Đồ án 2 – Đa Chương

Phần 1. Hiểu về đa chương trên Nachos

Một hệ điều hành thực tế hiện nay hỗ trợ chạy đồng thời nhiều tiến trình (đa chương). Hệ điều hành Nachos thực ra là một tiến trình chạy trên Linux, do đó Nachos giả lập một tiến trình chạy trên nó là một tiểu trình trên Linux (lớp Thread).

Hiện thời, hệ điều hành Nachos chỉ hỗ trợ đơn chương và chưa hỗ trợ system call để thực thi một tiến trình.

Nhiệm vụ chính của đồ án này là hỗ trợ cho Nachos có thể thực thi nhiều tiến trình cùng lúc.

Các tập tin trong đồ án này:

- **progtest.cc** kiểm tra các thủ tục để chạy chương trình người dùng
- **syscall.h** system call interface: các thủ tục ở kernel mà chương trình người dùng có thể gọi
- **exception.cc** xử lý system call và các exception khác ở mức user, ví dụ như lỗi trang, trong phần mã chúng tôi cung cấp, chỉ có 'halt' system call được viết
- **bitmap.*** các hàm xử lý cho lớp bitmap (hữu ích cho việc lưu vết các ô nhớ vật lý) **filesys.h**
- **synchconsole.*** nhóm hàm cho việc quản lý nhập xuất I/O theo dòng trong Nachos.
- **../test/*** Các chương trình C sẽ được biên dịch theo MIPS và chạy trong Nachos.
- **thread.*:** Các hàm liên quan tới việc quản lý các thread bên trong hệ thống Nachos như: Cấp phát stack, đưa một thread vào trạng tháy sleep, thay đổi trạng thái hoạt động của một thread, lưu trữ trạng thái khi xuất hiện "context switching".
- **list.*:** Lớp dùng để quản lý danh sách các đối tượng.
- **addrspace.*:** Lớp dùng để quản lý việc cấp phát và thu hồi bộ nhớ cho tiến trình.

Phần 2. Hướng dẫn cài đặt đa chương cho Nachos

Trong đồ án này, chúng ta sẽ cài đặt để hỗ trợ đa chương trình trên Nachos. Đồng thời cài đặt thêm system call để thực thi tiến trình từ người lập trình trên hệ điều hành Nachos. Các bạn phải phát triển chương trình từ đồ án 1. Phải chắc rằng đồ án 1 của các bạn đã viết đúng và đầy đủ.

Chúng ta sẽ 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 cho các tiến trình/ tiểu trình. Thay đổi mã cho các exception khác (không phải system call exceptions) để tiến trình có thể hoàn tất, chứ không halt máy như trước đây. Một run time exception sẽ không gây ra việc HĐH phải shut down.

Sau khi hoàn thành đồ án 2, Nahos có thể chạy được đa tiến trình như sau:

❖ **Chương trình scheduler**

```
#include "syscall.h"
void main()
{       int pingPID, pongPID;
        PrintString("Ping-Pong test starting ...\n\n");
        pingPID = Exec("./test/ping", 4);
        pongPID = Exec("./test/pong", 4);
        /* Need to wait here by some ways */
}
```

❖ **Chương trình Ping**

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

❖ **Chương trình Pong**

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

Kết quả là 2 ký tự A, B sẽ được xuất tuỳ ý ra màn hình: AABABBBABABAAABAABA….

Hướng dẫn chi tiết cài đặt như sau:

1. Cài đặt system call **SpaceID Exec(char* name) hoặc SpaceID Exec(char* name, int priority (có khi cài đồ án cộng điểm))**. Exec gọi thực thi một chương trình mới trong một system thread mới. Bạn cần phải đọc hiểu hàm "StartProcess" trong progtest.cc để biết cách khởi tạo một user space trong 1 system thread. Exec trả về -1 nếu bị lỗi và thành công thì trả về Process SpaceID của chương trình người dùng vừa được tạo. Các bước như sau:
   a. Tạo ra một không gian địa chỉ mới
   b. Load chương trình vào khoảng bộ nhớ mới được cấp phát
   c. Sau đó tạo thread mới (bằng phương thức Thread::Fork()) để thực thi chương trình.

2. Cài đặt đa tiến trình. Chương trình hiện tại giới hạn bạn chỉ thực thi 1 chương trình, bạn phải có vài thay đổi trong file addrspace.h và addrspace.cc để chuyển hệ thống từ đơn chương thành đa chương. Bạn sẽ cần phải:
    a. 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.
    b. Phải xử lý giải phóng bộ nhớ khi user program kết thúc.
    c. Phần quan trọng là 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 chúng ta hỗ trợ đa chương trình, bộ nhớ sẽ không còn biểu diễn liên tiếp nhau nữa. Nếu chúng ta không lập trình đúng đắn thì khi nạp một chương trình mới có thể làm phá hỏng HĐH của bạn.

Chú ý: không để cho user có thể làm sụp HĐH, system call nên xử lý càng nhiều trường hợp càng tốt.

Phần 3. Nộp đồ án

Khi bạn hoàn thành, xóa các object file và các file thực thi. Tar hoặc nén file đó lại theo mã số sinh viên của 1 bạn trong nhóm.

Phần 4. Phụ Lục

```
int doSC_Exec()
{
      ......
      Thread *mythread;
      mythread = new Thread(...);
      ......
      mythread->Fork(StartProcess,pid);
      ......
}

AddrSpace::AddrSpace(OpenFile *executable)
{
    NoffHeader noffH;
    unsigned int i, size;

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    if ((noffH.noffMagic != NOFFMAGIC) &&
            (WordToHost(noffH.noffMagic) == NOFFMAGIC))
      SwapHeader(&noffH);
    ASSERT(noffH.noffMagic == NOFFMAGIC);

// how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
                + UserStackSize;  // we need to increase the size
                                  // to leave room for the stack
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;

    ASSERT(numPages <= NumPhysPages);           // check we're not trying
```

```
                                    // to run anything too big --
                                    // at least until we have
                                    // virtual memory

    DEBUG('a', "Initializing address space, num pages %d, size %d\n",
                                numPages, size);
// first, set up the translation
    pageTable = new TranslationEntry[numPages];
    for (i = 0; i < numPages; i++) {
      pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
      pageTable[i].physicalPage = i;
      pageTable[i].valid = TRUE;
      pageTable[i].use = FALSE;
      pageTable[i].dirty = FALSE;
      pageTable[i].readOnly = FALSE;  // if the code segment was entirely on
                                // a separate page, we could set its
                                // pages to be read-only
    }

// zero out the entire address space, to zero the unitialized data segment
// and the stack segment
    bzero(machine->mainMemory, size);

// then, copy in the code and data segments into memory
    if (noffH.code.size > 0) {
        DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
                  noffH.code.virtualAddr, noffH.code.size);
        executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
                  noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
                  noffH.initData.virtualAddr, noffH.initData.size);
        executable->ReadAt(&(machine-
>mainMemory[noffH.initData.virtualAddr]),
                  noffH.initData.size, noffH.initData.inFileAddr);
    }

}
```