Contents

1 code/test/start.S

Listing 1: *code/test/start.S*

```
/* Main */
    .globl __start
    .ent    __start
__start:
    jal     main
    move    $4,$0
    jal     UserThreadExit      /* if we return from main, we just need to exit the currentThread */
    .end __start

    .globl Halt
    .ent    Halt
Halt:
    addiu $2,$0,SC_Halt
    syscall
    j     $31
    .end Halt

    .globl Exit
    .ent    Exit
Exit:
    addiu $2,$0,SC_Exit
    syscall
    j     $31
    .end Exit

/* Exemple 1 */

    .globl PutString
    .ent PutString
PutString:
    addiu $2,$0,SC_PutString
    syscall
    j     $31
    .end PutString

/* Exemple 2 */

    .globl UserThreadCreate
    .ent UserThreadCreate
UserThreadCreate:
    addiu $2,$0,SC_UserThreadCreate
    addiu $6,$0,UserThreadExit // On passe en parametre la fonction de retour
    syscall
    j     $31
    .end UserThreadCreate
```

2 code/userprog/userthread.h

Listing 2: *code/userprog/userthread.h*

```
#ifndef USERTHREAD_H
#define USERTHREAD_H

#include "copyright.h"
#include "system.h"
#include "syscall.h"

class UserThread : public Thread {
    public:
        UserThread(const char *name, int f, int a, int callback);
        int func;
        int arg;
        void Fork (); // Make userthread run (*f)(arg)
        void UpdateCallBackRegister (int value); // $31 = value
};

extern int do_UserThreadCreate(int f, int arg, int callback);
extern void StartUserThread(int f);
extern void do_UserThreadExit();
extern int do_UserThreadJoin(int thread_id);

#endif
```

3 code/userprog/userthread.cc

Listing 3: *code/userprog/userthread.cc*

```
#include "userthread.h"
#include "forkprocess.h"


void StartUserThread(int thread) {
    UserThread *t = (UserThread *) thread;
    // L'id du thread informe egalement le num ro de page du thread
    currentThread->space->InitThreadRegisters(t->func, t->arg, t->getZone());
    currentThread->space->UpdateRunningThreads(1); // appel atomique
    machine->Run();
}

UserThread::UserThread(const char *debugName, int f, int a, int callback) : Thread(debugName) {
    this->func = f;
    this->arg = a;
    // A la fin du thread on appelle cette nouvelle fonction
    this->UpdateCallBackRegister(callback);
}

void UserThread::Fork () {
    DEBUG ('t', "Forking userThread \"%s\"\n", getName ());
    Thread::Fork (StartUserThread, (int) this);
}

void UserThread::UpdateCallBackRegister(int value) {
    this->userRegisters[31] = value;
}

int do_UserThreadCreate(int f, int arg, int callback) {
    UserThread* newThread = new UserThread((char*)f, f, arg, callback);
    if (newThread == NULL) { return -1; } // Erreur

    int zone = currentThread->space->GetNewZone();
    if (zone < 0) { delete newThread; return 0; }

    int thread_id = currentThread->space->GetNewThreadId(zone);
    if (thread_id < 0) { return 0; } // on verifie quand meme au cas ou

    newThread->setId(thread_id);
    newThread->setZone(zone);

    // Avant de commencer on prend le jetton, pour que tout thread qui appelle
    // userThreadJoin sur moi soit bloqu .
    // Ca permet egalement de ne pas lancer le thread tant que des thread sont
    // bloque sue le thread precedent qui utilisait cette zone
    currentThread->space->semJoinThreads[newThread->getZone()]->P();
    // Le nouveau thread s'execute sur le meme espace d'adressage que celui
    // qui fait l'appel systeme
    newThread->Fork();
    return newThread->getId();
}

void do_UserThreadExit() {
    // On d cremente le nombre de thread en cours d'execution
    currentThread->space->UpdateRunningThreads(-1); // appel atomique

    // Je libere les threads en attente sur moi
    currentThread->space->semJoinThreads[currentThread->getZone()]->V();
    // Plusieurs threads peuvent attendre que je me termine.
    // Il faut donc que dans la fonction join, les threads en attente se
    // reveillent les uns les autres (en chaine)

    // Mise a jour de la bitmap et de la map {thread-id <-> num ro zone}
    // On utilise cette structure pour ne pas avoir deux fois le meme numero de thread
    currentThread->space->FreeBitMap(); // appel atomique
    currentThread->space->RemoveId(currentThread->getZone()); // appel atomique

    // Si je suis le thread seul/dernier thread, je termine le processus
    if(currentThread->space->Alone()) {
        // Depuis l'etape 4 on appelle Exit() au lieu de Halt()
        // interrupt->Halt();
        do_Exit();
    }
    else {
        currentThread->Finish();
    }
}

int do_UserThreadJoin(int thread_id) {
    int zone = currentThread->space->GetZoneFromThreadId(thread_id);
    if (zone < 0)
        return zone;
    currentThread->space->semJoinThreads[zone]->P();
    // On reveille le suivant qui peut etre soit le prochain thread    qui
    // on a allouer la zone, soit un autre thread qui avait appeller join
    currentThread->space->semJoinThreads[zone]->V();
    return 0;
}
```

4 code/userprog/addspace.h

Listing 4: *code/userprog/addspace.h*

```
// addrspace.h
//      Data structures to keep track of executing user programs
//      (address spaces).
//
//      For now, we don't keep any information about address spaces.
//      The user level CPU state is saved and restored in the thread
//      executing the user program (see thread.h).
//
// Copyright (c) 1992-1993 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#ifndef ADDRSPACE_H
#define ADDRSPACE_H

#include "copyright.h"
#include "filesys.h"
#include "bitmap.h"
#include "synch.h"

#define UserStackSize        4096// increase this as necessary! (4k)
#define UserThreadNumPage    3 // 3 pages par thread


class AddrSpace
{
    public:
        AddrSpace (OpenFile * executable);
        // Create an address space,
        // initializing it with the program
        // stored in the file "executable"
        ~AddrSpace ();
        // De-allocate an address space

        // nombre de threads max
        const static int userMaxNumThread = (int) (UserStackSize / (UserThreadNumPage * PageSize));

        void InitRegisters ();
        // Initialize user-level CPU registers,
        // before jumping to user code

        void SaveState ();
        // Save/restore address space-specific
        void RestoreState ();
        // info on a context switch

        // Initialize user-level CPU registers,  before jumping to user
        // function f(arg)
        void InitThreadRegisters (int f, int arg, int thread_id);

        // Le nombre de thread en cours d'executions (   prot ger par un mutex)
        int runningThreads;
        // l'objet bitmap qui permet de trouver les zones libres pour les
        // nouveaux threads sans devoir g rer  a  nous meme..
        BitMap *stackBitMap;
        // Pour manipuler la variable runningThreads
        Semaphore *semRunningThreads;
        // Pour manipuler la bitmap
        Semaphore *semStackBitMap;
        // Pour permettre a un ou plusieurs threads de se bloquer en attendant
        // qu'un autre se termine
        Semaphore *semJoinThreads[userMaxNumThread];

        // Ces methodes permettent de manipuler les variables    prot ger d'une
        // utilisation multithread
        void UpdateRunningThreads(int i);
        // Permet de savoir si je suis le dernier thread
        int Alone();
        int GetNewZone();
        void FreeBitMap();

        // Permet de compter le nombre total de thread et donc d'avoir des ids
        // unique pour les threads
        int countThreads;
        // Ce tableau fait le mappage entre thread_id et num ro de la zone
        // correspondant    ce thread dans la pile
        int *threadZoneMap;
        // Pour manipuler les deux bitmap
        Semaphore *semThreadZoneMap;

        // Methodes qui permettent de manipuler les deux attributs pr c dent
        int GetNewThreadId(int zone);
        void RemoveId(int zone);
        int GetZoneFromThreadId(int thread_id);
        // Permet d'initialiser le thread main
        void InitMainThread();
        void ReleaseFrames();
        bool AvailFrames;
        bool ToBeDestroyed;

    private:
        TranslationEntry * pageTable;
        // Assume linear page table translation
        // for now!
        unsigned int numPages;
        // Number of pages in the virtual
        // address space
};

#endif // ADDRSPACE_H
```

5 code/userprog/addspace.cc

Listing 5: *code/userprog/addspace.cc*

```
// addrspace.cc
//      Routines to manage address spaces (executing user programs).
//
//      In order to run a user program, you must:
//
//      1. link with the -N -T 0 option
//      2. run coff2noff to convert the object file to Nachos format
//              (Nachos object code format is essentially just a simpler
//              version of the UNIX executable object code format)
//      3. load the NOFF file into the Nachos file system
//              (if you haven't implemented the file system yet, you
//              don't need to do this last step)
//
// Copyright (c) 1992-1993 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#include "copyright.h"
#include "system.h"
#include "addrspace.h"
#include "noff.h"

#include <strings.h>        /* for bzero */

//----------------------------------------------------------------------
// SwapHeader
//      Do little endian to big endian conversion on the bytes in the
//      object file header, in case the file was generated on a little
//      endian machine, and we're now running on a big endian machine.
//----------------------------------------------------------------------

static void
SwapHeader (NoffHeader * noffH)
{
    noffH->noffMagic = WordToHost (noffH->noffMagic);
    noffH->code.size = WordToHost (noffH->code.size);
    noffH->code.virtualAddr = WordToHost (noffH->code.virtualAddr);
    noffH->code.inFileAddr = WordToHost (noffH->code.inFileAddr);
    noffH->initData.size = WordToHost (noffH->initData.size);
    noffH->initData.virtualAddr = WordToHost (noffH->initData.virtualAddr);
    noffH->initData.inFileAddr = WordToHost (noffH->initData.inFileAddr);
    noffH->uninitData.size = WordToHost (noffH->uninitData.size);
    noffH->uninitData.virtualAddr =
    WordToHost (noffH->uninitData.virtualAddr);
    noffH->uninitData.inFileAddr = WordToHost (noffH->uninitData.inFileAddr);
}


void ReadAtVirtual( OpenFile *executable, int virtualaddr, int numBytes,
                    int position, TranslationEntry *pageTable,
                    unsigned int numPages ) {
    /* Ecriture dans la m moire virtuelle
    * On commence par initialis  les table de pages dans la machine
    * Ensuite on lit a partir de m moire physique pour recopie octet
    * par octet dans la m moire virtuelle (avec un buffer par s curit )
    */
    TranslationEntry * old_pageTable = machine->pageTable;
    unsigned int old_numPages = machine->pageTableSize;
    machine->pageTable = pageTable;
    machine->pageTableSize = numPages;
    //buffer to read the specified portion of executable
    char buffer[numBytes];
    //char * buffer = new char[numBytes];
```

```
    // On lit au plus numBytes octets
    int nb_read = executable->ReadAt(buffer, numBytes, position);
    // On ecrit dans la m moire virtuelle
    for (int i = 0; i < nb_read; i++)
        machine->WriteMem(virtualaddr+i, 1, buffer[i]);
    //delete buffer;
    // On restore le context
    machine->pageTable = old_pageTable;
    machine->pageTableSize = old_numPages;
}

//----------------------------------------------------------------------
// AddrSpace::AddrSpace
//      Create an address space to run a user program.
//      Load the program from a file "executable", and set everything
//      up so that we can start executing user instructions.
//
//      Assumes that the object code file is in NOFF format.
//
//      First, set up the translation from program memory to physical
//      memory.  For now, this is really simple (1:1), since we are
//      only uniprogramming, and we have a single unsegmented page table
//
//      "executable" is the file containing the object code to load into memory
//----------------------------------------------------------------------

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;

    // Le nombre de thread en cours d'executions (    prot ger par un mutex)
    this->runningThreads = 0; // 1 = le thread main//currentThread
    // l'objet bitmap qui permet de trouver les zones libres pour les
    // nouveaux threads, sans devoir g rer  a  nous meme..
    // Ici on gere X zones de `UserThreadNumPage` pages
    this->stackBitMap = new BitMap(this->userMaxNumThread);
    // Mutex pour manipuler la variable running_threads
    this->semRunningThreads = new Semaphore("semRunningThreads", 1);
    // Permet de prot ger la bitmap
    this->semStackBitMap = new Semaphore("semStackBitMap", 1);

    // On les initialise tous    1 jeton
    for (int j = 0; j<this->userMaxNumThread ; j++) {
        this->semJoinThreads[j] = new Semaphore("semJoinThread ", 1);
    }

    // Mise en place du tableau de mappage entre thread_ids et num ro de zone
    this->countThreads = 0;
    this->threadZoneMap = new int[this->userMaxNumThread];
    for(int j=0; j<this->userMaxNumThread; j++) {
        this->threadZoneMap[j] = -1;
    }

    this->semThreadZoneMap = new Semaphore("threadZoneMap", 1);

    DEBUG ('a', "Initializing address space, num pages %d, size %d\n",
        numPages, size);

    // first, set up the translation
    pageTable = new TranslationEntry[numPages];

    // NumAvailFrame == atomique
    int * frames = frameprovider->GetEmptyFrames((int) numPages);
    if (frames == NULL) {
        DEBUG ('p', "Pas suffisamment de memoire !\n");
        this->AvailFrames = false;
        return;
    } else {
        this->AvailFrames = true;
    }
    for (i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;
        // for now, virtual page # = phys page #
        pageTable[i].physicalPage = frames[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
    }
    // On supprime ce tableau car plus besoin..
    delete frames;

    // 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);
        ReadAtVirtual(executable, noffH.code.virtualAddr, noffH.code.size,
            noffH.code.inFileAddr, pageTable, numPages);
    }
    if (noffH.initData.size > 0) {
        DEBUG ('a', "Initializing data segment, at 0x%x, size %d\n",
            noffH.initData.virtualAddr, noffH.initData.size);
        ReadAtVirtual(executable, noffH.initData.virtualAddr,
          noffH.initData.size, noffH.initData.inFileAddr, pageTable, numPages);
    }
    this->ToBeDestroyed = false;
}

//----------------------------------------------------------------------
// AddrSpace::~AddrSpace
// Dealloate an address space.  Nothing for now!
//----------------------------------------------------------------------

AddrSpace::~AddrSpace ()
{
    // LB: Missing [] for delete
    ReleaseFrames();
    delete [] pageTable;
    delete [] threadZoneMap;
    delete stackBitMap;
    delete semRunningThreads;
    delete semStackBitMap;
    delete semThreadZoneMap;
    // End of modification
}

//----------------------------------------------------------------------
// AddrSpace::InitRegisters
//      Set the initial values for the user-level register set.
//
//      We write these directly into the "machine" registers, so
//      that we can immediately jump to user code.  Note that these
//      will be saved/restored into the currentThread->userRegisters
//      when this thread is context switched out.
//----------------------------------------------------------------------

void
AddrSpace::InitRegisters ()
{
    int i;
    for (i = 0; i < NumTotalRegs; i++)
    machine->WriteRegister (i, 0);

    // Initial program counter -- must be location of "Start"
    machine->WriteRegister (PCReg, 0);

    // Need to also tell MIPS where next instruction is, because
    // of branch delay possibility
    machine->WriteRegister (NextPCReg, 4);

    // Set the stack register to the end of the address space, where we
    // allocated the stack; but subtract off a bit, to make sure we don't
    // accidentally reference off the end!
    machine->WriteRegister (StackReg, numPages * PageSize - 16);
    DEBUG ('a', "Initializing stack register to %d\n",
       numPages * PageSize - 16);
}


void AddrSpace::InitThreadRegisters (int f, int arg, int thread_zone)
{
    machine->WriteRegister (PCReg, f);
    machine->WriteRegister (NextPCReg, f+4);
    // On ajoute l'arguments
    machine->WriteRegister (4, arg);
    // On se place sur la pile du thread
    int threadOffset = UserThreadNumPage * PageSize  * thread_zone;
    machine->WriteRegister (StackReg, numPages * PageSize - 16 - PageSize - threadOffset);

    DEBUG ('a', "Initializing thread stack register to %d\n",
       numPages * PageSize - 16 - threadOffset);
```

```
}

//----------------------------------------------------------------------
// AddrSpace::SaveState
//      On a context switch, save any machine state, specific
//      to this address space, that needs saving.
//
//      For now, nothing!
//----------------------------------------------------------------------

void
AddrSpace::SaveState ()
{
    pageTable = machine->pageTable;
    numPages = machine->pageTableSize;
}

//----------------------------------------------------------------------
// AddrSpace::RestoreState
//      On a context switch, restore the machine state so that
//      this address space can run.
//
//      For now, tell the machine where to find the page table.
//----------------------------------------------------------------------

void
AddrSpace::RestoreState ()
{
    machine->pageTable = pageTable;
    machine->pageTableSize = numPages;
}


void AddrSpace::UpdateRunningThreads(int value) {
    ASSERT (value != 1 || value !=-1);
    this->semRunningThreads->P();
    this->runningThreads += value;
    DEBUG ('t', "runningThread =  %d\n", runningThreads);
    this->semRunningThreads->V();
}

int AddrSpace::Alone() {
    int value = 0;
    this->semRunningThreads->P();
    if (this->runningThreads == 0)
        value = 1;
    this->semRunningThreads->V();
    return value;
}

int AddrSpace::GetNewZone() {
    int zone;
    this->semStackBitMap->P();
    zone = this->stackBitMap->Find();
    this->semStackBitMap->V();
    return zone;
}

void AddrSpace::FreeBitMap() {
    this->semStackBitMap->P();
    // On libere la zone
    this->stackBitMap->Clear(currentThread->getZone());
    this->RemoveId(currentThread->getZone());
    this->semStackBitMap->V();
}

int AddrSpace::GetNewThreadId(int zone) {
    this->semThreadZoneMap->P();
    int id = this->countThreads;
    threadZoneMap[zone]=id;
    this->countThreads++;
    this->semThreadZoneMap->V();
    return id;
}

int AddrSpace::GetZoneFromThreadId(int thread_id) {
    int zone = -1;
    this->semThreadZoneMap->P();
    for(int j = 0; j<userMaxNumThread; j++) {
        if (threadZoneMap[j] == thread_id) {
            if (this->stackBitMap->Test(j))
                zone = j;
            break;
        }
    }
    this->semThreadZoneMap->V();
    return zone;
}

void AddrSpace::RemoveId(int zone){
    this->semThreadZoneMap->P();
    threadZoneMap[zone]=-1;
    this->semThreadZoneMap->V();
}

void AddrSpace::InitMainThread() {
    this->UpdateRunningThreads(1); // appel atomique
    int zone = this->GetNewZone();
    currentThread->setZone(zone);
    currentThread->setId(this->GetNewThreadId(zone));
}


void AddrSpace::ReleaseFrames() {
    for (unsigned j = 0; j < this->numPages; j++) {
        frameprovider->ReleaseFrame(this->pageTable[j].physicalPage);
    }
}
```

6 code/userprog/exception.cc

Listing 6: *code/userprog/exception.cc*

```
// exception.cc
//      Entry point into the Nachos kernel from user programs.
//      There are two kinds of things that can cause control to
//      transfer back to here from user code:
//
//      syscall -- The user code explicitly requests to call a procedure
//      in the Nachos kernel.  Right now, the only function we support is
//      "Halt".
//
//      exceptions -- The user code does something that the CPU can't handle.
//      For instance, accessing memory that doesn't exist, arithmetic errors,
//      etc.
//
//      Interrupts (which can also cause control to transfer from user
//      code into the Nachos kernel) are handled elsewhere.
//
// For now, this only handles the Halt() system call.
// Everything else core dumps.
//
// Copyright (c) 1992-1993 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#include "copyright.h"
#include "system.h"
#include "syscall.h"
#include "synchconsole.h"
#include "userthread.h"
#include "forkprocess.h"

//----------------------------------------------------------------------
// UpdatePC : Increments the Program Counter register in order to resume
// the user program immediately after the "syscall" instruction.
//----------------------------------------------------------------------
static void
UpdatePC ()
{
    int pc = machine->ReadRegister (PCReg);
    machine->WriteRegister (PrevPCReg, pc);
    pc = machine->ReadRegister (NextPCReg);
    machine->WriteRegister (PCReg, pc);
    pc += 4;
    machine->WriteRegister (NextPCReg, pc);
}


//----------------------------------------------------------------------
// ExceptionHandler
//      Entry point into the Nachos kernel.  Called when a user program
//      is executing, and either does a syscall, or generates an addressing
//      or arithmetic exception.
//
//      For system calls, the following is the calling convention:
//
//      system call code -- r2
//              arg1 -- r4
//              arg2 -- r5
//              arg3 -- r6
//              arg4 -- r7
//
//      The result of the system call, if any, must be put back into r2.
//
// And don't forget to increment the pc before returning. (Or else you'll
```

```
// loop making the same system call forever!
//
//      "which" is the kind of exception.  The list of possible exceptions
//      are in machine.h.
//----------------------------------------------------------------------


char * ReadStringFromMachine(int from, unsigned max_size) {
  /* On copie octet par octet, de la m moire user vers la m moire noyau (buffer)
   * en faisant attention    bien convertir explicitement en char
   */
  int byte;
  unsigned int i;
  char * buffer = new char[max_size];
  for(i = 0; i < max_size-1; i++) {
    machine->ReadMem(from+i,1, &byte);
    if((char)byte=='\0')
      break;
    buffer[i] = (char) byte;
  }
  buffer[i] = '\0';
  return buffer;
}

void WriteStringToMachine(char * string, int to, unsigned max_size) {
  /* On copie octet par octet, en faisant attention    bien convertir
   * explicitement en char
   */
  char * bytes = (char *)(&machine->mainMemory[to]);
  for(unsigned int i = 0; i < max_size-1; i++) {
    bytes[i] = string[i];
    if(string[i]=='\0')
      break;
  }
}

void
ExceptionHandler (ExceptionType which)
{
    int type = machine->ReadRegister (2);

    if (which == SyscallException) {
        switch (type) {

        case SC_Halt: {
            DEBUG('a', "Shutdown, initiated by user program.\n");
            interrupt->Halt();
            break;
        }

        case SC_Exit: {
            DEBUG('p', "Explicit Exit, initiated by user program.\n");
            // Par defaut le thread main appel UserThreadExit et attend donc
            // les threads utilisateurs; mais Un appel explicite de Exit
            // n'attend aucun threads et quitte
            do_Exit();
            break;
        }

        case SC_PutChar: {
          DEBUG('a', "PutChar, initiated by user program.\n");
          synchconsole->SynchPutChar((char)(machine->ReadRegister(4)));
          break;
        }

        case SC_PutString: {
          DEBUG('a', "PutString, initiated by user program.\n");
          // Le premier argument (registre R4) c'est l'adresse de la chaine de caractere
          // Que l'ont recopie dans le monde linux (noyau)
          // R4 >> pointeur vers la m moire  MIPS
          // MAX_STRING_SIZE est defni prealablement dans code/threads/system.h
          char *buffer = ReadStringFromMachine(machine->ReadRegister(4), MAX_STRING_SIZE);
          synchconsole->SynchPutString(buffer);
          // /!\ big probleme qui fait planter Nachos
          //delete [] buffer;
          break;
        }

        case SC_GetChar: {
          DEBUG('a', "GetChar, initiated by user program.\n");
          machine->WriteRegister(2,(int) synchconsole->SynchGetChar());
          break;
        }

        case SC_GetString: {
          DEBUG('a', "GetString, initiated by user program.\n");

          // le premier argument est une adresse (char *)
          int to = machine->ReadRegister(4);
          // le second est un int >> la taille
          int size = machine->ReadRegister(5);
          // On donne pas acceder    la  m moire directement ,on ecrit ecrit dans
          // un buffer..
          // Peut etre pas oblig  , mais au cas ou on utilise un buffer..
          char buffer[MAX_STRING_SIZE];
          synchconsole->SynchGetString(buffer, size);
          WriteStringToMachine(buffer, to, size);
          break;
        }

        case SC_PutInt: {
          DEBUG('a', "PutInt, initiated by user program.\n");
          // le premier est la valeur int
          int value = machine->ReadRegister(4);
          synchconsole->SynchPutInt(value);
          break;
        }

        case SC_GetInt: {
          DEBUG('a', "GetInt, initiated by user program.\n");
          int value = synchconsole->SynchGetInt();
          machine->WriteRegister(2, value);
          break;
        }

        case SC_UserThreadCreate:
        {
          DEBUG('t', "UserThreadCreate, initiated by user program.\n");
          int f = machine->ReadRegister(4);
          int arg = machine->ReadRegister(5);
          int callback = machine->ReadRegister(6);
          int ret = do_UserThreadCreate(f,arg, callback);
          machine->WriteRegister(2,ret);
          break;
        }

        case SC_UserThreadExit:
        {
          DEBUG('t', "UserThreadExit, initiated by user program.\n");
          // Laisse les autres threads s'executer et attends jusqu'a ce qu'il se
          // termine tous
          do_UserThreadExit();
          break;
        }

        case SC_UserThreadJoin:
        {
          DEBUG('t', "UserThreadJoin, initiated by user program.\n");

          int thread_id = machine->ReadRegister(4);
          int ret = do_UserThreadJoin(thread_id);
          machine->WriteRegister(2,ret);
          break;
        }

        case SC_ForkExec:
        {
            DEBUG('p', "ForkExec, initiated by user program.\n");
            char *buffer = ReadStringFromMachine(machine->ReadRegister(4), MAX_STRING_SIZE);
            int ret = do_ForkExec(buffer);
            // On delete pas car le nom du fichier sert de nom pour le thread
            // main du nouveau processus, il sera delete deletea  la destruction du
            // thread
            // delete [] buffer;
            machine->WriteRegister(2,ret);
            break;
        }

        default: {
          printf("Unexpected user mode exception %d %d\n", which, type);
          ASSERT(FALSE);
        }
      }
    }


    // LB: Do not forget to increment the pc before returning!
    UpdatePC ();
    // End of addition
}
```

7 code/userprog/synconsole.h

Listing 7: *code/userprog/synconsole.h*

```
#ifndef SYNCHCONSOLE_H
#define SYNCHCONSOLE_H

#include "copyright.h"
#include "utility.h"
#include "console.h"

class SynchConsole {
    public:
        SynchConsole(char *readFile, char *writeFile);
        ~SynchConsole();
        void SynchPutChar(const char ch);
        char SynchGetChar();
        // Unix putchar(3S)
        // Unix getchar(3S)
        void SynchPutString(char *s); // Unix puts(3S)
        void SynchGetString(char *s, int n);
        void SynchGetString(char *s, int n, char delim);
        void SynchPutInt(int value);
        int SynchGetInt();
        Semaphore * putStringMutex;
        // Unix fgets(3S)
    private:
        Console *console;
};
#endif // SYNCHCONSOLE_H
```

8 code/userprog/synconsole.cc

Listing 8: *code/userprog/synconsole.cc*

```
#include "copyright.h"
#include "system.h"
#include "synchconsole.h"
#include "synch.h"


static Semaphore *readAvail;
static Semaphore *writeDone;
static Semaphore *writeMutex;
static Semaphore *readMutex;

static void ReadAvail(int arg) {
    readAvail->V();
}

static void WriteDone(int arg) {
    writeDone->V();
}

SynchConsole::SynchConsole(char *readFile, char *writeFile) {
    readAvail  = new Semaphore("read avail", 0);
    writeDone  = new Semaphore("write done", 0);
    writeMutex = new Semaphore("writeMutex", 1);
    readMutex  = new Semaphore("readMutex", 1);
    this->putStringMutex = new Semaphore("putStringMutex", 1);
    console    = new Console (readFile, writeFile, ReadAvail, WriteDone, 0);
}

SynchConsole::~SynchConsole() {
    delete console;
    delete writeDone;
    delete readAvail;
    delete writeMutex;
    delete readMutex;
    delete putStringMutex;
}

void SynchConsole::SynchPutChar(const char ch) {
    /* On  crit  un char on se bloque en attendant que la console appelle
     * (WriteDone>V())
     */
    writeMutex->P();
    console->PutChar(ch);
    writeDone->P();
    writeMutex->V();
}

char SynchConsole::SynchGetChar() {
    /* Lorsqu'il y a rien    lire, on se bloque, et d s qu'il y a quelques chose
     *    lire, on sait qu'on sera d bloqu  (ReadAvail->V())
     */
    readMutex->P();
    readAvail->P();
    return console->GetChar();
    readMutex->V();
}

void SynchConsole::SynchPutString(char * string) {
    /* On utilise un mutex pour que les appels SynchPutString soient atomiques
     * C'est    dire que deux appels    SynchPutString() affichent correctement
     * les chaines de caract res...
     * * */
    this->putStringMutex->P();
    for(int i=0; i<MAX_STRING_SIZE-1;i++) {
        if(string[i] == '\0')
            break;
        this->SynchPutChar(string[i]);
    }
    this->putStringMutex->V();
}

void SynchConsole::SynchGetString(char *buffer, int n, char delim) {
  /* On utilise un mutex pour que tous les appels SynchGetString soient
   * atomiques.
   * * */
  int i;
  char c;
  for (i=0; i<n-1; i++) {
    c = this->SynchGetChar();
    // CTRL+D pour arr ter la saisie
    if(c == delim)
      break;
    else
      buffer[i] = c;
  }
  buffer[i] = '\0';
}


void SynchConsole::SynchGetString(char *buffer, int n) {
  this->SynchGetString(buffer, n, EOF);
}

void SynchConsole::SynchPutInt(int value) {
  char * buffer = new char[MAX_STRING_SIZE];
  // on ecrit dans le buffer la valeur avec sprintf
  snprintf(buffer,MAX_STRING_SIZE, "%d", value);
  this->SynchPutString(buffer);
  delete [] buffer;
}

int SynchConsole::SynchGetInt() {
  int value;
  char * buffer = new char[MAX_STRING_SIZE];
  this->SynchGetString(buffer, MAX_STRING_SIZE, '\n');
  sscanf(buffer, "%d", &value);
  delete [] buffer;
  return value;
}
```

9 code/userprog/syscall.h

Listing 9: *code/userprog/syscall.h*

```
/* syscalls.h
 *  Nachos system call interface.  These are Nachos kernel operations
 *  that can be invoked from user programs, by trapping to the kernel
 *  via the "syscall" instruction.
 *
 *  This file is included by user programs and by the Nachos kernel.
 *
 * Copyright (c) 1992-1993 The Regents of the University of California.
 * All rights reserved.  See copyright.h for copyright notice and limitation
 * of liability and disclaimer of warranty provisions.
 */

#ifndef SYSCALLS_H
#define SYSCALLS_H

#include "copyright.h"

/* system call codes -- used by the stubs to tell the kernel which system call
 * is being asked for
 */
#define SC_Halt     0
#define SC_Exit     1
#define SC_Exec     2
#define SC_Join     3
#define SC_Create   4
```

```
#define SC_Open     5
#define SC_Read     6
#define SC_Write    7
#define SC_Close    8
#define SC_Fork     9
#define SC_Yield    10
#define SC_PutChar  11
#define SC_PutString    12
#define SC_GetChar  13
#define SC_GetString    14
#define SC_PutInt   15
#define SC_GetInt   16
#define SC_UserThreadCreate 17
#define SC_UserThreadExit 18
#define SC_UserThreadJoin 19
#define SC_ForkExec 20

#ifdef IN_USER_MODE

void Halt () __attribute__((noreturn));

[...]

void PutChar(char c);
void PutString(char *s);
char GetChar();
void GetString(char *buffer, int size);
void PutInt(int value);
int GetInt();

// Threads : etape 3

int UserThreadCreate(void * f, void *arg);
void UserThreadExit();
int UserThreadJoin(int thread_id);
int ForkExec(char * filename);

#endif // IN_USER_MODE

#endif /* SYSCALL_H */
```

10 code/userprog/forkprocess.cc

Listing 10: *code/userprog/forkprocess.cc*

```
#include "forkprocess.h"
#include "system.h"

void StartForkedProcess(int arg) {
    currentThread->space->RestoreState();
    currentThread->space->InitRegisters ();
    currentThread->space->InitMainThread();
    machine->Run();
}

int do_ForkExec (char *filename)
{
    OpenFile *executable = fileSystem->Open (filename);
    AddrSpace *space;

    if (executable == NULL) {
        printf ("Unable to open file %s\n", filename);
        delete [] filename;
        return -1;
    }
    // Creation d'un nouvel espace d'adressage
    space = new AddrSpace (executable);

    // Si c'est null ou qu'il n'y a pas assez de memoire on arrete la
    if (space == NULL || !space->AvailFrames) {
        printf("%s : Insufficient memory to start the process.\n",
                    filename);
        delete executable;
        delete [] filename;
        return -1;
    }
    delete executable;

    // Creation du nouveau thread main du nouveau processus
    Thread * mainThread = new Thread(filename);
    mainThread->space = space;
    machine->UpdateRunningProcess(1); // appel atomique
    mainThread->Fork (StartForkedProcess, 0);

    return 0;
}

void do_Exit() {
    DEBUG('p', "ExitProcess : %s", currentThread->getName());
    machine->UpdateRunningProcess(-1);
    if (machine->Alone()) {
        interrupt->Halt();
    }
    currentThread->space->ToBeDestroyed = true;
    currentThread->Finish();
}
```

11 code/userprog/frameprovider.cc

Listing 11: *code/userprog/frameprovider.cc*

```
#include <time.h>

#include "frameprovider.h"
#include "system.h"


FrameProvider::FrameProvider (int n) {
    this->lenght = n;
    this->bitmap = new BitMap(this->lenght);
    this->semFrameBitMap = new Semaphore("semFrameBitMap", 1);
}

FrameProvider::~FrameProvider () {
    delete bitmap;
}

void FrameProvider::ReleaseFrame(int n) {
    this->semFrameBitMap->P();
    this->bitmap->Clear(n);
    this->semFrameBitMap->V();
}

int * FrameProvider::GetEmptyFrames(int n) {
    RandomInit(0);
    this->semFrameBitMap->P();
    int * frames = NULL;
    if (n <= this->bitmap->NumClear()) {
        frames = new int[n];
        for(int i=0; i<n; i++) {
            int frame = Random()%NumPhysPages;
            // Recherche d'une page libre
            while(this->bitmap->Test(frame)) {
                frame = Random()%NumPhysPages;
            }
            this->bitmap->Mark(frame);
            bzero(&(machine->mainMemory[ PageSize * frame ] ), PageSize );
            frames[i] = frame;
        }
    }
    this->semFrameBitMap->V();
    return frames;
}
```

12 code/filesys/directory.cc

Listing 12: *code/filesys/directory.cc*

```

Directory::Directory(int size) {
    table = new DirectoryEntry[size];
    tableSize = size;
    for (int i = 0; i < tableSize; i++)
        table[i].inUse = false;
    int sector = 1;
    int parentSector = 1; // pas de parent pour la racine
    makeDirHierarchy(sector, parentSector);
}

Directory::Directory(int size, int sector, int parentSector) {
    table = new DirectoryEntry[size];
    tableSize = size;
    for (int i = 2; i < tableSize; i++)
        table[i].inUse = false;
```

```
    makeDirHierarchy(sector, parentSector);
}

Directory::~Directory()
{
    delete [] table;
}

//----------------------------------------------------------------------
// Directory::FetchFrom
// Read the contents of the directory from disk.
//
// "file" -- file containing the directory contents
//----------------------------------------------------------------------

void Directory::FetchFrom(OpenFile *file)
{
    (void) file->ReadAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
}

//----------------------------------------------------------------------
// Directory::WriteBack
//     Write any modifications to the directory back to disk
//
//    "file" -- file to contain the new directory contents
//----------------------------------------------------------------------

void
Directory::WriteBack(OpenFile *file)
{
    (void) file->WriteAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
}

//----------------------------------------------------------------------
// Directory::FindIndex
//     Look up file name in directory, and return its location in the table of
//    directory entries.  Return -1 if the name isn't in the directory.
//
//    "name" -- the file name to look up
//----------------------------------------------------------------------

int Directory::FindIndex(const char *name)
{
    for (int i = 0; i < tableSize; i++)
        if (table[i].inUse && !strncmp(table[i].name, name, FileNameMaxLen))
            return i;
    return -1;        // name not in directory
}

//----------------------------------------------------------------------
// Directory::Find
//     Look up file name in directory, and return the disk sector number
//    where the file's header is stored. Return -1 if the name isn't
//    in the directory.
//
//    "name" -- the file name to look up
//----------------------------------------------------------------------

int Directory::Find(const char *name)
{
    int i = FindIndex(name);

    if (i != -1)
        return table[i].sector;
    return -1;
}

//----------------------------------------------------------------------
// Directory::Add
//     Add a file into the directory.  Return TRUE if successful;
//    return FALSE if the file name is already in the directory, or if
//    the directory is completely full, and has no more space for
//    additional file names.
//
//    "name" -- the name of the file being added
//    "newSector" -- the disk sector containing the added file's header
//----------------------------------------------------------------------

bool
Directory::Add(const char *name, int newSector)
{
    if (FindIndex(name) != -1)
    return FALSE;

    for (int i = 2; i < tableSize; i++)
        if (!table[i].inUse) {
            table[i].inUse = TRUE;
            strncpy(table[i].name, name, FileNameMaxLen);
            table[i].sector = newSector;
            return TRUE;
        }
    return FALSE;    // no space.  Fix when we have extensible files.
}

//----------------------------------------------------------------------
// Directory::Remove
//     Remove a file name from the directory.  Return TRUE if successful;
//    return FALSE if the file isn't in the directory.
//
//    "name" -- the file name to be removed
//----------------------------------------------------------------------

bool
Directory::Remove(const char *name)
{
    int i = FindIndex(name);

    if (i == -1)
        return FALSE;          // name not in directory
    table[i].inUse = FALSE;
    return TRUE;
}

bool Directory::Remove(int sector)
{
    for (int i = 2; i < tableSize; i++)
        if (table[i].sector == sector) {
            table[i].inUse = false;
            return true;
        }
    return false;
}

//----------------------------------------------------------------------
// Directory::List
//     List all the file names in the directory.
//----------------------------------------------------------------------

void
Directory::List()
{
   for (int i = 0; i < tableSize; i++)
    if (table[i].inUse)
        printf("%s\n", table[i].name);
}

//----------------------------------------------------------------------
// Directory::Print
//     List all the file names in the directory, their FileHeader locations,
//    and the contents of each file.  For debugging.
//----------------------------------------------------------------------

void
Directory::Print()
{
    FileHeader *hdr = new FileHeader;

    printf("Directory contents:\n");
    for (int i = 0; i < tableSize; i++)
    if (table[i].inUse) {
        printf("Name: %s, Sector: %d\n", table[i].name, table[i].sector);
        hdr->FetchFrom(table[i].sector);
        hdr->Print();
    }
    printf("\n");
    delete hdr;
}

void Directory::makeDirHierarchy(int sector, int parentSector) {
    // Ajout les dossiers "." et ".."
    table[0].inUse = true;
    table[0].sector = sector;
    strcpy(table[0].name, ".");

    table[1].inUse = true;
    table[1].sector = parentSector;
    strcpy(table[1].name, "..");
}


bool Directory::isFull() {
  for (int i = 2; i < tableSize; i++) {
    if(table[i].inUse == false)
      return false;
  }
  return true;
}

//return true if there is nothing in the directory
bool Directory::isEmpty() {
  //we don't check the "." and ".." entries
  for (int i = 2; i < tableSize; i++) {
```

```
    if (table[i].inUse == true)
      return false;
  }
  return true;
}


bool Directory::isRoot() {
    // ". == ".."
    return (table[0].sector == table[1].sector);
};

char * Directory::getNameFromSector(int sector) {
    for (int i = 2; i < tableSize; i++)
        if (table[i].sector == sector) {
            return table[i].name;
        }
    return NULL;
};

int Directory::getSector(int position) {
    return this->table[position].sector;
}

int Directory::getCurrentSector() {
    return this->getSector(0);
}

int Directory::getParentSector() {
    return this->getSector(1);
}

// Affiche le nom du fichier complet exemple : /dossier1/test/image.jpg
char * Directory::getDirName() {
    Directory * currentDir = this;
    Directory * parentDir;
    int parentSector;
    char * fullname = new char[MAX_DIRNAME_SIZE];
    char * temp = new char[MAX_DIRNAME_SIZE];
    strcpy(fullname, "/");
    char * currentName = new char [MAX_DIRNAME_SIZE];
    while (!currentDir->isRoot()) {
        parentSector = currentDir->getParentSector();
        OpenFile * parentDirFile = new OpenFile(parentSector);
        parentDir = new Directory(this->tableSize);
        parentDir->FetchFrom(parentDirFile);
        currentName = parentDir->getNameFromSector(currentDir->getCurrentSector());

        strcpy(temp, "/");
        strcat(temp, currentName);
        strcat(temp, fullname);
        strcpy(fullname, temp);
        currentDir = parentDir;
    }
    return fullname;
}
```

13 code/filesys/filehdr.cc

Listing 13: *code/filesys/filehdr.cc*

```
bool FileHeader::Allocate(BitMap *freeMap, int fileSize)
{
    numBytes = fileSize;
    numSectors  = divRoundUp(FileLength(), SectorSize);
    if (freeMap->NumClear() < numSectors)
        return FALSE;// not enough space

    int * indirectList;
    int allocatedSectors = 0;
    int i;
    int j;
    for (i = 0; i < (int) NumDirect && allocatedSectors < (int) numSectors; i++) {
        dataSectors[i] = freeMap->Find();
        indirectList = new int[NumIndirect];
        for (j=0; (j < (int) NumIndirect) && (allocatedSectors < numSectors); j++) {
            indirectList[j] = freeMap->Find();
            allocatedSectors++;
        }
        synchDisk->WriteSector(dataSectors[i], (char *)indirectList);
    }

    return TRUE;
}
```

```
//----------------------------------------------------------------------
// FileHeader::Deallocate
//  De-allocate all the space allocated for data blocks for this file.
//
//  "freeMap" is the bit map of free disk sectors
//----------------------------------------------------------------------

void
FileHeader::Deallocate(BitMap *freeMap)
{
    int * indirectList;
    int deallocatedSectors = 0;
    int i;
    int j;
    for (i = 0; i < (int) NumDirect && deallocatedSectors < (int) numSectors; i++) {
        ASSERT(freeMap->Test((int) dataSectors[i]));  // ought to be marked!

        indirectList = new int[NumIndirect];
        synchDisk->ReadSector(dataSectors[i], (char *)indirectList);

        for (j=0; (j < (int) NumIndirect) && (deallocatedSectors < numSectors); j++) {
            ASSERT(freeMap->Test((int) indirectList[j]));
            deallocatedSectors++;
        }
        freeMap->Clear((int) dataSectors[i]);
    }
}

//----------------------------------------------------------------------
// FileHeader::FetchFrom
//  Fetch contents of file header from disk.
//
//  "sector" is the disk sector containing the file header
//----------------------------------------------------------------------

void
FileHeader::FetchFrom(int sector)
{
    synchDisk->ReadSector(sector, (char *)this);
}

//----------------------------------------------------------------------
// FileHeader::WriteBack
//  Write the modified contents of the file header back to disk.
//
//  "sector" is the disk sector to contain the file header
//----------------------------------------------------------------------

void
FileHeader::WriteBack(int sector)
{
    synchDisk->WriteSector(sector, (char *)this);
}

//----------------------------------------------------------------------
// FileHeader::ByteToSector
//  Return which disk sector is storing a particular byte within the file.
//      This is essentially a translation from a virtual address (the
//  offset in the file) to a physical address (the sector where the
//  data at the offset is stored).
//
//  "offset" is the location within the file of the byte in question
//----------------------------------------------------------------------

int
FileHeader::ByteToSector(int offset)
{
    int sector = offset / SectorSize;
    int numList = sector / NumIndirect;
    int posInList = sector % NumIndirect;

    int * indirectList = new int[NumIndirect];
    synchDisk->ReadSector(dataSectors[numList], (char *)indirectList);

    return(indirectList[posInList]);
}

//----------------------------------------------------------------------
// FileHeader::FileLength
//  Return the number of bytes in the file.
//----------------------------------------------------------------------

int
FileHeader::FileLength()
{
    return abs(numBytes);
}

bool FileHeader::isDirectoryHeader()
{
    return (numBytes < 0);
```

```
}
```

14 code/filesys/filesys.cc

Listing 14: *code/filesys/filesys.cc*

```
#include "copyright.h"

#include "disk.h"
#include "bitmap.h"
#include "directory.h"
#include "filehdr.h"
#include "filesys.h"

// Sectors containing the file headers for the bitmap of free sectors,
// and the directory of files.  These file headers are placed in well-known
// sectors, so that they can be located on boot-up.
#define FreeMapSector          0
#define DirectorySector        1

// Initial file sizes for the bitmap and directory; until the file system
// supports extensible files, the directory size sets the maximum number
// of files that can be loaded onto the disk.
#define FreeMapFileSize        (NumSectors / BitsInByte)
#define NumDirEntries           10
#define DirectoryFileSize       (sizeof(DirectoryEntry) * NumDirEntries * 1)
#define MAX_PATH_DEPTH 20
#define MAX_DIRNAME_SIZE 150
//----------------------------------------------------------------------
// FileSystem::FileSystem
//      Initialize the file system.  If format = TRUE, the disk has
//     nothing on it, and we need to initialize the disk to contain
//     an empty directory, and a bitmap of free sectors (with almost but
//     not all of the sectors marked as free).
//
//     If format = FALSE, we just have to open the files
//     representing the bitmap and the directory.
//
//     "format" -- should we initialize the disk?
//----------------------------------------------------------------------

// Parse le path
void parse_path(char *buffer, char** args, int *nargs)
{
    char *buf_args[MAX_PATH_DEPTH];
    char **cp;
    char *wbuf;
    int i, j;

    wbuf=buffer;
    buf_args[0]=buffer;
    args[0] = buffer;

    for(cp=buf_args; (*cp=strsep(&wbuf, "/")) != NULL ;){
        if ((*cp != '\0') && (++cp >= &buf_args[MAX_PATH_DEPTH]))
            break;
    }

    for (j=i=0; buf_args[i]!=NULL; i++){
        if(strlen(buf_args[i])>0)
            args[j++]=buf_args[i];
    }

    *nargs=j;
    args[j]=NULL;
}


FileSystem::FileSystem(bool format)
{
    DEBUG('f', "Initializing the file system.\n");
    if (format) {
        BitMap *freeMap = new BitMap(NumSectors);
        Directory *directory = new Directory(NumDirEntries);
    FileHeader *mapHdr = new FileHeader;
    FileHeader *dirHdr = new FileHeader;

        DEBUG('f', "Formatting the file system.\n");

    // First, allocate space for FileHeaders for the directory and bitmap
    // (make sure no one else grabs these!)
    freeMap->Mark(FreeMapSector);
    freeMap->Mark(DirectorySector);

    // Second, allocate space for the data blocks containing the contents
    // of the directory and bitmap files.  There better be enough space!

    ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
    ASSERT(dirHdr->Allocate(freeMap, -1 * DirectoryFileSize));

    // Flush the bitmap and directory FileHeaders back to disk
    // We need to do this before we can "Open" the file, since open
    // reads the file header off of disk (and currently the disk has garbage
    // on it!).

        DEBUG('f', "Writing headers back to disk.\n");
    mapHdr->WriteBack(FreeMapSector);
    dirHdr->WriteBack(DirectorySector);

    // OK to open the bitmap and directory files now
    // The file system operations assume these two files are left open
    // while Nachos is running.

        freeMapFile = new OpenFile(FreeMapSector);
        directoryFile = new OpenFile(DirectorySector);

    // Once we have the files "open", we can write the initial version
    // of each file back to disk.  The directory at this point is completely
    // empty; but the bitmap has been changed to reflect the fact that
    // sectors on the disk have been allocated for the file headers and
    // to hold the file data for the directory and bitmap.

        DEBUG('f', "Writing bitmap and directory back to disk.\n");
    freeMap->WriteBack(freeMapFile);     // flush changes to disk
    directory->WriteBack(directoryFile);

    if (DebugIsEnabled('f')) {
        freeMap->Print();
        directory->Print();

        delete freeMap;
    delete directory;
    delete mapHdr;
    delete dirHdr;
    }
    } else {
    // if we are not formatting the disk, just open the files representing
    // the bitmap and directory; these are left open while Nachos is running
        freeMapFile = new OpenFile(FreeMapSector);
        directoryFile = new OpenFile(DirectorySector);
    }
    workingDir = new char[MAX_DIRNAME_SIZE];
}

//----------------------------------------------------------------------
// FileSystem::Create
//     Create a file in the Nachos file system (similar to UNIX create).
//    Since we can't increase the size of files dynamically, we have
//    to give Create the initial size of the file.
//
//    The steps to create a file are:
//      Make sure the file doesn't already exist
//        Allocate a sector for the file header
//       Allocate space on disk for the data blocks for the file
//      Add the name to the directory
//      Store the new file header on disk
//      Flush the changes to the bitmap and the directory back to disk
//
//    Return TRUE if everything goes ok, otherwise, return FALSE.
//
//     Create fails if:
//           file is already in directory
//         no free space for file header
//         no free entry for file in directory
//         no free space for data blocks for the file
//
//     Note that this implementation assumes there is no concurrent access
//    to the file system!
//
//    "name" -- name of file to be created
//    "initialSize" -- size of file to be created
//----------------------------------------------------------------------

bool
FileSystem::Create(const char *name, int initialSize)
{
    Directory *directory;
    BitMap *freeMap;
    FileHeader *hdr;
    int sector;
    bool success;

    DEBUG('f', "Creating file %s, size %d\n", name, initialSize);

    directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
```

```
    if (directory->Find(name) != -1)
      success = FALSE;            // file is already in directory
    else {
        freeMap = new BitMap(NumSectors);
        freeMap->FetchFrom(freeMapFile);
        sector = freeMap->Find();     // find a sector to hold the file header
        if (sector == -1)
            success = FALSE;         // no free block for file header
        else if (!directory->Add(name, sector))
            success = FALSE;     // no space in directory
        else {
                hdr = new FileHeader;
            if (!hdr->Allocate(freeMap, initialSize))
                    success = FALSE;     // no space on disk for data
            else {
                success = TRUE;
            // everthing worked, flush all changes back to disk
                hdr->WriteBack(sector);
                directory->WriteBack(directoryFile);
                freeMap->WriteBack(freeMapFile);
            }
            delete hdr;
        }
        delete freeMap;
    }
    delete directory;
    return success;
}

//----------------------------------------------------------------------
// FileSystem::Open
//     Open a file for reading and writing.
//    To open a file:
//      Find the location of the file's header, using the directory
//      Bring the header into memory
//
//    "name" -- the text name of the file to be opened
//----------------------------------------------------------------------

OpenFile *
FileSystem::Open(const char *name)
{
    Directory *directory = new Directory(NumDirEntries);
    OpenFile *openFile = NULL;
    int sector;
    DEBUG('f', "Opening file %s\n", name);
    directory->FetchFrom(directoryFile);
    sector = directory->Find(name);
    if (sector >= 0)
    openFile = new OpenFile(sector);    // name was found in directory
    delete directory;
    return openFile;                // return NULL if not found
}

Directory * FileSystem::CurrentDir()
{
    Directory *directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
    return directory;
}

//----------------------------------------------------------------------
// FileSystem::List
//     List all the files in the file system directory.
//----------------------------------------------------------------------

void FileSystem::List()
{

    Directory *directory = this->CurrentDir();
    directory->List();
    delete directory;
}

void FileSystem::List(char * name)
{
    int currentSector = this->CurrentDir()->getCurrentSector();
    if (this->MoveToLastDir(name) != -1) {
        int sector = this->CurrentDir()->Find(name);
        OpenFile * remoteFile = new OpenFile(sector);
        // Si c'est un dossier on liste son contenu
        if (remoteFile->isDirectoryFile()) {
            this->MoveToDir(name);
            this->List();
        } else {
            // On affiche son nom si c'est fichier

            printf("Name : %s\tLength : %d Bytes\n", name, remoteFile->Length());
        }
        this->MoveToSector(currentSector);
    }
}

//----------------------------------------------------------------------
// FileSystem::Print
//      Print everything about the file system:
//        the contents of the bitmap
//        the contents of the directory
//        for each file in the directory,
//            the contents of the file header
//            the data in the file
//----------------------------------------------------------------------

void FileSystem::Print()
{
    FileHeader *bitHdr = new FileHeader;
    FileHeader *dirHdr = new FileHeader;
    BitMap *freeMap = new BitMap(NumSectors);
    Directory *directory = new Directory(NumDirEntries);

    printf("Bit map file header:\n");
    bitHdr->FetchFrom(FreeMapSector);
    bitHdr->Print();

    printf("Directory file header:\n");
    dirHdr->FetchFrom(DirectorySector);
    dirHdr->Print();

    freeMap->FetchFrom(freeMapFile);
    freeMap->Print();

    directory->FetchFrom(directoryFile);
    directory->Print();

    delete bitHdr;
    delete dirHdr;
    delete freeMap;
    delete directory;
}

//----------------------------------------------------------------------
// FileSystem::Remove
//      Delete a file from the file system.  This requires:
//         Remove it from the directory
//         Delete the space for its header
//         Delete the space for its data blocks
//         Write changes to directory, bitmap back to disk
//
//    Return TRUE if the file was deleted, FALSE if the file wasn't
//    in the file system.
//
//    "name" -- the text name of the file to be removed
//----------------------------------------------------------------------

bool FileSystem::Remove(char *name)
{
    bool error = false;
    int currentSector = this->CurrentDir()->getCurrentSector();
    if (this->MoveToLastDir(name) == -1)
        return true;
    // si je fais rm / > name = "\0"
    if (strcmp(name, "\0") == 0) {
        printf("rm: impossible supprimer le r pertoire  /  \n");
        return false;
    }

    Directory *directory = this->CurrentDir();
    BitMap *freeMap;
    FileHeader *fileHdr;
    int sector;

    sector = directory->Find(name);
    if (sector == -1) {
       printf("rm: le fichier ou dossier  %s  n'existe pas\n", name);
       error = true;
    }

    fileHdr = new FileHeader;

    if (!error) {
        fileHdr->FetchFrom(sector);
        // Si le fichier a supprimer est un dossier
        if (fileHdr->isDirectoryHeader()) {
            OpenFile * removeDirFile = new OpenFile(sector);
            Directory * removeDir = new Directory(NumDirEntries);
            removeDir->FetchFrom(removeDirFile);
            if (!removeDir->isEmpty()) {
                delete removeDirFile;
                delete removeDir;
                printf("rm : le dossier n'est pas vide\n");
                error = true;
```

```
            } else if (removeDir->isRoot()) {
                delete removeDirFile;
                delete removeDir;
                printf("rm: impossible supprimer le r pertoire  /  \n");
                error = true;
            }
            // si je me retrouve dans le dossier actuellement je remonte au parent
            // rm .
            else if (sector == currentSector) {
                currentSector = removeDir->getParentSector();
                MoveToSector(currentSector);
                directory = this->CurrentDir();
                delete removeDirFile;
                delete removeDir;
            }
            // si je fais un : rm /dir1/dir2/dir3/.
            // on se retrouve alors dans dir3 entrain de supprimer "."
            // et il faut revenir   / le dossier courant
            else if (sector == removeDir->getCurrentSector()) {
                MoveToSector(removeDir->getParentSector());
                directory = this->CurrentDir();
                delete removeDirFile;
                delete removeDir;
            }
        }
    }
    if (!error) {
        freeMap = new BitMap(NumSectors);
        freeMap->FetchFrom(freeMapFile);

        directoryFile = new OpenFile(directory->getCurrentSector());

        // suppression
        fileHdr->Deallocate(freeMap);          // remove data blocks
        freeMap->Clear(sector);            // remove header block
        directory->Remove(sector);

        // sauvegarde en m moire persistante
        freeMap->WriteBack(freeMapFile);        // flush to disk
        directory->WriteBack(directoryFile);        // flush to disk
        delete freeMap;
    }
    delete fileHdr;
    delete directory;
    this->MoveToSector(currentSector);

    return TRUE;
}

bool FileSystem::Exist(char * name) {
    Directory *currentDir = this->CurrentDir();
    int dirSector = currentDir->Find(name);
    if (dirSector == -1) {
        delete currentDir;
        return false;
    }
    delete currentDir;
    return true;
}

int FileSystem::MakeDir(char *name) {
    bool error = false;
    int originalSector = this->CurrentDir()->getCurrentSector();
    if (this->MoveToLastDir(name) == -1)
        return -1;
    // si je fais mkdir / > name = "\0"
    if (strcmp(name, "\0") == 0) {
        printf("mkdir: impossible de cr er le r pertoire  /  : Le fichier existe\n");
        return -1;
    }

    Directory *currentDir = this->CurrentDir();

    if(strlen(name) > FileNameMaxLen) {
        printf("mkdir: nom de fichier trop long\n");
        error = true;
    }

    if (!error && currentDir->Find(name) != -1) {
       printf("mkdir: impossible de cr er le r pertoire  %s%s  : Le fichier existe\n",
                                                currentDir->getDirName(), name);
       error = true;
    }

    if (!error && currentDir->isFull()) {
        printf("mkdir: le dossier %s est plein\n", currentDir->getDirName());
        error = true;
    }

    BitMap *freeMap = new BitMap(NumSectors);
    int freeSector;
    if (!error) {
        freeMap->FetchFrom(freeMapFile);
        freeSector = freeMap->Find();
        if (freeSector == -1) {
            printf("mkdir: plus de secteurs libres\n");
            error = true;
        }
    }

    if (!error) {
        int currentSector = currentDir->getCurrentSector();
        // Ajout du dossier dans le dossier courant (le parent)
        currentDir->Add(name, freeSector);

        // Cr ation du header du nouveau dossier
        FileHeader *newDirHeader = new FileHeader;
        // - DirectoryFileSize pour detecter que c'est un dossier
        ASSERT(newDirHeader->Allocate(freeMap, -1 * DirectoryFileSize));
        // Ecriture en m moire
        newDirHeader->WriteBack(freeSector);

        // creation du dossier avec le bon parent
        Directory *newDir = new Directory(NumDirEntries, freeSector, currentSector);
        // Ouverture du DirFile pour savegarder le dossier
        OpenFile *newDirFile = new OpenFile(freeSector);
        // Ecriture en m moire
        newDir->WriteBack(newDirFile);
        // Sauvegarde du dossier courant
        currentDir->WriteBack(directoryFile);
        // Sauvegarde de la freemap
        freeMap->WriteBack(freeMapFile);

        delete freeMap;
        delete newDirHeader;
        delete newDirFile;
        delete newDir;
    }
    this->MoveToSector(originalSector);
    delete currentDir;
    return 0;
}

int FileSystem::ChangeDir(char *name) {

    // Pour ne pas devoir reecrire toute les fonction on utilise une fonction
    // qui permet d'aller    l'avant dernier dossier dans le path, puis de faire
    // l'operation
    if (this->MoveToLastDir(name) == -1)
        return -1;
    if (strcmp(name, "\0") == 0)
        return 0;
    return this->MoveToDir(name);
}

int FileSystem::MoveToDir(char *name) {
    Directory *currentDir = this->CurrentDir();
    int dirSector = currentDir->Find(name);
    if (dirSector == -1) {
        printf("Le dossier %s%s n'existe pas\n", CurrentDir()->getDirName(), name);
        delete currentDir;
        return -1;
    }
    int result = MoveToSector(dirSector);
    if (result == -1)
        printf("%s%s n'est pas un dossier\n", CurrentDir()->getDirName(), name);
    delete currentDir;
    return result;
}

int FileSystem::MoveToSector (int dirSector) {
    OpenFile * newDirectoryFile = new OpenFile(dirSector);
    if (!newDirectoryFile->isDirectoryFile()) {
        delete newDirectoryFile;
        return -1;
    }
    directoryFile = newDirectoryFile;
    return 0;
}

int FileSystem::MoveToRoot() {
    return this->MoveToSector(1);
}

int FileSystem::MoveToLastDir(char * name)
{
    if (strcmp(name, "/") == 0) {
        this->MoveToRoot();
        strcpy(name, "\0");
        return 0;
    }
    if (name[0] == '/') {
```

```
        this->MoveToRoot();
    }
    char *paths[MAX_PATH_DEPTH];
    int npath;
    int i;
    parse_path(name, paths, &npath);
    if (npath > 0) {
        for(i = 0; i < npath-1; i++) {
            // On ignore les deplacement dans "." pour optimiser
            if (strcmp(paths[i], ".") != 0) {
                if (this->MoveToDir(paths[i]) != 0) {
                    return -1;
                }
            }
        }
        strcpy(name, paths[i]);
    }
    return 0;
}

int FileSystem::MakeParentDir(char *name) {
    bool error = false;
    int currentSector = this->CurrentDir()->getCurrentSector();
    if (strcmp(name, "/") == 0) {
        return this->MakeDir(name);
    }
    if (name[0] == '/') {
        this->MoveToRoot();
    }
    char *paths[MAX_PATH_DEPTH];
    int npath;
    int i;
    parse_path(name, paths, &npath);
    for(i = 0; i < npath;) {
        // On ignore les "." pour optimiser
        if (this->Exist(paths[i])) {
            if (this->ChangeDir(paths[i]) != 0) {
                error = true;
                break;
            }
            i++;
        } else {
            if (this->MakeDir(paths[i]) != 0) {
                error = true;
                break;
            }
            // Pour eviter les boucles infinies si le fonctionement attendu des
            // fonction n'est pas le bon, on change de dossier ici
            if (this->Exist(paths[i])) {
                if (this->ChangeDir(paths[i]) != 0) {
                    error = true;
                    break;
                }
            }
            else {
                error = true;
                break;
            }
            i++;
        }
    }
    this->MoveToSector(currentSector);
    if (error) {
        printf("mkdir: erreur lors de la creation recurssive des dossiers");
        return -1;
    }
    return 1;
}


char * FileSystem::WorkingDirectory () {
    workingDir = new char[MAX_DIRNAME_SIZE];
    workingDir = CurrentDir()->getDirName();
    return workingDir;
}
```

15 code/filesys/fstest.cc

Listing 15: *code/filesys/fstest.cc*

```
// fstest.cc
//    Simple test routines for the file system.
//
//    We implement:
//       Copy -- copy a file from UNIX to Nachos
//      Print -- cat the contents of a Nachos file
//      Perftest -- a stress test for the Nachos file system
//       read and write a really large file in tiny chunks
//       (won't work on baseline system!)
//
// Copyright (c) 1992-1993 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#include "copyright.h"

#include "utility.h"
#include "filesys.h"
#include "system.h"
#include "thread.h"
#include "disk.h"
#include "stats.h"

#define TransferSize     10     // make it small, just to be difficult
#define SHELL_BUFFER_SIZE 1<<16
#define SHELL_ARGS_SIZE 1<<16

void Print(char *name);
void Copy(const char *from, const char *to);
void PerformanceTest();
static void FileWrite();
static void FileRead();

void parse_args(char *buffer, char** args, size_t args_size, size_t *nargs)
{
    char *buf_args[args_size];
    char **cp;
    char *wbuf;
    size_t i, j;

    wbuf=buffer;
    buf_args[0]=buffer;
    args[0] =buffer;

    for(cp=buf_args; (*cp=strsep(&wbuf, " \n\t")) != NULL ;){
        if ((*cp != '\0') && (++cp >= &buf_args[args_size]))
            break;
    }

    for (j=i=0; buf_args[i]!=NULL; i++){
        if(strlen(buf_args[i])>0)
            args[j++]=buf_args[i];
    }

    *nargs=j;
    args[j]=NULL;
}

void prompt() {
    printf("nachos %s $ ", fileSystem->WorkingDirectory());
}

void test_prompt(const char * cmd) {
    prompt();
    printf("%s\n", cmd);
}

void show_help() {
    printf("\nCommandes disponibles :\n");
    printf("exit\n");
    printf("ls [<path>]\n");
    printf("cd [<path>]\n");
    printf("touch <filename>\n");
    printf("print <filename>\n");
    printf("cp <src> <dest>\n");
    printf("rm <path>\n");
    printf("mkdir <dirname>\n");
    printf("mkdir -p <path>\n");
    printf("pwd\n");
    printf("test  : lance les tests\n");
    printf("format\n------------------------\n\n");
}

void run_test() {
[...]
}


void shell() {
    char buffer[SHELL_BUFFER_SIZE];
    size_t nargs;
    char *args[SHELL_ARGS_SIZE];
    show_help();
    while(1) {
        prompt();
        if (fgets(buffer, SHELL_BUFFER_SIZE, stdin) == NULL) break;
```

```
        parse_args(buffer, args, SHELL_ARGS_SIZE, &nargs);

        if (nargs==0) continue;
        else if (!strcmp(args[0], "exit" )) break;
        else if (!strcmp(args[0], "ls" ) && (nargs == 1)) {
            fileSystem->List ();
        }
        else if (!strcmp(args[0], "ls" ) && (nargs == 2)) {
            fileSystem->List (args[1]);
        }
        else if (!strcmp(args[0], "cd") && (nargs == 1) ) {
            fileSystem->MoveToRoot();
        }
        else if (!strcmp(args[0], "cd") && (nargs == 2) ) {
            fileSystem->ChangeDir(args[1]);
        }
        else if (!strcmp(args[0], "touch") && (nargs == 2) ) {
            fileSystem->Create (args[1], 0);
        }
        else if (!strcmp(args[0], "print") && (nargs == 2) ) {
            Print (args[1]);
        }
        else if (!strcmp(args[0], "cp") && (nargs == 3) ) {
            Copy(args[1], args[2]);
        }
        else if (!strcmp(args[0], "rm") && (nargs == 2) ) {
            fileSystem->Remove (args[1]);
        }
        else if (!strcmp(args[0], "mkdir") && (nargs == 2) ) {
            fileSystem->MakeDir (args[1]);
        }
        else if (!strcmp(args[0], "mkdir") && (nargs == 3) ) {
            if (!strcmp(args[1], "-p"))
                fileSystem->MakeParentDir (args[2]);
            else
                show_help();
        }
        else if (!strcmp(args[0], "pwd") && (nargs == 1) ) {
            printf("%s\n", fileSystem->WorkingDirectory());
        }
        else if (!strcmp(args[0], "format")) {
            fileSystem = new FileSystem (true);
        }
        else if (!strcmp(args[0], "test")) {
            // on lance quelques tests
            run_test();
        }
        else {
            show_help();
        }
    }

    printf("\nBye\n");
    interrupt->Halt();
}


//----------------------------------------------------------------------
// Copy
// Copy the contents of the UNIX file "from" to the Nachos file "to"
//----------------------------------------------------------------------
void Copy(const char *from, const char *to) {
    FILE *fp;
    OpenFile* openFile;
    int amountRead, fileLength;
    char *buffer;

    // Open UNIX file
    if ((fp = fopen(from, "r")) == NULL) {
        printf("Copy: couldn't open input file %s\n", from);
        return;
    }

    // Figure out length of UNIX file
    fseek(fp, 0, 2);
    fileLength = ftell(fp);
    fseek(fp, 0, 0);

    // Create a Nachos file of the same length
    DEBUG('f', "Copying file %s, size %d, to file %s\n", from, fileLength, to);
    if (!fileSystem->Create(to, fileLength)) {     // Create Nachos file
        printf("Copy: couldn't create output file %s\n", to);
        fclose(fp);
        return;
    }

    openFile = fileSystem->Open(to);
    ASSERT(openFile != NULL);

    // Copy the data in TransferSize chunks
    buffer = new char[TransferSize];
    while ((amountRead = fread(buffer, sizeof(char), TransferSize, fp)) > 0)
    openFile->Write(buffer, amountRead);
    delete [] buffer;

    // Close the UNIX and the Nachos files
    delete openFile;
    fclose(fp);
}

//----------------------------------------------------------------------
// Print
// Print the contents of the Nachos file "name".
//----------------------------------------------------------------------

void Print(char *name)
{
    OpenFile *openFile;
    int i, amountRead;
    char *buffer;

    if ((openFile = fileSystem->Open(name)) == NULL) {
    printf("Print: unable to open file %s\n", name);
    return;
    }

    buffer = new char[TransferSize];
    while ((amountRead = openFile->Read(buffer, TransferSize)) > 0)
    for (i = 0; i < amountRead; i++)
        printf("%c", buffer[i]);
    delete [] buffer;

    delete openFile;        // close the Nachos file
    return;
}

//----------------------------------------------------------------------
// PerformanceTest
//      Stress the Nachos file system by creating a large file, writing
//     it out a bit at a time, reading it back a bit at a time, and then
//     deleting the file.
//
//     Implemented as three separate routines:
//       FileWrite -- write the file
//       FileRead -- read the file
//       PerformanceTest -- overall control, and print out performance #'s
//----------------------------------------------------------------------

#define FileName     "TestFile"
#define Contents     "1234567890"
#define ContentSize     strlen(Contents)
#define FileSize     ((int)(ContentSize * 5000))

static void FileWrite() {
    OpenFile *openFile;
    int i, numBytes;

    printf("Sequential write of %d byte file, in %zd byte chunks\n",
    FileSize, ContentSize);
    if (!fileSystem->Create(FileName, 0)) {
      printf("Perf test: can't create %s\n", FileName);
      return;
    }
    openFile = fileSystem->Open(FileName);
    if (openFile == NULL) {
        printf("Perf test: unable to open %s\n", FileName);
        return;
    }
    for (i = 0; i < FileSize; i += ContentSize) {
        numBytes = openFile->Write(Contents, ContentSize);
        if (numBytes < 10) {
            printf("Perf test: unable to write %s\n", FileName);
            delete openFile;
            return;
        }
    }
    delete openFile;    // close file
}

static void
FileRead()
{
    OpenFile *openFile;
    char *buffer = new char[ContentSize];
    int i, numBytes;

    printf("Sequential read of %d byte file, in %zd byte chunks\n",
    FileSize, ContentSize);

    if ((openFile = fileSystem->Open(FileName)) == NULL) {
        printf("Perf test: unable to open file %s\n", FileName);
        delete [] buffer;
```

```
        return;
    }
    for (i = 0; i < FileSize; i += ContentSize) {
        numBytes = openFile->Read(buffer, ContentSize);
        if ((numBytes < 10) || strncmp(buffer, Contents, ContentSize)) {
            printf("Perf test: unable to read %s\n", FileName);
            delete openFile;
            delete [] buffer;
            return;
        }
    }
    delete [] buffer;
    delete openFile;    // close file
}

void
PerformanceTest()
{
    printf("Starting file system performance test:\n");
    stats->Print();
    FileWrite();
    FileRead();
    if (!fileSystem->Remove((char *)FileName)) {
      printf("Perf test: unable to remove %s\n", FileName);
      return;
    }
    stats->Print();
}
```