THỰC HÀNH HỆ ĐIỀU HÀNH

BÁO CÁO ĐỒ ÁN 2

I. Thành viên:

Giáo viên hướng dẫn: thầy Lê Giang Thanh

Họ và tên	MSSV
Nguyễn Nhật Minh Khôi	19120020
Bạch Ngọc Minh Tâm	19120034
Hoàng Minh Huy	19120241

II. Tổng kết công việc:

Công việc	Thành viên thực hiện	Mức độ hoàn thành
Cấp phát bộ nhớ và tổ chức hệ thống	Khôi	100%
Syscall Exec	Khôi, Tâm	100%
Syscall Join	Tâm	100%
Syscall Exit	Tâm	100%
Syscall CreateSemaphore	Huy	100%
Syscall Wait	Huy	100%
Syscall Signal	Huy	100%
Tổ chức quản lý file	Khôi, Tâm	100%
Syscall Create	Khôi	100%
Syscall Open	Tâm	100%
Syscall Close	Huy	100%
Syscall Read	Huy	100%
Syscall Write	Khôi	100%
Syscall Append	Tâm, Huy	100%
Syscall ProcessID	Khôi	100%
Chương trình demo	Khôi, Tâm	100%

III. Các file đính kèm:

- Báo cáo này
- Các syscall được khai báo/cài đặt trong 4 file: exception.cc, ksyscall.h, syscall.h nằm trong thư mục *source/NachOS-4.0/code/userprog* và start.S trong thư mục *source/NachOS-4.0/code/test.*
- Các phần liên quan đa chương và đồng bộ được cài đặt các file *pcb.h, pcb.cc, ptable.h, ptable.cc, sem.h, stable.h, stable.cc* và *kernel.h* (thêm các thuộc tính mới) trong thư mục *source/NachOS-4.0/code/threads*.

- Các phần liên quan quản lý file được cài trong 2 file chính: *ftable.h, ftable.cc*. Ngoài ra còn có interface được cài trong 4 file: *pcb.h, pcb.cc, ptable.h, ptable.cc*. Tất cả đều nằm trong thư mục *source/NachOS-4.0/code/threads*.
- Các chương trình demo được cài đặt trong thư mục *source/NachOS-4.0/code/test* với tên file là *project2.c* và *sinhvien.c*.

IV. Chi tiết công việc:

1) Đa chương và đồng bộ

a) Cách **cấp phát bộ nhớ** cho một chương trình

Để có thể chạy nhiều tiến trình, ta cần thiết kế lại hàm cấp phát bộ nhớ cho chương trình, đó là hàm *AddrSpace::Load*, sơ lược hàm này có các nhiệm vụ như sau:

- Tính toán số lượng trang nhớ cần dùng cho chương trình, ở đây giả sử không tồn tại bộ nhớ cấp phát động

```
int numCodePages = divRoundUp(noffH.code.size, PageSize);
int numDataPages = divRoundUp(noffH.initData.size, PageSize);
int numRemainPages = divRoundUp(noffH.uninitData.size + UserStackSize, PageSize);
int numReadOnlyPages = divRoundUp(noffH.readonlyData.size, PageSize);
numPages = numCodePages + numDataPages + numReadOnlyPages + numRemainPages;
```

- Cấp phát các trang trống cho chương trình và đánh dấu đã dùng

```
pageTable = new TranslationEntry[numPages];
for (i = 0; i < numPages; i++)
{
        ...
        pageTable[i].physicalPage = kernel->gPhysPageBitMap->FindAndSet();
        ...
}
```

- Chép dữ liệu (code segment, readonly segment và initial data segment) vào bộ nhớ chính (main memory) của hệ điều hành.

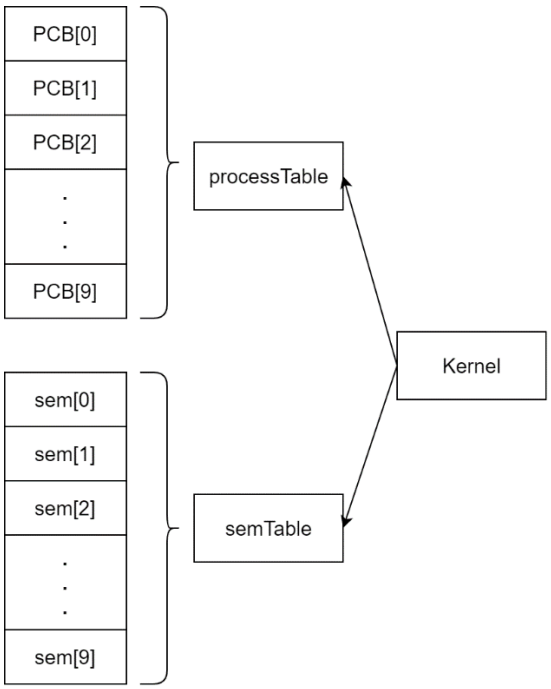
Để có thể làm hàm *Load* như trên, yêu cầu trong class *Kernel* phải có thêm hai thuộc tính mới:

- *addrLock* (*Semaphore*): dùng để đồng bộ việc chép bộ nhớ, tránh chép hai chương trình cùng lúc.
- *gPhysPageBitMap (Bitmap)*: dùng để đánh dấu các trang vật lý đã dùng, tránh ghi đè.

b) Tổ chức hệ thống

Ta sẽ có các class để quản lý việc đa chương là class PCB đại diện cho một tiến trình trong hệ điều hành. Tiếp theo, để chứa các process đang còn hoạt động trong hệ điều hành, ta có class PTable. Trong class Kernel đại diện cho nhân của hệ điều hành, ta tạo một biến processTable kiểu PTable để quản lý các process đang chạy.

Tương tự, để cài đặt tính năng đồng bộ, ta kiểm soát các semaphore của người dùng bằng class STable, class sẽ có một mảng semTab có độ lớn 10 (số semaphore tối đa), mỗi phần tử của mảng là một đối tượng kiểu Sem. Kiểu Sem này là một wrapper, tạo ra interface của lớp Semaphore (hệ điều hành cài sẵn) cho người dùng sử dụng. Tương tự, ta cũng tạo một biến semTable kiểu STable để quản lý các process đang chạy trong class Kernel.



Lưu ý là process table khi khởi tạo sẽ luôn có một process ở vị trí pcb[0] tượng trưng cho chương trình main trong hệ điều hành.

c) System call **Exec**

Đó là hàm *SysExec(char* name)* mà nhóm đã cài đặt trong file *ksyscall.h*. Hàm này có kiểu *int* và có 2 giá trị trả về: trả về 0 nếu thực hiện thành công, hoặc trả về -1 nếu thất bại. System call sẽ yêu cầu *kernel->processTable* thực thi file với tên là tham số *name*. Khi nhận được yêu cầu, processTable sẽ thực hiện các việc sau:

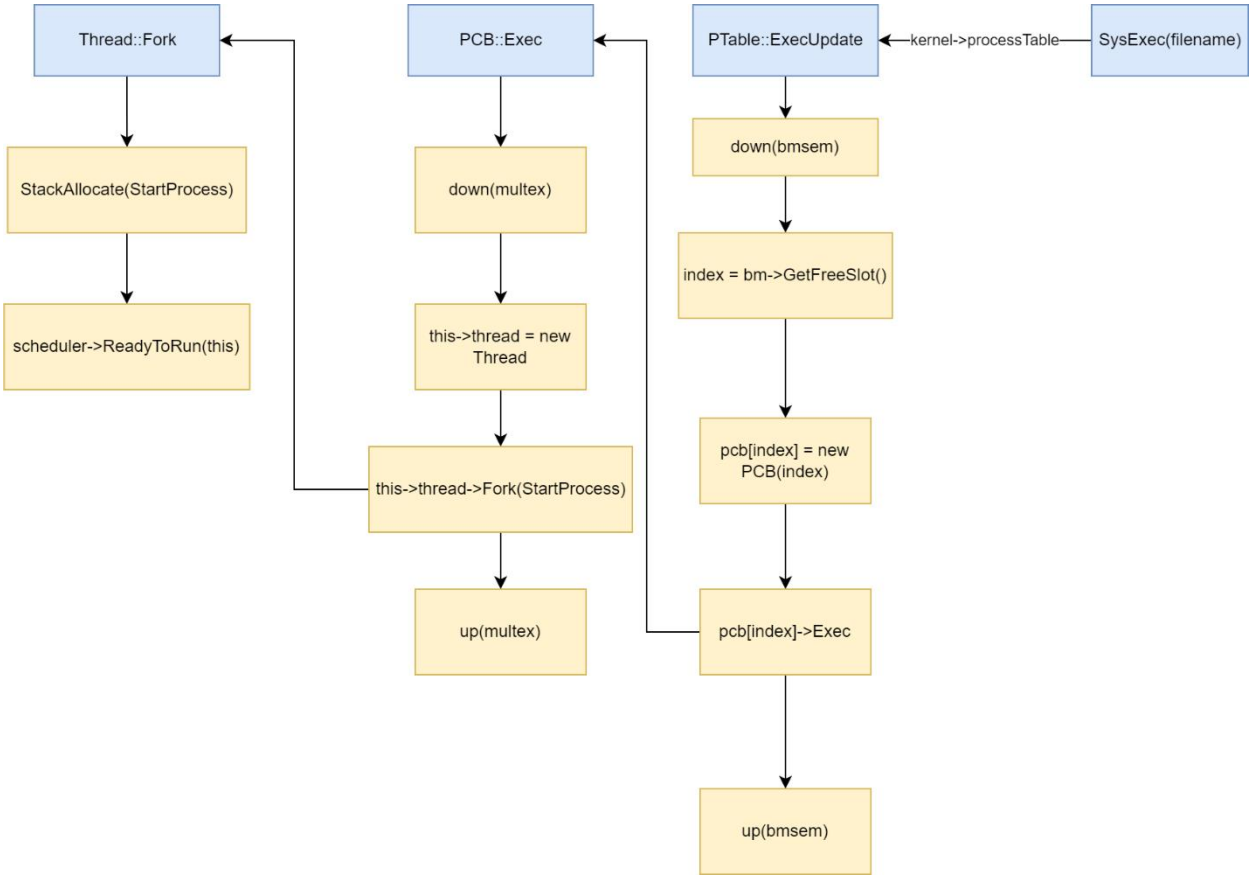
- Thực hiện down semaphore *bmsem* để tránh việc process table load hai chương trình cùng lúc.
- Tìm một vị trí trống trong process table để load chương trình lên, thao tác này do thuộc tính bitmap trong class PTable thực hiện.

- Tạo một đối tượng PCB mới ở vị trí trống đã tìm và yêu cầu process thực thi (hàm PCB::Exec) với hai thông tin là filename và id của process.
- Thực hiện up semaphore bmsem để báo hiệu có thể load process khác lên.

Khi một process (đối tượng PCB) được yêu cầu thực thi, nó sẽ thực hiện các việc sau:

- Thực hiện down semaphore *multex* để tránh việc process chạy các hàm khác trước khi được thực thi.
- Tạo một thread mới từ chương trình ở trong file có tên filename được đưa.
- Khởi tạo các thông số của thread và chính nó.
- Dùng *this->thread->Fork* để yêu cầu thread đặt nó vào scheduler, chuyển qua trạng thái READY để chờ được load lên. Lưu ý ta cũng truyền hàm *Start_Process* như một callback để cấp phát bộ nhớ cho thread.
- Thực hiện up semaphore *multex* để báo hiệu có thể load process khác lên.

Toàn bộ quá trình ở trên được mô tả ở hình dưới



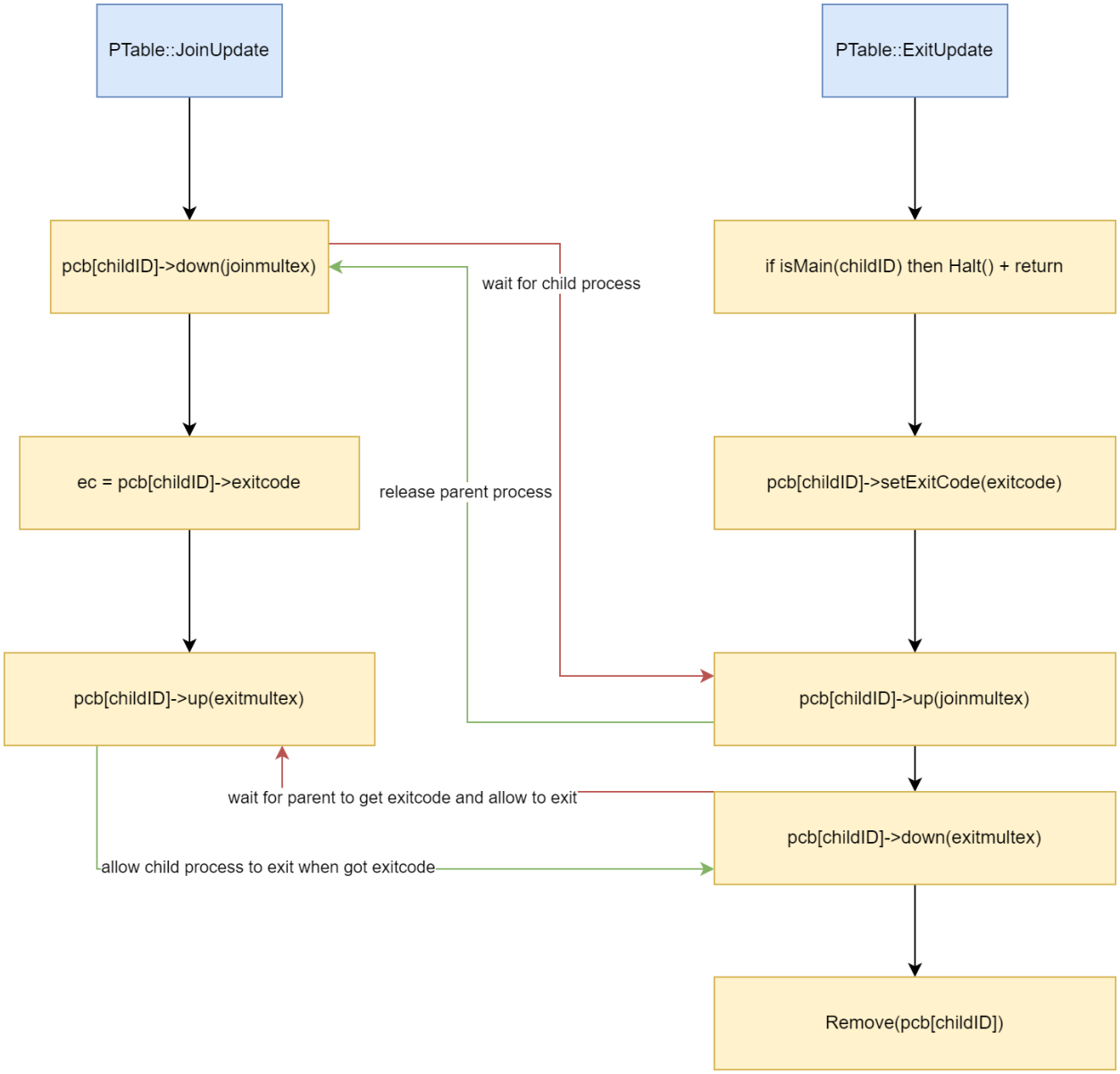
Một số ngoại lệ được xử lý và trả về -1 cho người dùng:

- Filename là null.
- Không thể mở file để đọc.

- Filename trùng với process hiện tại (một process không được exec chính nó vì sẽ gây ra đệ quy vô hạn).
- Không còn chỗ trống trong process table để load chương trình lên (đã đủ 10 process).
- Không thể cấp phát thread do đã hết bộ nhớ

d) System call **Join** và **Exit:**

Đó là hai hàm *SysJoin(int id)* và *SysExit(int id)* mà nhóm đã cài đặt trong file *ksyscall.h*. Ta sẽ xem luồng thực hiện của hai hàm này song song do chúng có quan hệ mật thiết với nhau. Ta sẽ gọi *processTable->JoinUpdate(id)* và *processTable->ExitUpdate(id)* tương ứng với hai thao tác join và exit. Giả sử chỉ số của process cha và con lần lượt là *parentID* và *childID*, ta có sơ đồ sau:



Khi join, tiến trình cha sẽ *down(joinmultex)* của tiến trình con để chờ tiến trình con thực hiện xong, tiến trình con thực hiện xong sẽ *up(joinmultex)* của chính mình để báo cho tiến trình cha dừng đợi. Tuy nhiên lúc này tiến trình con chưa được phép dừng mà phải *down(exitmultex)*, chờ tiến trình cha lấy được *exitcode* của tiến trình con thì tiến trình cha mới *up(exitmultex)* để cho phép tiến trình con thoát. Cuối cùng, ta huỷ tiến trình con và trả về *exitcode*.

Có hai chú ý: (1) nếu ta đang exit tiến trình main thì sẽ không thực hiện các thao tác up, down mà sẽ gọi *kernel->interrupt->Halt()* để dừng hệ thống nachos; (2) mặc định các tiến trình cha sẽ không đợi tiến trình con.

Một số ngoại lệ được xử lý và trả về -1 cho người dùng:

- Process id truyền vào không tồn tại
- Process được join không phải là con của process gọi hàm join.

e) System call **CreateSemaphore**

Đó là hàm *SysCreateSem (char* name, int semval)* mà nhóm đã cài đặt trong file *ksyscall.h*. Hàm này có kiểu int và có 2 giá trị trả về: trả về 0 nếu thực hiện thành công, hoặc trả về -1 nếu thất bại. Kịch bản của system call này như sau:

System call này sẽ nhận vào tên semaphore cần được tạo mới (được biểu diễn bởi chuỗi ký tự name dưới dạng con trỏ kiểu char) và giá trị khởi tạo của semaphore (được biểu diễn bằng một số nguyên).

Hàm sẽ gọi tính năng *Create()* của semTable (cú pháp*: kernel->semTable->Create()*).

Hàm *Create()* sẽ tìm một vị trí trống trong bảng quản lý semaphore để tạo semaphore mới với tên *name* và giá trị *semval*, lưu lại vào bảng semaphore rồi trả về 0.

Các ngoại lệ được xử lý và trả về -1 cho người dùng:

- Có semaphore trùng tên đã tồn tại trước đó.
- Tên của semaphore không tồn tại (*name == NULL*)
- Không còn chỗ để tạo semaphore (đã đủ 10 semaphore)

f) System call **Wait:**

Đó là hàm *SysWait (char* name)* mà nhóm đã cài đặt trong file *ksyscall.h.* Hàm này có kiểu int và có 2 giá trị trả về: trả về 0 nếu thực hiện thành công, hoặc trả về -1 nếu thất bại. System call này có kịch bản như sau:

System call này sẽ nhận vào tên semaphore cần được chờ, được biểu diễn bằng chuỗi ký tự name (truyền vào dưới dạng con trỏ)

Hàm gọi tính năng Wait() của semTable (cú pháp: kernel->semTable->Wait()).

Hàm Wait() khiến cho tiến trình chờ đợi (sleep) cho đến khi giá trị semaphore > 0 thì giảm giá trị semaphore xuống rồi ngắt. Sau khi thực hiện thành công, hàm trả về 0.

Các ngoại lệ được xử lý và trả về -1 cho người dùng:

- Tên của semaphore không tồn tại (*name == NULL*)
- Không tìm thấy semaphore có tên là name

g) System call **Signal:**

Đó là hàm *SysSignal (char* name)* mà nhóm đã cài đặt trong file *ksyscall.h*. Hàm này có kiểu *int* và có 2 giá trị trả về: trả về 0 nếu thực hiện thành công, hoặc trả về -1 nếu thất bại. Kịch bản của system call này về cơ bản khá tương đồng với system call Wait:

System call cũng nhận vào tên semaphore cần được chờ, được biểu diễn bằng chuỗi ký tự name (truyền vào dưới dạng con trỏ).

Hàm gọi tính năng *Signal()* của semTable (cú pháp: *kernel->semTable->Signal()*).

Hàm *Signal()* tăng giá trị của semaphore lên nhằm đánh thức một tiến trình đang chờ đợi. Sau khi thực hiện thành công, hàm trả về 0.

Các ngoại lệ được xử lý và trả về -1 cho người dùng:

- Tên của semaphore không tồn tại (*name == NULL*)
- Không tìm thấy semaphore có tên là *name*

h) System call **ProcessID**

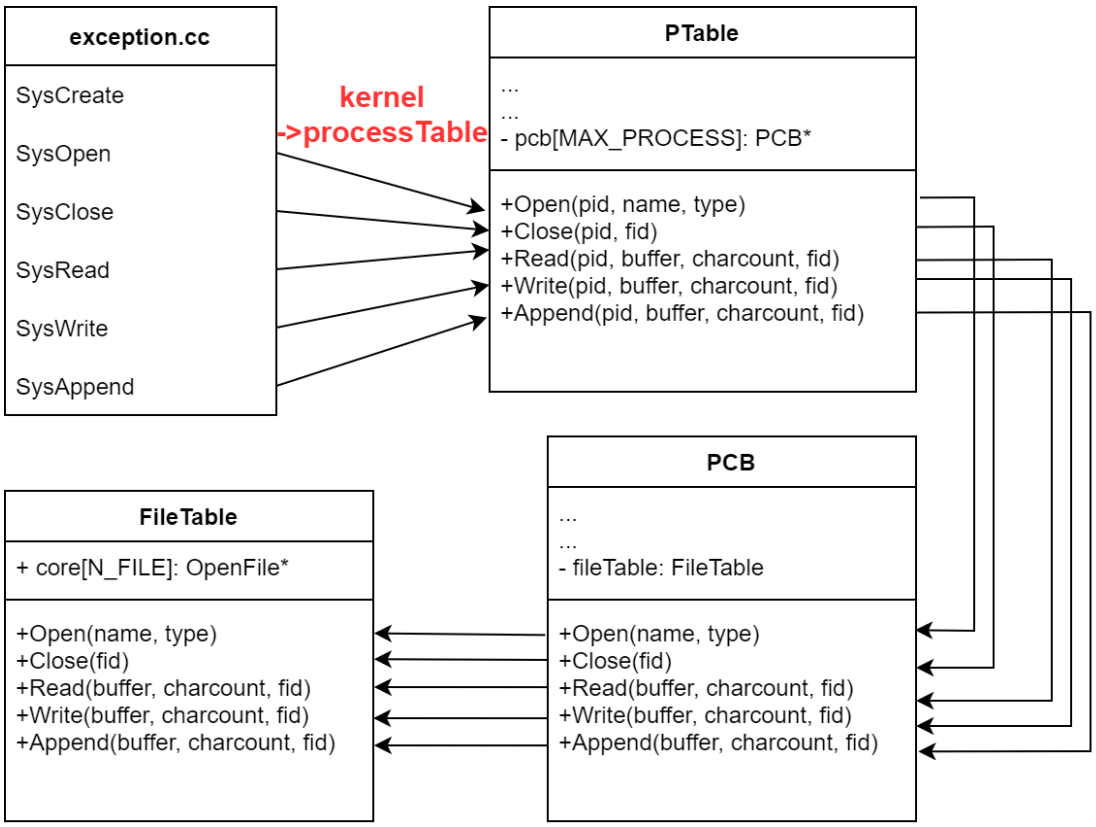
Đây là system call phụ trợ cho chương trình demo ở phần 3, có tác dụng trả về id của tiến trình đang chạy hiện tại. Cách cài đặt cũng rất đơn giản, chỉ cần trả về *kernel->currentThread->processID* cho người dùng qua thanh ghi số 2.

2) Hệ thống quản lý file

a) Tổ chức **quản lý file** trong một tiến trình

Với hệ thống đa chương đã cài đặt ở phần 1, ta sẽ tạo một lớp *FileTable* mới để quản lý các file của tiến trình.

Đồng thời, ta cũng sẽ cài đặt các interface *Open, Close, Read, Write, Append* của *FileTable* trong lớp *PCB* và lớp *PTable* để lúc xử lý system call trong exception.cc ta có thể gọi các chức năng này thông qua *kernel->processTable* (hình dưới).



b) System call **Create**:

Đó là hàm ***SysCreateFile(char* filename)*** mà nhóm đã cài đặt trong file ***ksyscall.h.***

Hàm này có kiểu int và có 2 giá trị trả về: trả về 0 nếu thực hiện thành công việc tạo file, hoặc trả về -1 nếu việc tạo file thất bại. Kịch bản của system call này như sau:

System call này sẽ nhận vào tên file mà người dùng muốn tạo mới, được biểu diễn bằng chuỗi ký tự *filename*.

Hàm sẽ gọi hàm *Create()* của kernel (cú pháp: *kernel->fileSystem->Create()*).

Hàm *Create()* thực hiện việc tạo file và có kiểu trả về là *bool*: trả về ***true*** nếu tạo file thành công hoặc trả về ***false*** nếu tạo file thất bại

Dựa trên kết quả trả về của hàm *Create()*, system call trả về kết quả tương ứng.

Một số ngoại lệ được xử lý và trả về -1 cho người dùng:

- Không đủ vùng nhớ để lưu trữ chuỗi filename
- Không đủ vùng nhớ để tạo file

c) System call **Open:**

Đó là hàm ***SysOpen*** *(char* filename, int type)* mà nhóm đã cài đặt trong file *ksyscall.h.* Hàm này có kiểu *int* và trả về ID của *filename* (được quản lý bởi directory) nếu mở thành công hoặc -1 nếu mở thất bại. Kịch bản của system call này như sau:

System call này sẽ nhận vào tên thư mục mà người dùng muốn mở (được biểu diễn bằng chuỗi ký tự *filename*) và biến type kiểu int cho biết kiểu mở file (Nhóm quy định như sau: 0 là kiểu mở đọc và ghi, 1 là chỉ đọc, 2 là nhập từ bàn phím STDIN, 3 là xuất ra màn hình STDOUT).

Hàm kiểm tra kiểu mở *type*:

- Nếu *type = 0* hoặc *type = 1*, hàm sẽ gọi tính năng *OpenFile()* của file table thông qua process table (cú pháp: *kernel->processTable->OpenFile()*) để tìm ra ID của *filename* cần mở và trả về ID của nó. Về cơ bản, hàm *OpenFile()* trước hết sẽ tìm ra một vùng nhớ trống để làm vùng nhớ tạm lưu trữ thông tin có thể xuất ra khi mở file. Nếu đã tìm thấy vùng nhớ, hàm sẽ thực hiện mở file để đọc ghi thông qua hàm *OpenForReadWrite.* Nếu không thể mở file, hàm sẽ trả về NULL. Nếu mở thành công, hàm sẽ lưu nội dung của file vào vùng nhớ tạm có ID đã tìm thấy trước đó rồi trả về ID của file.
- Nếu kiểu mở là 2, hàm trả về hằng số STDIN (ID của kiểu stdin, ở đây là 0)
- Nếu kiểu mở là 3, hàm trả về hằng số STDOUT (ID của kiểu stdout, ở đây là 1)

Một số trường hợp ngoại lệ được xử lý và trả về -1 cho người dùng:

- Không đủ dung lượng lưu lại filename
- Không tìm thấy ID của file được gọi
- Không tìm thấy file có tên filename
- Kiểu mở file khác các giá trị đã quy định
- Nếu không tìm thấy vùng nhớ trống, hàm sẽ trả về -1

d) System call **Close:**

Đó là hàm *SysClose (OpenFileID fid)* mà nhóm đã cài đặt trong file *ksyscall.h*. Hàm này có kiểu int và trả về ID của filename (được quản lý bởi directory) nếu mở thành công hoặc -1 nếu mở thất bại. Kịch bản của system call này như sau:

System call này sẽ nhận vào ID của file cần đóng. Đầu tiên tìm ID của tiến trình đang chạy (*kernel->currentThread->processID*). Sau đó fọi tính năng *CloseFile()* của file table thông qua process table (cú pháp: *kernel->processTable->CloseFile()*) để cập nhật trạng thái của file tương ứng và trả về kết quả là -1.

Các ngoại lệ cần xử lý và trả về -1 cho người dùng:

- File cần đóng chưa được mở
- ID của file (*fid*) không hợp lệ

- Không tìm thấy file có ID = *fid*

e) System call **Read:**

Đó là hàm *SysRead (char* buffer, int charcount, OpenFileID fid)* mà nhóm đã cài đặt trong file *ksyscall.h*. Hàm này có kiểu int và trả về số lượng ký tự đã đọc được thành công; -1 nếu đọc thất bại hoặc -2 nếu file cần đọc là file rỗng, không có ký tự nào để đọc.

Lý do hàm chỉ trả về số ký tự chứ không trả về nội dung đọc được là vì buffer được truyền vào dưới dạng con trỏ kiểu char. Do đó việc đọc dữ liệu sẽ ảnh hưởng trực tiếp đến dữ liệu mà buffer đang lưu trữ. Nói cách khác, ta gần như truyền tham chiếu của buffer vào hàm, nên không cần trả về giá trị của buffer. Kịch bản của system call này như sau:

- System call này sẽ nhận vào ID của thư mục cần đọc (số nguyên *fid*), một mảng ký tự *buffer* có vai trò vùng nhớ tạm, và một số nguyên *charcount* cho biết số lượng ký tự cần đọc từ file nhằm tránh việc đọc những ký tự rác không cần thiết.
- Đầu tiên, hàm kiểm tra ID của file cần mở
  - Nếu file cần mở là file có kiểu sdtin (ID = 0), hàm sẽ cho phép người dùng nhập ký tự thông qua hàm *GetChar()* của *synchConsoleIn* (đã cài đặt ở đồ án 1, cú pháp: *kernel->synchConsoleIn->GetChar()*). Tuy nhiên buffer chỉ ghi nhận tối đa *charcount* ký tự.
  - Nếu file cần mở là file bình thường, hàm sẽ lấy vị trí hiện tại *oldPos* của con trỏ hiện hành trong file trước khi đọc thông qua hàm *GetCurrentPos()* (thông thường, khi khởi đầu, hàm này sẽ trả về giá trị 0) rồi thực hiện đọc file từ vị trí bắt đầu *oldPos* và lưu lại thông tin vào buffer thông qua hàm *core[fid]->Read()* (với *fid* là số nguyên biểu diễn ID của file).
- Sau khi đọc xong, hàm sẽ lưu lại vị trí mới newPos của con trỏ trong file sau khi đã đọc file và trả về độ chênh lệch giữa 2 vị trí *newPos – oldPos* (cũng chính là số ký tự đã đọc được).

Các ngoại lệ mà hệ thống xử lý :

- Không tìm thấy file: trả về -1
- File cần tìm chưa được mở: trả về -1
- File cần đọc có kiểu stdout: trả về -1
- File cần tìm không có nội dung để đọc: trả về -2

f) System call **Write:**

Đó là hàm *SysWrite (char* buffer, int charcount, OpenFileID fid)* mà nhóm đã cài đặt trong file *ksyscall.h*. Hàm này có kiểu *int* và trả về số lượng ký tự đã xuất ra thành công hoặc -1 nếu viết thất bại.

Tương tự như system call Read, do buffer được truyền vào dưới dạng con trỏ nên nội dung được ghi vào sẽ được chỉnh sửa trực tiếp trên buffer. Kịch bản của system call này như sau:

System call này sẽ nhận vào ID của thư mục cần đọc, một mảng ký tự *buffer* có vai trò vùng nhớ tạm, và một số nguyên *charcount* cho biết số lượng ký tự cần được xuất ra nhằm tránh viết xuất giá trị rác hoặc không cần thiết.

Hàm xác định ID của tiến trình đang chạy rồi gọi tính năng *WriteFile()* của kernel (cú pháp: *kernel->processTable->WriteFile()*). Hàm *WriteFile()* sẽ xử lý dựa trên ID của file truyền vào:

- Nếu ID = STDOUT = 0, hàm sẽ gọi hàm *PutChar*() của *synchConsoleOut* (đã cài đặt ở đồ án 1, cú pháp: *kernel->synchConsoleOut->PutChar()*) và cho phép người dùng ghi vào file. Tuy nhiên, hàm sẽ ngừng ghi nhận khi đạt đến *charcount* ký tự hoặc đã chạm đến cuối chuỗi ký tự. Sau khi nhập liệu kết thúc, hàm trả về số lượng ký tự đã nhập vào được.
- Nếu ID không phải 2 trường hợp trên: hàm sẽ thực hiện mở file, gọi hàm *core[fid]->Write()* (với *fid* là số nguyên biểu diễn ID của file), ghi dữ liệu vào file rồi trả về số lượng ký tự đã ghi được. Cụ thể như sau:
    - Hàm sẽ kiểm tra xem file đã mở chưa rồi lấy vị trí hiện tại *oldPos* của con trỏ hiện hành trong file trước khi ghi vào thông qua hàm *GetCurrentPos()* (trong trường hợp này, *oldPos* = 0)
    - Thực hiện ghi vào file từ vị trí bắt đầu oldPos và lưu lại thông tin vào buffer thông qua hàm *core[fid]->Write()*. Việc ghi bắt đầu ở đầu file, nên dữ liệu mới sẽ được ghi đè lên một phần (hoặc toàn bộ) dữ liệu cũ trước đó.

Trong quá trình ghi, hàm sẽ liên tục cập nhật vị trí hiện hành của con trỏ trong file. Kết thúc quá trình ghi, hàm lưu lại vị trí mới *newPos* của con trỏ trong file sau khi đã đọc file và trả về độ chênh lệch giữa 2 vị trí *newPos – oldPos* (cũng chính là số ký tự đã ghi được)

Một số ngoại lệ sẽ trả về -1 cho người dùng:

- ID của file không hợp lệ
- File cần ghi có kiểu stdin hoặc read-only
- File cần viết chưa được mở

g) System call **Append:**

Đó là hàm ***SysAppend (char* buffer, int charcount, OpenFileID fid)*** mà nhóm đã cài đặt trong file *ksyscall.h.* Hàm này có kiểu *int* và trả về số lượng ký tự đã xuất ra thành công hoặc -1 nếu viết thất bại.

Tương tự như system call Read và Write, do buffer được truyền vào dưới dạng con trỏ nên nội dung được ghi vào sẽ được chỉnh sửa trực tiếp trên *buffer*. Về cơ bản, kịch bản của system call này gần như tương tự system call Write:

System call cũng nhận vào ID của thư mục cần đọc, một mảng ký tự *buffer* và một số nguyên *charcount* với vai trò tương tự system call Write.

Hàm xác định ID của tiến trình đang chạy rồi gọi hàm *AppendFile*() của file table thông qua process table (cú pháp: *kernel->processTable->AppendFile()*) để xử lý:

- Nếu ID = STDOUT = 1, hàm gọi hàm *PutChar()* của kernel, cho phép người dùng ghi vào file và trả về số lượng ký tự đã nhập y hệt như trong hàm *WriteFile*().
- Nếu ID là một file bình thường đã mở, hàm kiểm tra xem file đã mở chưa rồi gọi hàm con *core[fid]->Append()* (với *fid* là số nguyên biểu diễn ID của file), ghi nối tiếp nội dung vào cuối file và trả về số lượng ký tự đã ghi được

Do system call Append được dùng để viết tiếp dữ liệu vào trong một file mà không làm mất đi dữ liệu vốn có, hàm được gọi này sẽ không cần quan tâm vị trí khởi đầu như sysstem call Write. Thay vào đó, nó di chuyển con trỏ về cuối file và cho phép người dùng ghi vào file. Trong lúc ghi, hàm sẽ liên tục cập nhật số lượng ký tự đã ghi được, thay vì vị trí của con trỏ (do không còn cần thiết nữa).

Các ngoại lệ cần xử lý và trả về -1 cho người dùng:

- ID của file *fid* truyền vào không hợp lệ.
- File cần append chưa mở
- File được mở là file stdin hoặc file read-only.

3) Chương trình minh hoạ

a) Đề bài:

Trong một ký túc xá, có một vòi nước và n sinh viên, n sẽ input từ người dùng (n < 5). Mỗi sinh viên sẽ **cần lấy 10 lít nước từ vòi**, nhưng ở mỗi lần lấy sinh viên chỉ có thể lấy **tối đa 1 lít nước** và phải trả vòi nước lại (sau khi trả vòi nước sinh viên có quyền yêu cầu sử dụng vòi nước ngay lập tức). **Thời gian lấy một lít nước của mỗi sinh viên là ngẫu nhiên** (có thể dùng vòng lặp for để mô phỏng thời gian đợi). Hãy viết một hệ thống đáp ứng được các yêu cầu trên:

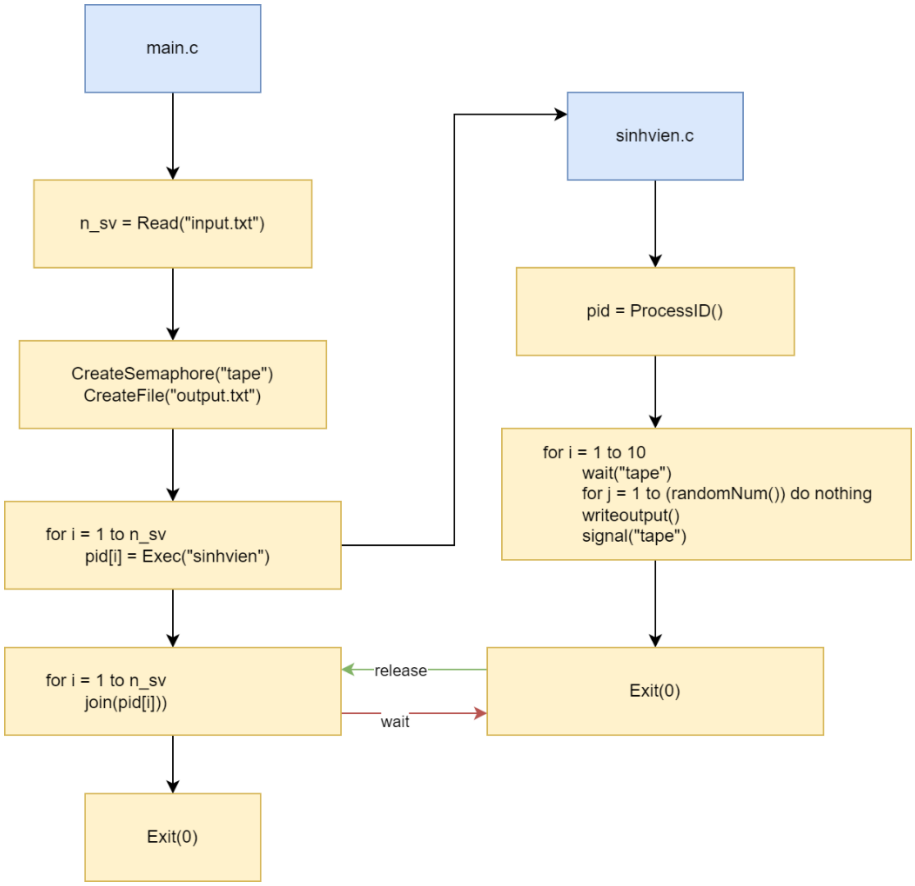
Input: số lượng sinh viên (có thể đọc từ file hoặc nhập từ bàn phím)

Output: Thứ tự lấy nước của sinh viên cho đến khi sinh viên cuối cùng lấy được 10 lít nước. Ví dụ nếu có 3 sinh viên và các sinh viên có số thứ tự từ 1-3 thì output sẽ có dạng: 1, 2, 1, 1 ,2, 3.....

b) Cách giải:

Ý tưởng chung của chương trình là sẽ có một chương trình ***main*** thực thi việc đọc số lượng sinh viên từ người dùng, sau đó chương trình *main* mô phỏng việc các sinh viên lấy nước bằng việc yêu cầu thực thi chương trình ***sinhvien*** n lần, với n là số lượng sinh viên đã đọc từ lúc nãy và gọi join để chờ các tiến trình *sinhvien* kết thúc thì mới thoát. Chương trình *main* cũng phụ trách việc chuẩn bị các tài nguyên cần thiết cho chương trình *sinhvien*, đó là một semaphore tượng trưng cho tài nguyên vòi nước và một file *output.txt* rỗng.

Chương trình main sẽ thực hiện các công việc chính như hình sau:



Trong đây, ta cũng thấy được luồng xử lý công việc chính của chương trình *sinhvien*. Chương trình sẽ có một vòng for từ 1 đến 10 tượng trưng cho 10 lần lấy nước của sinh viên, mỗi lần 1 lít. Mỗi lần như vậy, sinh viên sẽ chờ semaphore vòi nước (tape) được trả lại (lớn hơn 0) và thực hiện lấy nước, thao tác lấy nước được mô tả bằng hai việc:

- Một vòng for với số lượng lặp **ngẫu nhiên**, tượng trưng cho thời gian lấy nước ngẫu nhiên của sinh viên
- Sau khi đã lấy xong, sinh viên viết tên mình vào file kết quả

Cuối cùng, sinh viên trả lại tài nguyên vòi nước bằng cách *Signal(tape)*. Khi kết thúc chương trình, sinh viên sẽ gọi *Exit(0)* để báo cho chương trình cha biết.

Một số ngoại lệ được xử lý trong chương trình:

- Nếu số lượng sinh viên không thể đọc được từ file đầu vào, chương trình sẽ yêu cầu người dùng nhập từ console số lượng sinh viên.
- Nếu file kết quả không tồn tại, chương trình sẽ ghi kết quả ra màn hình console.

Để sử dụng chương trình, ta cần thực hiện các bước sau:

- Tạo file input.txt trong thư mục code và điền một số nguyên vào file (tương ứng với số lượng sinh viên của chương trình).
- Mở console, vào thư mục build.linux gõ make để compile hệ điều hành nachos
- Mở console, vào thư mục test gõ make để compile chương trình test (chương trình có tên project2)
- Chạy chương trình: *./build.linux/nachos -rs 1234 -x ./test/project2*. Option -rs để truyền random seed, giúp thời gian context switch được ngẫu nhiên và option -x để truyền file thực thi.