TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

# HỆ ĐIỀU HÀNH

## ĐỒ ÁN 2: LẬP TRÌNH NACHOS

**Giảng viên:** Trần Trung Dũng

**Hướng dẫn:** Lê Giang Thanh

**Lớp**: 19TN

Người thực hiện:



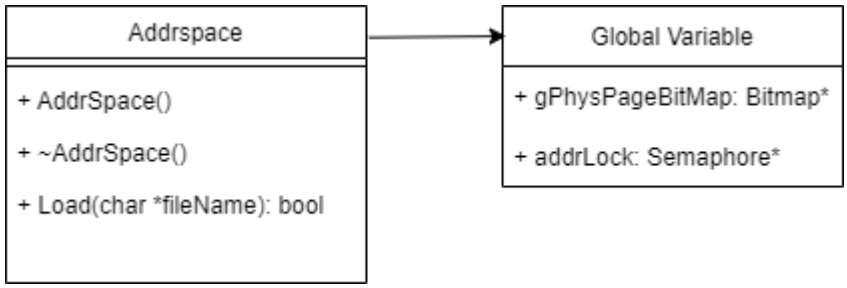
| Họ và tên | MSSV |
|---|---|
| Mai Duy Nam | 19120298 |
| Đặng Thái Duy | 19120491 |
| Võ Văn Toàn | 19120690 |

# MỤC LỤC

# 1. Cài đặt tổng quan

## 1.1 AddrSpace

Lớp AddrSpace đại diện cho một không gian địa chỉ của một tiến trình, có nhiệm vụ chính là quản lý không gian địa chỉ này (khởi tạo, cấp phát và hủy bỏ). Lớp này có chức năng mấu chốt khi ta chuyển NachOS từ hệ điều hành đơn chương sang hệ điều hành đa chương.




Để hoạt động đúng, AddrSpace nhờ sự giúp đỡ của hai biến toàn cục sau:

- gPhysPageBitMap: một bitmap dùng để quản lý các frame vật lý, có chức năng kiểm tra frame còn trống hay đã được dùng, đánh dấu frame được dùng và giải phóng frame
- addrLock: một semaphore dùng để khóa truy cập vào bộ nhớ khi AddrSpace thực hiện các thao tác nạp dữ liệu từ chương trình

Các thay đổi chính về lớp AddrSpace:

- Constructor: chỉ khởi tạo biến pageTable = NULL và numPages = 0
- Destructor: ngoài việc giải phóng vùng nhớ cho pageTable, ta còn giải phóng các frame vật lý bằng cách đánh dấu chúng là clear thông qua gPhysPageBitMap
- Phương thức Load(char *fileName): phương thức này nhận vào tên của chương trình (định dạng NOFF) và thực hiện nạp chương trình này vào bộ nhớ chính. Load làm việc này theo các bước sau

  B1: Mở file thông qua fileSystem và đọc các thông số về chương trình từ NoffHeader

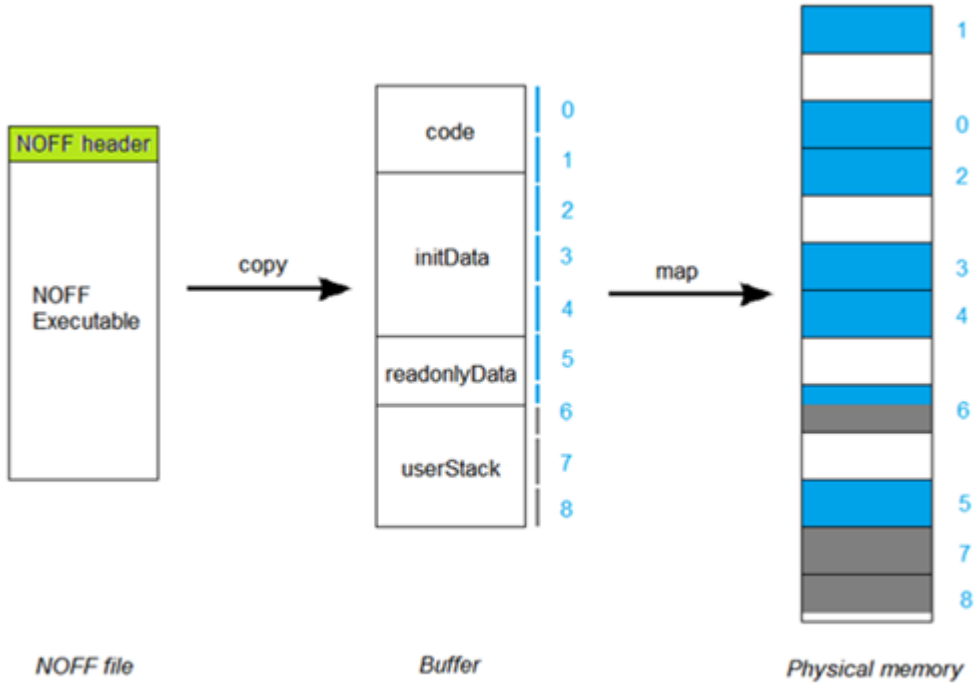
  B2: Xác định không gian vật lý cần thiết cho chương trình, từ đó xác định số trang bộ nhớ ảo (numPages)

  B3: Khóa truy cập vào bộ nhớ. Thực hiện ánh xạ từ các trang bộ nhớ ảo vào các frame bộ nhớ vật lý còn trống bằng việc gán các giá trị phù hợp cho biến virtualPage và physicalPage của pageTable

B4: Nạp mã nguồn và dữ liệu của chương trình từ file NOFF vào bộ nhớ. Ta thực hiện qua hai bước:

- Nạp toàn bộ chương trình vào buffer, theo đúng thứ tự mà các segment sẽ xuất hiện trong vùng nhớ vật lý
- Chia nhỏ buffer thành các trang và nạp từng trang vào frame vật lý tương ứng đã được ánh xạ trong pageTable

B5: Mở khóa truy cập bộ nhớ và dọn dẹp




*Cách nạp mã nguồn và dữ liệu từ file NOFF vào bộ nhớ vật lý.*

## 1.2 PCB

Lớp PCB (Process Control Block) có nhiệm vụ lưu các thông tin liên quan đến một tiến trình đang được thực thi trong hệ thống. Cấu trúc lớp PCB như sau:

| Thuộc tính | Phương thức |
|---|---|
| parentID | Exec() |
| _pid | JoinWait() |
| _exitCode | JoinRelease() |
| _file | ExitWait() |
| _thread | ExitRelease() |
| _numwait | Các hàm setter |

| _joinsem | Các hàm getter |
|---|---|
| _exitsem | |
| _mutex | |

a) Thuộc tính

- parentID: PID của tiến trình cha gọi khởi động tiến trình này
- _pid: PID của tiến trình này
- _exitCode: exit code của tiến trình (được gán khi tiến trình kết thúc)
- _file: tên file thực thi
- _thread: Thread object của tiến trình
- _numwait: số tiến trình con mà tiến trình này đang đợi
- _joinsem, _exitsem, _mutex: các semaphore cần thiết cho quá trình join, exit

Một đối tượng của lớp PCB được sử dụng làm một entry trong process table (sẽ được quản lý bởi lớp PTable). Để thuận tiện, ta quy định PID của một tiến trình là index của PCB của tiến trình đó trong process table. Tiến trình đầu tiên được chạy (main) không có tiến trình cha nên được quy định parentID = -1.

b) Phương thức

- Exec(char *fileName, int pid): phương thức này cập nhật PID (tham số pid được truyền vào từ PTable), lưu tên file thực thi, tạo Thread object cho tiến trình và fork Thread object này. Fork một Thread object là việc ta định nghĩa entry point cho tiến trình và đưa tiến trình vào ready queue (nhờ đến dịch vụ của hệ điều hành).
    - Entry point là một hàm dùng làm điểm xuất phát cho tiến trình khi nó được thực thi lần đầu tiên. Entry point được định nghĩa qua hàm StartProcess (trong file threads/pcb.cc)
    - StartProcess nhận vào tên của file thực thi, tạo một AddrSpace và nạp chương trình vào AddrSpace đó thông qua phương thức Load. Nếu thành công, StartProcess gọi phương thức Execute của AddrSpace để chạy tiến trình.
- JoinWait(): chờ trên _joinsem của tiến trình này. Phương thức này được gọi bởi tiến trình cha khi nó muốn join vào tiến trình này
- JoinRelease(): giải phóng _joinsem, từ đó giải phóng tiến trình cha đang join. JoinRelease() được gọi bởi chính tiến trình hiện tại khi nó chuẩn bị kết thúc
- ExitWait(): chờ trên _exitsem, được gọi bởi tiến trình hiện tại. Phương thức này dùng để block tiến trình con khi nó kết thúc cho đến khi tiến trình cha kết thúc, do ta không thể để tiến trình con kết thúc theo cách bình thường vì nó sẽ shutdown cả hệ thống NachOS trước khi cha hoàn thành (điều ta không mong muốn)

- ExitRelease(): giải phóng _exitsem. Phương thức này được gọi bởi tiến trình cha khi nó được giải phóng khỏi _joinsem.
- Các getter và setter cho các biến của PCB

### 1.3 PTable

Lớp PTable dùng để quản lý process table của toàn bộ hệ thống. Nhiệm vụ chính của nó là cập nhật process table một cách hợp lý với mỗi yêu cầu exec, join và exit của các process. Cấu trúc lớp Ptable như sau:

| Thuộc tính | Phương thức |
|---|---|
| _pcbs[] | ExecUpdate() |
| _num_processes | JoinUpdate() |
| _slotManager | ExitUpdate() |
| __sem | |

a) Thuộc tính

- _pcbs[]: mảng chứa các PCB, là process table cần quản lý
- _num_processes: số lượng tiến trình đang chạy của hệ thống
- _slotManager: một bitmap dùng để quản lý các entry còn trống trong _pcbs
- _sem: semaphore dùng để khóa truy cập vào _pcbs khi PTable đang thực hiện cập nhật

b) Phương thức

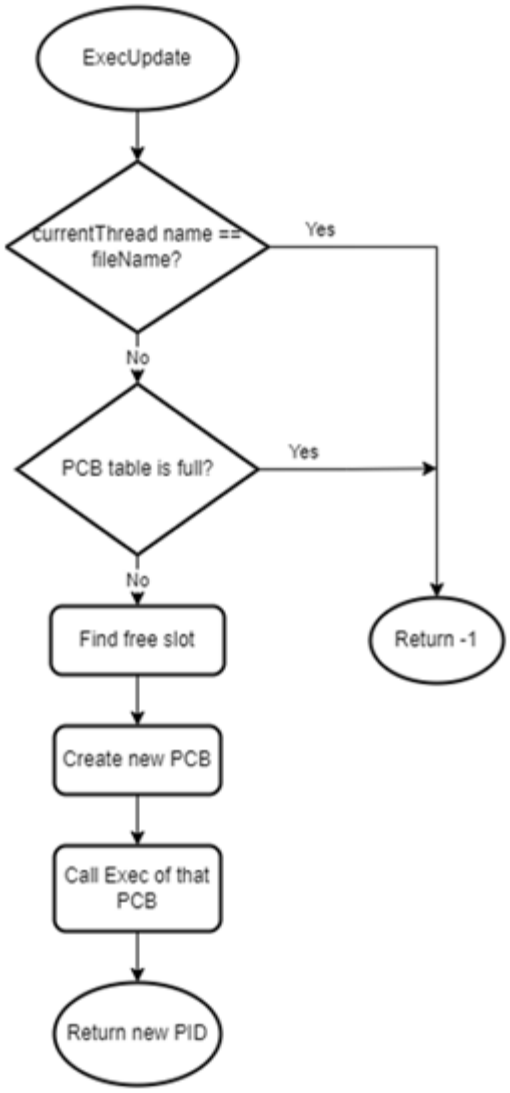
PTable cung cấp ba phương thức chính là ExecUpdate, JoinUpdate và ExitUpdate. Ba phương thức này sẽ được sử dụng bởi ba system call Exec, Join và Exit.

- ExecUpdate(char *fileName): cập nhật process table khi một tiến trình gọi Exec. Phương thức này nhận vào tên file thực thi cần khởi động. Hoạt động như sau:

  B1: Kiểm tra tiến trình có tự gọi Exec chính mình hay không

  B2: Tạo một PCB mới, xác định PID mới cho PCB này, gán parentID của PCB này với PID của tiến trình hiện tại

  B3: Gọi phương thức Exec của PCB mới để nó cập nhật các thông tin của mình




*Flowchart của ExecUpdate*

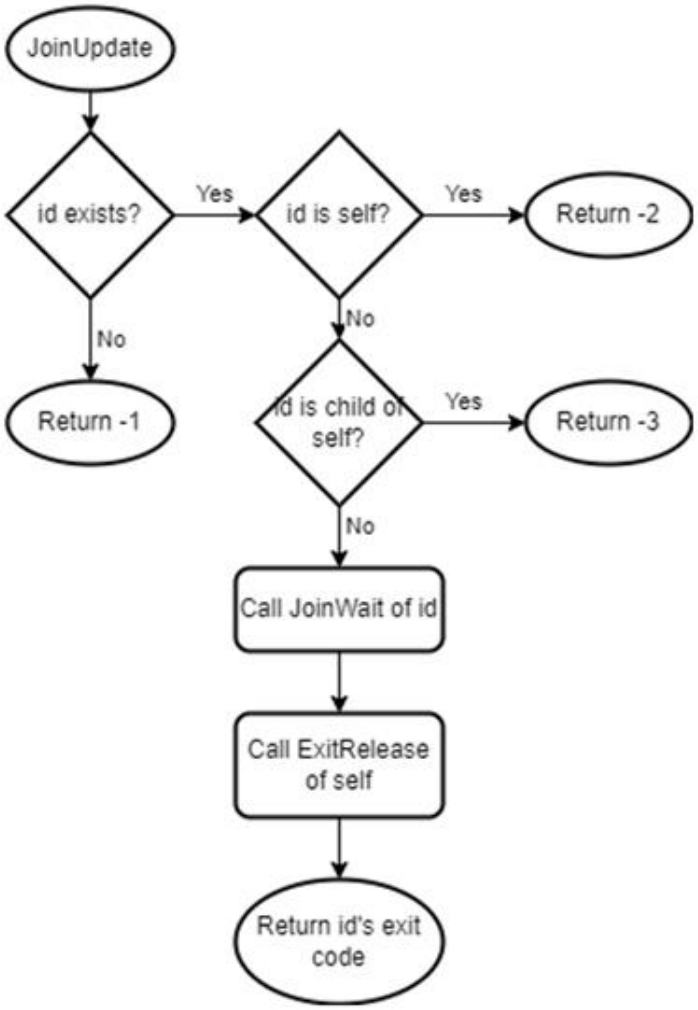
- JoinUpdate(int id): cập nhật các PCB khi một tiến trình cha join vào tiến trình con. Phương thức này nhận vào PID của tiến trình con. Hoạt động như sau:

  B1: Kiểm tra id này có là PID của tiến trình con của tiến trình hiện tại hay không

  B2: Gọi phương thức JoinWait của PCB của tiến trình con để block tiến trình cha hiện tại

  B3: Khi tiến trình con kết thúc, tiến trình cha quay trở lại, nó gọi ExitRelease để tiến trình con kết thúc

  B4: Trả về exit code của tiến trình con




*Flowchart của JoinUpdate*

- ExitUpdate(int ec): cập nhật PCB khi một tiến trình kết thúc. Hoạt động như sau:

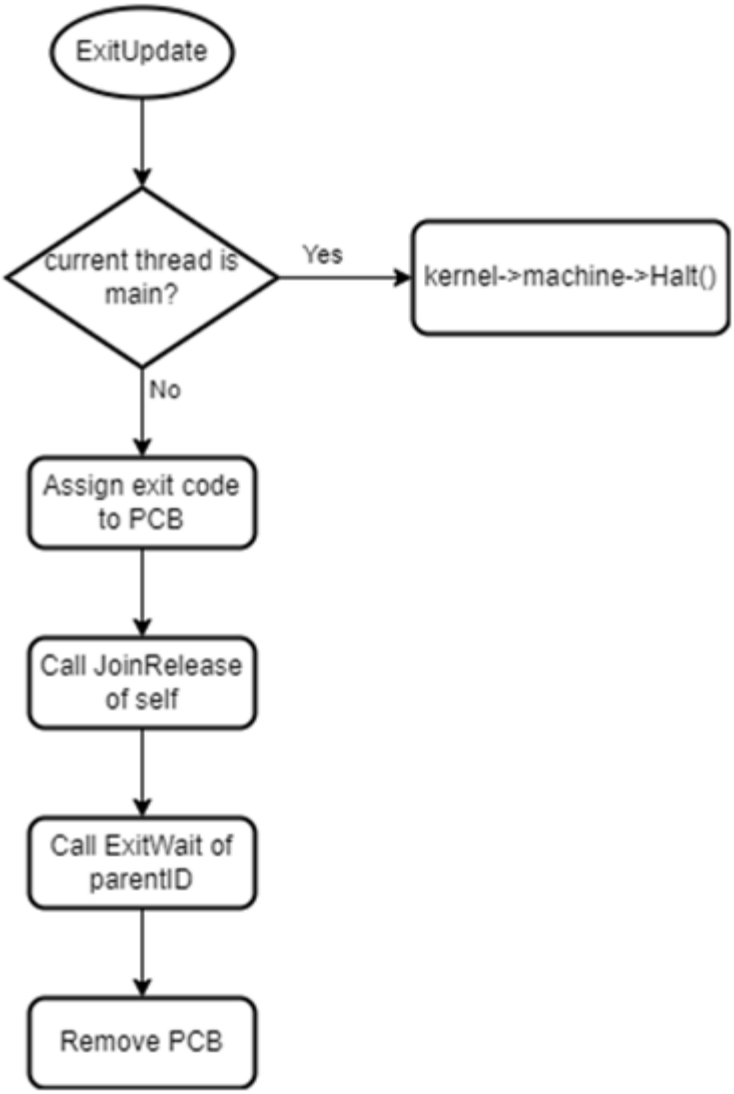
  B1: Kiểm tra xem tiến trình hiện tại có phải main. Nếu phải thì gọi Halt.

  B2: Nếu không phải, nó gán exit code vào PCB của tiến trình hiện tại

  B3: Gọi JoinRelease của tiến trình hiện tại để giải phóng bất kỳ tiến trình cha nào đang join vào nó

  B4: Gọi ExitWait để chờ tiến trình cha cho nó kết thúc

  B5: Dọn dẹp và giải phóng vùng nhớ cho PCB




*Flowchart của ExitUpdate*

Trên đây là những phương thức cơ bản dành cho quản lý đa chương trình. Tuy nhiên, các phương thức trên chỉ hỗ trợ cho việc quản lý các tiến trình được gọi mới từ Exec, còn tiến trình đầu tiên được chạy khi khởi động NachOS (tiến trình main) thì chưa được quản lý.

Do đó PTable có thêm phương thức InitializeFirstProcess(const char *fileName, Thread *thread). Phương thức này cấp phát PCB cho main với PID mặc định là 0, cùng với việc lưu lại tên file và Thread object của main. Các tiến trình về sau sẽ được sinh ra bởi main. Để lưu được các thông tin này, ta gọi InitializeFirstProcess trong hàm main() (threads/main.cc) ngay trước khi AddrSpace của main được nạp.

Lớp PTable có một đối tượng pTab được khai báo toàn cục (khai báo nằm trong file main.h).

## 1.4 Sem

Lớp Sem dùng để đóng gói lại class Semaphore có sẵn của NachOS 4.0. Nhiệm vụ chính của nó là tạo một lớp để người dùng tự define các phương thức cho cho Semphore nhưng không ảnh hưởng đến mã nguồn ban đầu. Quan trọng nhất là nó tạo ra vùng nhớ mới để lưu tên Semaphore nếu không mọi tên Semaphore đều trỏ tới cùng một vùng nhớ. Cấu trúc lớp Sem như sau:

| Thuộc tính | Phương thức |
|---|---|
| name | Wait() |
| sem | Signal() |
| | getName() |

a) Thuộc tính
- name: tên của Semaphore được đóng gói
- sem: con trỏ đến Semaphore mà Sem quản lý

b) Phương thức
- Wait(): gọi đến P() của Semaphore
- Signal(): gọi đến V() của Semaphore
- getName(): trả về tên của Sem (Semaphore đang quản lý)

## 1.5 STable

Lớp STable dùng để quản lý semaphore table của toàn bộ hệ thống. Nhiệm vụ chính của nó là cập nhật semaphore table một cách hợp lý với mỗi yêu cầu create, wait và signal của process. Cấu trúc lớp Stable như sau:

| Thuộc tính | Phương thức |
|---|---|
| table[] | Create() |
| | Wait() |
| | Signal() |

a) Thuộc tính
- table[]: mảng chứa các Sem, là semaphore table cần quản lý

b) Phương thức

STable cung cấp ba phương thức chính là Create, Wait và Signal. Ba phương thức này sẽ được sử dụng bởi ba system call CreateSemaphore, Wait và Signal.

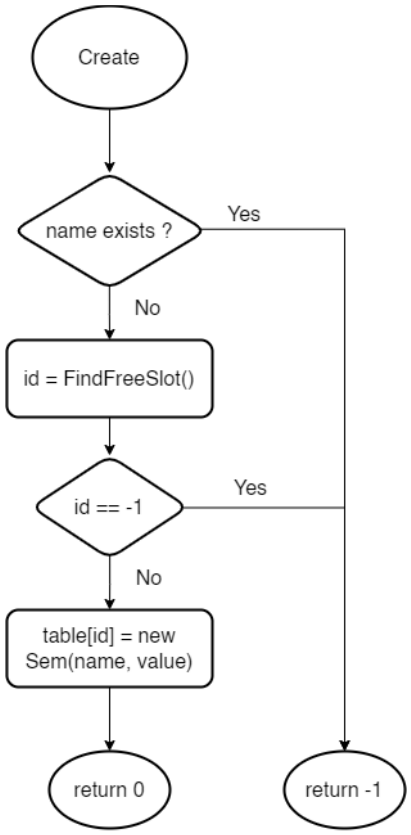
- Create(char *name, int value): Tạo một semaphore mới trong table

  B1: Kiểm tra semaphore với name đã tồn tại chưa. Nếu không thì trả về -1

  B2: Tìm vị trí còn trống trong table bằng phương thức FindFreeSlot(). Nếu không có thì trả về -1

  B3: Tạo object Semphore(name, value) và gán con trỏ vào vị trí còn trống.

  B4: Trả về 0



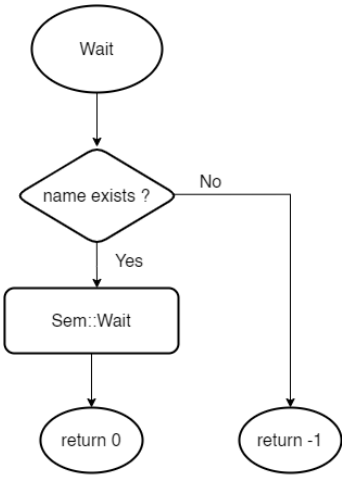

*Flowchart của Create*

- Wait(char *name): Giảm giá trị semaphore

  B1: Kiểm tra semaphore với name đã tồn tại chưa. Nếu không thì trả về -1

  B2: Nếu tồn tại thì down semaphore đó bằng phương thức Wait của Sem

  B3: Trả về 0



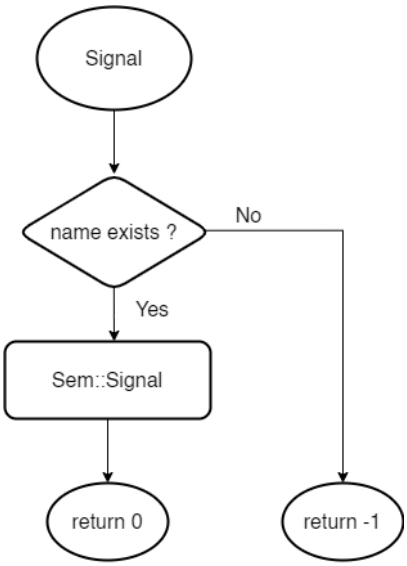

*Flowchart của Wait*

- Signal(char *name): Tăng giá trị semaphore

  B1: Kiểm tra semaphore với name đã tồn tại chưa. Nếu khôn thì trả về -1

  B2: Nếu tồn tại thì up semaphore đó bằng phương thức Signal của Sem

  B3: Trả về 0




*Flowchart của Signal*

Bên cạnh đó còn những hàm construct, destruct của class STable dể tạo ra và xóa đối tượng Bitmap bm và bảng semaphore table. Hàm FindFreeSlot để gọi gián tiếp đến phương thức FindAndSet của Bitmap.

Lớp STable có một đối tượng semTab được khai báo toàn cục (khai báo nằm trong file main.h).

# 2. Cài đặt System Call

## 2.1 SC_CreateFile: int CreateFile(char* name)

- SC_CreateFile sẽ sử dụng NachOS FileSystem Object để tạo một file rỗng.
- Mô tả:
  - Input: tên file cần tạo
  - Output: trả về 0 nếu thành công và -1 nếu có lỗi
  - Chức năng: Tạo ra 1 file mới
- Cài đặt: Đọc tham số name từ thanh ghi r4. Dùng User2System để chuyển filename từ user space sang system space. Nếu tạo file thành công, ta ghi kết quả 0 vào thanh ghi r2. Ngược lại, nếu xảy ra bất kỳ lỗi nào trong lúc tạo file, ta ghi kết quả -1 vào thanh ghi r2 và kết thúc chương trình ngay tại đó.

## 2.2 SC_Open: OpenFileID Open(char* name, int type)

- SC_Open có thể dùng để mở 2 loại file, file chỉ đọc và file đọc và ghi. Mỗi tiến trình được cấp một bảng mô tả file là openTable với kích thước cố định. Ở đây, ta cài đặt openTable cho lưu được đặc tả của 10 files, trong đó 2 phần tử đầu mặc định lần lượt là console input và console output.
- Mô tả:
  - Input: tên file và loại file tương ứng dùng để mở
  - Output: trả về OpenFileID tương ứng của file đó nếu thành công và -1 nếu có lỗi
  - Chức năng: Mở một file với loại cho phép
- Cài đặt: Đọc các tham số name và type tương ứng lần lượt từ thanh ghi r4 và r5. Ta tìm freeSlot là chỗ trống còn dư trong openTable để xem có thể tiếp tục mở file được hay không. Nếu còn thì ta xét tiếp các trường hợp type để ghi kết quả tương ứng vào thanh ghi r2 như sau:

| Trường hợp | type | r2 |
|---|---|---|
| đọc và ghi | 0 | freeSlot |
| chỉ đọc | 1 | freeSlot |
| stdin | 2 | 0 |
| stdout | 3 | 1 |

Ngoài các trường hợp trên, nếu có lỗi xảy ra trong quá trình mở file thì ta ghi kết quả -1 vào thanh ghi r2 và kết thúc chương trình ngay tại đó.

## 2.3 SC_Close: int Close(OpenFileID id)

- SC_Close được dùng để đóng một file được mở trước đó
- Mô tả:
  - Input: id của file đó trong bảng openTable
  - Output: 0 nếu thành công và -1 nếu bị lỗi
  - Chức năng: đóng 1 file
- Cài đặt: Đọc tham số fileID từ thanh ghi r4. Nếu fileID ∈ [1,10], ta sẽ hủy vùng nhớ của nó trong bảng openTable và ghi kết quả 0 vào thanh ghi r2. Ngược lại, nếu xảy ra bất kỳ lỗi nào trong lúc tạo file, ta ghi kết quả -1 vào thanh ghi r2 và kết thúc chương trình ngay tại đó.

## 2.4 SC_Read: int Read(char* buffer, int size, OpenFileID id)

- SC_Read sử dụng hàm Read được cài đặt trong lớp SynchConsoleInput để hỗ trợ việc đọc 1 chuỗi ký tự. Có sự khác biệt trong việc đọc giữa Console IO (OpenFileID 0,1) và file.
  - Đọc từ console input sẽ sử dụng dữ liệu ASCII
  - Đọc từ file sẽ dùng các lớp được cung cấp sẵn trong file system
- Mô tả:
  - Input: vùng nhớ cấp phát để chứa các ký tự được đọc từ file, số lượng byte cho việc đọc, id của file
  - Output: số lượng byte đọc được nếu thành công, -1 nếu lỗi và -2 nếu ở vị trí cuối file EOF
  - Chức năng: đọc dữ liệu từ file cho trước
- Cài đặt: Đọc các tham số buffer, size và id lần lượt từ thanh ghi r4, r5 và r6. Ta xét tiếp các trường hợp type để ghi kết quả tương ứng vào thanh ghi r2 như sau, trong đó count là số lượng ký tự thật sự đọc được:

| Trường hợp | type | r2 |
|---|---|---|
| đọc và ghi | 0 $(count > 0)$ | count |
| | 0 $(count \leq 0)$ | -2 |
| chỉ đọc | 1 $(count > 0)$ | count |
| | 1 $(count \leq 0)$ | -2 |
| stdin | 2 | count |
| stdout | 3 | -1 |

Ngoài các trường hợp trên, nếu có lỗi xảy ra trong quá trình đọc file thì ta ghi kết quả -1 vào thanh ghi r2 và kết thúc chương trình ngay tại đó.

## 2.5 SC_Write: int Write(char* buffer, int size, OpenFileID id)

- SC_Write sử dụng hàm Write được cài đặt trong lớp SynchConsoleOutput để hỗ trợ việc in 1 chuỗi ký tự. Có sự khác biệt trong việc ghi giữa Console IO (OpenFileID 0,1) và file.
  - Ghi vào console ouput sẽ sử dụng dữ liệu ASCII
  - Ghi vào file sẽ dùng các lớp được cung cấp sẵn trong file system
- Mô tả:
  - Input: vùng nhớ chứa các ký tự để ghi vào file, số lượng byte cho việc ghi, id của file
  - Output: số lượng byte đọc được nếu thành công và -1 nếu lỗi
  - Chức năng: ghi dữ liệu vào file cho trước
- Cài đặt: Đọc các tham số buffer, size và id lần lượt từ thanh ghi r4, r5 và r6. Ta xét tiếp các trường hợp type để ghi kết quả tương ứng vào thanh ghi r2 như sau, trong đó count là số lượng ký tự trong buffer:

| Trường hợp | type | r2 |
|---|---|---|
| đọc và ghi | 0 | count |
| chỉ đọc | 1 | -1 |
| stdin | 2 | -1 |
| stdout | 3 | count |

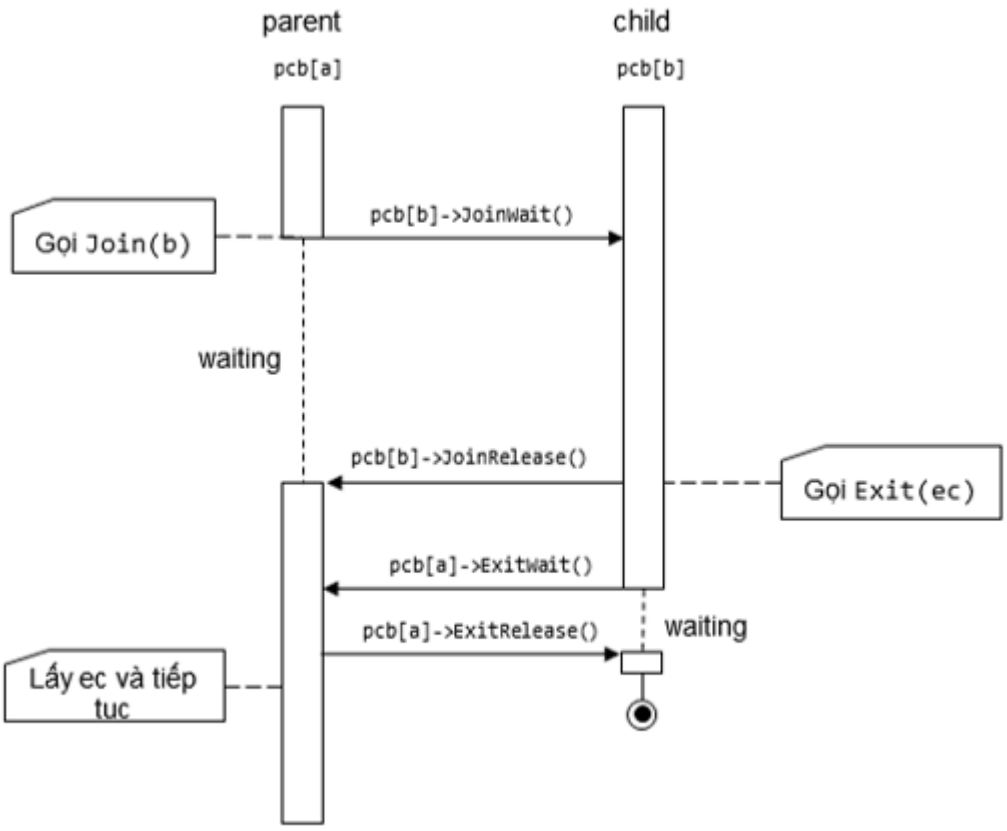
Ngoài các trường hợp trên, nếu có lỗi xảy ra trong quá trình ghi file thì ta ghi kết quả -1 vào thanh ghi r2 và kết thúc chương trình ngay tại đó.

## 2.6 SC_Exec: SpaceID Exec(char* fileName)

- System call Exec sử dụng lớp PTable để khởi động một tiến trình lưu trong fileName
- Mô tả:
  - Input: Tên chương trình cần khởi động
  - Output: PID của tiến trình nếu khởi động thành công, -1 nếu thất bại
  - Chức năng: gọi thực thi chương trình fileName
- Cài đặt: Ta đọc địa chỉ tên chương trình từ thanh ghi 4, sau đó copy tên chương trình vào kernel space. Tiếp theo, ta gọi phương thức ExecUpdate của biến toàn cục pTab để thêm chương trình này vào process table (nếu thành công). Ta trả về kết quả của phương thức ExecUpdate này về cho user process.

## 2.7 SC_Join: int Join(SpaceID id)

- System call Join dùng để join tiến trình cha vào tiến trình con, có nghĩa là block tiến trình cha cho đến khi tiến trình con kết thúc. System call này cần sự phối hợp với system call Exit (xem **Error! Reference source not found.**)
- Mô tả:
  - Input: PID của tiến trình con
  - Output: PID của tiến trình con, hoặc -1 nếu tiến trình cha join vào tiến trình không phải tiến trình con của mình
  - Chức năng: block tiến trình gọi Join (tiến trình cha) cho đến khi tiến trình con (tiến trình được join) kết thúc
- Cài đặt: Ta đọc PID của tiến trình con từ thanh ghi 4, sau đó gọi phương thức JoinUpdate của biến toàn cục pTab. Trả về cho user process kết quả của phương thức này.




*Sơ đồ phối hợp giữa hai system call Join và Exit*

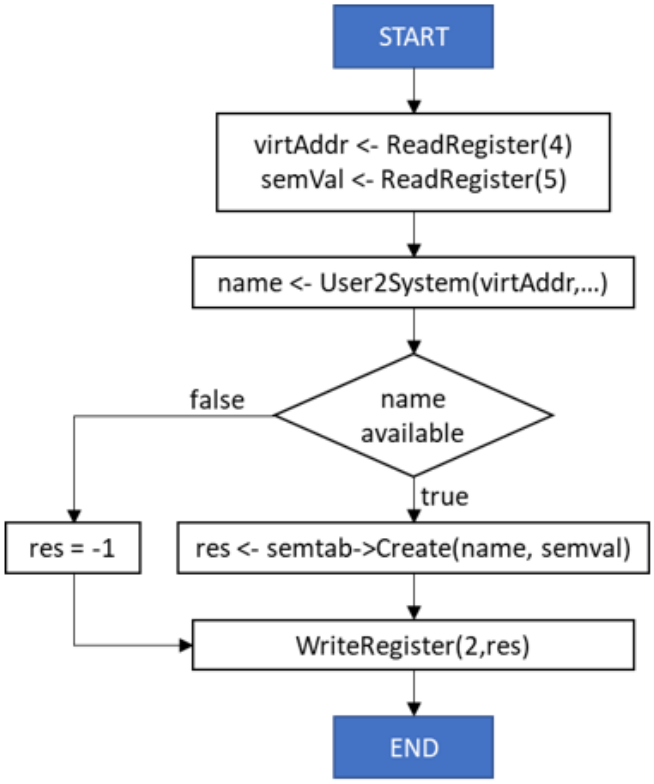
## 2.8 SC_Exit: void Exit(int exitCode)

- System call Exit dùng để thoát tiến trình hiện tại và trả về exit code.
- Mô tả:

  - Input: exit code của chương trình muốn trả về
  - Output: không có
  - Chức năng: dừng tiến trình hiện tại, gỡ block cho bất kỳ tiến trình nào đang join vào tiến trình này (nếu có), hoặc thoát NachOS nếu tiến trình này là main
- Cài đặt: Ta đọc exit code của tiến trình từ thanh ghi 4, sau đó gọi phương thức ExitUpdate của biến toàn cục pTab. Ta không trả về giá trị cho user.

### 2.9 SC_CreateSemaphore: int CreateSemaphore(char* name, int semval)

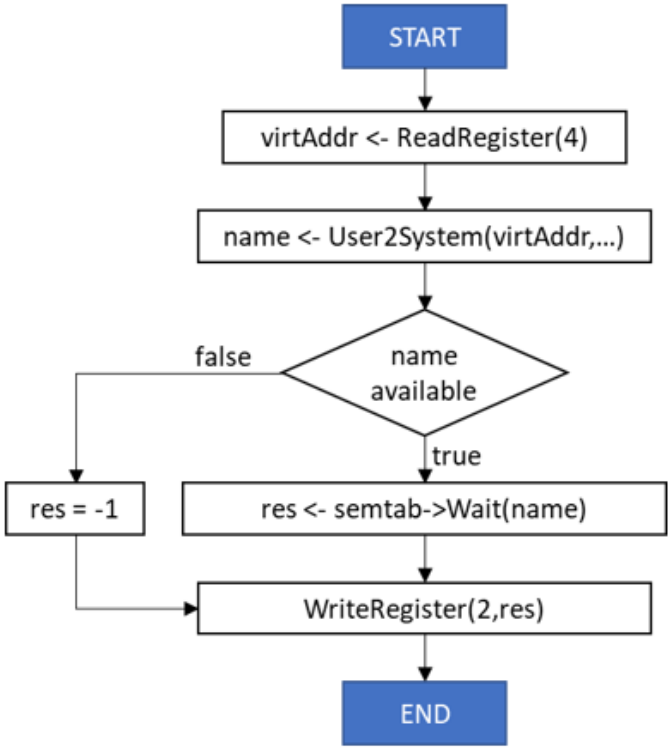
- Mục đích: Tạo một semaphore mới bằng phương thức CreateSemaphore của STable
- Mô tả:
  - Input: tên của Semaphore muốn tạo và giá trị ban đầu
  - Output: 0 nếu thành công và -1 nếu thất bại
  - Chức năng: Tạo một semaphore trong semTab
- Cài đặt: Đọc địa chỉ vùng nhớ tên và giá trị ban đầu của semaphore lần lượt ở thanh ghi 4 và 5. Load tên của Semaphore vào biến name. Kiểm tra biến name có null không. Nếu không thì tạo Semaphore mới bằng Create của semTab (biến global). Kiểm tra kết quả trả về, thành công thì trả về 0. Bất kỳ bước nào ở trên thất bại thì trả về -1. Xóa buffer name và kết quả trả về ghi vào thanh ghi 2.




*Mô hình cài đặt CreateSemaphore*

## 2.10 SC_Wait: int Wait(char* name)

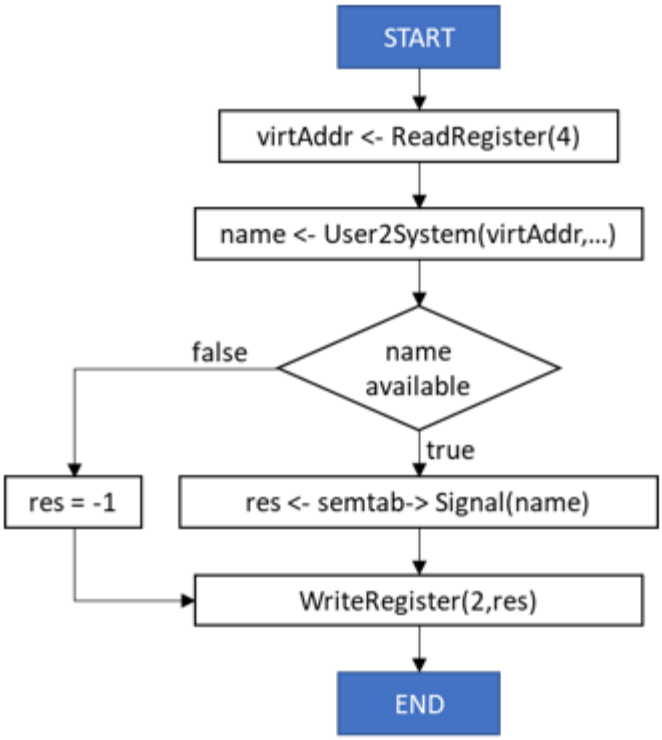
- Mục đích: Down semaphore bằng phương thức Wait của STable
- Mô tả:
  - Input: tên của Semaphore
  - Output: 0 nếu thành công và -1 nếu thất bại
  - Chức năng: Kiểm tra tồn tại và down semaphore
- Cài đặt: Đọc địa chỉ vùng nhớ tên của semaphore ở thanh ghi 4. Load tên của Semaphore vào biến name. Kiểm tra biến name có null không. Nếu không thì down Semaphore bằng Wait của semTab (biến global). Kiểm tra kết quả trả về, thành công thì trả về 0. Bất kỳ bước nào ở trên thất bại thì trả về -1. Xóa buffer name và kết quả trả về ghi vào thanh ghi 2.




*Mô hình cài đặt Wait*

## 2.11 SC_Signal: int Signal(char* name)

- Mục đích: Up semaphore bằng phương thức Signal của STable
- Mô tả:
  - Input:
  - Output: 0 nếu thành công và -1 nếu thất bại
  - Chức năng: Kiểm tra tồn tại và up semaphore
- Cài đặt: Đọc địa chỉ vùng nhớ tên của semaphore ở thanh ghi 4. Load tên của Semaphore vào biến name. Kiểm tra biến name có null không. Nếu không thì up Semaphore bằng Signal của semTab (biến global). Kiểm tra kết quả trả về, thành công thì trả về 0. Bất kỳ bước nào ở trên thất bại thì trả về -1. Xóa buffer name và kết quả trả về ghi vào thanh ghi 2.



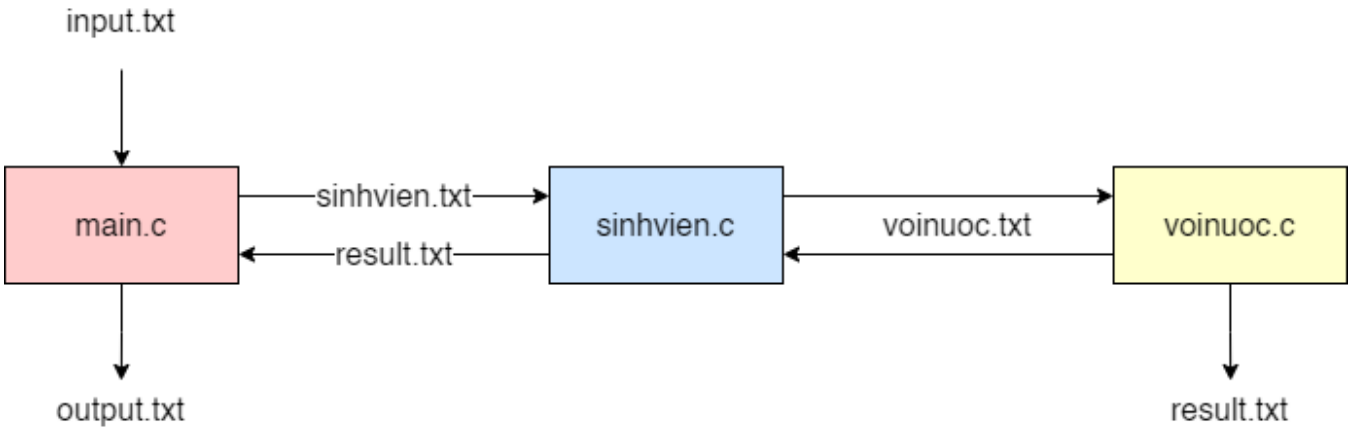

*Mô hình cài đặt Signal*

## 3. Chương trình minh họa

Ứng dụng "Thống kê sử dụng máy nóng lạnh"

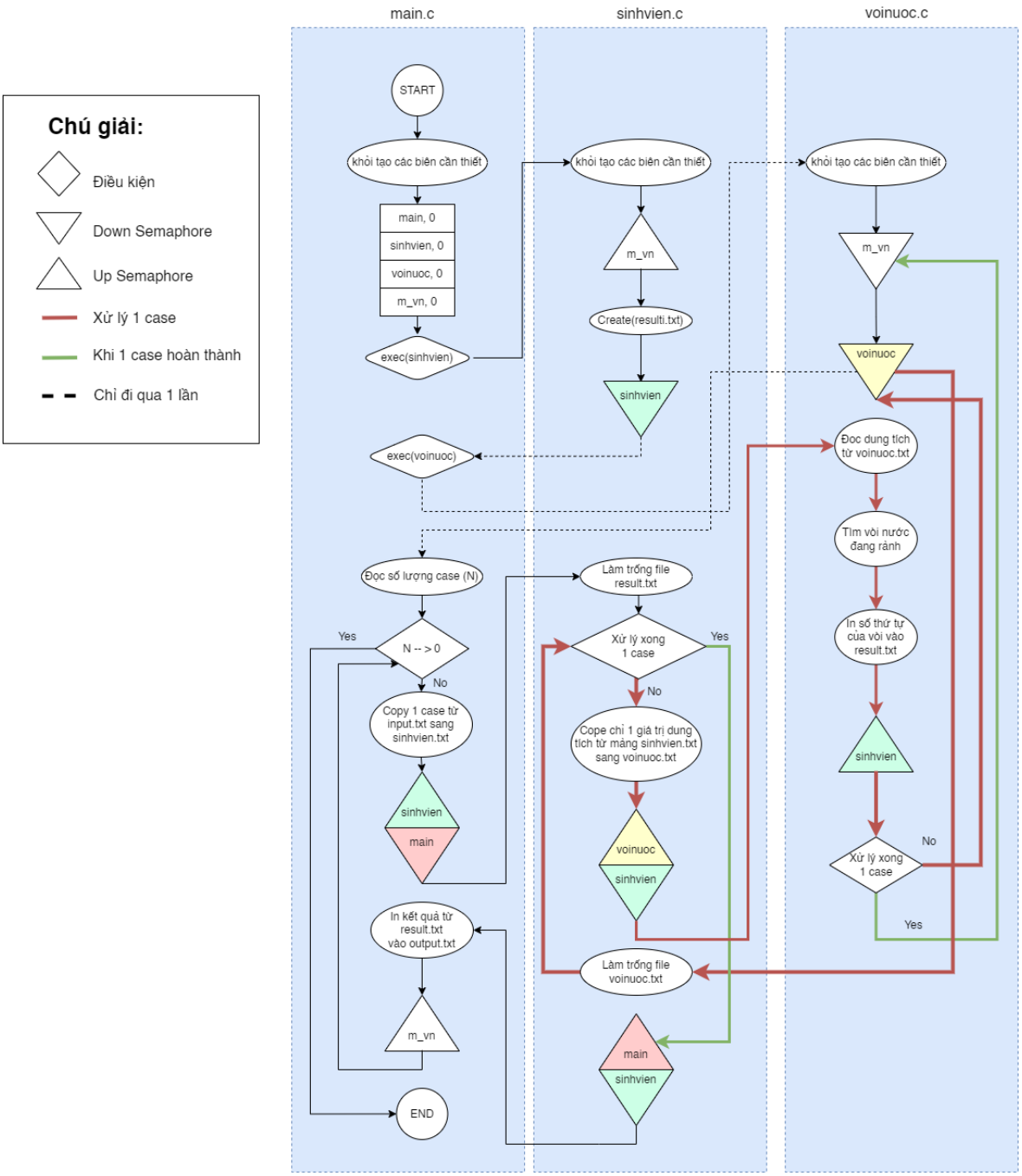
### 3.1 Cài đặt

Mô hình:



Cách tiếp cận bài toán của nhóm là ủy quyền. Trong đó main.c dùng để điều khiển có nhiệm vụ đưa từng case (dòng) vào sinhvien.txt; sinhvien.c có nhiệm vụ tạo sẵn file trống là result.txt và đưa từng giá trị dung tích cho voinuoc xử lý; voinuoc.c sẽ quản lý các vòi nước, mỗi khi nhận giá trị dung tích khác không thì sẽ tìm vòi đang rảnh và ghi vào result.txt. Sau khi xử lý một case (dòng) thì main.c sẽ chuyển kết quả trong result.txt vào output.txt.

Giao tiếp giữa các tiến trình:

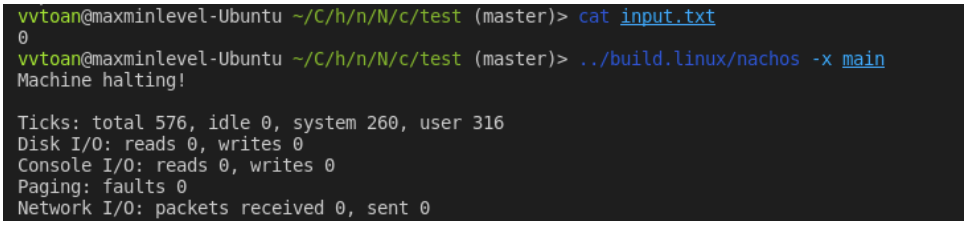


## 3.2 Demo

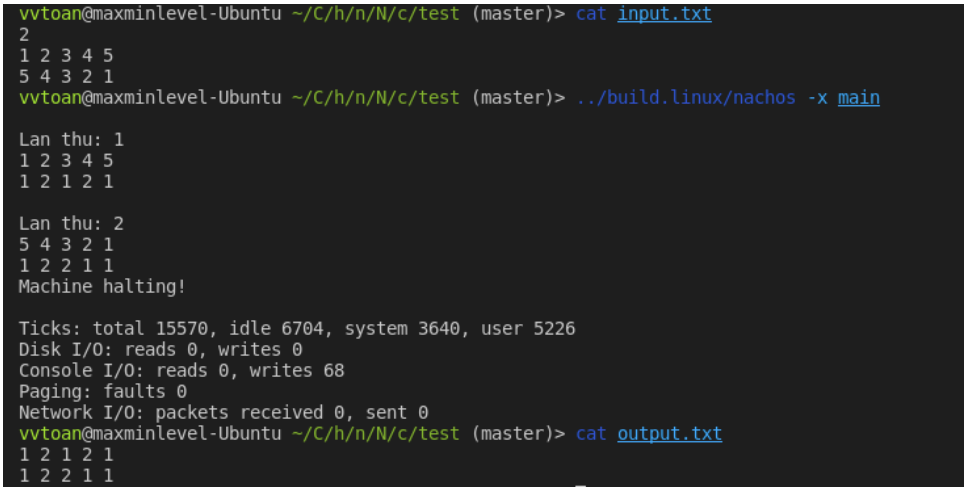
- File input.txt:
  - Dòng đầu là số nguyên dương N cho biết số thời điểm được xét.
  - N dòng tiếp theo, mỗi dòng chứa n số nguyên dương cho biết có n Sinh viên đến uống nước và giá trị của mỗi số nguyên dương cho biết dung tích của chai nước mà sinh viên mang theo.
- File output.txt:
  - N dòng, mỗi dòng chứa thứ tự vòi nước sinh viên dùng tương ứng
- Hướng dẫn sử dụng máy nóng lạnh:
  - Mở Terminal ở thư mục test
  - Chạy lệnh sau: # ../build.linux/nachos -x main

Kết quả chương trình:

- 0 case, 2 vòi nước



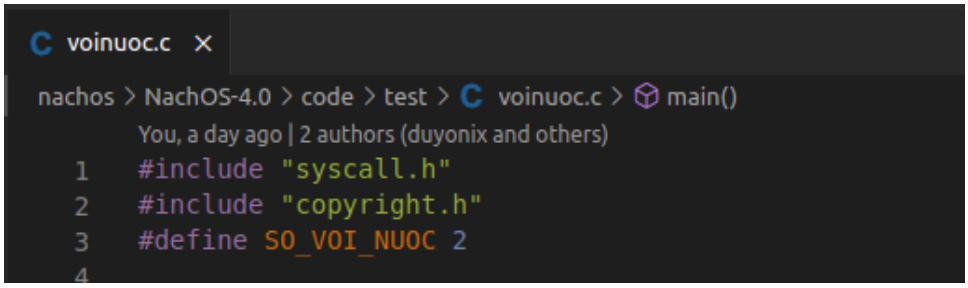

- 2 case, 2 vòi nước




- 4 case, 2 vòi nước, trong đó có case dung tích bằng 0

```
vvtoan@maxminlevel-Ubuntu ~/C/h/n/N/c/test (master)> cat input.txt
4
1 2 3 4 5
5 4 3 2 1
1 0 0
1 1 1
vvtoan@maxminlevel-Ubuntu ~/C/h/n/N/c/test (master)> ../build.linux/nachos -x main

Lan thu: 1
1 2 3 4 5
1 2 1 2 1

Lan thu: 2
5 4 3 2 1
1 2 2 1 1

Lan thu: 3
1 0 0
1

Lan thu: 4
1 1 1
1 2 1
Machine halting!

Ticks: total 25556, idle 11406, system 5970, user 8180
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 116
Paging: faults 0
Network I/O: packets received 0, sent 0
vvtoan@maxminlevel-Ubuntu ~/C/h/n/N/c/test (master)> cat output.txt
1 2 1 2 1
1 2 2 1 1
1
1 2 1
```

Kết quả làm thêm được, có thể thay đổi số lượng vòi nước bằng cách thay đổi giá trị của SO_VOI_NUOC trong voinuoc.c




Kết quả chương trình với 4 case, 3 vòi nước, trong đó có case dung tích bằng 0

```
vvtoan@maxminlevel-Ubuntu ~/C/h/n/N/c/test (master)> cat input.txt
4
1 2 3 4 5
5 4 3 2 1
1 0 0
1 1 1
vvtoan@maxminlevel-Ubuntu ~/C/h/n/N/c/test (master)> ../build.linux/nachos -x main

Lan thu: 1
1 2 3 4 5
1 2 3 1 2

Lan thu: 2
5 4 3 2 1
1 2 3 3 2

Lan thu: 3
1 0 0
1

Lan thu: 4
1 1 1
1 2 3
Machine halting!

Ticks: total 26136, idle 11431, system 6030, user 8675
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 116
Paging: faults 0
Network I/O: packets received 0, sent 0
vvtoan@maxminlevel-Ubuntu ~/C/h/n/N/c/test (master)> cat output.txt
1 2 3 1 2
1 2 3 3 2
1
1 2 3
```

# 4. Đánh giá

## 4.1 Những vấn đề đã làm được

- Hiểu và cài đặt thành công các System Call cũng như xây dựng các chương trình để chạy kiểm thử các System Call tương ứng đó
- Từ các tài liệu và code mẫu dựa trên NachOS 3.4, đã đọc hiểu cũng như triển khai cài đặt lại cho phù hợp với NachOS 4.0
- Giải quyết được bài toán "Thống kê sử dụng máy nóng lạnh" và chạy thành công. Mở rộng bài toán với n vòi nước.
- Qua đồ án đã giúp nhóm hiểu rõ hơn lý thuyết về đa chương, lên lịch và đồng bộ hóa

## 4.2 Những vấn đề chưa làm được

- Nhóm đã hoàn thành được các yêu cầu giao trong đồ án, tuy nhiên vẫn phải dựa trên các nguồn tài liệu tham khảo sẵn có từ NachOS 3.4 để cài đặt lại.
- Vẫn chưa hiểu rõ sâu về cài đặt gốc bên trong NachOS, đa phần chỉ hiểu được cách hoạt động từ phạm vi kernel trở đi.
- Với input từ 5 case trở lên sẽ bị lỗi full OpenTable.

# 5. Tài liệu tham khảo

[1] – Sách Hệ điều hành, Trần Trung Dũng, Phạm Tuấn Sơn, Nhà xuất bản Khoa học và Kỹ thuật.

[2] – Video hướng dẫn:

https://www.youtube.com/watch?v=t0jtY1C129s&list=PLRgTVtca98hUgCN2_2vzsAAXPiTFbvHpO

[3] – NachOS 3.4 Project, Nguyễn Thành Chung, https://github.com/nguyenthanhchungfit/Nachos-Programing-HCMUS

[4] – NachOS-Lab-2, https://github.com/hiendang7613/OS-Nachos-Lab2