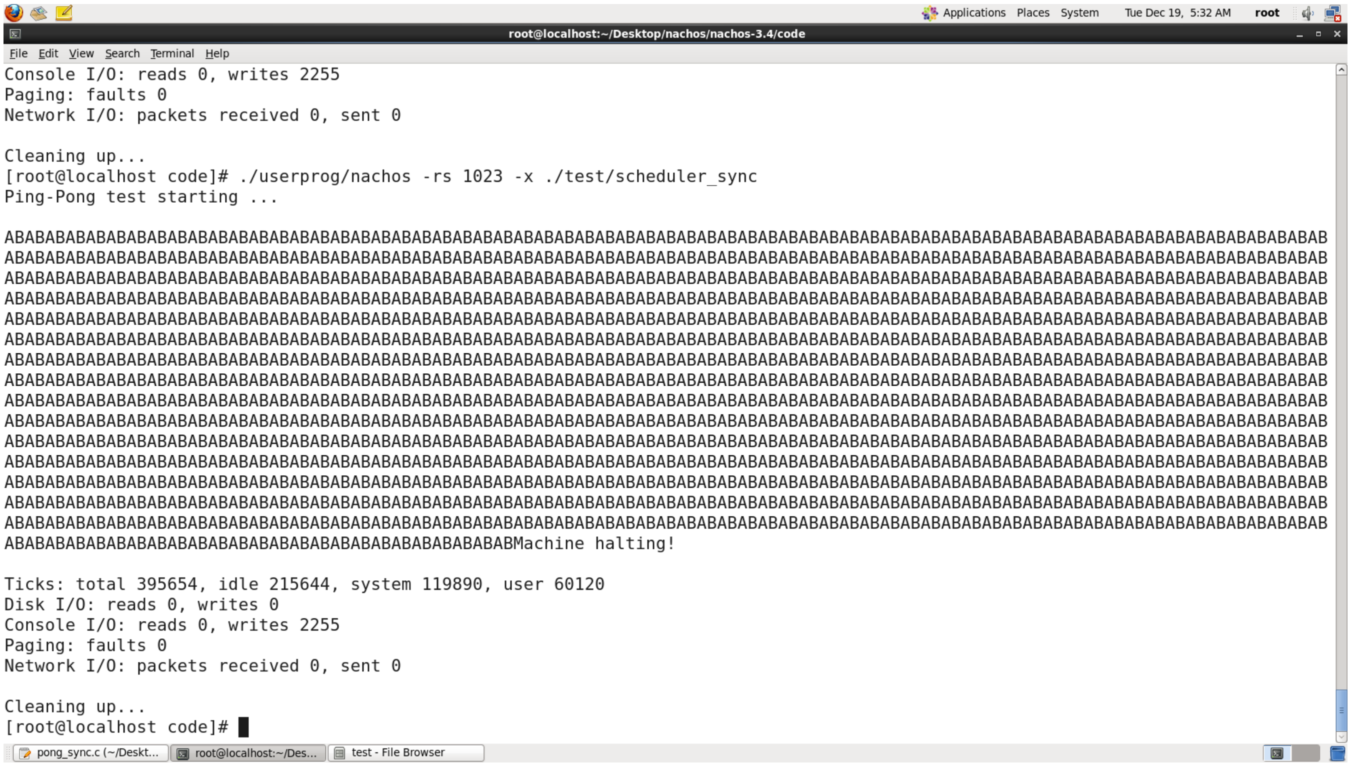

Figure 3. Result of scheduler_sync

Link youtube: https://youtu.be/29KU5Lw-WWE

5 Tài liệu tham khảo

Các file tài liệu do giảng viên hướng dẫn thực hành cung cấp. Bao gồm có:

- Huong Dan Cac Syscall Ve Da Chuong.pdf
- [5] Da Chuong Dong Bo Hoa.doc
- Seminar_HDH_Project 2.pptx
- Constructor_Cua_AddrSpace_2.pdf
- pcb.h
- ptable.h