# I. CHUẨN BỊ CHO QUÁ TRÌNH CÀI ĐẶT SYSTEM CALL VỀ ĐA CHƯƠNG, LẬP LỊCH VÀ ĐỒNG BỘ:

## Xác định yêu cầu:

- Thiết kế và cài đặt để hỗ trợ đa chương trình trên Nachos.
- Nachos hiện tại chỉ là môi trường đơn chương. Chúng ta sẽ phải lập trình để cho mỗi tiến trình được duy trì trong system thread của nó. Chúng ta phải quản lý việc cấp phát và thu hồi bộ nhớ, quản lý phần dữ liệu và đồng bộ hóa các tiến trình/ tiểu trình.

## Giải pháp:

**1. Cài đặt đa tiến trình:**

Chương trình hiện tại giới hạn chỉ thực thi 1 chương trình → cần phải có vài thay đổi trong file addrspace.h và addrspace.cc để chuyển hệ thống từ đơn chương thành đa chương. Cụ thể như sau:

+ Giải quyết vấn đề cấp phát các frames bộ nhớ vật lý sao cho nhiều chương trình có thể nạp lên bộ nhớ cùng một lúc→ sử dụng biến toàn cục ***Bitmap *gPhysPageBitMap*** để quản lý các frames.
+ Xử lý giải phóng bộ nhớ khi user program kết thúc.
+ Thay đổi đoạn lệnh nạp user program lên bộ nhớ. Hiện tại, việc cấp phát không gian địa chỉ giả thiết rằng một tiến trình được nạp vào các đoạn liên tiếp nhau trong bộ nhớ. Một khi hỗ trợ đa chương trình, bộ nhớ sẽ không còn biểu diễn liên tiếp nhau nữa→ tạo một ***pageTable = new TranslationEntry[numPages],*** tìm trang trống bằng phương thức Find() của lớp Bitmap, sau đó nạp chương trình lên bộ nhớ chính.

**2. Thiết kế các lớp theo hướng đối tượng, bao gồm 4 lớp sau:**

- Lớp PCB (system **data segment):** Lưu các thông tin để quản lý process. Cụ thể như sau:

```
private:
    Semaphore* joinsem;          // semaphore cho quá trình join
    Semaphore* exitsem;          // semaphore cho quá trình exit
    Semaphore* multex;           // semaphore cho quá trình truy xuất đọc quyền
    int exitcode;
    int numwait;                 // số tiến trình đã join
public:
    int parentID;                // ID của tiến trình cha
PCB();
PCB(int id);                     // contructor
~PCB();                          // deconstructor
// nạp chương trình có tên lưu trong biến filename và processID là pid
int Exec(char *filename, int pid);   // Tạo 1 thread mới có tên là filename và process là pid
```

```
int GetID();                          // Trả về ProcessID của tiến trình gọi thực hiện
int GetNumWait();                     // Trả về 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 của tiến trình
int GetExitCode();                    // Trả về exitcode

void SetFileName(char* fn);           // Đặt tên của tiến trình
char* GetFileName();                  // Trả về tên của tiến trình
```

- Lớp Ptable: dùng để quản lý các tiến trình được chạy, gồm thuộc tính pcb là một bảng mô tả tiến trình, có cấu trúc mảng một chiều có số phần tử tối đa là 10, mỗi phần tử của mảng thuộc kiểu dữ liệu PCB. Hàm constructor của lớp sẽ khởi tạo tiến trình cha (là tiến trình đầu tiên) ở vị trí thứ 0 tương đương với phần tử đầu tiên của mảng. Từ tiến trình này, chúng ta sẽ tạo ra các tiến trình con thông qua system call Exec(). Các thuộc tính và phương thức:

```
private:
    BitMap bm;                        // đánh dấu các vị trí đã được sử dụng trong pcb
    PCB* pcb[MAX_PROCESS];
    int psize;
    Semaphore* bmsem;                 // dùng để ngăn chặn trường hợp nạp 2 tiến trình cùng lúc
public:
// khởi tạo size đối tượng PCB để lưu size process. Gán giá trị ban đầu là null
// nhớ khởi tạo bm và bmsem để sử dụng
PTable(int size);
~PTable();                            // hủy các đối tượng đã tạo
int ExecUpdate(char* name);           // Xử lý cho system call SC_Exec
int ExitUpdate(int ec);               // Xử lý cho system call SC_Exit
int JoinUpdate(int id);               // Xử lý cho system call SC_Join
int GetFreeSlot();                    // tìm free slot để lưu thông tin cho tiến trình mới
bool IsExist(int pid);                // kiểm tra tồn tại processID này không?
void Remove(int pid);                 // khi tiến trình kết thúc, delete processID ra khỏi
```

```
mảng quản lý nó
char* GetFileName(int id);                    // Trả về tên của tiến trình
```

- Lớp Sem: dùng để quản lý Semaphore. Các thuộc tính và phương thức:

```
private:
    char name[50];
    Semaphore *sem;                    // Tạo Semaphore để quản lý
public:
// khởi tạo đối tượng Sem. Gán giá trị ban đầu là null
// nhớ khởi tạo bm sử dụng
Sem(char* na, int i){
    strcpy(this->name,na);
    sem = new Semaphore(name,i);
}
~Sem(){                                // hủy các đối tượng đã tạo
    delete sem;
}
void wait(){                           // thực hiện thao tác chờ
    sem->P();
}
void signal(){                         // thực hiện thao tác giải phóng Semaphore
    sem->V();
}
char* GetName(){                       // Trả về tên của Semaphore
    return name;
}
```

- Lớp Stable: gồm thuộc tính semTab là một bảng mô tả semaphore, có cấu trúc mảng một chiều có số phần tử tối đa là 10, mỗi phần tử của mảng thuộc kiểu dữ liệu Sem. Các thuộc tính và phương thức:

```
private:
    BitMap* bm;                        // quản lý slot trống
    Sem* semTab[MAX_SEMAPHORE];        // 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.
```

➢ Các biến toàn cục cần khai báo:

***Semaphore* addrLock;***

***BitMap* gPhysPageBitMap;***

***STable* semTab;***

***PTable* pTab;***

## II. CÀI ĐẶT CÁC SYSTEM CALL:

### 1. SC_Exec:

Khai báo hàm: **SpaceID Exec(char* name).**

Exec system call sử dụng lớp PCB và Ptable để gọi thực thi một chương trình mới trong một system thread mới.

Trước khi cài đặt system call Exec ta cần cài đặt các phương thức;

Cài đặt Exec(char* name, int pid) ở lớp PCB:

- Gọi mutex->P(); để giúp tránh tình trạng nạp 2 tiến trình cùng 1 lúc.
- Kiểm tra thread đã khởi tạo thành công chưa, nếu chưa thì báo lỗi là không đủ bộ nhớ, gọi mutex->V() và return.
- Đặt processID của thread này là id.
- Đặt parrentID của thread này là processID của thread gọi thực thi Exec
- Gọi thực thi Fork(StartProcess_2,id) => Ta cast thread thành kiểu int, sau đó khi xử ký hàm StartProcess ta cast Thread về đúng kiểu của nó.
- Trả về id.

Cài đặt ExecUpdate(char* name) ở lớp Ptable:

- Gọi mutex->P(); để giúp tránh tình trạng nạp 2 tiến trình cùng 1 lúc.
- Kiểm tra tính hợp lệ của chương trình "name".
- Kiểm tra sự tồn tại của chương trình "name" bằng cách gọi phương thức Open của lớp fileSystem.

So sánh tên chương trình và tên của currentThread để chắc chắn rằng chương trình này không gọi
- thực thi chính nó.
  Tìm slot trống trong bảng Ptable.
- Nếu có slot trống thì khởi tạo một PCB mới với processID chính là index của slot này, parrentID là
- processID của currentThread.
  Đánh dấu đã sử dụng.
- Gọi thực thi phương thức Exec của lớp PCB.
- Gọi bmsem->V().
- Trả về kết quả thực thi của PCB->Exec.
-

Quá trình xử lý của system call Exec:

Đọc địa chỉ tên chương trình "name" từ thanh ghi r4.
- Tên chương trình lúc này đang ở trong user space. Gọi hàm *User2System* đã được khai báo trong
- lớp *machine* để chuyển vùng nhớ user space tới vùng nhớ system space.
  Nếu bị lỗi thì báo "Không mở được file" và gán -1 vào thanh ghi 2.
- Nếu không có lỗi thì gọi pTab-> ExecUpdate(name), trả về và lưu kết quả thực thi phương thức này
- vào thanh ghi r2.

**2. SC_Join và SC_Exit:**

- **SC_Join**

Khai báo hàm: **int Join(SpaceID id).**

Join system call sử dụng lớp PCB và Ptable để thực hiện đợi và block dựa trên tham số "SpaceID id".

Trước khi cài đặt system call này ta phải cài đặt các phương thức sau:

Cài đặt JoinWait() ở lớp PCB:

Gọi joinsem->P() để tiến trình chuyển sang trạng thái block và ngừng lại, chờ JoinRelease để thực
- hiện tiếp.

Cài đặt ExitRelease() ở lớp PCB:

Gọi exitsem-->V() để giải phóng tiến trình đang chờ.

Cài đặt JoinUpdate(int id) ở lớp Ptable.:

Ta kiểm tra tính hợp lệ của processID id và kiểm tra tiến trình gọi Join có phải là cha của tiến trình
- có processID là id hay không. Nếu không thỏa, ta báo lỗi hợp lý và trả về -1.
  Tăng numwait và gọi JoinWait() để chờ tiến trình con thực hiện.
- Sau khi tiến trình con thực hiện xong, tiến trình đã được giải phóng
- Xử lý exitcode.
- ExitRelease() để cho phép tiến trình con thoát.
-

Quá trình xử lý của system call Join:

- Đọc id của tiến trình cần Join từ thanh ghi r4.
- Gọi thực hiện pTab->JoinUpdate(id) và lưu kết quả thực hiện của hàm vào thanh ghi r2.
-

➢ **SC_Exit**

Khai báo hàm: **void Exit(int exitCode).**

Exit system call sử dụng lớp PCB và Ptable để thực hiện thoát tiến trình nó đã join.

Trước khi cài đặt system call này ta phải cài đặt các phương thức sau:

Cài đặt JoinRelease() ở lớp PCB:

- Gọi joinsem->V() để giải phóng tiến trình gọi JoinWait().

Cài đặt ExitWait () ở lớp PCB:

- Gọi exitsem-->V() để tiến trình chuyển sang trạng thái block và ngừng lại, chờ ExitReleaseđể thực hiện tiếp.

Cài đặt ExitUpdate(int exitcode) ở lớp Ptable:

- Nếu tiến trình gọi là main process thì gọi Halt().
- Ngược lại gọi SetExitCode để đặt exitcode cho tiến trình gọi.
- Gọi JoinRelease để giải phóng tiến trình cha đang đợi nó(nếu có) và ExitWait() để xin tiến trình cha cho phép thoát.

Quá trình xử lý của system call Exit:

- Đọc exitStatus từ thanh ghi r4
- Gọi thực hiện pTab->ExitUpdate(exitStatus) và lưu kết quả thực hiện của hàm vào thanh ghi r2.

**3. SC_CreateSemaphore:**

Khai báo hàm: **int CreateSemaphore(char* name, int semval).**

CreateSemaphore system call để tạo Semaphore mới.

Quá trình xử lý của system call CreateSemaphore:

- Đọc địa chỉ “name” từ thanh ghi r4.
- Đọc giá trị “semval” từ thanh ghi r5.
- Tên địa chỉ “name” lúc này đang ở trong user space. Gọi hàm *User2System* đã được khai báo trong lớp *machine* để chuyển vùng nhớ user space tới vùng nhớ system space.
- Gọi thực hiện hàm semTab->Create(name,semval) để tạo Semaphore, nếu có lỗi thì báo lỗi.
- Lưu kết quả thực hiện vào thanh ghi r2.
-

### 4. SC_Up và SC_Down:

- **SC_Up**

Khai báo hàm: **int Up(char* name).**

Up system call sử dụng lớp Stable để giải phóng tiến trình đang chờ.

Quá trình xử lý của system call Up:

- Đọc địa chỉ "name" từ thanh ghi r4.
- Tên địa chỉ "name" lúc này đang ở trong user space. Gọi hàm *User2System* đã được khai báo trong lớp *machine* để chuyển vùng nhớ user space tới vùng nhớ system space.
- Kiểm tra Semaphore "name" này có trong bảng sTab chưa, nếu chưa có thì báo lỗi.
- Gọi phương thức Signal() của lớp Stable.
- Lưu kết quả thực hiện vào thanh ghi r2.

- **SC_Down:**

Khai báo hàm: **int Down(char* name).**

Down system call sử dụng lớp Stable để thực hiện thao tác chờ.

Quá trình xử lý của system call Down:

- Đọc địa chỉ "name" từ thanh ghi r4.
- Tên địa chỉ "name" lúc này đang ở trong user space. Gọi hàm *User2System* đã được khai báo trong lớp *machine* để chuyển vùng nhớ user space tới vùng nhớ system space.
- Kiểm tra Semaphore "name" này có trong bảng sTab chưa, nếu chưa có thì báo lỗi.
- Gọi phương thức Wait() của lớp Stable.
- Lưu kết quả thực hiện vào thanh ghi r2.

## III. SỬ DỤNG CÁC SYSTEM CALL:

Sử dụng các system call trên để cài đặt chương trình shell:

Chương trình dùng để nhận một lệnh tại một thời điểm và thực thi chương trình tương ứng.

Quá trình xử lý của chương trình shell:

- Chạy vòng lặp do..while(1) với điều kiện dừng khi người dùng nhập lệnh "exit".
- Yêu cầu người dùng nhập vào tên chương trình cần chạy.
- Kiểm tra chương trình có tồn tại ở trên máy không, nếu không thì yêu cầu nhập lại.
- Kiểm tra chương trình kí tự đầu trong chuỗi buffer người dùng nhập vào.