

**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**

**KHOA CÔNG NGHỆ THÔNG TIN**

**BỘ MÔN MẠNG MÁY TÍNH - VIỄN THÔNG**



## **BÀI BÁO CÁO**

### **Project 3 – Đa chương và đồng bộ hóa**

**Môn học: Hệ điều hành**

**Lớp: 22CLC06**

**Giáo viên hướng dẫn: Ths. Lê Viết Long**

**Sinh viên thực hiện:**

- + 22127022 - Võ Hoàng Anh**
- + 22127154 - Nguyễn Gia Huy**
- + 22127210 - Phạm Anh Khôi**
- + 22127413 - Phạm Hoàng Tiên**

# Mục lục

I. Thông tin nhóm.....	3
II. Bảng phân công công việc.....	3
III. Đánh giá mức độ hoàn thành.....	4
IV. Các bước thực hiện.....	5
A. Tóm tắt lí thuyết.....	5
1. Đa chương - Multiprogram.....	5
2. Tiến trình - Process.....	6
3. Tiểu trình - Thread.....	8
4. Đa tiểu trình - Multithread.....	8
5. Lập lịch và phân công - Scheduling & Dispatching.....	9
6. Đồng bộ hóa - Synchronization.....	11
B. Thiết kế và cài đặt.....	13
1. Chú thích.....	13
2. Hiểu các lớp có sẵn.....	13
a. Lớp BitMap.....	13
b. Lớp Semaphore.....	14
c. Lớp Thread.....	15
d. Lớp AddrSpace.....	19
3. Thiết kế các lớp mới.....	21
a. Lớp PCB.....	21
b. Lớp PTable.....	22
c. Lớp STable.....	24
4. Cài đặt các System Call.....	25
a. Exec, Join, Exit.....	25
b. CreateSemaphore, Down, Up.....	27
C. Viết chương trình minh họa.....	29
1. Ping - Pong.....	29
2. Thống kê sử dụng máy quét.....	30
V. Hình ảnh demo chương trình.....	31
A. Chương trình PING-PONG.....	31
B. Chương trình máy quét.....	31
VI. Tài liệu tham khảo.....	32

## I. Thông tin nhóm

- Môn: Hệ điều hành
- Lớp: 22CLC06
- Project 3 - Đa chương và đồng bộ hóa
- Thành viên nhóm
- + 22127022 - Võ Hoàng Anh
- + 22127154 - Nguyễn Gia Huy
- + 22127210 - Phạm Anh Khôi
- + 22127413 - Phạm Hoàng Tiên

## II. Bảng phân công công việc

MSSV	Họ và tên	Công việc
22127022	Võ Hoàng Anh	Tìm hiểu và điều lớp <b>BitMap</b> , <b>Semaphore</b> , <b>Addrspace</b> , <b>Thread</b> , viết báo cáo
22127154	Nguyễn Gia Huy	Thêm lớp <b>PCB</b> , <b>PTable</b> , System call <b>Exec</b> , <b>Join</b> , <b>Exit</b> , viết chương trình máy quét, viết báo cáo
22127210	Phạm Anh Khôi	Thêm lớp <b>STable</b> , System call <b>CreateSemaphore</b> , <b>Down</b> , <b>Up</b> , viết báo cáo
22127413	Phạm Hoàng Tiên	Viết chương trình PINGPONG, viết báo cáo

**III. Đánh giá mức độ hoàn thành**

STT	Tên công việc	Mức độ hoàn thành
1	Tìm hiểu lớp <b>BitMap, Semaphore</b>	Hoàn thành
2	Tìm hiểu và điều chỉnh lớp <b>AddrSpace, Thread</b>	Hoàn thành
3	Thiết kế lớp <b>PCB, PTable</b>	Hoàn thành
4	Thiết kế lớp <b>STable</b>	Hoàn thành
5	Viết System Call <b>Exec, Join, Exit</b>	Hoàn thành
6	Viết System Call <b>CreateSemaphore, Down, Up</b>	Hoàn thành
7	Viết chương trình PING-PONG	Hoàn thành
8	Viết chương trình thống kê sử dụng máy quét	Hoàn thành

## IV. Các bước thực hiện

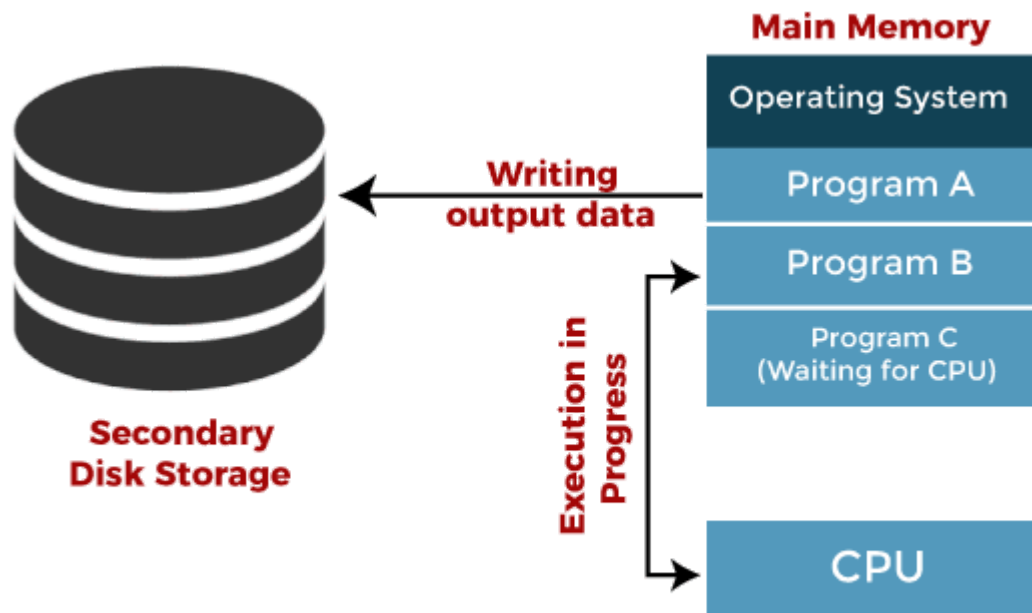
### A. Tóm tắt lý thuyết

#### 1. Đa chương - Multiprogram

*Đa chương (multiprogram)* là một kỹ thuật trong lập trình máy tính và quản lý hệ thống, cho phép máy tính thực thi nhiều chương trình đồng thời.

Thay vì chờ đợi một chương trình kết thúc trước khi bắt đầu chương trình tiếp theo, hệ thống đa chương cho phép máy tính chạy nhiều chương trình song song.

Trong khi một chương trình đang chờ tài nguyên hoặc I/O, hệ thống có thể chuyển sang thực thi một chương trình khác, tận dụng tài nguyên máy tính một cách hiệu quả hơn.



**Jobs in multiprogramming system**

**Lưu ý:** Đa chương khác với đa nhiệm (Multitasking trong dạng hệ điều hành chia sẻ thời gian)

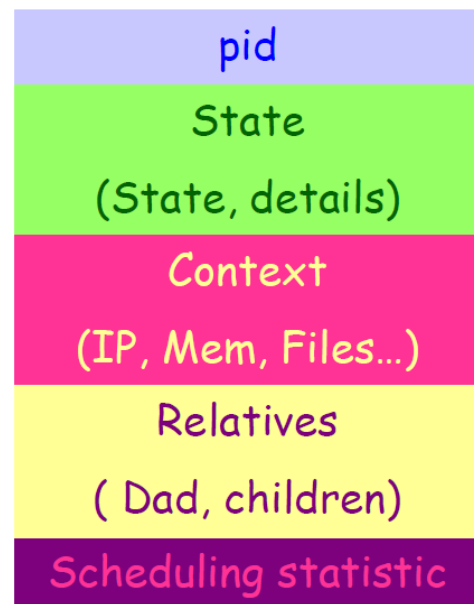
## 2. Tiến trình - Process

Một *tiến trình (process)* là một *chương trình máy tính (program)* đang được thực thi

Một tiến trình cần sử dụng các tài nguyên: CPU, bộ nhớ, tập tin, thiết bị nhập xuất để hoàn tất công việc của nó

Mỗi tiến trình có một khối quản lý tiến trình (*PCB - Process Control Block*) để quản lý các thông tin của nó như hình minh họa bên dưới

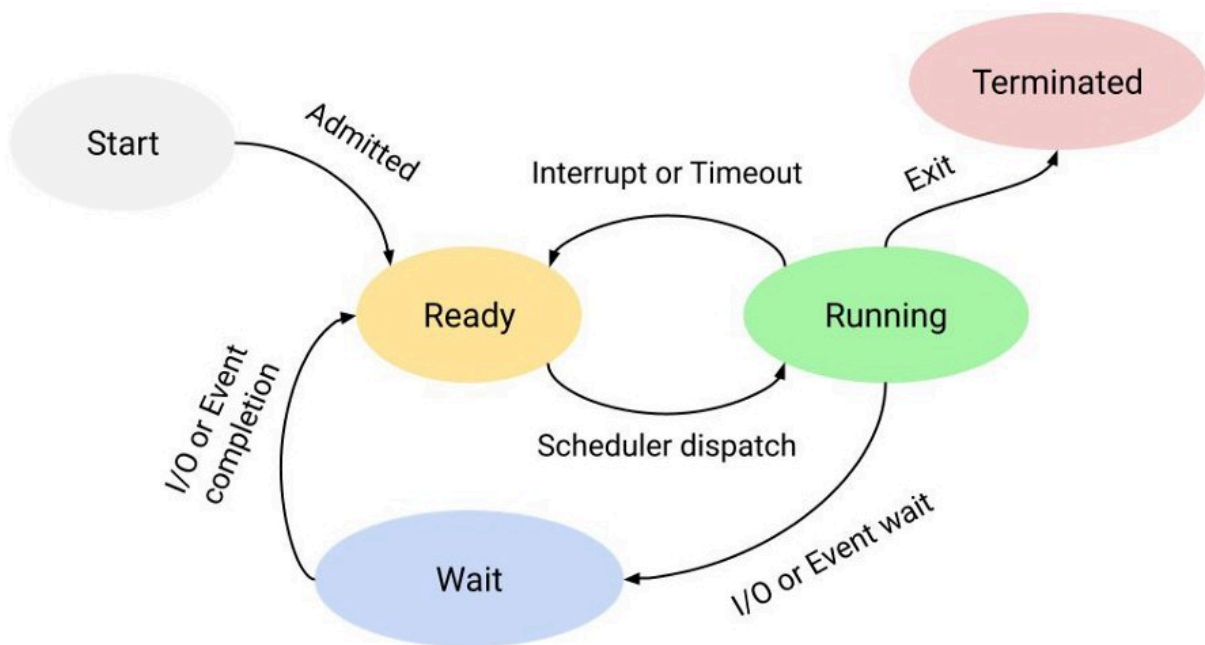
- Định danh (Process ID)
- Trạng thái tiến trình
- Ngưỡng cảnh tiến trình
  - Trạng thái CPU
  - Bộ xử lý (cho máy nhiều CPU)
  - Bộ nhớ chính
  - Tài nguyên sử dụng /tạo lập
- Thông tin giao tiếp
  - Tiến trình cha, tiến trình con
  - Độ ưu tiên
- Thông tin thống kê



Process control Block - PCB

Hoạt động của tiến trình có những trạng thái như sau:

- ❖ *Start (Bắt đầu)*: Tiến trình được tạo ra hoặc khởi động bởi hệ điều hành. Trong trạng thái này, tiến trình chờ đợi để được chuyển sang trạng thái sẵn sàng.
- ❖ *Ready (Sẵn sàng)*: Tiến trình đã được tạo ra và sẵn sàng để thực thi, nhưng chưa được chọn để chạy trên CPU. Trong trạng thái này, tiến trình đợi CPU để thực thi.
- ❖ *Running (Đang chạy)*: Tiến trình được chọn để thực thi trên CPU và đang thực hiện các lệnh của nó.
- ❖ *Wait (Chờ đợi)*: Tiến trình chờ đợi để thực hiện một sự kiện nào đó xảy ra trước khi tiếp tục thực thi. Điều này có thể là chờ dữ liệu từ I/O, hoặc chờ tài nguyên hệ thống khác.
- ❖ *Terminated (Kết thúc)*: Tiến trình đã hoàn thành thực thi và được hệ điều hành kết thúc. Trong trạng thái này, tài nguyên đã được giải phóng và không còn được sử dụng.



### **3. Tiểu trình - Thread**

Một tiểu trình là một dòng xử lý trong một tiến trình

Tiểu trình là tác vụ cơ sở độc lập nhìn từ CPU, nó bao gồm định danh tiểu trình, một con trỏ lệnh, một tập thanh ghi và stack

Mỗi tiểu trình có một khối quản lý tiểu trình (TCB - Thread Control Block) để chứa các thông tin riêng của nó như:

- ❖ ID của tiểu trình
- ❖ Không gian lưu các thanh ghi
- ❖ Con trỏ tới vị trí xác định trong ngăn xếp
- ❖ Trạng thái của tiểu trình

Và thông tin chia sẻ giữa các tiểu trình trong một tiến trình

- ❖ Các biến toàn cục
- ❖ Các tài nguyên sử dụng như tập tin, ...
- ❖ Các tiến trình con
- ❖ Thông tin thống kê

### **4. Đa tiểu trình - Multithread**

Mỗi tiến trình luôn có một tiểu trình chính (dòng xử lý cho hàm main())

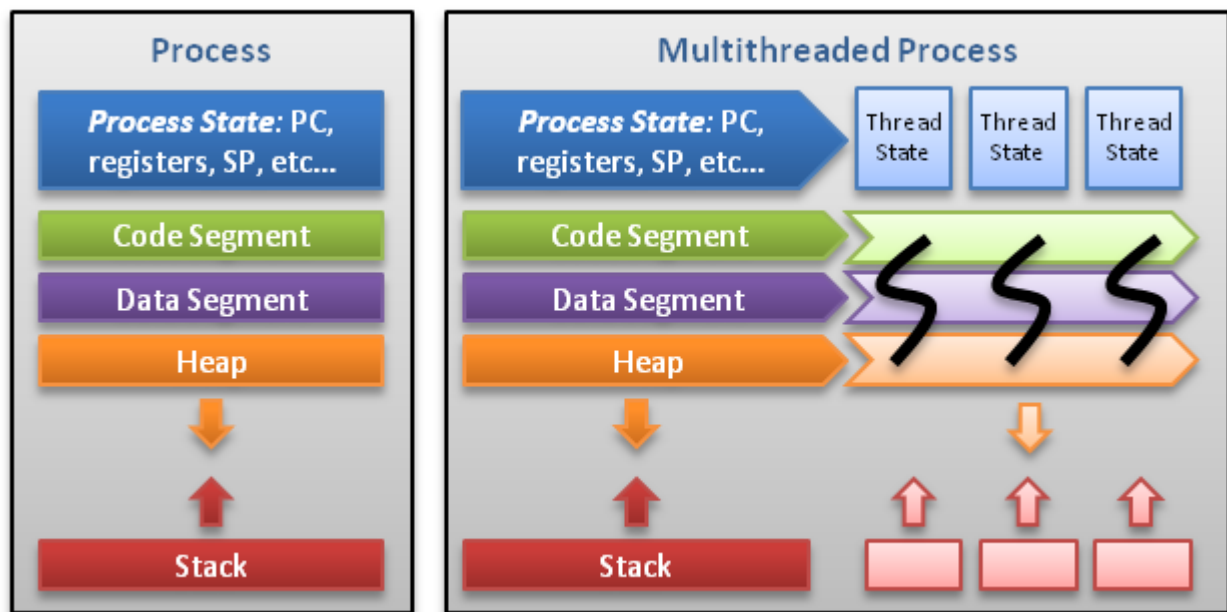
Ngoài tiểu trình chính, tiến trình còn có thể có nhiều tiểu trình con khác, là việc thực hiện nhiều tác vụ trong cùng một thời gian

Các tiểu trình trong một tiến trình có thể cùng chia sẻ không gian vùng code, data và có vùng stack riêng



Rõ ràng một tiến trình gồm đa tiểu trình giúp giải quyết được bài toán đa nhiệm trong một chương trình hiện đại

Hình vẽ dưới đây mô tả sự khác nhau giữa tiến trình đơn tiểu trình và tiến trình đa tiểu trình



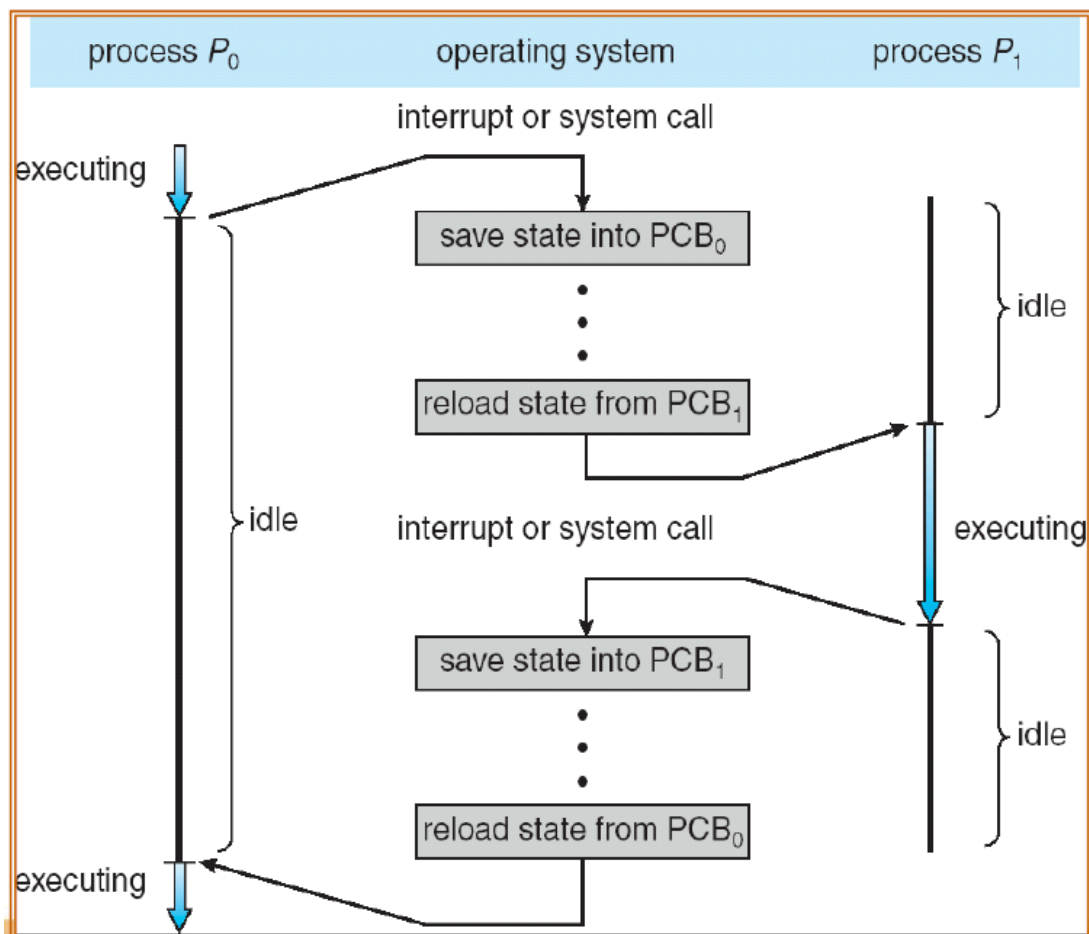
## 5. Lập lịch và phân công - Scheduling & Dispatching

- ❖ **Scheduler** là một thành phần của hệ điều hành, chịu trách nhiệm quản lý việc lập lịch (scheduling) các tiến trình và luồng trên CPU, nó quyết định xem tiến trình nào sẽ được thực thi tiếp theo dựa trên các tiêu chí như độ ưu tiên, thời gian chờ đợi và các chiến lược khác. Các chức năng chính:
  - + Quản lý hàng đợi các tiến trình
  - + Lập lịch thời gian cho các tiến trình dựa trên các thuật toán lập lịch
  - + Quyết định xem tiến trình nào sẽ được chạy trên CPU và trong bao lâu.

❖ **Dispatcher** là thành phần khác của hệ điều hành, có nhiệm vụ thực hiện việc chuyển đổi giữa các tiến trình khi một tiến trình mới được chọn để thực thi trên CPU. Các chức năng chính:

- + Chuẩn bị môi trường thực thi cho tiến trình mới, bao gồm sao chép trạng thái của tiến trình từ bộ nhớ vào CPU
- + Chuyển giao điều khiển từ tiến trình hiện tại sang tiến trình mới được chọn
- + Cập nhật thông tin về tiến trình sau khi thực thi

Dưới đây là hình ảnh minh họa về công việc chuyển đổi ngữ cảnh (Context switching) của Dispatcher



## 6. Đồng bộ hóa - Synchronization

Đồng bộ hóa là quá trình đảm bảo rằng các tiến trình, luồng hoặc tác vụ hoạt động song song truy cập và thay đổi dữ liệu cùng một lúc một cách an toàn và chính xác

Một phương pháp giải quyết tốt bài toán đồng bộ hoá cần 4 điều kiện sau:

- ❖ Mutual Exclusion: Không có hai tiến trình cùng ở trong miền găng cùng lúc.
- ❖ Progress: Một tiến trình tạm dừng bên ngoài miền găng không được ngăn cản các tiến trình khác vào miền găng
- ❖ Bounded Waiting: Không có tiến trình nào phải chờ vô hạn để được vào miền găng.
- ❖ Không có giả thiết nào đặt ra cho sự liên hệ về tốc độ của các tiến trình, cũng như về số lượng bộ xử lý trong hệ thống.

Có 2 nhóm giải pháp kinh điển gồm

- ❖ Nhóm giải pháp Busy Waiting bao gồm:

- Các giải pháp phần mềm:

- Sử dụng các biến cờ hiệu.
- Sử dụng việc kiểm tra luân phiên.
- Giải pháp của Peterson.

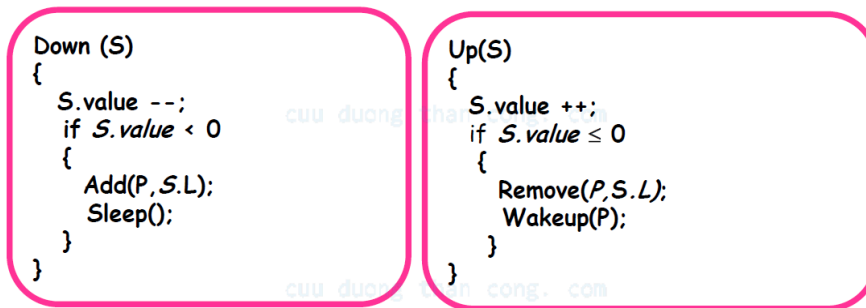
- Các giải pháp phần cứng

- Cấm ngắt.
- Test&Set lock Instruction.

- ❖ Nhóm giải pháp Sleep & Wakeup bao gồm:

- Semaphore.
- Monitor.
- Send & Receive

Trong đồ án lần này, chúng ta sử dụng Semaphore để đồng bộ hóa các tiến trình



Ứng dụng của Semaphore vào bài toán đồng bộ hóa Một Semaphore s có cấu trúc như sau:

```

Struct Semaphore {
    Int value; // giá trị bên trong của semaphore
    Process* L; // danh sách các tiến trình đang bị block đợi
                // semaphore nhận giá trị dương
};

```

Các đặc tính của Semaphore:

- + Có 1 giá trị
- + Chỉ được thao tác bởi 2 primitives: **Down(s)** và **Up(s)**
- + Các primitive **Down(s)** và **Up(s)** được thực hiện không thể phân chia

Dưới đây là hình minh họa 2 primitive của Semaphore với sự hỗ trợ từ 2 System Call của hệ điều hành **Sleep()** và **Wakeup()**

- Tổ chức “Độc quyền truy xuất”:

```

Semaphore s = 1
Process P:
    Down(s)
    CS; // Miền găng
    Up(s)

```

- Tổ chức phối hợp hoạt động:



## B. Thiết kế và cài đặt

### 1. Chú thích

Phần này dùng các màu sắc để thể hiện một số tính chất sau

- **Xanh lá**: Một thuộc tính (có thể là tên hoặc đối tượng)
- **Xanh dương**: Một kiểu dữ liệu hoặc một lớp

### 2. Hiểu các lớp có sẵn

#### a. Lớp BitMap

**BitMap** là một kỹ thuật quản lý bộ nhớ trong hệ điều hành, trong đó bộ nhớ được chia thành các đơn vị phân bổ (allocation units), mỗi đơn vị thường tương ứng với một bit trong **BitMap**. Gồm hai loại giá trị là bật hoặc tắt (bằng giá trị 0 hoặc 1). Điều này giúp quản lý bộ nhớ hiệu quả tới từng bit, giúp tìm kiếm các chuỗi 0 (hoặc 1) liên tiếp một cách hiệu quả.

Trong đoạn mã được thiết kế, tại (`./userprog/bitmap.*`), **BitMap** là một lớp định nghĩa một bitmap như một mảng các bit, gồm các hàm để kiểm soát việc thay đổi giá trị, kiểm tra giá trị, xuất ra giá trị, tìm kiếm giá trị. Trong chủ đề NachOS thì được dùng một cách tối ưu cho việc quản lý phân bổ vùng nhớ của khối đĩa và phân trang. Mỗi bit đại diện cho một vùng / khối / trang đang được sử dụng hay không.

Bảng dưới đây mô tả chi tiết một số phần quan trọng của **BitMap**

Thuộc tính - Phương thức	Ý nghĩa
<code>numBits: int</code>	Chỉ số bit được sử dụng trong bitmap
<code>numWords: int</code>	Chỉ số từ được sử dụng trong bitmap (làm tròn lên)

<code>map: unsigned int*</code>	Lưu trữ bit
<code>Mark(int which): void</code>	Đánh dấu bit thứ <code>which</code>
<code>Clear(int which): void</code>	Bỏ đánh dấu bit thứ <code>which</code>
<code>Test(int which): bool</code>	Kiểm tra xem bit thứ <code>which</code> đã được đặt chưa
<code>Find(): int</code>	Trả về số bit đầu tiên chưa được đánh dấu và đặt nó đã sử dụng Nếu tất cả đều đã được đánh dấu thì trả về -1
<code>NumClear(): int</code>	Trả về số bit chưa được đánh dấu

### b. Lớp Semaphore

**Semaphore** là một công cụ đồng bộ hóa được sử dụng để đồng bộ hóa các luồng trong một hệ thống đa luồng. Nó hoạt động dựa trên một giá trị số nguyên không âm, với hai hoạt động chính là tăng và giảm **Semaphore** với điều kiện nhất định (đợi tới khi lớn hơn 0 thì giảm, hoặc tăng giá trị và gọi luồng đang chờ thức dậy). **Semaphore** không cho phép đọc giá trị trực tiếp, để tránh nhận xét sai ngữ cảnh hay mất đồng bộ hoá thời gian khi đọc và viết trong quá trình thực hiện các thao tác tăng giảm **Semaphore**.

Trong đoạn mã được thiết kế, tại (`./threads/synch.*`), **Semaphore** là một lớp định nghĩa gồm các phương thức để thiết kế để hoạt động theo một cách thống nhất (sao cho chúng không bị gián đoạn khi đang thực hiện) và nguyên thủy (sao cho không có hai luồng cùng chạy một thời điểm). Vậy nên không thể thay đổi giá trị **Semaphore** trong khi một luồng nào đó đang chạy.

Bảng dưới đây mô tả chi tiết một số phần quan trọng của **Semaphore**

Thuộc tính - Phương thức	Ý nghĩa
<code>name: char*</code>	Tên của <b>Semaphore</b> để tiện debug
<code>value: int</code>	Giá trị nguyên của <b>Semaphore</b> (luôn không âm) Phần âm có thể dùng để debug
<code>queue: List*</code>	Thông tin các luồng đang chờ
<code>Semaphore(char* name, int value)</code>	Hàm tạo <b>Semaphore</b> với tên phân biệt <code>name</code> và giá trị khởi tạo <code>value</code>
<code>P(): void</code>	Sẽ đợi cho đến khi giá trị <b>Semaphore</b> lớn hơn 0, sau đó giảm giá trị <b>Semaphore</b>
<code>V(): void</code>	Tăng giá trị <b>Semaphore</b> , và nếu có luồng đang chờ trong <code>P()</code> , nó sẽ được thức dậy

### c. Lớp Thread

**Thread** là một đơn vị cơ bản của hệ thống đa luồng, cho phép thực thi các đoạn mã đồng thời. Trong hệ thống đa luồng, mỗi **Thread** đại diện cho một luồng thực thi độc lập, có thể chạy đồng thời với các thread khác. **Thread** giúp cải thiện hiệu suất và độ phản hồi của các ứng dụng bằng cách cho phép chúng thực hiện nhiều tác vụ cùng một lúc.

Trong đoạn mã được thiết kế, tại (`./threads/thread.*`), **Thread** là một lớp định nghĩa các cấu trúc dữ liệu và phương thức cần thiết để quản lý và thực thi các thread. Mỗi thread có một stack riêng biệt để lưu trữ trạng thái thực thi, cũng như một tập hợp các trạng thái (`JUST_CREATED`, `RUNNING`, `READY`, `BLOCKED`) để quản lý việc chạy của nó.

Bảng dưới đây mô tả chi tiết một số phần quan trọng của **Thread**

Phương thức	Ý nghĩa
<code>Thread(char *debugName)</code>	Tạo một <b>Thread</b> với tên <code>debugName</code> . Đặt trạng thái <b>JUST_CREATED</b>
<code>~Thread()</code>	Giải phóng bộ nhớ theo phong cách RAII. <b>Thread</b> được đảm bảo là đã kết thúc.
<code>Fork(VoidFunctionPtr func, int arg): char*</code>	Tạo một <b>Thread</b> để chạy hàm <code>func</code> với tham số <code>arg</code>
<code>Yield(): void</code>	Cho phép các thread khác chạy. Đặt trạng thái <b>READY</b>
<code>Sleep(): void</code>	Được sử dụng khi thread đang chờ một sự kiện hoặc đồng bộ hóa. Đặt trạng thái <b>BLOCKED</b>
<code>Finish(): void</code>	Đánh dấu <b>Thread</b> hiện tại đã hoàn thành và nó sẽ không chạy lại.
<code>CheckOverflow(): void</code>	Kiểm tra xem <b>Thread</b> có đã vượt quá kích thước stack đã cấp phát hay không.
<code>StackAllocate(VoidFunctionPtr func, int arg): void</code>	Cấp phát và khởi tạo stack cho thread
<code>SaveUserState(): void</code>	Lưu trạng thái của CPU khi chuyển đổi code người dùng và kernel.
<code>RestoreUserState(): void</code>	Phục hồi trạng thái của CPU khi chuyển đổi code người dùng và kernel.



- Các phương thức đặc biệt

Phương thức	Ý nghĩa
<code>ThreadPrint(int arg)</code>	Gọi <code>Thread::Print()</code> để debug
<code>SWITCH(Thread *oldThread, Thread *newThread):</code>	Chuyển đổi hai luồng. Dừng luồng cũ <code>oldThread</code> Khởi chạy luồng mới <code>newThread</code>
<code>ThreadRoot()</code>	Tạo khung xử lý cho một luồng khi một luồng được chạy, để nhận diện các hoạt động nhằm phòng tránh việc bị gián đoạn.

- Các thuộc tính của `Thread`:

Thuộc tính	Ý nghĩa
<code>stackTop: int*</code>	Đại diện cho con trỏ đến đầu của stack hiện tại. <code>stackTop</code> sẽ di chuyển theo thời gian để phản ánh việc cấp phát và giải phóng không gian trên stack.
<code>machineState: int[MachineStateSize]</code>	Lưu trữ trạng thái các thanh ghi CPU khi không chạy. Khi luồng được tạo hoặc chuyển đổi ngữ cảnh, trạng thái của các thanh ghi CPU được lưu vào <code>machineState</code> để có thể phục hồi lại khi luồng đó được chạy lại.
<code>stack: int*</code>	Đại diện cho đầu của ngăn xếp để cấp phát cho luồng. Ngăn xếp này lưu trữ các khung gọi và biến tạm. Khi luồng được tạo, <code>stack</code> trỏ đến đầu ngăn xếp được cấp phát. Khi luồng kết thúc, <code>stack</code> này sẽ được giải phóng.

<code>status: ThreadStatus</code>	Đại diện cho trạng thái hiện tại của luồng ( <code>JUST_CREATED</code> , <code>RUNNING</code> , <code>READY</code> , <code>BLOCKED</code> ). Trạng thái này được sử dụng bởi lịch trình <code>scheduler</code> để quyết định luồng nào sẽ được chạy tiếp theo.
<code>name: char*</code>	Tên của luồng, hữu ích cho việc debug
<code>space: AddrSpace*</code>	Đối với các luồng chạy chương trình người dùng, <code>space</code> sẽ trỏ đến không gian địa chỉ của chương trình đó. Điều này cho phép hệ thống điều hành quản lý và ánh xạ giữa địa chỉ ảo và địa chỉ vật lý của chương trình.
<code>processID: int</code>	Khi một luồng được tạo, <code>processID</code> được thiết lập để là ID của quá trình mà luồng thuộc về. Điều này giúp quản lý và xác định các luồng thuộc về cùng một quá trình.
<code>exitStatus: int</code>	Trạng thái thoát của luồng. Khi một luồng kết thúc, <code>exitStatus</code> được thiết lập để là trạng thái thoát của luồng. Điều này có thể được sử dụng để truyền thông tin về kết quả thực thi của luồng cho quá trình cha.
<code>userRegisters: int[NumTotalRegs]</code>	Lưu trữ trạng thái của các thanh ghi CPU khi luồng đang chạy mã người dùng. Điều này giúp quản lý và ánh xạ giữa trạng thái của CPU khi chạy mã người dùng và khi chạy mã hệ thống.

**d. Lớp AddrSpace**

**AddrSpace** là một khái niệm quan trọng trong hệ thống điều hành, đặc biệt là trong hệ thống đa luồng, nơi mà nhiều chương trình người dùng có thể chạy cùng một lúc. Trong hệ thống như vậy, mỗi chương trình người dùng cần có một không gian địa chỉ riêng biệt để quản lý và thực thi mã của nó mà không gây ra xung đột với các chương trình khác.

Trong đoạn mã được thiết kế, tại (`./userprog/addrspace.*`), **AddrSpace** đóng vai trò như một "bản đồ" cho mỗi chương trình người dùng, cho phép hệ thống điều hành biết về vị trí và trạng thái của mã và dữ liệu của chương trình đó trong bộ nhớ. Sử dụng lớp đó để quản lý không gian địa chỉ của các chương trình người dùng. Cụ thể, nó thực hiện các nhiệm vụ sau

**Phân phương thức chính bao gồm:**

Thuộc tính - Phương thức	Ý nghĩa
<code>AddrSpace(OpenFile *executable)</code>	Tạo một không gian địa chỉ để chạy tệp. Tải chương trình từ một tệp executable và thiết lập để thực thi các chỉ thị của người dùng Chỉ thực hiện đơn chương
<code>AddrSpace(char *filename)</code>	Tạo một không gian địa chỉ để chạy tệp. Tải chương trình từ một tệp executable và thiết lập để thực thi các chỉ thị của người dùng Thực hiện được đa chương
<code>~AddrSpace()</code>	Giải phóng một không gian địa chỉ.
<code>InitRegisters()</code>	Thiết lập giá trị ban đầu cho tập đăng ký CPU cấp độ người dùng.

<code>SaveState()</code>	Lưu trạng thái của máy, liên quan đến không gian địa chỉ này, khi có sự chuyển đổi ngữ cảnh.
<code>RestoreState()</code>	Phục hồi trạng thái của máy để không gian địa chỉ này có thể chạy.
<code>pageTable: TranslationEntry*</code>	Đại diện cho bảng ánh xạ giữa địa chỉ ảo và địa chỉ vật lý. Bảng ánh xạ này cho phép hệ thống điều hành quản lý và ánh xạ giữa địa chỉ ảo (được sử dụng bởi chương trình người dùng) và địa chỉ vật lý (được sử dụng bởi bộ nhớ phân cứng). Cho phép chương trình người dùng chạy mà không cần phải quan tâm đến cách bộ nhớ vật lý được sắp xếp.
<code>numPages: unsigned int</code>	Số lượng trang trong không gian địa chỉ ảo của chương trình người dùng mà cần thiết để lưu trữ toàn bộ chương trình, bao gồm cả mã, dữ liệu khởi tạo, dữ liệu không khởi tạo, và stack.
<code>usedPhyPage: bool[NumPhysPages]</code>	Mảng theo dõi các trang bộ nhớ vật lý đã được sử dụng bởi các không gian địa chỉ (True là đang sử dụng). Mỗi phần tử trong mảng <code>usedPhyPage</code> tương ứng với một trang bộ nhớ vật lý.

### 3. Thiết kế các lớp mới

Để thêm một lớp mới vào NachOS, tham khảo [Cách thêm Class SynchConsole vào Nachos](#) trong [báo cáo đề án 2](#) của nhóm tại trang 8. Phần bên dưới sẽ trình bày thiết kế của các lớp cần dùng để hỗ trợ cho đề án 3 lần này

#### a. Lớp PCB

Lớp **PCB** dùng để biểu diễn khối quản lý tiến trình (Process Control Block). Mỗi tiến trình có một khối quản lý tiến trình (PCB - Process Control Block) để quản lý các thông tin của nó như:

- ❖ Định danh (Process ID)
- ❖ Trạng thái tiến trình
- ❖ Ngưỡng cảnh tiến trình (Trạng thái CPU, Bộ nhớ chính, tài nguyên sử dụng / tạo lập)
- ❖ Thông tin giao tiếp (tiền trình cha, tiến trình con, độ ưu tiên)
- ❖ Thông tin thống kê

Trong mã nguồn, lớp **PCB** được cài đặt chi tiết trong `./userprog/PCB.*`, bảng dưới đây là tóm lược những thông tin quan trọng của lớp **PCB**

Thuộc tính	Ý nghĩa
<code>joinsem: Semaphore*</code>	Semaphore được sử dụng để đồng bộ hóa việc tham gia (join) quy trình.
<code>exitsem: Semaphore*</code>	Semaphore được sử dụng để đồng bộ hóa việc thoát (exit) quy trình.
<code>mutex: Semaphore*</code>	Semaphore được sử dụng để đảm bảo tính đồng bộ hóa khi truy cập vào các thuộc tính của <b>PCB</b> .
<code>exitcode: int</code>	Mã thoát của quy trình, được sử dụng để truyền thông tin về trạng thái thoát của quy trình.
<code>numwait: int</code>	Số lượng quy trình đang chờ quy trình hiện tại thoát.
<code>thread: Thread*</code>	Đối tượng <b>Thread</b> đại diện cho quy trình.

<code>parentID: int</code>	ID của quy trình cha để xác định quy trình cha của quy trình hiện tại.
----------------------------	------------------------------------------------------------------------

Phương thức	Ý nghĩa
<code>PCB(int id)</code>	Hàm khởi tạo, nhận vào <code>id</code> của quy trình và khởi tạo các semaphore.
<code>Exec(char* filename, int pID): int</code>	Tạo một luồng mới, gán tên tệp và ID quy trình chỉ định, và phân nhánh để thực thi chương trình. Trả về ID quy trình mới.
<code>JoinWait(): void</code>	Được gọi bởi một quy trình con để đợi quy trình cha thoát.
<code>JoinRelease(): void</code>	Báo hiệu cho quy trình con đang đợi rằng quy trình cha đã thoát.
<code>ExitWait(): void</code>	Được gọi bởi một quy trình trước khi thoát để đợi bất kỳ quy trình con nào có thể đang đợi nó.
<code>ExitRelease(): void</code>	Báo hiệu cho các quy trình con đang đợi rằng quy trình này đã thoát.

### b. Lớp PTable

Lớp **PTable** là lớp trừu tượng biểu diễn bảng quản lý các tiến trình trong chương trình, cung cấp các phương thức hỗ trợ cho đa chương như **Exec**, **Join**, **Exit**

Trong mã nguồn, lớp **PTable** được cài đặt chi tiết trong `./userprog/PTable.*`, bảng dưới đây là tóm lược những thông tin quan trọng của lớp **PTable**

Thuộc tính	Ý nghĩa
<code>psize: int</code>	Kích thước của đối tượng PCB để lưu kích thước của quy trình. Để xác định số lượng quy trình có thể được quản lý bởi <b>PTable</b> .

<code>bm: BitMap*</code>	Đối tượng <b>BitMap</b> được sử dụng để kiểm tra vị trí đã được sử dụng và theo dõi quy trình nào đã được khởi tạo và quy trình nào chưa.
<code>pcb: PCB*[MAX_PROCESS]</code>	Mảng các đối tượng đại diện cho quy trình đang chạy. <b>MAX_PROCESS</b> là số lượng tối đa của quy trình có thể được quản lý.
<code>bmsem: Semaphore*</code>	Đối tượng <b>Semaphore</b> được sử dụng để ngăn chặn trường hợp tải 2 quy trình cùng một lúc, đảm bảo tính đồng bộ trong việc quản lý quy trình.

Phương thức	Ý nghĩa
<code>PTable(int size): void</code>	Hàm khởi tạo, thiết lập kích thước của đối tượng PCB và khởi tạo giá trị ban đầu cho bm và bmsem. Nó cũng đánh dấu vị trí đầu tiên đã được sử dụng cho tiến trình cha (scheduler).
<code>ExecUpdate(char* name): int</code>	Hàm xử lý cho hệ thống gọi <b>SC_Exit</b> , thực hiện việc khởi tạo và thực thi một quy trình mới dựa trên tên được cung cấp.
<code>ExitUpdate(int exitcode): int</code>	Hàm xử lý cho hệ thống gọi <b>SC_Exit</b> , cập nhật mã thoát của quy trình hiện tại và thực hiện các thao tác liên quan đến việc kết thúc quy trình.
<code>JoinUpdate(int id): int</code>	Hàm xử lý cho hệ thống gọi <b>SC_Join</b> , cho một quy trình chờ cho đến khi quy trình khác hoàn thành.

### c. Lớp STable

Lớp **STable** là lớp trừu tượng biểu diễn bảng quản lý các **Semaphore** trong chương trình, cung cấp các phương thức hỗ trợ cho đồng bộ như **CreateSemaphore**, **Down**, **Up**

Trong mã nguồn, lớp **STable** được cài đặt chi tiết trong `./userprog/STable.*`, bảng dưới đây là tóm lược những thông tin quan trọng của lớp **STable**

Thuộc tính - Phương thức	Chức năng
<code>bm: BitMap*</code>	Đối tượng <b>BitMap</b> được sử dụng để theo dõi các vị trí đã được sử dụng trong bảng <b>Semaphore</b> .
<code>semTab: int</code>	Mảng các đối tượng <b>Sem</b> , mỗi đối tượng đại diện cho một <b>Semaphore</b> .
<code>Create(char* name, int init): int</code>	Hàm tạo một <b>Semaphore</b> mới với tên và giá trị khởi tạo cho trước. Nếu tên <b>Semaphore</b> đã tồn tại, hàm sẽ trả về -1
<code>Wait(char* name): int</code>	Hàm thực hiện thao tác giảm (wait) trên <b>Semaphore</b> có tên được chỉ định. Nếu <b>Semaphore</b> không tồn tại, hàm sẽ trả về -1.
<code>Signal(char* name): int</code>	Hàm thực hiện thao tác tăng (signal) trên <b>Semaphore</b> có tên được chỉ định. Nếu <b>Semaphore</b> không tồn tại, hàm sẽ trả về -1.



#### 4. Cài đặt các System Call

Trong phần này, chúng ta chỉ đề cập đến việc thiết kế logic hoạt động của các **System Call** thông qua flow chart. Chi tiết về cách cài đặt mời bạn tham khảo [mã nguồn](#) trong file `./code/userprog/exception.cc` và phần [Cài đặt các System Call](#) trong [báo cáo đồ án 2](#)

Chúng ta cần thêm các biến toàn cục sau để hỗ trợ cho việc cài đặt các **System Call**, tham khảo [Thêm biến toàn cục lớp SynchConsole](#) để biết cách cài đặt chi tiết để thêm một biến sử dụng toàn cục trong chương trình

```
extern Semaphore *addrLock;      // semaphore
extern BitMap *gPhysPageBitMap; // manages physical frames
extern PTable *pTable;           // manages processes
extern STable *sTable;           // manages semaphores
```

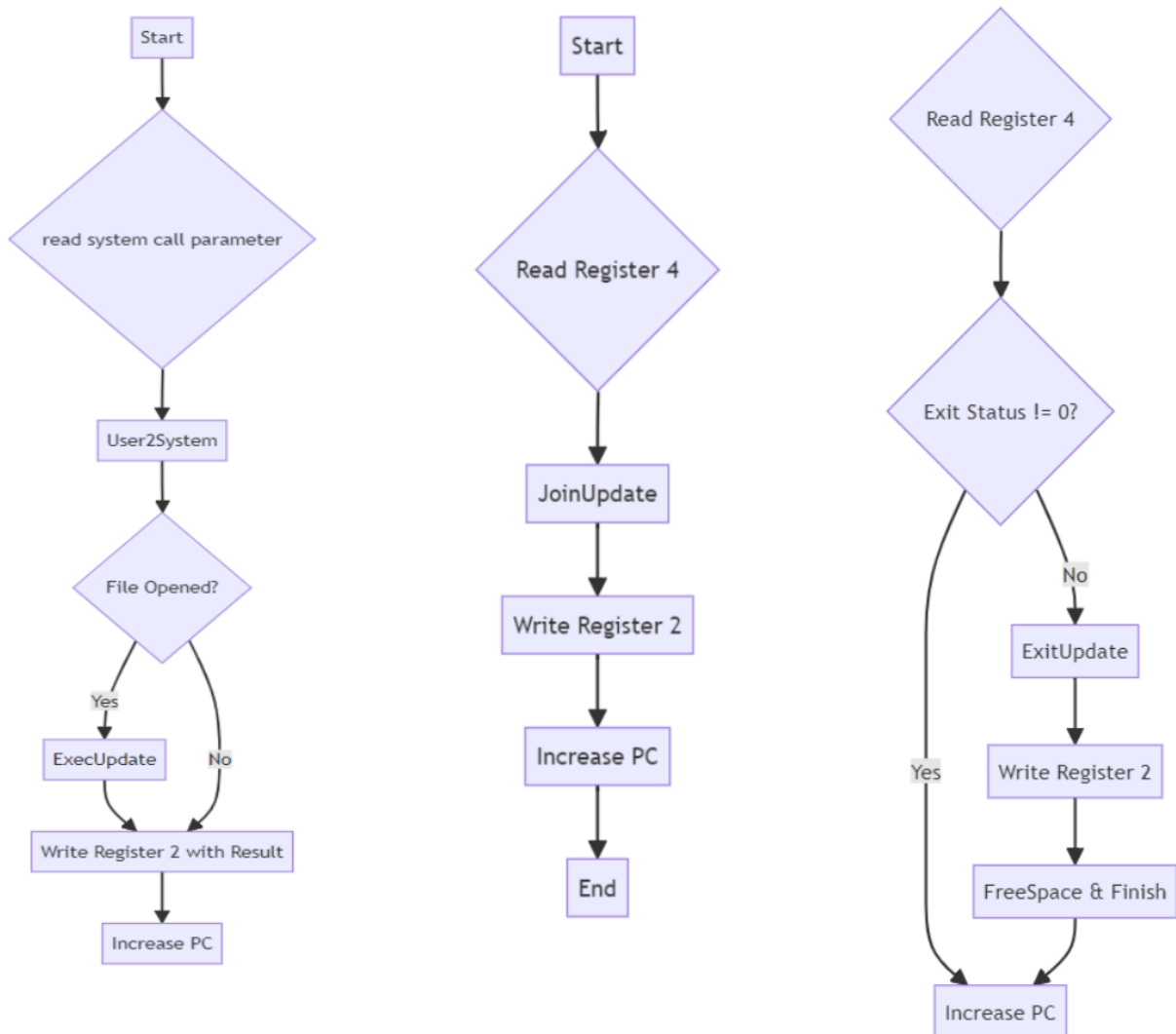
##### a. Exec, Join, Exit

Với các **System call** hỗ trợ đa chương này, ta cần dùng các phương thức **ExecUpdate**, **JoinUpdate**, **ExitUpdate** của lớp **PTable** để cài đặt lần lượt cho các **System call Exec, Join, Exit**

- **SpaceID Exec(char\* name)**: gọi thực thi một chương trình mới trong một system thread mới, cần tạo user space trong một system thread
- **int Join(SpaceID id)**: đợi và block dựa trên tham số **id**, trả về exit code cho tiến trình nó đã đang block trong đó, -1 nếu join bị lỗi. Một user program chỉ có thể join vào những tiến trình mà đã được tạo bằng **Exec**, không thể **Join** vào các tiến trình khác hoặc chính tiến trình mình.

- `void Exit(int exitCode)`: khi một tiến trình đã hoàn thành xong, mã exit code trả về là 0 thì gọi `Exit` để giải phóng vùng nhớ đã cấp phát cho tiến trình và kết thúc tiến trình

Hình bên dưới là lược đồ flow-chart mô tả các cài đặt 3 `System call Exec`, `Join`, `Exit` lần lượt từ trái sang phải (gần như giống nhau, chỉ khác ở phần gọi phương thức từ lớp `PTable`)

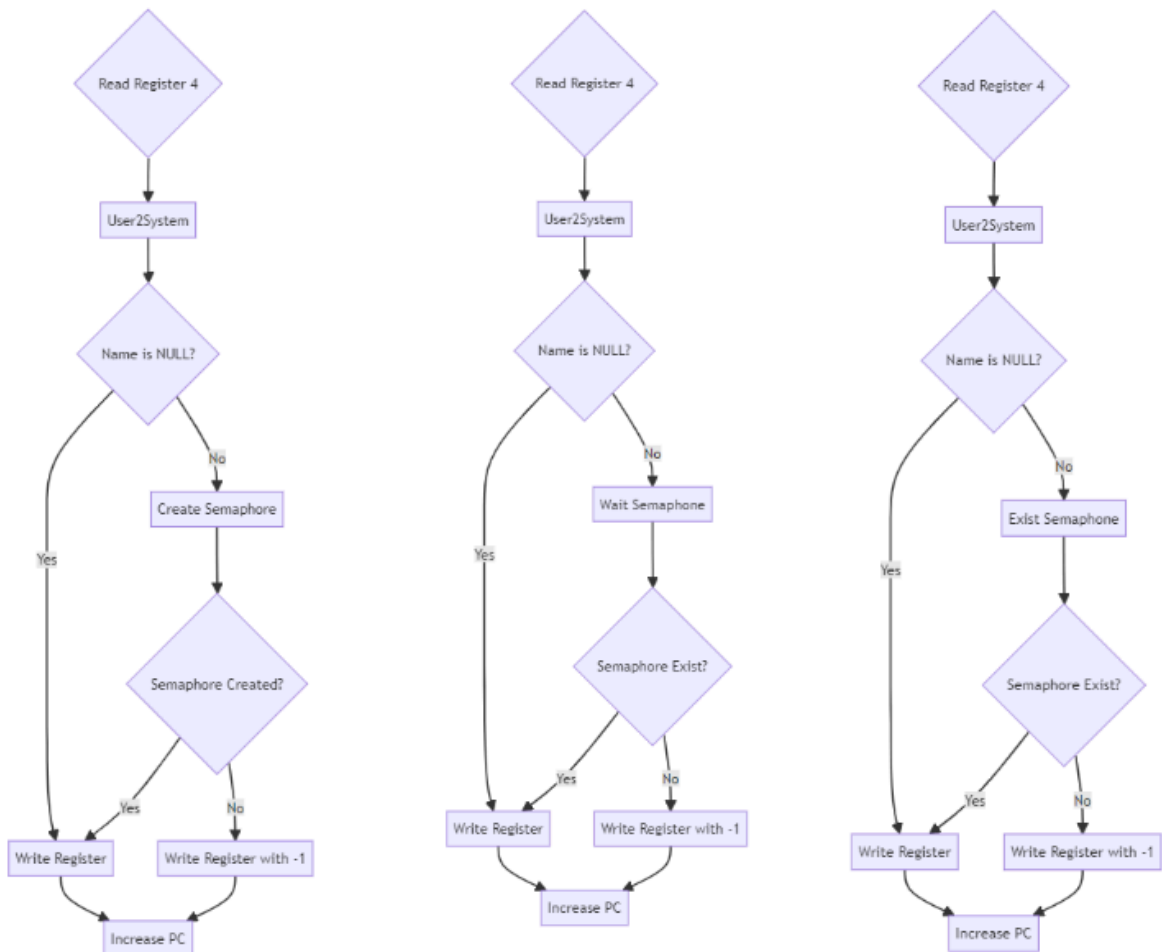


**b. CreateSemaphore, Down, Up**

Với các **System call** hỗ trợ đồng bộ, ta cần dùng các phương thức **Create, Wait, Signal** của lớp **STable** để cài đặt lần lượt cho các **System call** **CreateSemaphore, Down, Up**

- **int CreateSemaphore(char\* name, int semval)**: tạo một **Semaphore** với tên phân biệt **name** và giá trị khởi tạo **semval** để hỗ trợ cho bài toán đồng bộ “độc quyền truy xuất” hoặc “phối hợp hoạt động”
- **int Down(char\* name)**: có ý nghĩa tương tự với primitive **Down** của **Semaphore**, đưa tiến trình vào trạng thái sleep nếu **Semaphore** chuyển từ mang giá trị dương sang âm
- **int Up(char\* name)**: có ý nghĩa tương tự primitive **Up** của **Semaphore**, wakeup một tiến trình nếu giá trị **Semaphore** của nó chuyển từ âm sang dương

Hình bên dưới là lược đồ flow-chart mô tả các cài đặt 3 **System call** **CreateSemaphore**, **Down**, **Up** lần lượt từ trái sang phải (gần như giống nhau, chỉ khác ở phần gọi phương thức từ lớp **STable**)



## C. Viết chương trình minh họa

### 1. Ping - Pong

Chương trình PING-PONG gồm 3 chương trình chính:

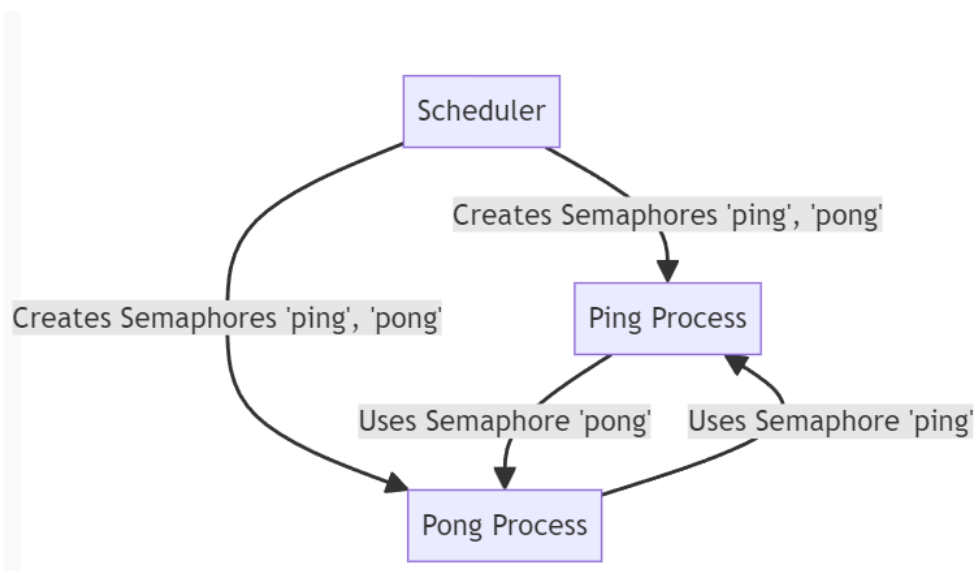
- + **ping**: in ra 1000 chữ A
- + **pong**: in ra 1000 chữ B
- + **main**: phối hợp để in A, B xen kẽ nhau

Chúng ta sử dụng 2 **Semaphore** để thực hiện phối hợp hoạt động giữa 2 chương trình **ping** và **pong**. Dưới đây là đoạn mã giả trong chương trình **ping** sử dụng 2 **Semaphore** để đồng bộ hóa trong file **ping.c**

```
For (i: 1 to 1000)
{
    Down("pong")
    Print('A')
    Up("ping")
}
```

Ta thực hiện tương tự với file **pong.c**, trong file **main.c**, ta tạo 2 **Semaphore** bằng **System call CreateSemaphore** và gọi lệnh **Exec** để thực thi 2 chương trình **ping** và **pong**

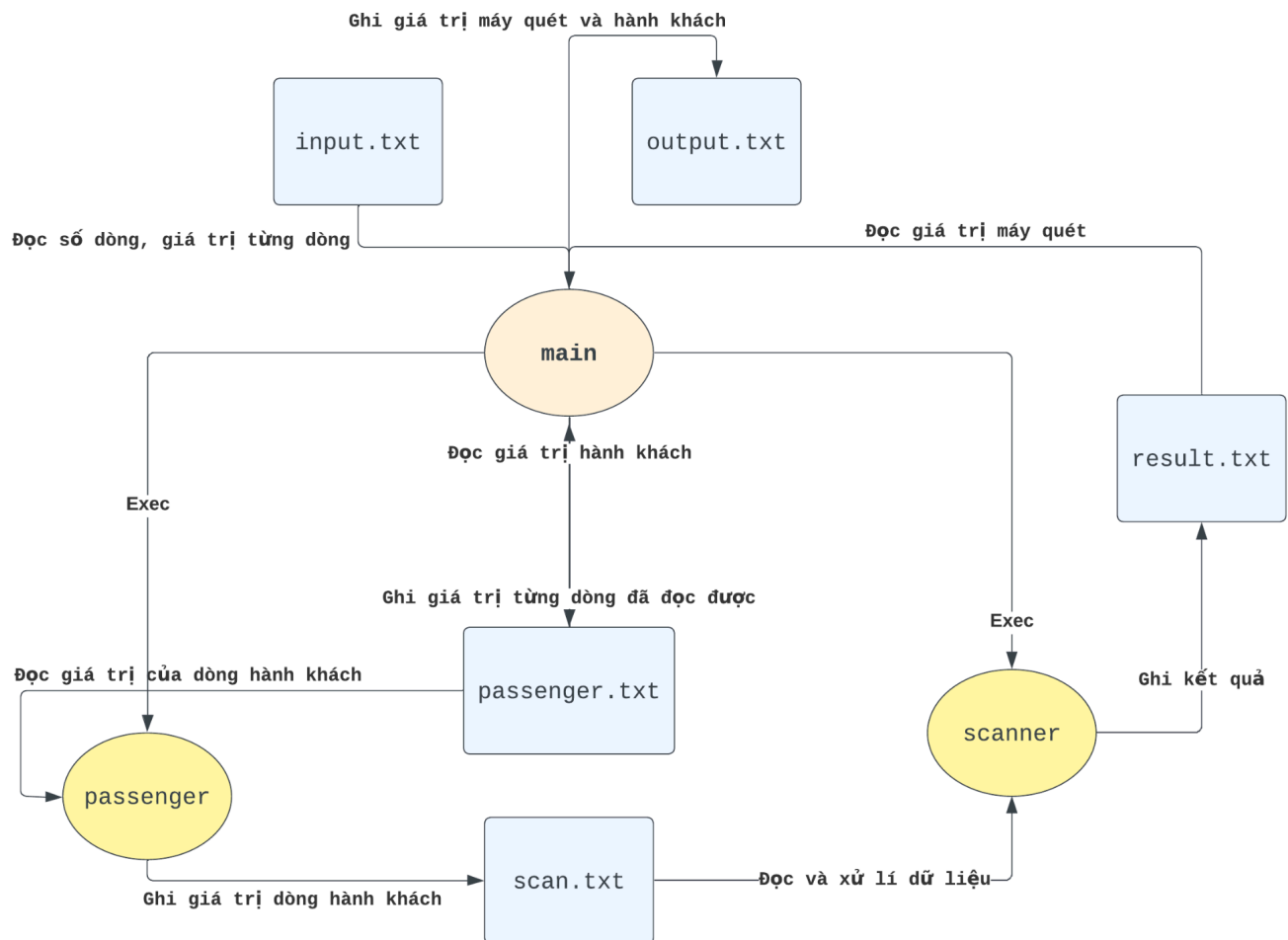
Hình vẽ bên dưới là sơ đồ thiết kế của chương trình



## 2. Thống kê sử dụng máy quét

Để xử lý bài toán này, ta cần áp dụng đa chương và đồng bộ cũng như các thao tác với tập tin như tạo tập tin, đọc, ghi, ... Chi tiết các **System call** liên quan đến xử lý tập tin bạn có thể tham khảo mã nguồn và phần [2. Syscall Create, Open, Close, Read, Write cho File](#), bên cạnh đó, ta cần viết thêm **System call Seek** để hỗ trợ giải quyết bài toán

Hình bên dưới minh họa một hướng giải quyết bài toán bằng cách sử dụng các **Semaphore** để phối hợp các công việc, lưu trữ giá trị xử lý trên các file và phối hợp xử lý trên các file đó. Từ thiết kế bên dưới, cài đặt chi tiết được viết trong các file [scan.c](#), [passenger.c](#), [scan\\_passenger.c](#) ở thư mục [./test/](#)



## V. Hình ảnh demo chương trình

### A. Chương trình *PING-PONG*

```

→ code git:(main) x ./userprog/nachos -x ./test/scheduler
Ping-Pong Program

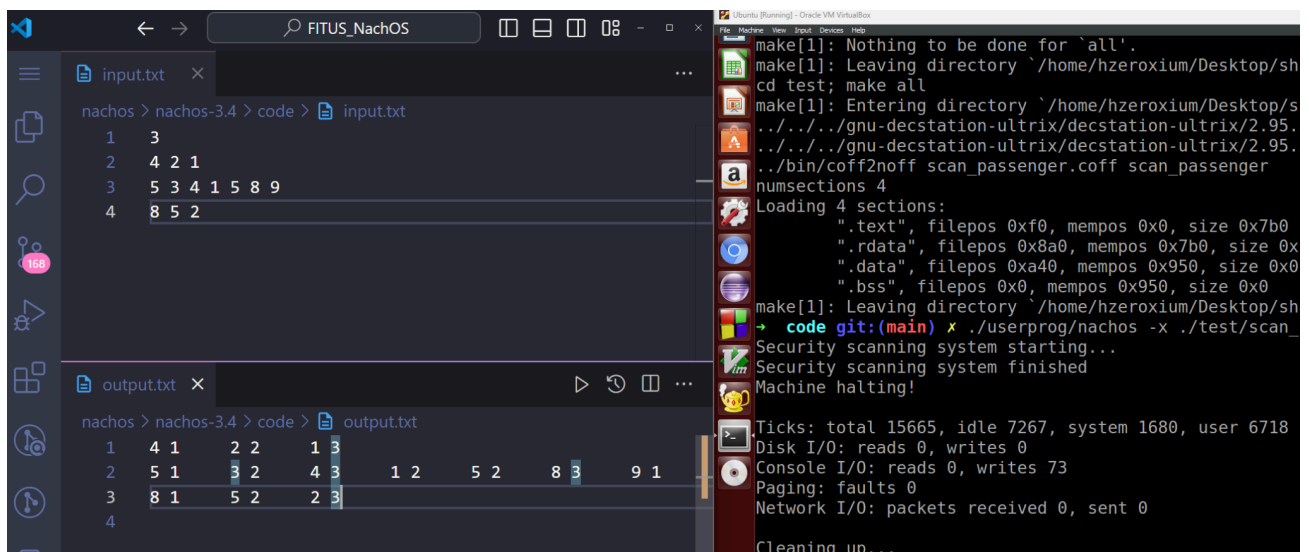
AAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBB
BBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBB
ABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAA
AAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBB
BBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAA
AAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBB
BBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBB
ABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAA
AAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBB
BBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAA
AAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBB
BBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBB
ABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAA
BBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBB
ABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAA
AAAAAAAAABBBBBBBBBBAAAAAMachine halting!

Ticks: total 359431, idle 198751, system 100550, user 60130
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 2020
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

```

### B. Chương trình máy quét



```

make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/hzeroxium/Desktop/sh'
cd test; make all
make[1]: Entering directory '/home/hzeroxium/Desktop/s
../../../../gnu-decstation-ultrix/decstation-ultrix/2.95.
../../../../gnu-decstation-ultrix/decstation-ultrix/2.95.
../bin/coff2noff scan_passenger.coff scan_passenger
numsections 4
Loading 4 sections:
".text", filepos 0xf0, mempos 0x0, size 0x7b0
".rdata", filepos 0x8a0, mempos 0x7b0, size 0x
".data", filepos 0xa40, mempos 0x950, size 0x0
".bss", filepos 0x0, mempos 0x950, size 0x0
make[1]: Leaving directory '/home/hzeroxium/Desktop/sh'
→ code git:(main) x ./userprog/nachos -x ./test/scan_
Security scanning system starting...
Security scanning system finished
Machine halting!

Ticks: total 15665, idle 7267, system 1680, user 6718
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 73
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

```

## VI. Tài liệu tham khảo

[1]. Ths. Lê Viết Long, *Tài liệu Project 3 - 20240412*

[2]. Nguyễn, T. C. [@thanhchungnguyen2618]. (n.d.). Lập trình nachos HCMUS.

Youtube. Retrieved March 31, 2024, from

[https://www.youtube.com/playlist?list=PLRgTVtca98hUgCN2\\_2vzsAAXPiTFbvHpO](https://www.youtube.com/playlist?list=PLRgTVtca98hUgCN2_2vzsAAXPiTFbvHpO)