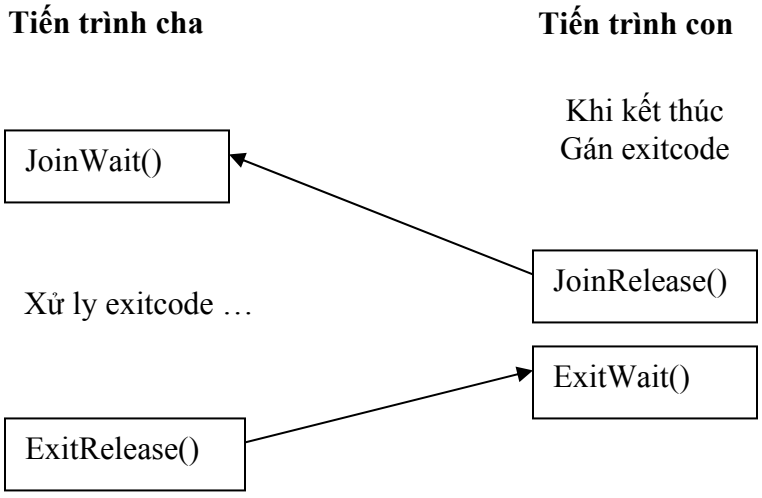
Hướng dẫn đồ án 2

Chúng ta dùng lớp Ptable để quản ly thông tin các tiến trình. Khai báo lớp Ptable có thể như bên dưới, chúng ta phải lập trình cho hoàn thiện lớp này. Trong lớp Ptable chúng ta quản ly 10 tiến trình, mỗi tiến trình lại được mô tả chi tiết, và các hành động của nó được cài đặt trong lớp PCB. Và class BitMap để lưu vết số lượng process hiện hành

Khai báo biến Ptable processTab toàn cục trong threads/system.h và khởi tạo trong threads/system.cc. (Trong đồ án 1, biến toàn cục FDTable fTab được khai báo trong lớp Thread. Vì chúng ta chỉ có 1 tiểu trình lúc đó)

Sơ lượt qua vòng đời của 1 process.

1. Nếu là tiến trình đầu tiên thì process ID của nó là 0.
2. Nếu là tiến trình con A, sau lời gọi system call Exec() thì nó gọi Join() vào tiến trình cha.Thật sự là việc báo cho tiến trình cha phải đợi cho tiến trình con A kết thúc mới thực thi tiếp. Như vậy, khi mà gọi system call Join() thì HĐH sẽ gọi A->JoinWait() (semaphore đợi cho đến khi A kết thúc) và đến khi A kết thúc bằng system call Exit() thì HĐH sẽ gọi A->JoinRelease(), và gọi A->ExitWait() để xin phép exit, HĐH lúc đó sẽ gọi A->ExitRelease() để cho phép tiến trình con kết thúc. Kết luận, trong chương trình, khi mà xử ly cho system call Join() thì HĐH gọi pcb[x]->JoinWait(), kế nhận mã exitcode để xử ly cần thiết rồi gọi pcb[x]->ExitRelease() để cho phép tiến trình con này kết thúc (pcb[x] = tiến trình mới tạo). Và khi tiến trình mới tạo này thoát bằng system call Exit() thì ta sẽ viết mã: pcb[x]->JoinRelease() và pcb[x]->ExitWait(), nghĩa là tiến trình con signal cho tiến trình cha thực thi tiếp, và nó phải đợi cho tiến trình cha cho phép nó thoát (gọi pcb[x]->ExitWait()), khi đó HĐH sẽ gọi pcb[x]->ExitRelease() (hàm mà ta viết trong sysctem call SC_Join)



```
#ifndef PTABLE_H
#define PTABLE_H
#include "bitmap.h"
#include "pcb.h"
#include "schandle.h"
#include "semaphore.h"

#define MAXPROCESS 10

class PTable {
 private:
  BitMap *bm;
  PCB    *pcb[MAXPROCESS];
  int     psize;
  Semaphore *bmsem; // dùng để ngăn chặn trường hợp nạp 2 tiến trình cùng lúc
 public:
  PTable(int size);
// 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();
// hủy các đối tượng đã tạo

  int ExecUpdate();// return PID
//Thực thi cho system call SC_EXEC, Kiểm tra chương trình được gọi có tồn tại trong máy không. Kiểm tra thử xem chương trình được gọi là chính nó không? Chúng ta không cho phép điều này. Kiểm tra còn slot trống để lưu tiến trình mới không (max là 10 process). Nếu thỏa các điều kiện trên thì ta lấy index của slot trống là ProcessID của tiến trình mới tạo này, giả sử là ID. Và gọi phương thức Exec của lớp PCB với đối tượng tương ứng quản ly process này, nghĩa là gọi pcb[ID]->Exec(…). Xem chi tiết mô tả lớp PCB ở bên dưới.
Phương thức này được gọi trong hàm xử ly system call SC_Exec

  int ExitUpdate(int ec);
//Được thủ tục sử ly cho system call SC_Exit sử dụng. Trong system call này chúng ta kiểm tra thử PID có tồn tại không, sau đó chúng ta mới xử ly kết thúc tiến trình, phải gọi JoinRelease() để giải phóng sự chờ đợi của tiến trình cha và ExitWait() để xin phép tiến trình cha cho kết thúc.

Chú y là sau khi gọi thủ tục này, chúng ta phải giải phóng tiểu trình hiện tại bằng các dòng lênh như sau, nếu là main process thì chúng ta gọi hàm Halt() luôn

  int pid = currentThread->processID;
```

```
  currentThread->FreeSpace();
  if(pid == 0){//exit main process
    interrupt->Halt();
  }
  currentThread->Finish();

  int JoinUpdate(int id);
// Được sử dụng khi mà chúng ta viết thủ tục cho system call Join(), trong thủ tục này thì tiến trình cha gọi pcb[id]->JoinWait() để chờ cho tới khi tiến trình con kết thúc (trước khi tiến trình con kết thúc thì tiến trình con gọi JoinRelease()). Sau đó tiến trình cha lại gọi ExitRelease() để cho phép tiến trình con được kết thúc.
Chú y là chúng ta không cho phép tiến trình join vào chính nó, hoặc là join vào tiến trình khác không phải là cha của nó.

  int GetFreeSlot();
//Tìm free slot để lưu thông tin cho tiến trình mới

bool IsExist(int pid);
//kiểm tra có tồn tại process ID này không

  void Remove(int pid);
//Xóa một processID ra khỏi mãng quản ly nó, khi mà tiến trình này đã kết thúc

};

#endif

#ifndef PCB_H
#define PCB_H
#include "thread.h"
#include "synch.h"
class PCB{
 private:
  Semaphore *joinsem;//semaphore cho quá trình join
  Semaphore *exitsem;//semaphore cho quá trình exit
  Semaphore *mutex;
  int       exitcode;
  Thread    *thread;
  int       pid;
  int       numwait ;// số tiến trình đã join

 public:
  int       parentID; //ID của tiến trình cha
  PCB(int id);
  ~PCB();
```

```
  int Exec(char *filename,int pid);// nạp chương trình có tên lưu trong biến filename và
//processID sẽ là pid
  int GetID() {return pid;}
  int GetNumWait();
  void JoinWait();
  void ExitWait();
  void JoinRelease();
  void ExitRelease();
  void IncNumWait();
  void DecNumWait();
  void SetExitCode(int ec){exitcode = ec;}
  int  GetExitCode(){return exitcode;}
};
#endif
```

Nạp chương trình trong môi tường đa chương

Khi nạp chương trình mới lên bộ nhớ, thì constructor của lớp AddrSpace có nhiệm vụ tính toán kích thước chương trình mới và tìm đủ vùng nhớ để nạp nó lên, tuy nhiên vì là môi trường đa chương, nên chúng ta phải thay đổi cách nạp chương trình cho phù hợp.

Giải thích các biến:
Input là tên của chương trình được lưu trong biến char *filename:
Sửa lại giá trị cho PageSize và NumPhysPages trong machine/machine.h. PageSize = 2*SectorSize (thay vì =SectorSize), và NumPhysPages = 256 (thay vì 32). Sở dĩ chúng ta phải tăng bộ nhớ lên, vì bay giờ máy tính có thể phải thực thi nhiều chương trình cùng lúc
Biến toàn cục **BitMap *gPhysPageBitmap**: là dùng để lưu các frame còn trống cũng như các frame đang sử dụng trên máy tính, mỗi khi nạp chương trình mới, chúng ta dựa thông tin trên đối tượng này để biết số frame còn trống đủ để nạp chương trình mới hay không.

```
AddrSpace::AddrSpace(char * filename)
{
   NoffHeader noffH;
   unsigned int i, size;
   OpenFile* executable = fileSystem->Open(filename);

   if (executable == NULL){
     printf("\nAddrspace::Error opening file:  %s",filename);
     DEBUG(dbgFile,"\n Error opening file.");
     return -1;
   }

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

```
        (WordToHost(noffH.noffMagic) == NOFFMAGIC))
      SwapHeader(&noffH);
    ASSERT(noffH.noffMagic == NOFFMAGIC);

    addrLock->Acquire();

//  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;

    // Check the available memory enough to load new process
    //debug
    //    printf("\n check enough memmory");

    if (numPages > số frame còn trống){
      printf("\nAddrSpace:Load: not enough memory for new process..!");
      numPages = 0;
      delete executable;
      addrLock->Release();
      return -1;
    }

    // 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 = frame còn trống, và đánh dấu frame này thành đã sử
dụng;
      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
      bzero(&(machine->mainMemory[pageTable[i].physicalPage*PageSize]), PageSize);

    }

    addrLock->Release();

// chúng ta nạp vào nộ nhớ từng trang 1, chứ không nạp cả chương trình như trước đây
    if(noffH.code.size > 0)
```

```
   for(i= 0 ; i < numPages ; i++){
     executable->ReadAt(&(machine-
>mainMemory[noffH.code.virtualAddr])+(pageTable[i].physicalPage*PageSize)     (địa
chỉ vật ly của trang),
                          PageSize,  noffH.code.inFileAddr+(i*PageSize)(xác  định  vị  trí
của trang thứ i trên file chương trình));
   }

// tương tự như trên cho phần nạp đoạn dữ liệu
if(noffH.initData.size > 0)
     for(i= 0 ; i < numPages ; i++){
          executable->ReadAt(&(machine-
>mainMemory[noffH.initData.virtualAddr])+(pageTable[i].physicalPage*PageSize),
                              PageSize, noffH.initData.inFileAddr+(i*PageSize));
     }

   delete executable;
}
```