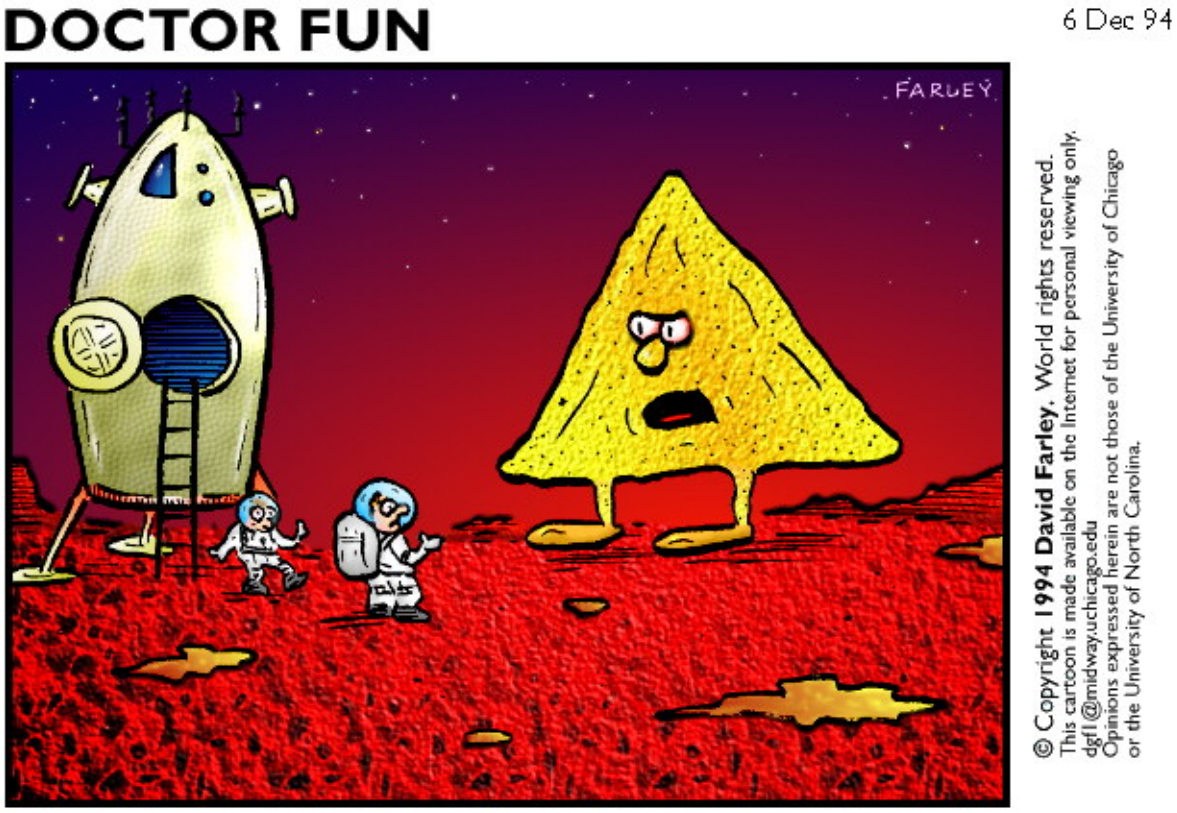
Nicolas MOBILIA, Remi BARRAQUAND, Romain FORNARA et Savas Ali TOKMEN



NachOS: compte rendu

Index

Etape 3: Multithreading

Partie 1: Mise en place de threads utilisateurs

Gestion des threads au niveau kernel dans NachOS

NachOS a un systeme de gestion de threads qui n'utilise pas les threads Linux.

La classe Thread

Cette classe contient les structures de donnees pour gerer les threads noyau (donc qui executent du code i386). Un objet Thread represente l'execution d'une sequence de code dans un programme; il contient donc les registres du processeur i386 dont le compteur programme (PCReg) et la pile d'execution.

Pour créer un thread, on doit créer une instance de la classe Thread:

```
Thread* t = new Thread();
```

Le nouvel objet Thread est vide, il faut donc l'initialiser.

L'initialisation se fait en specifiant un pointeur vers l'adresse du code a executer et l'argument de cette fonction. Par exemple, pour faire executer au thread t le code de le fonction toto avec comme argument titi:

```
t->Fork( &(toto), titi );
```

Cet appel provoque:

1- L'appel de StackAllocate: cette methode apelle AllocBoundedArray pour allouer un espace fixe pour la pile i386 (donc il faudra eviter de créer trop de variables dans les fonctions noyau executees). Cette espace est allouee dans la memoire de NachOS. Apres, il met dans le tableau MachineState tous les elements necessaires au lancement et a la terminaison du thread (donc les registres PCState, StartupPCState, InitialPCState, InitialArgState et WhenDonePCState). Lors du lancement du thread, ces informations seront copies dans la machine reelle.

2- L'appel scheduler->ReadyToRun (this), qui met en queue le thread dans le scheduler NachOS (attention, pas dans le scheduler Linux!).

La classe Scheduler

Cette classe est formee d'une liste de threads qui sont en attente pour l'execution (donc qui sont prets) et des methodes pour manipuler cette liste:

- La methode ReadyToRun qui marque le thread en « pret » (thread->setStatus (READY)) et le met en queue de la liste des prets.

- La methode FindNextToRun qui enleve sa tete de la liste, qui correspond au thread a etre execute.

- La methode Print pour deboguer.

Cette classe fournit aussi une methode Run, qui:

- Verifie qu'il n'y a pas eu debordement de la pile de la fonction executee par le thread courant.

- Change le thread courant et le thread suivant de place avec l'appel SWITCH. Cet appel, ecrit en assembleur:

    - Sauvegarde l'etat du thread courant dans sa variable MachineState

    - Prepare le chargement l'etat du thread a executer dans la machine

- Si c'est necessaire, detruit le thread d'avant (dans le cas ou il aurait termine)

- Execute le thread qu'il vient de SWITCHer.

Le scheduler n'est donc qu'un objet global qui est appele par les threads.

Le fonctionnement global

L'objet thread, une fois instancie, doit donc etre initialise. Une fois initialise, le thread se mettra en queue dans le scheduler. On notera que le thread qui a cree le nouveau thread et l'a mis en queue continue son execution.

Lorsque le thread est ordonnance par le scheduler, celui-ci reprend son execution la ou il etait avant, donc en se basant sur les valeurs stockees dans sa variable MachineState.

L'execution ne se terminera pas tant que le thread n'aura pas exlipcitement appele Sleep, Yield ou Finish. Par consequent, lorsqu'un thread commence a utiliser le processeur, celui-ci peut ne jamais se terminer.

- Le thread apelle Yield pour se remettre dans la liste des prets de scheduler et changer de contexte.

- Le thread apelle Sleep pour changer de contexte sans se remettre dans la liste des prets (donc dans le cas d'une attente passive)

- Le thread apelle Finish pour changer de contexte sans se remettre dans la liste des prets et en se declarant le thread a etre detruit.

Dans les trois cas, la methode Run du scheduler est appele, ce qui provoque le changement de contexte.

Fonctionnement des threads utilisateurs

On mappera un thread utilisateur vers un thread noyau. L'avantage de ce type d'implementation est sa simplicite. Par contre, cela pourra dans certains cas impliquer de mauvaises performances...

La fonction StartUserThread s'occupera de ceci:

```
void StartUserThread ( int which )
{
   machine->Run();
}
```

On notera que tout code MIPS peut etre reordonnance a tout moment, donc cette fonction bouclera jamais a l'infini.

La creation du thread utilisateur doit passer par quelques etapes:

1- Toute fonction appartenant au meme processus MIPS partage la meme pile. Il est donc important de calculer la taille de la pile necessaire a la fonction. Meme si ceci est pratiquement impossible, on peut au moins savoir la pile minimale requise par la fonction car le compilateur genere des fonctions qui ont comme premiere instruction de changer SP.

```
      machine->ReadMem( f, 4, &( nouvelle_pile ) );
      Instruction* instr = new Instruction();
      instr->value = nouvelle_pile;
      instr->Decode();
      // Le "16" sert a reserver un peu plus de place pour les variables en plus,
      // les adresses de retour et autre basoins dynamiques des fonctions
      // instr->extra (<0) contient la place utilisee pour la declaration des
      // variable de la fonction
      nouvelle_pile = machine->ReadRegister( StackReg ) + instr->extra - 16;
      delete instr;
```

2- Verification de l'espace disponible dans la pile. Pour ceci, on a prefere mettre dans AddrSpace une variable qui indique l'adresse minimale que peut avoir la pile (donc la ou se termine le code MIPS).

3- Mettre en place l'environnement MIPS du thread: il doit copier les registres MIPS ainsi que l'espace d'adressage du processus (SaveUserState) et modifier PC ainsi que les registres d'argument. D'ou le code:

```
      nouveauThread->SaveUserState();
      nouveauThread->ChangeUserReg( 4, arg );
      nouveauThread->ChangeUserReg( PCReg, f );
      nouveauThread->ChangeUserReg( NextPCReg, f + 4 );
      machine->WriteRegister( StackReg, nouvelle_pile );
```

Et voici l'appel systeme:

```
    // Recuperation des arguments de lancement de l'utilisateur
    fonction = machine->ReadRegister(4);
    argument = machine->ReadRegister(5);
    // Pour pouvoir retourner -1 si le thread ne peut etre cree
    machine->WriteRegister( 2, do_CreateUserThread( fonction, argument ) );
```

UserThreadExit ne fera que ne terminer le thread courant sans detruire l'espace d'adressage:

```
    // Destruction du thread
    currentThread->Finish();
```

On peut donc commencer a tester: on cree makethreads.c qui contient des fonctions affichant des « ping » et des « pong » avec des compteurs (les compteurs servent a voir si les piles des threads se melangent). La fonction main de ce programme va lancer une vingtaine de threads ping et pong differents. L'execution marche avec succes, et on remarque que la sortie indique de temps en temps qu'un thread ne peut etre cree, car il n'y a plus d'espace dans la pile MIPS.

On remarque que si makethreads.c est lancé sans l'option -rs, ceci desactive la preemption donc le gestionnaire de threads devient FIFO.

Partie 2: Plusiers threads par processus

Quand plusiers threads sont executes en meme temps, il se peut que les appels PutChar et PutString créent des exceptions s'ils ne sont pas synchronises. A l'etape 2, nous avions implemente ces fonctions en pensant au multi-threading, donc nous n'avons pas eu ce probleme.

Il se peut que le processus s'arrete alors que ses threads ne sont pas termines. Cela a pour consequence d'arreter la machine alors que les threads n'ont pas termines! On a donc cree une hierarchie entre objets crees: tout objet Thread doit attendre ses fils avant de terminer. Pour ceci:

- On a ajoute un mutex « `Il_y_a_des_fils` » qui est pris par le premier thread fils. Avant de terminer, le thread parent prend le mutex (donc attend que le dernier fils l'ait libere).

- On a aussi ajoute un entier « `Nombre_de_fils` » qui compte le nombre de fils, pour que seul le premier fils acquiert le mutex.

Ceci aura comme consequence que les threads utilisateurs vont eux aussi attendre la fin des threads utilisateurs qu'ils ont crees. Ceci pourrait etre logique au sens ou des threads peuvent obtenir et faire heriter, par exemple, des attributs de securite.

Donc, la nouvelle ligne ajoutee dans SC_Halt:

```
    currentThread->Il_y_a_des_fils->Acquire();
```

Ainsi que les lignes en plus dans la fonction UserThreadCreate:

```
    nouveauThread->Parent = currentThread;
    // Informations fils vers pere
    if( currentThread->Nombre_de_fils == 0 )
    {
          currentThread->Il_y_a_des_fils->Acquire();
    }
    currentThread->Nombre_de_fils++;
```

Et dans UserThreadExit:

```
    // on attend qu'on ait plus de fils qui tourne
    currentThread->Il_y_a_des_fils->Acquire();
    // Informations fils vers pere: mise a jour
    currentThread->Parent->Nombre_de_fils--;
    if( currentThread->Parent->Nombre_de_fils == 0 )
    {
          currentThread->Parent->Il_y_a_des_fils->Release();
    }
```

Le lancement d'un grand nombre de thread, avec des arguments passes en parametre, plusieurs variables locales ainsi que des pointeurs vers des variables globales ou locales passes en parametre ne pose aucun probleme.

On notera que le lancement sans l'option -rs a peu d'interet, vu que l'ordonnanceur devient un FIFO tout simple et que l'on ne voit pas les divers « ping » et « pong » entremêlés.

Partie 3: Terminaison automatique, PutString et semaphores utilisateur

Une fonction normale, une fois termine, retourne a l'appelant a l'endroit ou il etait lors de l'appel. Ce comportement n'a aucun sens avec des threads: un thread qui a termine devrait tout simplement terminer, et pas se rebrancher sur un appelant.

Donc, si UserThreadExit n'est pas present a la fin d'un thread, l'instruction suivant la creation du thread est execute une fois le thread termine. Pour remedier a ce probleme, on a decide de mettre comme adresse de retour (donc comme contenu de registre 31) la valeur -1. Il faudra aussi modifier l'interpreteur MIPS pour que les branchements OP_JR vers l'adresse -1 soient rediriges vers UserThreadExit. D'ou la ligne en plus dans UserThreadCreate:

```
    nouveauThread->ChangeUserReg( 31, -1 );
```

Ainsi que les lignes ajoutees dans mipssim.cc:

```
    if( registers[31] == -1 )
    {
          registers[2] = 31;  // SC_UserThreadExit
          RaiseException(SyscallException, 0);
          break;
    }
```

Dans l'etape 2, nous avions deja implemente l'appel systeme PutString. On observera que l'appel continue a marcher correctement dans un environnement multithreadé.

Pour la creation de semaphores utilisateur, on va faire correspondre chaque semaphore utilisateur à un semaphore noyau. Pour créer cette correspondance, on pourrait tout simplement renvoyer a l'utilisateur l'adresse du semaphore noyau crée... Mais ceci presente un risque: un utilisateur pourrait donner des adresses « au hasard » pour contourner le noyau ou modifier les semaphores des autres utilisateurs!

On a donc cree la classe Liste qui represente une liste chainee avec des cles (un peu comme un Map C++, mais malheurement pour nous Map n'est pas present sur les compilateurs des machines mandelbrot ou goedel!). On a ajoute un objet de cette classe a la classe AddrSpace pour avoir une liste des semaphores utilisateur. Les appels systemes impliques sont donc:

```
	case SC_InitSemaphore :
	{
		machine->WriteRegister( 2, CreateUserSemaphore(
					machine->ReadRegister(4) ) );
		break;
	}
```

```
	case SC_Sema_P :
	{
		UserSemaphoreP( machine->ReadRegister(4) );
		break;
	}
```

```
	case SC_Sema_V :
	{
		UserSemaphoreV( machine->ReadRegister(4) );
		break;
	}
```

Et voici les definitions de CreateUserSemaphore, UserSemaphoreP et UserSemaphoreV:

```
int CreateUserSemaphore( int nb )
{
   Semaphore* semaphore = new Semaphore( "Semaphore utilisateur", nb );
   return currentThread->space->semaphoresUtilisateur->Add( semaphore );
}
```

```
void UserSemaphoreP( int id )
{
   Semaphore* semaphore = (Semaphore*) currentThread->space->
                                          semaphoresUtilisateur->Find( id );
   if( semaphore )
   {
         semaphore->P();
   }
   else
   {
         printf( "Le semaphore utilisateur %d n'existe pas!\n", id );
   }
}
```

```
void UserSemaphoreV( int id )
{
   Semaphore* semaphore = (Semaphore*) currentThread->space->
                                          semaphoresUtilisateur->Find( id );
   if( semaphore )
   {
         semaphore->V();
   }
   else
   {
         printf( "Le semaphore utilisateur %d n'existe pas!\n", id );
   }
}
```

Pour tester, on cree semaphores.c qui fait exactement la meme chose que makethreads.c mais qui, au lieu d'imprimer « ping » et « pong » avec un nombre d'un seul coup obtient la semaphore, imprime « ping » ou « pong », imprime le nombre puis imprime un retour a la ligne. Donc, sans synchronisation, ces sorties seraient entremêlés.

On observe que l'implementation marche, et que le meme test sans les semaphores crees des sorties melanges (car le thread est reordonnance en plein milieu de l'impression!).

Etape 4: NachOS et la pagination

Partie 1: Adressage virtuel par une table de pages

Le lancement d'un processus

Pour lancer un processus, il faut tout d'abord le mettre en memoire. Pour ce faire, la version initiale de NachOS:

1- Lit l'entete de l'executable MIPS pour savoir la taille du code, la taille des donnees initialises et non-initialises. A cette taille est ajoutee la taille de la pile utilisateur pour deduire la taille memoire totale dont l'executable a besoin.

2- Si possible, alloue le nombre de pages necessaire et initialise ces pages (en utilisant la fonction bzero de la librairie <string.h>).

3- Copie l'executable du fichier vers cet espace en utilisant executable->ReadAt.

Ceci pose un probleme si on a envie d'utiliser une table des pages: executable->ReadAt est une fonction UNIX qui copie le contenu d'un fichier vers la memoire. Or, les pages ne sont pas obligatoirement physiquement contigues, donc on ne peut se permettre de lire un fichier directement dans la memoire.

Mise en memoire en utilisant l'adressage virtuel

On va donc implementer une fonction ReadAtVirtual, qui lui va:

1- Lire l'executable dans un buffer

2- Mettre ce buffer dans les pages virtuels donnees en argument.

Pour mettre le buffer dans la machine, on va utiliser la fonction WriteMem propose par Machine. Or, cet appel utilise la table des pages de la machine, il faut donc:

1- Sauvegarder l'etat courant de la machine (surtout la table des pages)

2- Ecraser la table des pages courante avec la table des pages donnee en argument

3- Mettre le buffer en memoire en utilisant WriteMem

4- Restaurer l'ancien etat de la machine

Pour les les etapes 1 et 4, nous avons mis en place dans Machine deux methodes:

- PushConfiguration, qui va sauvegarder l'etat courant de la machine

- PopConfiguration, qui va restaurer l'etat sauvegarde

On en deduit la fonction ReadAtVirtual:

```
DEBUG( 'p', "Lecture du code MIPS...\n" );
// Sauvegarde de la page des pages courante
machine->PushConfiguration();
// Modification (temporaire) de la table des pages
machine->pageTable = pageTable;
machine->pageTableSize = numPages;
// Lecture de l'executable dans le cache
char* cache = new char[ numBytes ];
ASSERT( executable->ReadAt( cache, numBytes, position ) == numBytes );
ASSERT( numBytes % 4 == 0 );
for( int i = 0 ; i < ( numBytes / 4 ) ; i++ )
{
      // Copie du buffer dans l'adresse physique
      ASSERT( machine->WriteMem( virtualaddr + 4*i, 4,
                  *( (int*) ( cache + 4*i ) ) ) );
}
// Plus besoin du buffer
delete[] cache;
// Restauration de la page des pages
machine->PopConfiguration();
DEBUG( 'p', "Lecture du code MIPS termine\n" );
```

Attention: on pourrait penser que l'on pourrait copier le buffer dans la memoire de la machine page par page. Ceci n'est malheurement pas possible, i386 et MIPS utilisant des formats de lecture (little et big endian) differents! La fonction avec la plus grande capacite de conversion qui nous est fournie est la fonction WordToHost, qui convertit par tranches de 32 bits (et est utilisee par WriteMem).

Le test avec un mappage des pages virtuels vers de divers de configuration de pages physiques (inclus « au hasard ») semble marcher...

Le fournisseur de frames

Nous avons cree une classe FrameProvider, qui s'occupe de l'allocation des frames. Vu que la memoire MIPS appartient a la machine MIPS, le meilleur endroit pour mettre une instance de cette classe est la classe Machine. FrameProvider a trois methodes:

- GetEmptyFrame, qui alloue, met a zero et retourne le numero d'une page physique.

- ReleaseFrame, qui desalloue une page physique si celle-ci est deja allouee.

- NumAvailableFrame, qui retourne me nombre de pages physiques disponibles (cette methode sera utile pour verifier la memoire disponible avant de créer des processus).

Nous utilisons la classe Bitmap pour savoir si une page est libre ou allouee. Les definitions des methodes sont:

```
FrameProvider::FrameProvider(int nelem)
{
   DEBUG( 'p', "FrameProvider initialise avec %d pages\n", nelem );
   my_map=new BitMap(nelem);
}
```

```
FrameProvider::~FrameProvider()
{
   delete my_map;
}
```

```
int FrameProvider::GetEmptyFrame()
{
   if( my_map->NumClear() > 0 )
   {
         int page = NumPhysPages - 1 - my_map->Find();
         bzero( &( machine->mainMemory[ PageSize * page ] ), PageSize );
         DEBUG('p',"Allocation du frame %d, reste %d\n",
                        page, my_map->NumClear());
         return page;
   }
   else
   {
         DEBUG('p',"Plus de frames !\n");
         return -1;
   }
}
```

```
void FrameProvider::ReleaseFrame(int num)
{
   DEBUG('p',"Liberation du frame %d\n",num);
   ASSERT( my_map->Test( NumPhysPages - num - 1 ) );
   my_map->Clear( NumPhysPages - num - 1 );
}

int FrameProvider::NumAvailFrame()
{
   return my_map->NumClear();
}
```

Notez que l'on retourne les pages dans une ordre inversee. Ceci a ete fait pour pouvoir avoir une allocation « inhabituelle » des pages sans trop d'efforts (donc sans utiliser rand()).

Pour utiliser cette classe dans AddrSpace:

```
// Initialisation de la table des pages
for (i = 0; i < numPages; i++)
{
   pageTable[i].virtualPage = i;
   pageTable[i].physicalPage = machine->frameprovider->GetEmptyFrame();
   pageTable[i].valid = TRUE;
   pageTable[i].use = FALSE;
   pageTable[i].dirty = FALSE;
   pageTable[i].readOnly = FALSE;
}
```

Apres toutes ces modifications, nos programmes fonctionnent toujours...

Partie 2: Executer plusiers programmes en meme temps

Implementation du multiprocessus

Vu que l'adressage virtuel et le fournisseur de frames marche, on peut maintenant mettre plusieurs programmes dans la meme memoire physique et les executer en meme temps... Notre implementation du multiprocessus ressemblera beaucoup a l'implementation du multithreading; a savoir:

- Tout processus utilisateur correspond a un thread noyau.

- Comme chaque thread noyau attend tous ces fils avant de terminer; chaque processus (ou thread) attendra la terminaison de ses fils avant de quitter.

- Main sera un processus particulier: il n'a pas de parent (donc sont parent est NULL). Par consequent, grace au systeme d'attente, il sera toujours le dernier a se terminer.

On commence par l'appel systeme:

```
    DEBUG('i', "Interuption SC_ForkExec levee par \"%s\".\n",
                currentThread->getName() );
    // Recuperation de l'executable
    char executable[ MAX_STRING_SIZE ];
    copyStringFromMachine( machine->ReadRegister(4),
                           executable, MAX_STRING_SIZE );
    // Lancement
    machine->WriteRegister( 2, do_CreateUserProcess( executable ) );
```

On notera que la fonction do_CreateUserProcess retourne -1 si la creation du processus est impossible (donc s'il n'y a plus de frames disponibles).

Pour créer un processus, il faut:

- Lire l'executable (et surtout decoder les entetes pour savoir la memoire dont l'executable a besoin)

- Verifier qu'il y a assez de memoire et l'allouer

- Mettre l'executable en memoire (donc dans un nouvel espace d'adressage)

- Créer le thread noyau associe et l'initialiser

- Mettre ce nouveau thread dans le scheduler

D'ou la fonction do_CreateUserProcess:

```
    // Ouvrir le fichier
    OpenFile *exec = fileSystem->Open (executable);
    AddrSpace *space;
    // Lecture impossible
    if (exec == NULL)
    {
          printf ("Lecture du fichier \"%s\" impossible\n", executable);
          return -1;
    }
    // Lecture du fichier dans la memoire de la machine
    space = new AddrSpace (exec);
    // Fermeture du fichier
    delete exec;
    if (space == NULL || space->adresse_fin_de_code == 0)
    {
          printf ("Pas assez de memoire pour charger le fichier \"%s\" !\n",
                        executable);
          if( space ){  delete space;  }
          return -1;
    }
    Thread* nouveauThread;
    char strBuf[ MAX_STRING_SIZE ];
    // Creation de l'objet thread
    snprintf( strBuf, sizeof(strBuf)-1, "Processus \"%s\"", executable );
    strBuf[ sizeof(strBuf)-1 ] = '\0';
    nouveauThread = new Thread( strBuf );
    // Pointeur vers le pere
    ASSERT( currentThread != NULL );
    nouveauThread->Parent = currentThread;
    nouveauThread->space = space;
    // Informations fils vers pere
    if( currentThread->Nombre_de_fils == 0 )
    {
          currentThread->Il_y_a_des_fils->Acquire();
    }
    currentThread->Nombre_de_fils++;
    currentThread->Liste_processus_fils->Add((void*)(space->id));
    // Mise en queue dans le scheduler
    nouveauThread->InitKernelThread( StartUserProcess, 0 );
    return 0;
```

Ce nouveau thread va donc appeler la fonction StartUserProcess. Cette fonction doit:

- Initialiser les registres MIPS

- Chargement de la table de pages (qui se fait automatiquement par Scheduler::Run)

- Lancement de la machine

D'ou le code:

```
    // Le processus a les registres dans leur etat initial
    currentThread->space->InitRegisters ();
    machine->Run ();
```

La destruction d'un processus se fait de la facon suivante:

- Attente de la fin des fils (noter que cette attente a ete mise dans exception.cc)

- Mise a jour des liens de parente

- Destruction de l'espace d'adressage

- Destruction du thread noyau

Donc:

```
	// Informations fils vers pere: mise a jour
	currentThread->Parent->Liste_processus_fils->Remove(
	                          (void*)(currentThread->space->id));
	currentThread->Parent->Nombre_de_fils--;
	if( currentThread->Parent->Nombre_de_fils == 0 )
	{
		currentThread->Parent->Il_y_a_des_fils->Release();
	}
	// Liberer la memoire du processus
	delete currentThread->space;
	// Destruction du thread
	currentThread->Finish();
```

Finalement, l'appel systeme SC_HALT qui est modifie:

```
	currentThread->Il_y_a_des_fils->Acquire();
	// on ne fait halt que pour le main
	if( currentThread->Parent == NULL )
	{
		delete currentThread->space;
		interrupt->Halt();
	}
	else
	{
		// Fin d'un processus fils
		do_DestroyUserProcess();
	}
```

On teste avec userpages.c, qui:

- Lance plusiers copies de userpages0 et userpages1, dont le premier est un programme de type « ping - pong » et le deuxieme un progamme de type « pang - peng »

- Lance userpages (donc lancera userpages0 et userpages1 jusqu'à saturation de la memoire!)

On notera que l'appel systeme SC_Halt (qui est utilise par SC_Exit) ayant ete modifie directement, les processus fils n'ont pas besoin d'appeler des commandes utilisateur pour terminer. Les terminaisons sont toujours automatiques, transparentes au programmeur.

Parties bonus

Detection des erreurs de (des)allocation memoire

Nous avons choisi de surcharger les operateurs new et delete de C++ afin de na pas avoir a modifier le code deja ecrit. La detection devra etre capable de detecter les tentatives de double desallocation memoire ainsi que les non-desallocations. Dans le fichier de journal de sortie, le systeme devra preciser le nom du fichier et de la ligne ou l'erreur apparaît. Optionnellement, le systeme pourra aussi donner un historique complet des allocations et desallocations.

Le compilateur C++ a deux variables speciales: __FILE__ et __LINE__ (qui correspondent donc au nom de fichier et au numero de ligne). Il faut donc surcharger les operateurs new et delete de telle maniere a prendre en compte ces deux informations.

La surcharge de new et delete

Plusieurs solutions étaient possible, comme la création d'une classe qui aurait surcharge les commandes new et delete, la creation de deux nouvels operateurs SAFE_NEW et SAFE_DELETE ou encore la surcharge globale des opérateurs new et delete. La derniere solution a été retenue.

Le standard C++ permet de redefinir les operateurs new et delete d'une facon globale:

```
inline void* operator new(std::size_t size, const char* file, int line)
{
    return CMemory::Instance()._new(size, file, line, false);
}

inline void* operator new[](std::size_t size, const char* file, int line)
{
    return CMemory::Instance()._new(size, file, line, true);
}

inline void operator delete(void* ptr)
{
    CMemory::Instance()._delete(ptr, false);
}

inline void operator delete[](void* ptr)
{
    CMemory::Instance()._delete(ptr, true);
}
```

Maintenant que les operateurs sont surchargés, il faut mapper les anciens operateur vers les nouveaux, ainsi pour reecrire directement tous les new de la facon desiree, rien de plus simple:

```
#define new new( __FILE__, __LINE__ )
```

La definition de cette marcro ne pose donc aucun problème ici, et les deux operateurs new et new[] fonctionnent. Mais on voit apparaître un premier probleme pour l'operateur delete[], pour créer un tableau, on apelle:

```
variable = new type[ nombre ]
```

alors que pour delete les crochets sont avant le nom de la variable:

```
delete[] variable
```

Or, les symboles « [» et «] » etant des symboles reserves, on ne peut pas ecrire:

```
#define delete[] delete[]( __FILE__, __LINE__ )
```

De meme, la macro suivante est invalide:

```
#define delete delete( __FILE__, __LINE__ )
```

car, a l'appel d'un

```
delete[] variable
```

le preprocesseur C++ genere:

```
delete( __FILE__, __LINE__ )[] variable
```

Ce qui n'est pas valide!

En effet, il aurait ete preferable que C++ ait suivi le meme standard d'ecriture pour new et delete, a savoir ecrire:

```
delete variable[]
```

au lieu de

```
delete[] variable
```

D'ou la petite astuce d'étendre delete de la maniere suivante:

```
#define delete CMemory::Instance().InitLocal(__FILE__,__LINE__),delete
```

La classe CMemory

Comme vue precedement nos nouveaux operateurs utilisent des méthodes d'allocation et de désallocation de la classe CMemory. CMemory est donc le coeur de notre logger, c'est ici que la memoire sera allouée, désallouée et que toutes les operations seront loggés dans un fichier de log.

Pour chaque bloque memoire alloué par l'utilisateur un maillon d'une liste chainée est créée, ce maillon contient de multiples informations sur le bloc alloué et sera utilisé pour le delete et la partie finale du logger. Un maillon de chaine est du type :

```
struct BlockMem
    {
        void*       ptr;
        std::size_t size;
        char*       file;
        int         line;
        bool        array;
        BlockMem*   next;
    };
```

Le booléen « array » indique s'il sagit d'un new ou d'un new[].

Le code d'allocation est simple, tout d'abord nous allouons l'espace demandé à l'aide de malloc, puis nous créeons un nouveau maillon et l'initialisons avec les bonnes valeurs, si l'option est demandé nous loggons l'allocation.

```
void* CMemory::_new(std::size_t size, const char* file, int line, bool array)
{
    // Allocation de la memoire
    void* ptr = malloc(size);
    // Ajout du bloc à la liste des blocs alloues
    BlockMem* NewBlock = InitBlock(ptr, size, NULL, file, line, array);
    InsertBlock(NewBlock);
    // On log le truc
#ifdef ALL_LOG
    logfile << "|>> Allocation [0x" << ptr << ", " << (int)size << "]" << std::endl;
    logfile << "| dans le fichier " << file << ", ligne "<< line << std::endl;
    logfile << "+" << std::endl;
#endif
    return ptr;
}
```

Pour la desallocation cela se complique un petit peu, premierement nous devons faire passer les informations de ligne et de fichier ou la desallocation est effectuée, ceci par la methode :

```
void InitLocal(char* file, int line) { _FILE = file; _LINE = line; }
```

Ensuite nous procedons à l'effacement du bloc memoire par la methode:

```
void CMemory::_delete(void* ptr, bool array)
{
    BlockMem* temp = FindBlock(ptr);
    // Si aucun block alors !!!
    if(temp == NULL)
    {
        logfile << "|>> Desallocation erronee !! [0x" << ptr << "]" <<
                std::endl;
        logfile << "| dans le fichier " << _FILE << ", ligne "<< _LINE <<
                std::endl;
        logfile << "+" << std::endl;
    }
    // Sinon on l'efface et on log
    else
    {
#ifdef ALL_LOG
        logfile << "|>> Desallocation [0x" << ptr << ", " << (int)temp->size <<
                "]" << std::endl;
        logfile << "| dans le fichier " << _FILE << ", ligne "<< _LINE <<
                std::endl;
        logfile << "+" << std::endl;
#endif
        RemoveBlock(temp);
    }
}
```

Nous faisons une recherche dans notre liste chainée avec comme clé le pointeur à effacer, si le bloque demandé n'existe pas nous loggons l'erreur de desallocation erronée. Sinon nous effaçons le bloque de la liste chainée et loggons la réussite de la desallocation.

Lors de la destruction de la classe CMemory, c'est a dire lors de la fermeture du programme NachOS, nous loggons l'état de la memoire. Pour cela dans le destructeur de la classe Cmemory nous plaçons la methode suivante:

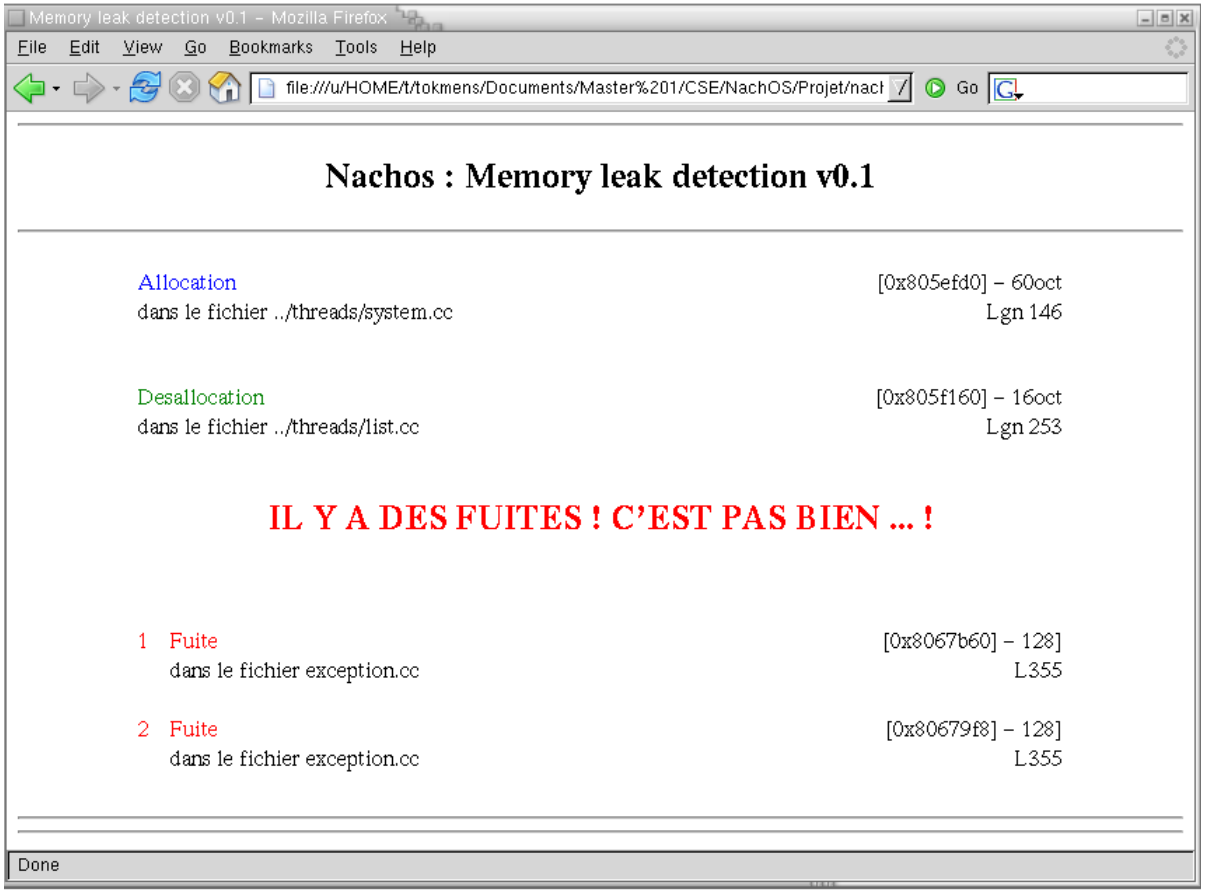
```
void CMemory::LogMemState()
{
    BlockMem* temp = _stack;

    // Si il reste juste le block de tete alors c'est bon
    if(temp == NULL)
    {
        logfile << "-----------------------------------------------" << std::endl;
        logfile << " PAS DE FUITES ... BRAVO ! " << std::endl;
        logfile << "-----------------------------------------------" <<
                std::endl;
    }
    else
    {
        // sinon c'est pas bien !!
        logfile << "-----------------------------------------------" << std::endl;
        logfile << " IL Y A DES FUITES ! C'EST PAS BIEN ..." << std::endl;
        logfile << "-----------------------------------------------" << std::endl;
        while(temp)
        {
            if( temp->file != NULL)
            {
                logfile << "|>> Fuite : [0x" << temp->ptr << ", " <<
                       (int)temp->size << "]" << std::endl;
                logfile << "| dans le fichier " << temp->file << ", ligne "<<
                       temp->line << std::endl;
                logfile << "+" << std::endl;
            }
            temp = temp->next;
        }
    }
}
```

Si la liste chainée est vide, alors tout est correct, il n'y a pas de leak memoire, par contre si elle n'est pas vide alors nous loggons tous les leaks et par sympathie nous effaçons la memoire correctement...

Notez qu'il y a un grand probleme: les executables generes et executes par mandelbrot creent, dans certains cas, des journaux d'erreur qu'il est logiquement impossible d'avoir. Ces problemes ne se manifestent pas sur goedel. Il faudra probablement reviser le compilateur present sur mandelbrot !

Notez que dans la version originale les logs sont generes en format HTML, mais que dans ce fichier la sortie HTML a ete coupee pour rendre le code plus lisible.

Voici une image de la sortie HTML sous Mozilla Firefox:



Shell

Choix d'implementation

Nous avons cree un shell utilisateur. Nous avons choisi de le faire en tant que processus utilisateur car cela nous permet de tester dans un « gros » programme les fonctionnalites qu'offre NachOS. De plus, si nous avions plusieurs fenetres, cela nous permettrait de lancer facilement plusieurs shells simultanement. Par contre, cela impose de faire un appel systeme pour chaque fonctionnalite que le shell propose.

Un shell est la repetition d'une suite d'actions :

- Saisie de la commande a executer
- Traitement de cette chaine, notament du path de la commande et de l'attente de la fin de l'execution
- Execution de la commande

La saisie

L'invite de commande indique que le shell est prêt a recevoir une commande. Cette invite indique aussi le repertoire courant.

La saisie proprement dite se fait via la fonction GetString.

Le traitement

On enleve les espaces au debut et a la fin de la chaine saisie.

Si le dernier caractere est le caractere « & », on met a VRAI un flag indiquant que le shell devra attendre la fin de l'execution de la commande avant de demander une autre commande.

On decoupe ensuite la chaine en deux : une partie qui contient le path de la commande et une autre qui contient la commande elle meme. On entend par « path de la commande » la partie « path/ » de la commande : « path/cmde ».

L'execution

Nous avons mis en place une astuce permettant d'executer une commande UNIX : il suffit de faire preceder la commande par le caractere '!' pour envoyer la commande a UNIX et donc ne pas la faire executer par NachOS. On peut, par exemple, lancer xterm ou encore firefox avec cette methode.

Tout d'abord, nous changons de repertoire pour se mettre dans le repertoire de la commande. Pour ne pas perdre le repertoire courant, on le sauvegarde en utiliant la fonction pushd(). Ensuite, on peut faire utiliser la fonction cd(). Apres l'execution de la commande, on retourne dans l'ancien repertoire courant via la fonction popd.

Les commandes disponibles

Nous avons implemente dans le shell une serie de commande qui utilise les fonctionnalite proposees par NachOS :

- cd ; md / mkdir ; rd / rmdir ; pwd
- exit
- ls / dir
- open / close ; cat / list ; write

Si la commande tapee par l'utilisateur ne correspond a aucune de ces fonctionnalites, on execute alors le programme sur le disque correspondant au nom rentre. Si le flag indiquant que le shell doit attendre la fin de l'execution du processus avant de demander une autre commande, est a VRAI, alors le shell se met en attente en utilisant un appel systeme.

Malloc et free utilisateur

Limites des programmes

Un petit programme de test (mem_protect.c) revele qu'il n'y a aucune protection memoire dans NachOS:

```
int main ()
{
   int i;
   int* start = 0;

   PutString( "je me suicides !!\n" );

   for( i = 0 ; i < 2048 ; i++ )
   {
          start[ i ] = 0;
   }

   PutString( "suicide OK\n" );

   return 0;
}
```

Ce programme, qui marche sans probleme, va remplir tout son code avec des 0 (en ne va donc jamais ecrire « Suicide OK »). En effet, la seule chose qu'un processus ne peut pas faire c'est ecrire au dela de son espace adressage (donc en particulier sur l'espace d'un autre processus ou sur le noyau).

On pourrait donc, dans un premier temps, essayer de resoudre ce probleme. En effet, si on regarde dans la classe TranslationEntry, on voit que des pages peuvent etre declares en lecture seule. Une idee serait donc de declarer les pages de code en lecture seule, ce qui va les verouiller...

Cet action a malheurement des grandes consequences: le compilateur MIPS cree les donnees initialisees et non-initialisees (donc les variables globales) juste apres le code. Par consequent, verouiller les pages de code aura comme consequence de verouiller aussi les pages contenant les variables globales!

On ne pourra donc pas, avec la version actuelle de NachOS, implementer un systeme de verrou de parties de la memoire...

Les pages supplementaires

La memoire d'un processus est represente par une adressage virtuelle. Donc, pour allouer plus de memoire a un processus, il suffit d'allouer un nouveau frame en utilisant le FrameProvider de Machine et l'ajouter a la table d'adressage virtuelle. Vu que le mode d'adressage est virtuel, le processus ne saura pas qu'il est en train de travailler sur un espace physique non-contigûe...

Pour allouer, il faut donc:

- Obtenir un frame

- Ajouter le frame dans l'espace d'adressage du processus

Ce qui correspond a:

```
	if( (int) n > machine->frameprovider->NumAvailFrame() )
	{
		DEBUG( 'p', "Pas assez de memoire !\n" );
		return -1;
	}
	// Allocation de la nouvelle table des pages
	TranslationEntry* tempPageTable = new TranslationEntry[ brk + n ];
	// Copie de l'ancienne sous-partie
	memcpy( tempPageTable, pageTable, sizeof( TranslationEntry ) * brk );
	// Mise en place du reste
	for ( unsigned int i = 0; i < n; i++)
	{
		tempPageTable[i + brk].virtualPage = i;
		tempPageTable[i + brk].physicalPage =
				machine->frameprovider->GetEmptyFrame();
		tempPageTable[i + brk].valid = TRUE;
		tempPageTable[i + brk].use = FALSE;
		tempPageTable[i + brk].dirty = FALSE;
		tempPageTable[i + brk].readOnly = FALSE;
	}
	brk += n;
	numPages = brk;
	pageTable = tempPageTable;
	machine->pageTableSize = brk;
	machine->pageTable = tempPageTable;
	return brk;
```

Attention: brk est le nombre total de pages. Donc, avant de le retourner dans l'appel malloc, il ne faut pas oublier de le convertir en adresse memoire:

```
	int brk = currentThread->space->Sbrk( 1 );
	if( brk < 0 )
	{
		DEBUG( 'p', "Plus de pages !\n", brk, PageSize );
		return 0;
	}
	else
	{
		DEBUG( 'p', "Nombre total de pages: %d (%d octets par page)\n",
				brk, PageSize );
		// Attention: brk indique l'indice * SUIVANT !! *
		return ( brk - 1 ) * PageSize;
	}
```

Dans une structure TranslationEntry, le booleen valid indique si la page est valide. Pour desallouer une page (donc pour free), il suffit donc:

- De marquer la propriete valid de cette page comme FALSE
- De relacher le frame

D'ou le code de free (on fera encore attention a convertir les unites d'adresse memoire en numero de page):

```
    if( p % PageSize )
    {
          printf( "Cette page n'est pas allouee!\n" );
          return;
    }
    // Liberer: attention a ne pas liberer completement:
    // il se peut que la page soit suivie d'une autre page !
    p /= PageSize;
    machine->pageTable[p].valid = FALSE;
    machine->frameprovider->ReleaseFrame( machine->pageTable[p].physicalPage );
```

Test de malloc et free

Pour tester malloc et free, on prepare un petit programme malloc.c, qui:

- Alloue de la memoire
- Ecrit dedans
- Lit les valeurs qu'il a ecrit
- Libere la memoire
- Re-essaye d'ecrire (ce qui devrait le faire planter)

Donc on teste:

```
> userprog/nachos -rs -x test/malloc
   Allocation...
   Ecriture dans 2560 (donc la page 20)
   Lecture:
     Quatre: 4
     Quarante cinq: 45
     Quatre cent cinquante six: 456

   Ecriture dans 2688 (donc la page 21)
   Lecture:
     Quatre: 4
     Quarante cinq: 45
     Quatre cent cinquante six: 456

   Free de la page 21
   Essai d'ecriture a la page 21 (devrait planter)
   La page virtuelle numero 21 n'est pas valide !
   Unexpected user mode exception 2 2688
   Assertion failed: line 608, file "exception.cc"
   Abandon
```

C'est bon, malloc et free marchent.

Etape 5: Disque

Système de fichiers NachOS

Nachos dispose d'un disque de 128Ko, réparti en 32 pistes, chacune composée de 32 secteurs de 128 octets. Il y a donc très peu de mémoire comparé aux standards d'aujourd'hui. Il nous permet néanmoins de créer des fichiers de taille fixe (3,75 Ko soit exactement 30 secteurs), et de les gérer dans un unique répertoire racine.

```
#define SectorSize          128    // number of bytes per disk sector
#define SectorsPerTrack     32     // number of sectors per disk track
#define NumTracks           32     // number of tracks per disk
```

Il nous était impossible de créer des sous-répertoires ou de créer des fichiers de taille plus importantes.

Le système de gestion de fichiers repose à la base sur un vecteur de bits (classe BitMap), dont l'entete est stocké dans le secteur 0 du disque (le FreeMapSector), chaque bit de ce secteur représente un unique secteur du disque.

Si un secteur est alloué, son bit correspondant sera marque à 1 ; si on contraire, on désalloue un secteur, son bit sera effacé donc remis à 0.

```
freeMap->Mark(secteurLibre); // on marque le bit correspondant au secteur a allouer
freeMap->Clear(sector);      // on retire la marque pour ce repertoire
```

Nous pouvons le manipuler à l'aide du fichier FreeMapFile, il nous suffit d'instancier un FreeMap puis de charger son contenu en mémoire principale. Toute modification devra etre écrite dans le FreeMapFile que devra etre lui-meme ecrit sur le disque, sans quoi les manipulations ne seraient pas prises en compte.

```
BitMap *freeMap = new BitMap(NumSectors);        // instanciation du bitmap
freeMap->FetchFrom(freeMapFile);                 // récupération des données
freeMap->WriteBack(freeMapFile);                 // écriture sur le disque
```

Nous disposons aussi d'un fichier pour manipuler le repertoire racine : directoryFile, dont l'entete est stocké dans le secteur 1 (directoryFileSector), il contient les données concernant les entrées du repertoire. A la différence du freeMapFile, ce dernier permet de récupérer des informations sur le répertoire racine et ainsi de les modifier.

```
Directory *directory = new Directory(NumDirEntries);
                                                 // instanciation du repertoire racine
directory->FetchFrom(directoryFile);             // récupération des données
currentD->WriteBack(directoryFile);              // écriture sur le disque
```

```
#define FreeMapSector        0
#define DirectorySector      1
```

Les fichiers FreeMapFile et DirectoryFile restent ouverts tout au long de l'execution de NachOS.

La classe FileSystem

Elle contient les méthodes necessaires à la creation et à la manipulation des fichiers ainsi que du repertoire racine.

A la base, chaque fichier possède :

- Un entête de fichier, stockée sur un secteur du disque (la taille de cette structure de données est égale à précisement la taille d'un secteur)
- Un nombre de blocs de données (30 pour être précis)
- Une entrée dans le répertoire racine du système de fichiers

Le système de fichier, en lui-même est composé de plusieurs structures de données :

- Un vecteur de bit pour les secteurs libres du disque (BitMap)
- Un répertoire avec les noms de fichiers et les entetes de ces fichiers.

Le vecteur de bit et le repertoire sont représentés par des fichiers normaux :

```
OpenFile* freeMapFile;
OpenFile* directoryFile;

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

Leurs entetes respectives sont stockées dans des secteurs spécifiques, ils pourront le système de fichier pourra les trouver au démarrage.

Ces deux fichiers restent ouverts tout au long de l'exécution de NachOS.

Pour la manipulation de fichiers, nous utilisons les méthodes Create, Open et Remove entre autres. Si l'opération réussit, les changements sont écrits directement sur le disque. Si l'opération échoue, il nous suffit de ne pas écrire les informations sur le disque.

La classe Directory

Le repertoire est une table d'entrée de taille fixe.

```
#define NumDirEntries        10
DirectoryEntry *table;       // Table of pairs:
                             // <file name, file header location>
```

Chaque entrée représente un unique fichier, et contient les informations suivantes :

- une chaine de caractère, pour le nom du repertoire
- un entier, pour le numéro de secteur où est stockée l'entete du repertoire
- un booléen, pour signaler une entrée libre ou non.

```
class DirectoryEntry {
  public:
    bool inUse;                     // Is this directory entry in use?
    int sector;                     // Location on disk to find the
                                    //   FileHeader for this file
    char name[FileNameMaxLen + 1];  // Text name for file, with +1 for
                                    // the trailing '\0'
};
```

La taille fixe des entêtes de fichiers imposent un nom de taille limitées (neuf caractères) :

```
#define FileNameMaxLen              9
```

Le constructeur initialise un repertoire vide d'une certaine taille, les méthodes FetchFrom et WriteBack sont utilisees pour récupérer le contenu du repertoire et pour ecrire les modificiations sur le disque.

La classe FileHdr

Elle contient les méthodes nécéssaires à la gestion des entetes de fichier du disque.

Les entetes de fichier sont utilisees pour situer la zone du disque où les données du fichier sont stockées. Ils sont implémentés comme étant une table de pointeurs de taille fixe, chaque entrée dans la table pointe vers le secteur du disque contenant la portion de données du fichier, l'allocation est donc directe.

```
#define NumDirect     ((SectorSize - 2 * sizeof(int)) / sizeof(int))      // soit 30

  private:
    int numBytes;                   // Number of bytes in the file
    int numSectors;                 // Number of data sectors in the file
    int dataSectors[NumDirect];     // Disk sector numbers for each data
                                    // block in the file
};
```

La taille de cette table a été implémentée par les concepteurs de NachOS pour qu'elle soit contenue dans un secteur du disque uniquement (2 entiers + un tableau de 30 entrées = 128 octets de données).

Un entete de fichier peut être initialisé de deux façons :

- pour un nouveau fichier, en modifiant les structures de données pour qu'elles pointent vers les nouveaux blocs du disque alloués, les secteurs alloués seront marqués dans le vecteur de bit

```
numSectors  = divRoundUp(fileSize, SectorSize);// calcule le nombre
                                               //de secteurs nécéssaires pour le nouveau fichier
for (int i = 0; i < numSectors; i++)
   dataSectors[i] = freeMap->Find();
```

- pour un fichier déjà existant, en chargeant l'entete de fichier déjà présent sur le disque.

```
FileHeader *fileHdr = new FileHeader;
fileHdr->FetchFrom(sector);
```

La désallocation se fait, elle, en retirant la marque des secteurs concernés dans le vecteur de bit :

```
    for (int i = 0; i < numSectors; i++)
    {
          ASSERT(freeMap->Test((int) dataSectors[i]));  // ought to be marked!
          freeMap->Clear((int) dataSectors[i]);
    }
```

Partie 1: implantation d'une hiérarchie de répertoires

Le problème donné est donc de créer une hierarchie de repertoires, à savoir des sous-répertoires et des moyens pour savoir où l'on se trouve et pour revenir en arrière.

Par convention, aucun nom de fichier ou de répertoire ne doit contenir d'espace, sous peine de recevoir un message d'erreur.

Modifications apportées

Etant donné que NachOS devra gérer à la fois des fichiers et des répertoires à partir des memes entrées, la convention de base pour ces memes entrées a été modifiée :

- les numéros de secteur seront initialisés à -1, cette valeur nous indique que l'entrée n'est pas utilisée
- le booléen inUse a été renommé en isFile, et est vrai si l'entrée correspond à un fichier, il sera à faux pour une entrée de repertoire
- le champ name reste inchangé

```
class DirectoryEntry {
  public:
    bool isFile;
    int sector;
    char name[FileNameMaxLen + 1];
};
```

```
#define PremiereEntreeNormale       2      // les entrees restantes dans le repertoire

   Hierarchie(1, 1);
         for (int i = PremiereEntreeNormale; i < tableSize; i++){
         table[i].sector = -1;
         table[i].isFile = FALSE;
   }
```

Le fichier directoryFile correspond maintenant au repertoire courant, c'est donc maintenant un pointeur sur le secteur d'entete du repertoire courant.

Les deux premières entrées du repertoire seront maintenant utilisées pour contenir les informations sur le repertoire courant « . » et le repertoire pere « .. ».

```
#define EntreeRepertoireCourant 0   // repertoire .
#define EntreeRepertoirePere    1   // repertoire ..

#define currentName "."
#define fatherName  ".."
```

Nous aurons aussi besoin de connaître les informations sur ces entrées, ie leur numéro de secteur respectifs, d'où l'ajout de deux attributs privés :

```
   int secteurPere;

   int secteurCourant;
```

Lorsque nous serons dans un repertoire, les informations concernant ces deux entrées seront ainsi récupérées, ce qui nous permettra une plus grande facilité de manipulation.

Un nouveau constructeur a été implémenté pour répondre à ce besoin d'information :

```
Directory::Directory(int size, char *name, int secteur, int secteurPere)
```

Nous pouvons ainsi passer les informations pour une bonne gestion de la hierarchie, et les faire correspondre à une méthode qui les assignera :

```
void Directory::Hierarchie(int secteur, int secteurPere){
   // l'entree .
   table[EntreeRepertoireCourant].isFile = FALSE;       // par convention
   table[EntreeRepertoireCourant].sector = secteur;     // son secteur
   strcpy(table[EntreeRepertoireCourant].name, currentName);

   // l'entree ..
   table[EntreeRepertoirePere].isFile = FALSE;
   table[EntreeRepertoirePere].sector = secteurPere;    // le secteur de son pere
   strcpy(table[EntreeRepertoirePere].name, fatherName);
}
```

CreateDirectory

```
bool FileSystem::CreateDirectory(char *name)
```

La creation d'un repertoire se fait en trois temps :

- La vérification du nom de repertoire passé en paramètre
- L'instanciation et la vérification des entrées du repertoire courant
- La création du repertoire

La première étape consiste uniquement à vérifier que le nom du répertoire à créer est conforme aux conventions du système de fichiers : sa longueur et les éventuels espaces qu'il contiendrait :

```
   if(strlen(name) > FileNameMaxLen) { ... }
   if(strchr(name, ' ') != NULL) { ... }
```

La seconde, elle, va instancier le repertoire courant et récupérer les informations le concernant à partir du disque :

```
   Directory *currentD = new Directory(NumDirEntries, "",
                              secteurCourant, secteurPere);    // repertoire courant
   currentD->FetchFrom(directoryFile);     // on recupere ses donnees
```

Nous devons ensuite nous assurer qu'il contient au moins une entrée libre et qu'il ne contient pas déjà un répertoire portant le nom passé en paramètre :

```
   if(currentD->IsFull())     // vérifie si toutes les entrées du repertoire
                              // sont occupées ou non
   if(currentD->FindIndexRep(name) != -1) // vérifie si un sous-répertoire
                                          // du répertoire courant ne porte
                                          // pas déjà ce nom
```

Une fois que l'on est sur de pouvoir créer un répertoire, nous devons chercher un secteur libre à partir du vecteur de bit et allouer un entête pour ce repertoire :

```
    int secteurLibre = freeMap->Find();
    FileHeader *dirHdr = new FileHeader;
    ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));
    dirHdr->WriteBack(secteurLibre);
```

Un nouveau dossier sera instancié, ainsi que son fichier correspondant, il aura comme secteur courant celui trouvé plus haut, et comme père le numéro de secteur du repertoire courant :

```
    // on cree un nouveau repertoire
    Directory *d = new Directory(NumDirEntries, name, secteurLibre, secteurCourant);
    OpenFile* fichierRep = new OpenFile(secteurLibre);         // on alloue
```

Il ne reste plus qu'à marquer le secteur correspondant à l'entete dans le vecteur de bit, à ajouter le nouveau repertoire dans le repertoire courant et à écrire les modifications en mémoire :

```
    currentD->AddRep(name, secteurLibre);   // on ajoute le repertoire aux
                                            // entrees du repertoire courant
    freeMap->Mark(secteurLibre); // on marque le bit correspondant au secteur a allouer
    d->WriteBack(fichierRep);    // on ecrit en memoire les entrees modifiees

    currentD->WriteBack(directoryFile);
    freeMap->WriteBack(freeMapFile);
```

Le nouveau répertoire a été créé et peut être atteint à partir du shell.

RemoveDirectory

```
bool FileSystem::RemoveDirectory(char *name)
```

La suppression d'un répertoire se fait en deux étapes :

- La vérification et l'instanciation des données nécéssaires
- La suppression du repertoire correspondant

Dans un premier temps, le système de fichier va vérifier que tous les paramètres sont corrects pour pouvoir poursuivre la suppression, tel qu'il y ait bien un répertoire correspondant au nom passé en paramètr, le fait que ce ne soit pas un fichier et le fait que le répertoire à supprimer soit vide).

```
if((indice = currentD->FindIndexRep(name)) == -1)
if(indice == -1 && currentD->Find(name) != -1)
if(!d->IsEmpty())
```

Par la suite, l'entete du repertoire sera désallouée puis le bit correspondant à ce secteur, effacé.

```
    dirHdr->Deallocate(freeMap);
    freeMap->Clear(sector);                 // on retire la marque pour ce repertoire
```

Et l'entrée correspondante dans le répertoire courant, supprimée :

```
    currentD->RemoveDir(name);
```

ChangeDirectory

```
bool FileSystem::ChangeDirectory(char *name)
```

Le parcours des répertoires se fait en deux étapes également :

- La vérification de la chaine de caractères passée en paramètre
- Le changement de répertoire

Comme pour les méthodes précédentes, des vérifications s'imposent :

```
    if((indice = currentD->FindIndexRep(name)) == -1)
    if(indice == -1 && currentD->Find(name) != -1)
```

Ensuite, nous pouvons nous occuper du changement en lui-même. Selon la chaine passée en paramètre, trois chemins s'offrent à nous :

- la chaine est « . », auquel cas, nous n'avons rien à faire
- la chaine est « .. », dans ce cas, nous allons rechercher et charger les informations correspondant au répertoire père

```
directoryFile = new OpenFile(secteurPere);       // on instancie le nouveau repertoire
secteurCourant = secteurPere; // on modifie les variables des secteurs courant et pere
Directory *fatherDir = new Directory(NumDirEntries, "",
                                                 secteurPere, 0);// 0 peu importe
fatherDir->FetchFrom(directoryFile);
secteurPere = fatherDir->ReturnSector(EntreeRepertoirePere);
```

- la chaine n'est ni « . », ni « .. », nous allons donc avancer d'un pas dans la hierarchie

```
directoryFile = new OpenFile(indice2);// directoryFile associé au répertoire courant
secteurPere = secteurCourant;       // secteur père du nouveau répertoier courant
secteurCourant = indice2;    // secteur du repertoire auquel on veut accéder
```

PrintDirectory

```
void FileSystem::PrintDirectory()
```

Nous appellons la méthode List de la classe FileSystem.

Partie 2: Implantation de la table système des fichiers ouverts

Pour ce problème, nous enregistrons la liste des secteurs des entetes des fichiers ouverts. Cette liste est propre a chaque processus.

```
    Liste* fichierOuvert;            // table des fichiers ouverts
```

Etant une liste et non pas un tableau, elle n'a pas de limite de taille.

Vu qu'il n'y a pas de fonction de fermeture pour les fichiers ouverts, on doit enregistrer et supprimer les elements de la liste lors des appels au constructeur et destructeur de la classe OpenFile.

```
OpenFile::OpenFile(int sector)
{
   [...]
   if(sector!=FreeMapSector && sector!=DirectorySector)
          fileSystem->fichierOuvert->Add((void*)sector);
}
```

```
OpenFile::~OpenFile()
{
   [...]
   if(secteurHdr!=FreeMapSector && secteurHdr!=DirectorySector)
          fileSystem->fichierOuvert->Remove((void*)secteurHdr);
}
```

Les fichiers freeMapFile et directoryFile restant ouverts tout au long de l'exécution, il n'est pas nécéssaire de les insérer dans la liste.

Partie 3: Support des accès concurrents

Une solution est d'interdire a un thread d'ouvrir un fichier deja ouvert. Pour ce faire, on utilise la liste des fichiers ouverts du systeme. Si un thread veut d'ouvrir un fichier, on regarde si ce fichier est deja present dans cette liste des fichiers ouverts, et si c'est le cas, on l'en empeche.

```
// on ouvre qu'une seule fois un fichier
if( fichierOuvert->FindVal((void*)sector)==NULL )
   openFile = new OpenFile(sector);
```

Nous avons aussi ajoute, dans la classe thread, une liste des fichiers ouverts par le thread. Cette liste contient les adresses des « openfile » ouvert par le thread.Grace a cette liste, on peut donner aux utilisateurs un identifiant unique pour chaque fichier ouvert sans avoir a donner un lien sur un espace systeme. De plus, cette liste permet de connaître les fichiers ouverts par le thread et de verifier que le thread n'accede bien qu'a des fichiers qu'il a ouverts :

D'ou le code dans le traitant de Read et de Write :

```
int id=machine->ReadRegister(6); // id du fichier
if(currentThread->fichierOuvert->Find(id)!=NULL)
{
	[...]
	OpenFile* fichier=(OpenFile*)currentThread->fichierOuvert->Find(id);
	[...]
}
```

Et pour terminer, quand le thread se termine, on peut s'assurer qu'il a bien ferme tous les fichiers qu'il a ouvert.

```
if(!fichierOuvert->IsEmpty())
{
	printf("Le processus %s a encore des fichiers ouvert.\n",name);
}
ASSERT(fichierOuvert->IsEmpty());
delete fichierOuvert;
```

Partie 4: Augmentation de la taille maximale des fichiers

Introduction

Pour augmenter la taille maximale des fichiers, plusieurs solution s'offrent a nous. La plus simple, consiste a faire correspondre chaque entree de l'entete a une table d'indirection. Cette methode permet d'avoir une taille maximale de 120Ko (30*32*128o). Nous avons cependant prefere implementer une autre methode : nous transformons trois entree des directes en :

- une entree qui pointe vers une indirection simple
- une entree qui pointe vers une indirection double
- une entree qui pointe vers une indirection triple.

Et nous laissons les entrees restante en tant que entree sans indirection.

L'entete devient donc :

```
   #define NumDirect         ( (int) ((SectorSize - 5 * sizeof(int)) / sizeof(int)) )
[...]
 private:
   int numBytes;                    // Number of bytes in the file
   int numSectors;                  // Number of data sectors in the file
   int Sector1Indirection;          // secteur de l'indirection simple
   int Sector2Indirection;          // secteur de l'indirection double
   int Sector3Indirection;          // secteur de l'indirection triple
   int dataSectors[NumDirect];      // Disk sector numbers for each data
```

Cette methode presente plusieurs avantages : elle est plus rapide avec des petits fichiers, elle authorise des fichiers de taille maximale de plus de 4,13Mo. Par contre, elle est plus complexe a implementer.

La classe indirect

Nous avons cree une nouvelle classe d'indirection qui contient une tableau d'entree et le numero de secteur de l'indirection. Il y a deux methodes qui permettent d'ecrire la table sur le disque et de lire la table a partir du disque qui s'appellent respectivement FetchFrom() et WriteBack().

On obtient donc le code suivant :

```
#define TailleTableIndir     ( (int) (SectorSize/sizeof(int)) )
class Indirect {
  public:
   // position : secteur ou sera stocke la table d'indirection
   Indirect(int position);

   // ecrit "table" sur le disque
   void WriteBack();
   //lit "table" du disque
   void FetchFrom();

   // contient des numeros de secteur
   int table[TailleTableIndir];
private:
   // secteur ou est stockee la table
   int secteur;
};
```

Nous avons donc modifie les fonctions Allocate, Desallocate et ByteToSector.

Allocate

Pour allouer un espace disque, on commence par allouer les entrees sans indirections. Si la place allouable avec les entrees sans indirections n'est pas suffisante, on utilise l'indirection simple pour allouer les secteurs avec une indirection. Si la place allouable n'est pas encore suffisante, on alloue les secteurs avec deux indirections, et finalement si la place n'est encore pas assez grande, on alloue les secteurs avec trois indirections.

Pour allouer un secteur avec une indirection:

- on alloue un secteur a l'indirection

- on alloue le nombre de secteurs de donnees necessaire

- on stocke les numeros des secteurs alloues dans l'indirection

- on ecrit l'indirection sur le disque// une indirection

```
if(fileSize>=Taille1IndirMin )
{
        // on met dans la table d'indirection au plus "TailleTableIndir" entree
   int NbCopie=min(TailleTableIndir,NbSectRestant);
        // numero de secteur de la premiere table d'indirection
   Indirect *indir1;
   this->Sector1Indirection=freeMap->Find();
   indir1=new Indirect(this->Sector1Indirection);
   DEBUG('f',"Allocation du secteur d'indirection1 %d\n",this->Sector1Indirection);
   for(i=0;i<NbCopie;++i)
   {
        indir1->table[i]=freeMap->Find();
        DEBUG('f',"Allocation du secteur de donnees %d\n",indir1->table[i]);
   }
   NbSectRestant-=NbCopie;
        // on ecrit sur le disque l'indirection
   indir1->WriteBack();
   delete indir1;
}
```

Pour allouer des secteurs a deux indirections, on procede de maniere similaire :

- on alloue l'indirection double

- on alloue une indirection simple

- on enregistre le numero du secteur de l'indirection simple dans la table de l'indirection double

- on alloue le nombre de secteur necessaire

- on enregistre les numeros de secteur dans la table de l'indirection simple

- on ecrit la table d'indirection simple sur le disque

- si necessaire, on alloue une nouvelle indirection simple

- on enregistre le numero du secteur de l'indirection simple dans la table de l'indirection double

- etc...

- on enregistre sur le disque l'indirection double

```
   // deux indirections
   if(fileSize>=Taille2IndirMin && numBytes<Taille2IndirMax )
   {
            // on enregistre au plus "TailleTableIndir*TailleTableIndir" entree
            // nombre d'indirection pour la premiere page
      int NbCopie1=min(TailleTableIndir,divRoundUp(NbSectRestant,TailleTableIndir));
      Indirect *indir1;
      this->Sector2Indirection=freeMap->Find();
      indir1=new Indirect(this->Sector2Indirection);
      DEBUG('f',"Allocation du secteur d'indirection2 niveau1 %d\n",
                                                  this->Sector2Indirection);
      for(i=0;i<NbCopie1;++i)
      {
               // nombre d'entree a copier dans la deuxieme table indirection
            int NbCopie2=min(NbSectRestant,TailleTableIndir);
            Indirect *indir2;
            if(indir1->table[i]!=0)
            indir1->table[i]=freeMap->Find();
            indir2=new Indirect(indir1->table[i]);
            DEBUG('p',"Allocation du secteur d'indirection2 niveau2 %d\n",
                                                        indir1->table[i]);
            for(int j=0;j<NbCopie2;++j)
            {
                  indir2->table[j]=freeMap->Find();
                  DEBUG('f',"Allocation du secteur de donnees %d\n",indir2->table[j]);
            }
            NbSectRestant-=NbCopie2;
            indir2->WriteBack();
            delete indir2;
      }
      indir1->WriteBack();
      delete indir1;
   }
```

On procede de maniere similaire pour allouer des secteurs avec trois indirections.

Desallocate

Pour liberer l'espace disque allouer a un fichier, on desalloue les secteurs sans indirections. Ensuite, on desalloue les secteurs avec une indirection, apres ceux avec deux indirections et finalement, ceux avec trois indirections.

Pour desallouer des secteurs avec une indirection :

- on charge a partir du disque l'indirection simple
- on libere chaque secteur alloue dans la table d'indirection simple
- on libere la table d'indirection simple

```
   if(numBytes>=Taille1IndirMin)
   {
          // debut devient relatif aux secteurs correspondant a une indirection
      if(!freeMap->Test(Sector1Indirection))
          printf("On accede un secteur qui n'est pas alloue.\n");
      ASSERT(freeMap->Test(Sector1Indirection));

      int NbDesaloc1=min(TailleTableIndir,NbSectRestant);
      Indirect *indir1=new Indirect(this->Sector1Indirection);
      indir1->FetchFrom();
      for(int i=0;i<NbDesaloc1;++i)
      {
          if(!freeMap->Test(indir1->table[i]))
          {
              printf("On desalloue le secteur %d qui n'est pas alloue.\n",
                                                          indir1->table[i]);
          }
          ASSERT(freeMap->Test(indir1->table[i]));
          DEBUG('f',"Desallocation du secteur de donnees %d\n",indir1->table[i]);
          freeMap->Clear(indir1->table[i]);
          indir1->table[i]=0;
          --NbSectRestant;
      }
      indir1->WriteBack();
      delete indir1;
      DEBUG('f',"Desallocation du secteur d'indirection1 %d\n",
                                                    this->Sector1Indirection);
      freeMap->Clear(this->Sector1Indirection);
      this->Sector1Indirection=0;
   }
```

On procede de maniere similaire pour liberer les secteurs avec deux et avec trois indirections.

ByteToSector

Pour renvoyer le secteur ou est stocke l'octet dont l'offset est passe en parametre, il faut savoir si on accede a l'octet en question sans indirection, avec une indirection, deux indirections ou trois indirections. Pour se faire, il suffit de comparer l'offset avec la taille maximale atteignable sans indirection, avec une indirection, deux indirections et trois indirections. Si l'octet est accessible sans indirection, on renvoie son indice dans le tableau de entrees sans indirections :

```
      if(offset<=TailleDirectMax-1)
      {
          return dataSectors[offset/SectorSize];
      }
```

Si il faut une indirection :

- on charge l'indirection simple
- on calcule l'indice du secteur desire
- on renvoie le numero du secteur correspondant

```
    else if(offset>=Taille1IndirMin-1 && offset<=Taille1IndirMax-1)
    {
            // offset devient relatif au debut des secteurs avec une indirection
        offset-=(Taille1IndirMin-1);
            // on recupere le resultat dans la table d'indirection
        Indirect *indir=new Indirect(Sector1Indirection);
        indir->FetchFrom();
        int result=indir->table[offset/SectorSize];
            // liberation de la memoire allouee
        delete indir;
            // renvoie du resultat
        return result;
    }
```

Si il faut deux indirections, on procede de maniere similaire :

- on charge l'indirection double
- on calcule l'indice de la table d'indirection simple dans la table d'indirection double
- on charge l'indirection simple
- on calcule l'indice du secteur voulu dans la table de l'indirection simple

D'ou le bout de code suivant :

```
else if(offset>=Taille2IndirMin-1 && offset<=Taille2IndirMax-1)
{
        // offset devient relatif au debut des secteurs avec deux indirections
    offset-=(Taille2IndirMin-1);
        // on va chercher l'indice du secteur cherche dans la premiere table
                                                                    // d'indirection
    Indirect *indir1=new Indirect(Sector2Indirection);
    indir1->FetchFrom();
    int indiceIndir1=offset / (SectorSize*TailleTableIndir);
        // offset de vient relatif au debut de la "indiceIndir1" sous-indirection
    offset-=indiceIndir1*SectorSize*TailleTableIndir;
        // on recupere l'indice du secteur dans la deuxieme table d'indirection
    Indirect *indir2=new Indirect(indir1->table[indiceIndir1]);
    indir2->FetchFrom();
    int result=indir2->table[offset/SectorSize];
        // liberation memoire et retour
    delete indir2;
    delete indir1;
    return result;
}
```

On procede de maniere similaire pour un octet accessible avec trois indirections.

On peut remarquer qu'on enleve toujours 1 lors des comparaisons entre l'offset et les tailles minimales et maximales atteignablse avec 0, 1, 2 ou 3 indirections. Cette difference est du au fait qu'un offset de 0 correspond au premier secteur.

Partie 5: Implantation de fichiers de taille variable

Pour permettre a un fichier de changer de taille, nous avons implementer une fonction Reallocate. Cette fonction appelle soit Allocate si la nouvelle taille est supperieure a l'ancienne taille, soit DesallocateFrom si la nouvelle taille est inferieure a l'ancienne.

Cette fonction est apellee a chaque fois que l'on essaye de lire ou d'ecrire au dela de la taille courante du fichier. Nous avons donc modifier les methodes ReadAt et WriteAt :

```
if(position+numBytes > fileLength)
{
   BitMap *freeMap = new BitMap(NumSectors);
   freeMap->FetchFrom(fileSystem->getfreeMapFile());
   hdr->Reallocate(freeMap,position+numBytes);
   freeMap->WriteBack(fileSystem->getfreeMapFile());
   hdr->WriteBack(secteurHdr);
   delete freeMap;
}
```

Nous avons donc modifier la fonction Allocate pour qu'elle puisse allouer des secteurs en plus des secteurs deja alloues. Cette modification ne gene pas l'ancienne utilisation de cette fonction car lorsqu'on alloue pour la premiere fois, la taille du fichier est nulle. Pour savoir si une entree est occupee ou non, on fixe la convention suivante : « si l'entree vaut 0, alors, elle n'est pas utilise ». Nous avons pour cela, cree un constructeur pour fileHdr qui initialise toutes les entrees a 0 :

```
FileHeader::FileHeader()
{
   numBytes=0;
   numSectors=0;
   for(int i=0;i<NumDirect;++i)
          dataSectors[i]=0;
   Sector1Indirection=0;
   Sector2Indirection=0;
   Sector3Indirection=0;
}
```

Avant d'allouer une page, on doit donc regarder si l'entree est nulle ou non. Pour les pages sans indirection, on obtient donc (sur la pages suivante) :

```
   // NbSectRestant est le nombre de secteur a alloue
   int NbSectRestant=divRoundUp(fileSize, SectorSize);

      // pas d'indirection, acces direct
      // si l'ancienne taille est superieur a TailleDirectMax, alors,
      //   le tableau d'acces direct est deja plein.
   if(fileSize>=TailleDirectMin && numBytes<TailleDirectMax )    // TailleDirectMin = 0
   {
            // on copie au plus NumDirect secteur dans la table directe
      int NbCopie=min(NumDirect,NbSectRestant);
      for(i=0;i<NbCopie;++i)
            if(dataSectors[i]==0)
            {
                  dataSectors[i]=freeMap->Find();
                  DEBUG('f',"Allocation du secteur de donnees %d\n",dataSectors[i]);
            }
      NbSectRestant-=NbCopie;
   }
      // si on avait deja alloue ces secteurs, il ne sont plus a allouer
   else if(numBytes>=TailleDirectMax)
      NbSectRestant-=NumDirect;
```

Dans le cas ou on alloue des secteur avec une indirection, il faut regarder si l'entree de l'indirection ets nulle ou non et eventuellement allouer un secteur pour l'indirection. A partir de la, on peut allouer dans des entrees non-allouees, les secteurs de donnees.

```
   // une indirection
   if(fileSize>=Taille1IndirMin && numBytes<Taille1IndirMax )
   {
          // on met dans la table d'indirection au plus "TailleTableIndir" entree
      int NbCopie=min(TailleTableIndir,NbSectRestant);
          // numero de secteur de la premiere table d'indirection
      Indirect *indir1;
      if(Sector1Indirection!=0)
      {           // indirection simple existant deja, on la charge du disque
          indir1=new Indirect(this->Sector1Indirection);
          indir1->FetchFrom();
      }
      else
      {           // indirection simple pas encore allouee, on l'alloue
          this->Sector1Indirection=freeMap->Find();
          indir1=new Indirect(this->Sector1Indirection);
          DEBUG('f',"Allocation du secteur d'indirection1 %d\n",
                                                  this->Sector1Indirection);
      }
      for(i=0;i<NbCopie;++i)
      {
          if(indir1->table[i]==0)
          {
              indir1->table[i]=freeMap->Find();
              DEBUG('f',"Allocation du secteur de donnees %d\n",indir1->table[i]);
          }
      }
      NbSectRestant-=NbCopie;
          // on ecrit sur le disque l'indirection
      indir1->WriteBack();
      delete indir1;
   }
   else if(numBytes>=Taille1IndirMax)
      NbSectRestant-=TailleTableIndir;
```

On voit que si la taille deja allouee est supperieure a la taille accessible avec une indirection, on enleve du nombre de secteurs a allouer le nombre de secteurs allouable avec une indirection.

On procede de maniere similaire pour deux et trois indirections.

Nous avons aussi implementer la fonction DesallocateFrom(BitMap *freeMap,int debut) qui desalloue les secteurs a partir de l'offset « debut ». On commence par trouver le secteur correspondant a « debut » et a partir de ce secteur, on supprime les secteurs suivant. Pour savoir si il faut supprimer la pages qui contient l'octet « debut », on regarde si « debut » est aligne sur le debut de la page. Si c'est le cas, on peut supprimer la page, sinon, on ne doit pas la supprimer.

Pour les secteurs accessibles sans indirection, on obtient :

```
// pas d'indirection
if(debut<=TailleDirectMax)
{
        // NbDesaloc est le nombre de secteur accessible sans indirection a liberer
   int NbDesaloc=min(NumDirect,NbSectRestant);
   for(int i=debut/SectorSize+(aligne?0:1);i<NbDesaloc;++i)
   {
        if(!freeMap->Test(dataSectors[i]))
              printf("On desalloue le secteur %d qui n'est pas alloue.\n",
                                                                  dataSectors[i]);
        ASSERT(freeMap->Test(dataSectors[i]));
        DEBUG('f',"Desallocation du secteur de donnees %d\n",dataSectors[i]);
        freeMap->Clear(dataSectors[i]);
        dataSectors[i]=0;
        --NbSectRestant;
   }
}
```

De meme, pour savoir si on desalloue une indirection, si c'est une indirection accessible a partir de l'entete, on regarde si « debut » est inferieur a la taille allouable avec cette entree. Si c'est le cas, on libere l'entree. Si l'indirection n'est pas accessible directement a partir de l'entete du fichier, on regarde si « debut » est aligne sur un multiple de la taille allouable avec une entree de la table de l'indirection. Pour les secteurs accessible avec deux indirections, on obtient le code qui vous sera donne sur la page suivante :

```
    //deux indirections
    if(numBytes>=Taille2IndirMin && debut<=Taille2IndirMax )
    {
            // debut devient relatif aux secteurs correspondant a deux indirection
       debut-=Taille2IndirMin;
       if(debut<0)
       {            // si debut etait inferieur a Taille2IndirMin, il serai devenu <0
              debut=0;
              aligne=TRUE;
       }
       if(!freeMap->Test(this->Sector2Indirection))
              printf("On accede a un secteur qui n'est pas alloue.\n");
       ASSERT(freeMap->Test(this->Sector2Indirection));

              //nombre de premiere indirection a liberer
       int NbDesaloc1=min(TailleTableIndir,divRoundUp(NbSectRestant,TailleTableIndir));
       Indirect *indir1=new Indirect(this->Sector2Indirection);
       indir1->FetchFrom();
       for(int i=debut/(SectorSize*TailleTableIndir);i<NbDesaloc1;++i)
       {
              if(!freeMap->Test(indir1->table[i]))
                     printf("On accede a un secteur qui n'est pas alloue.\n");
              ASSERT(freeMap->Test(indir1->table[i]));

                     // on calule le nombre de secteur a desallouer pour cette indirection
              int NbDesaloc2=min(TailleTableIndir,NbSectRestant);
              Indirect *indir2=new Indirect(indir1->table[i]);
              indir2->FetchFrom();
              int j_min=(debut-i*SectorSize>=0)?(debut-i*SectorSize)/SectorSize+
                                                                    (aligne?0:1):0;
              for(int j=j_min;j<NbDesaloc2;++j)
              {
                     if(!freeMap->Test(indir2->table[j]))
                            printf("On desalloue le secteur %d qui n'a pas ete alloue.\n",
                                                              indir2->table[j]);
                     ASSERT(freeMap->Test(indir2->table[j]));
                     DEBUG('f',"Desallocation du secteur de donnees %d\n",
                                                                  indir2->table[j]);
                     freeMap->Clear(indir2->table[j]);
                     indir2->table[j]=0;
                     --NbSectRestant;
              }
              indir2->WriteBack();
              delete indir2;
              if(j_min==0)
              {
                     DEBUG('f',"Desallocation du secteur d'indirection2 niveau2 %d\n",
                                                                  indir1->table[i]);
                     freeMap->Clear(indir1->table[i]);
                     indir1->table[i]=0;
              }
       }
       indir1->WriteBack();
       delete indir1;
       if(debut==0)
       {
              DEBUG('f',"Desallocation du secteur d'indirection2 niveau1 %d\n",
                                                            this->Sector2Indirection);
              freeMap->Clear(this->Sector2Indirection);
              Sector2Indirection=0;
       }
    }
```

Partie 6: Implantation des noms de chemin

```
bool FileSystem::ParsePath(char *path)
```

Un parseur de chaine a été implémenté pour répondre à ce problème, il s'agit d'un automate qui évoluera selon le morceau de chaine qu'il recevra (« / », « . », « .. » ou une suite de caractères).

```
enum Etat {E_INIT, E_SLASH, E_POINT, E_POINTPOINT, E_STRING, E_FIN, E_ERR};
```

Si un morceau est correct et est contenu dans les entrées du repertoire courant, on change automatiquement de répertoire pour se placer dans celui dont le nom est contenu dans le morceau de chaine. Nous utilisons un compteur pour nous permettre à tout moment de revenir au début si la chaine passée en paramètre se révèle être incorrecte.

```
   int nbPas = 0;    // pour revenir en arrière si besoin est
```

```
case E_STRING :                     // si on a lu une chaine de caractère quelconque
   if(!ChangeDirectory(string))     // si on ne peut pas changer de répertoire
         etatCourant = E_ERR;
   else {
         nbPas++;                   // on avance d'un pas
         [...]                      // gestion des états
   }
   break;
```

```
case E_POINTPOINT :                 // si on a lu « .. »
   if(!ChangeDirectory(string))     // si on ne peut pas changer de répertoire
         etatCourant = E_ERR;
   else {
         nbPas--;                   // on revient d'un pas en arrière
         [...]                      // gestion des états
   }
   break;
```

```
case E_ERR :
   for(int i = 0 ; i < nbPas ; i++)       // si erreur il y a eu,
         ChangeDirectory(fatherName);     // on revient en arrière de nbPas
   [...]                                  // gestion de cet état
   return FALSE;
break;
```

Bonus : commande pwd

Parmi les commandes implémentées pour les besoins du shell, la commande pwd a été rajoutée pour parfaire le confort de l'utilisateur. Elle a la même utilité que la commande du même nom sous Unix, elle permet l'affichage du nom complet du repertoire en cours.

Son principe, sous NachOS, est d'utiliser une chaine de caractères (de longueur maximale fixe : 128 caractères) qui sera complétée ou partiellement effacée selon le répertoire vers lequel on se dirige avec la commande cd.

```
#define PathMaxSize          128
char *stringPath;                    // path courant dans la hierarchie de repertoires
```

Par défaut, donc dans le répertoire racine, le path contiendra « / », peut importe le répertoire où elle se trouve, la fin de chaine visible sera ce même caractère '/'. Chaque répertoire sera séparé par ce même tokem.

Si on avance d'un ou plusieurs pas dans la hierarchie, le nom des répertoires dans lesquels nous passerons seront concaténés à la chaine un par un :

```
void
FileSystem::StringPathMore(char *string){
   strcat(stringPath, string);// on concatène avec la chaine en paramètre
   strcat(stringPath,"/");          // puis on concatene la chaine avec le slash
}
```

Si, à l'inverse, nous reculons dans la hierarchie, la chaine sera analysée en arrière et sectionnée nom par nom jusqu'à arriver au répertoire courant :

```
void
FileSystem::StringPathLess(){
   int i = strlen(stringPath) - 2;  // longueur du path courant moins le \0
                                    // et le slash de fin de chaine
   while(i >= 0 && *(stringPath + i) !='/')
         i--;  // parcours de la chaine en arriere
   if(*(stringPath + i) == '/')           // la chaine est sectionnée
         *(stringPath + i + 1) = '\0';    // en forçant à \0 la fin de la nouvelle

}                                         // chaine
```

Etape 6: Reseau

Partie 1: Apprentissage des mecanismes reseau

Le fonctionnement du reseau sous NachOS

NachOS a deux classes principales pour le reseau: l'adaptateur reseau (Network) et le bureau de poste (PostOffice).

La classe Network

Cette classe n'est jamais directement utilisee par l'utilisateur. Elle fait le lien entre le bureau de poste NachOS et les sockets TCP/IP Linux. Elle dispose de quatre methodes:

- Send, qui envoie un paquet vers une machine NachOS

- Receive, qui recoit un paquet s'il y en a. Dans le cas ou il n'y a aucun paquet, retourne avec un paquet de taille 0.

- CheckPktAvail, qui est reveille regulierement et apelle un gestionnaire (donc le bureau de poste) quand un paquet arrive.

- SendDone, qui apelle un gestionnaire (donc le bureau de poste) quand l'envoi d'un paquet est termine (car la classe Network ne supporte pas les envois bufferises: il faut envoyer les paquets un par un).

A sa creation, Network commence a ecouter les requetes UDP sur le localhost. Donc, dans la configuration par defaut, deux machines NachOS doivent tourner sur la meme machine i386 pour pouvoir se communiquer.

Les entetes du reseau NachOS

Pour la communication, NachOS utilise deux entetes: PacketHeader et MailHeader.

PacketHeader contient trois informations:

- « to », qui est la machine de destination du paquet

- « from », qui est la machine d'origine du paquet

- « length », qui est la taille du paquet

PacketHeader est le seul entete dont la classe Network a besoin pour envoyer des donnees. Elle correspond a l'entete IP ou a l'entete Materiel de TCP/IP.

MailHeader contient les informations dont PostOffice a besoin pour mettre les paquets dans des boites de reception:

- « to », qui est la boite postale de destination du paquet

- « from », qui est la boite postale d'origine du paquet

- « length », qui est la taille du paquet

Similairement, MailHeader correspond a l'entete UDP ou TCP de TCP/IP.

La classe PostOffice

Cette classe est la classe utilisee pour les dialogues reseau par les applications. Elle marche par boites postales (ces boites postales correspondraient a des ports dans TCP/IP):

- Send est la methode a utiliser pour envoyer des donnees. La version initiale est une version qui n'est pas fiable.

- La methode Receive sert a recevoir des paquets: son premier argument est la boite a attendre. En effet, l'application attendra tant qu'il n'y a pas de message dans cette boite...

- PostalDelivery, PacketSent et IncomingPacket sont trois methodes utilisees pour que la classe Network puisse appeler PostOffice quand un message est recu ou l'envoi est termine.

Les premiers tests

nettest.cc propose un exemple d'utilisation de PostOffice:

- Il envoie un paquet au numero de machine donne en argument
- Attend pour un message et l'imprime une fois recu
- Quand un message est recu, envoie un autre paquet a la meme machine
- Attend pour un message et l'imprime une fois recu

Pour envoyer 10 messages, il suffit donc de faire un boucle tout simple:

```
    for( char i = 1 ; i < 11 ; i++ )
    {
          // A:  machine distante, boite 0
          // De: machine locale, boite 1
          outPktHdr.to = farAddr;
          outMailHdr.to = 0;
          outMailHdr.from = 1;
          outMailHdr.length = strlen(data) + 1;
          data[0] = i;
          // Premier message
          postOffice->Send(outPktHdr, outMailHdr, data);
          // Attendre le premier message
          postOffice->Receive(0, &inPktHdr, &inMailHdr, buffer);
          printf("Got \"%s\" with id %d from %d, box %d\n",
                       buffer + 1,buffer[0],inPktHdr.from,inMailHdr.from);
          fflush(stdout);
          ack[0] = i;
          // Repondre
          outPktHdr.to = inPktHdr.from;
          outMailHdr.to = inMailHdr.from;
          outMailHdr.length = strlen(ack) + 1;
          postOffice->Send(outPktHdr, outMailHdr, ack);
          // Lire la reponse
          postOffice->Receive(1, &inPktHdr, &inMailHdr, buffer);
          printf("Got \"%s\" with id %d from %d, box %d\n",
                       buffer + 1,buffer[0],inPktHdr.from,inMailHdr.from);
          fflush(stdout);
    }
```

Pour l'anneau, la configuation est un peu different:

- Le premiere machine envoie un message « hello » a la deuxieme machine et attend pour une reponse (a priori de la derniere machine de l'anneau)

- La deuxieme machine attend pour un message. Une fois recu, envoie le paquet vers la troisieme machine

- Idem pour les autres machines

- La derniere machine, qui attendait pour un message, le renvoie a la premiere une fois recue.

Donc le code:

```
    // Sauvegarder l'idenfiticateur
    char id = (char) postOffice->getNetAddr();
    data[0] = id + 1;

    printf("Membre %d de l'anneau a %d membres\n\n", id, farAddr );

    if( id == 0 )
    {
           printf("Demarrage de l'anneau a %d membres...\n", farAddr );

           // Premier membre de l'anneau
           outPktHdr.to = 1;
           outMailHdr.to = 0;
           outMailHdr.from = 1;
           outMailHdr.length = strlen(data) + 1;

           // Premier message
           postOffice->Send(outPktHdr, outMailHdr, data);

           printf("Anneau demarre :)\n");
    }

    // Attendre le message
    postOffice->Receive(0, &inPktHdr, &inMailHdr, buffer);
    printf("Message \"%s\" avec id %d recu du membre %d, dans la boite %d\n",
                  buffer + 1,buffer[0],inPktHdr.from,inMailHdr.from);
    fflush(stdout);

    if( id != 0 )
    {
           // Autre membre de l'anneau
           outPktHdr.to = ( 1 + id ) % farAddr;
           outMailHdr.to = 0;
           outMailHdr.from = 1;
           outMailHdr.length = strlen(data) + 1;

           // Envoyer
           postOffice->Send(outPktHdr, outMailHdr, data);
    }
```

Le test de l'anneau marche.

L'option de fiabilite fait perdre des paquets aleatoirement. On remarque que l'anneau ne marche plus!

Partie 2: Les transmissions fiables de taille fixe

On a choisi de tout implementer dans la classe PostOffice. Pour rendre la transmission fiable, la methode Send est remplace par SendReliable. Son fonctionnement est ainsi:

- PostOffice garde une queue de messages a envoyer par boite (la variable outbox)

- SendReliable ne fait que prendre le message et le mettre dans la queue

- Le thread « envoyeur » fait le tour de toutes les boites d'envoi, et reexpedie les messages pour lesquelles il n'a pas recu d'accuse de reception. Au bout de MAX_REEMISSIONS, le paquet est enleve de la liste.

Les boites d'envois sont definis comme ceci:

```
 struct ListeDenvoi {
       MailHeader mailHdr;
       PacketHeader pktHdr;
       char data[MaxMailSize];
       int retransmission_restant;
       ListeDenvoi* pnext;
};
```

On distingue deux types de messages:

- Les messages de donnees

- Les messages d'accuse (ACK)

```
typedef enum { normal, ACK } TypesDeMessage;
```

Pour etre fiable, chaque message a un identifiant d'accuse de reception. Cet identifiant, choisi par l'expediteur, sera repete dans l'accuse de reception et donc l'expediteur pourra savoir que son paquet a ete delivre. Pour les accuses de reception, on va modifier les entetes de courier (MailHeader):

```
struct MailHeader {
       TypesDeMessage type;
       int CompteurACK;
       MailBoxAddress to;
       MailBoxAddress from;
       unsigned length;
};
```

Donc, a chaque reception, le destinataire doit envoyer un ACK:

```
MailBoxAddress temp = mailHdr.to;
pktHdr.to = pktHdr.from;
mailHdr.to = mailHdr.from;
pktHdr.from = netAddr;
mailHdr.from = temp;
mailHdr.type = ACK;
Send( pktHdr, mailHdr, "" );
```

La fiabilite a une consequence: vu que les paquets sont reexpedies, il se peut qu'un client recoive le meme paquet a plusiers reprises. Il faut donc filtrer ces paquets:

```
if( boxes[mailHdr.to].dernier_recu == mailHdr.CompteurACK )
{
      DEBUG( 'n', "Doublon recu!\n" );
}
else
{
      DEBUG( 'n', "Message normal recu\n" );
      boxes[mailHdr.to].dernier_recu = mailHdr.CompteurACK;
      boxes[mailHdr.to].Put(pktHdr, mailHdr, buffer + sizeof(MailHeader));
}
```

Grace a la fiabilite, l'anneau marche meme si on met une fiabilite faible (attention, pas une fiabilite de 0 quand meme: il faut brancher un cable reseau pour envoyer des donnees!)

Partie 3: Les transmissions de longueur variable

Pour des transmissions de longueur variable, il faut un protocol qui definisse comment les paquets seront expedies. Pour nous, le protocole sera le suivant:

Du cote de l'expediteur:

- L'expediteur envoie un message contenent la taille de son envoi et attend pour une reponse « OK »

- Une fois « OK » recu, il commence a envoyer les paquets en utilisant SendReliable. On notera que SendReliable est un appel asynchrone: cet apel ne va en effet que mettre des paquets en queue. Donc, l'envoi ne sera probablement pas termine quand l'expediteur aura fait son dernier SendReliable.

- L'expediteur attend pour un message « Reception OK »

- Une fois ce message recu, BigSend retourne.

Du cote du destinataire:

- Le destinataire attend pour un message avec une taille

- Une fois ce message recu, envoie le message « OK » et boucle tant que la taille recue est inferieure a la taille totale

- Une fois que le gros message est recu, il envoie « Reception OK ». Quand ceci est termine, le paquet est reconstitue et BigReceive retourne.

On notera que dans BigReceive, il se peut qu'un paquet se perde (malgre les MAX_REEMISSIONS, par exemple on peut tout simplement faire un CTRL + C ou sur un vrai reseau deconnecter le cable reseau) et donc que l'on ne sorte jamais de la boucle. Pour eviter ce probleme, un timeout a ete mis pour la reception.

On notera aussi que le temps de transmission est retourne en secondes, donc le client FTP (voir apres) pourra calculer la vitesse de transmission.

Ce qui donne, pour BigSend:

```
    MailHeader t1;
    PacketHeader t2;
    char buffer[ MaxMailSize ];

    ASSERT( size > 0 );

    // Envoi de la taille
    mailHdr.length = sizeof( int );
    printf("Connexion vers %d sur %d\n", pktHdr.to, mailHdr.to );
     SendReliable(pktHdr, mailHdr, ( char* ) &( size ) );

    // Attend pour la reponse
    Receive(mailHdr.from, &t2, &t1, buffer, time( NULL ) + 10 );

    if( t1.CompteurACK < 1 )
    {
           printf( "Le serveur est indisponible... Re-essayez plus tard\n" );
           return -1;
    }

    printf("Connexion etablie vers %s! Envoi...\n", buffer );

    int demarrage = time( NULL );

    int current_size = 0;

    // Decouper & envoyer la donnee
    mailHdr.length = sizeof( BigMail );
    BigMail grande_donnee;
    while( current_size < size )
    {
           printf( "Envoi de l'octet %d sur %d\n", current_size, size) ;

           grande_donnee.size = min( ( int ) sizeof( grande_donnee.data ),
                                     size - current_size );

           memcpy( grande_donnee.data, data + current_size, grande_donnee.size );

           SendReliable(pktHdr, mailHdr, ( char* ) &( grande_donnee ) );

           current_size += grande_donnee.size;
    }

    // Attend pour la reponse
    Receive(mailHdr.from, &t2, &t1, buffer);

    printf("Envoi OK: %s\n", buffer );

    return ( time( NULL ) - demarrage );
```

On voit donc que BigSend retourn aussi le temps qu'il a mis pour envoyer la donnee.

Et pour BigReceive:

```
	BigMail donnee;
	int current_size = 0;
	char buffer[MaxMailSize];
	PacketHeader outPktHdr, inPktHdr;
	MailHeader outMailHdr, inMailHdr;
	// Attend pour une requete d'envoi
	Receive(box, &inPktHdr, &inMailHdr, buffer);
	current_size = *( ( int* ) buffer );
	printf("Reception du paquet de taille %d\n", current_size );
	ASSERT( current_size > 0 );
	*(size) = current_size;
	char* output = NULL;
	current_size = 0;

	if( *(size) > 0 ){  output = new char[ *(size) ];  }
	// OK, on peut commencer a recevoir
	outPktHdr.to = inPktHdr.from;
	outMailHdr.to = inMailHdr.from;
	outMailHdr.from = inMailHdr.to;
	outMailHdr.length = strlen( "NachOS" ) + 1;
	SendReliable(outPktHdr, outMailHdr, "NachOS");

	while( current_size < *(size) )
	{
		// On attend pour 5 secondes
		Receive(inMailHdr.to, &inPktHdr, &inMailHdr, ( char* ) &( donnee ),
					time( NULL ) + 5 );
		if( inMailHdr.CompteurACK > 0 )
		{
			if( outPktHdr.to == inPktHdr.from )
			{
				printf("Reception de l'octet %d sur %d\n",
					current_size + donnee.size, *(size) );
				memcpy( output + current_size,donnee.data, donnee.size );
				current_size += donnee.size;
			}
			else
			{
				printf( "Requete du deuxieme client rejete !\n" );
			}
		}
		else
		{
			// CompteurACK a 0 !!
			printf( "Timeout !\n" );
			delete[] output;
			*(mailHdr) = inMailHdr;
			*(pktHdr) = inPktHdr;
			*(size) = 0;
			return NULL;
		}
	}

	outMailHdr.length = strlen( "Reception OK" ) + 1;
	SendReliable(outPktHdr, outMailHdr, "Reception OK");
	*(mailHdr) = inMailHdr;
	*(pktHdr) = inPktHdr;
	return output;
```

Partie 4: Transfert de fichiers

Pour tester ce protocole, la methode la plus simple est sans doute d'envoyer des fichiers: le serveur, lance avec l'option -fs, fait tout simplement un BigReceive et le client, lance avec l'option -fc, fait tout simplement un BigSend.

Pour echanger les noms de fichier, le client va envoyer un grand paquet qui va contenir

- Le nom du fichier

- Un '\0'

- Le fichier

Le serveur, une fois le BigReceive reussi, va donc:

- Faire un strcpy, ce qui va lui permettre d'obtenir le nom du fichier

- Sauvegarde le reste de son buffer dans un fichier portant le nom specifie

L'implementation du cote serveur se fait donc de la facon suivante:

```
void FileReceive( int box )
{
   int size;
   char* buffer;
    MailHeader inMailHdr;
    PacketHeader inPktHdr;

   // Attend pour une requete d'envoi
   buffer = postOffice->BigReceive(box, &inPktHdr, &inMailHdr, &(size));

   if( size > 0 )
   {
          int length = strlen( buffer );
          char* file = new char[ length + 1 ];
          strncpy( file, buffer, length ); file[ length ] = '\0';

          printf( "Reception du fichier: %s\n", file );

          ofstream os;
          os.open( file, ios::binary );
          os.write( buffer + length + 1, size );
          os.close();

          printf("Reception et sauvegarde OK\n\n" );
          delete[] buffer;
   }
}
```

```
void FTPServer()
{
   for( ;; )
   {
          printf("Le serveur de transfert de fichiers est actif :)\n" );
          printf(" Veuillez faire un CTRL + C pour l'arreter...\n" );
          // Ce serveur ecoute la boite 0
          FileReceive( 0 );
   }
}
```

Et, du cote client:

```
void FileSend( int netId, int boxTo, int boxFrom, char* local, char* remote )
{
   printf( "Envoi du fichier: %s\n", local );

   int length;

   ifstream is;
   is.open( local, ios::binary );

   is.seekg (0, ios::end);
   length = is.tellg();
   is.seekg (0, ios::beg);

   if( length > 0 )
   {
          int temps;

          MailHeader outMailHdr;
          PacketHeader outPktHdr;
          int longueur_nom = strlen( remote );
          char* buffer = new char[ length + longueur_nom + 1 ];

          strcpy( buffer, remote );
          buffer[ longueur_nom ] = '\0';

          is.read( buffer + longueur_nom + 1, length );
          is.close();

          outPktHdr.to = netId;
          outMailHdr.to = boxTo;
          outMailHdr.from = boxFrom;

          if( -1 != ( temps = postOffice->BigSend(outPktHdr, outMailHdr, buffer,
                              length + longueur_nom ) ) )
          {
                 printf( "Envoi termine en %d secondes", temps );
                 if( temps ){  printf( ", donc %d KO/s",
                              length / ( temps * 1000 ) );  }
                 printf( "\n" );
          }

          delete[] buffer;
   }
   else
   {
          is.close();
          printf( "Ce fichier n'existe pas ou a une taille nulle !\n" );
   }
}
```

```
void FTPClient( int netId, char* local, char* remote )
{
   FileSend( netId, 0, 0, local, remote );

   interrupt->Halt();
}
```

Les tests avec de divers types de fichiers (texte, PNG, ZIP et PDF) ont marche avec succes, sauf dans le cas ou la taille est trop grande (et mandelbrot termine le processus).