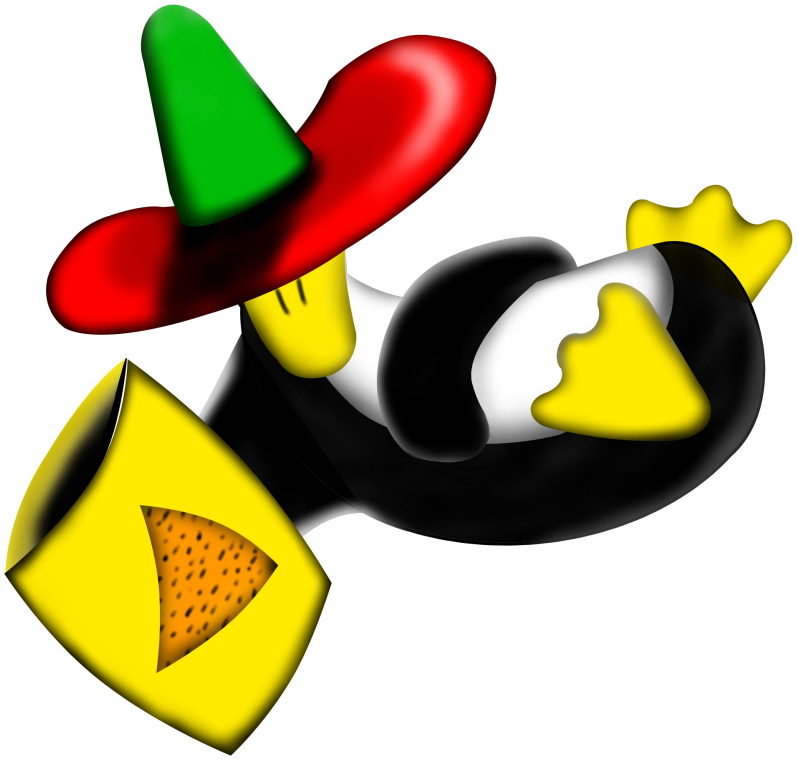
Travaux pratiques de système d'exploitation Première année de master informatique, U.E. SGP et SGM, parcours "systèmes et réseaux"

Année universitaire 2015-2016



Isabelle Puaut
Laurent Perraudeau

Table des matières

Introduction

Ce document contient les éléments de compréhension et les sujets de travaux pratiques des modules SGP (Systèmes d'exploitation, Gestion de Processus) et SGM (Systèmes d'exploitation, Gestion Mémoire), suivis par les étudiants de première année de master informatique, parcours "systèmes et réseaux".

Objectifs des travaux pratiques

Les travaux pratiques portent sur deux aspects complémentaires des systèmes d'exploitation :

- L'*utilisation* de l'interface de programmation d'un système d'exploitation (appels système). Ces travaux pratiques seront réalisés au dessus de Linux.
- La *réalisation* d'une partie de noyau de système d'exploitation. Pour ce faire, les noyaux de systèmes réels étant trop complexes, les travaux pratiques se dérouleront sur un petit noyau de système d'exploitation, nommé Nachos, qui possède comme bonnes propriétés : d'être relativement simple et peu volumineux, de s'exécuter au dessus d'une machine émulée par logiciel, ce qui facilite la mise au point.

Organisation du document

Le document est organisé comme suit. La structure interne de Nachos est présentée dans le chapitre 1. Les sujets de TP sont donnés dans les chapitres 2 et 3 pour chacun des deux semestres. Enfin, en annexe figurent quelques notes sur l'utilisation de *gdb*, un débogueur bien utile pour la mise au point du noyau (annexe A) et une FAQ d'utilisation de Nachos dans le cadre des TPs (annexe B).

Bugs, suggestions d'améliorations

Si vous trouvez un bug dans le code source de Nachos ou dans sa documentation, ou que vous avez envie de proposer des améliorations ou extensions au code source, merci de le signaler (e-mail : puaut@irisa.fr).

Chapitre 1

Un aperçu de Nachos

Ce chapitre a pour objectif de faciliter la compréhension de NACHOS et se propose de rentrer progressivement dans les détails de sa mise en œuvre. Il est important de noter que son contenu décrit le résultat final *attendu* suite à la réalisation des travaux pratiques. Certaines parties détaillées ici ne sont pas dans les fichiers sources qui vous seront fournis, et seront à réaliser pendant les travaux pratiques.

Après un bref historique sur la version de NACHOS que vous utiliserez, nous présentons dans le paragraphe 1.2 l'organisation générale du système, chacun de ses modules étant détaillé dans la suite. Le paragraphe 1.3 est consacré aux composants matériels, alors que le paragraphe 1.4 présente les pilotes de périphériques. Le paragraphe 1.5 explique le fonctionnement du cœur du noyau (gestion des tâches légères – threads – et de leur synchronisation). Les paragraphes suivants portent successivement sur le fonctionnement des pilotes de périphériques (paragraphe 1.4), sur le cœur du noyau (paragraphe 1.5), sur les appels système (paragraphe 1.6), et sur le système de gestion des fichiers (paragraphe 1.7). Les mécanismes de gestion de la mémoire virtuelle sont détaillés dans le paragraphe 1.8. Le paragraphe 1.9 présente les outils et fonctions utiles à la mise au point du noyau, et le dernier paragraphe explique les étapes de réalisation et d'exécution d'un programme utilisateur avec NACHOS.

1.1 Historique

NACHOS (*Not Another Completely Heuristic Operating System*) est un système d'exploitation à vocation pédagogique conçu à l'origine à l'université de Berkeley [1]. Son objectif est de permettre à des étudiants d'appréhender le fonctionnement interne d'un système d'exploitation. Pour ce faire, il dispose des mécanismes utilisés dans les systèmes commerciaux.

Ce logiciel a été modifié et étendu depuis 1999 à Rennes, tout d'abord lors d'un projet de développement de la quatrième année option informatique (équivalent master-1) de l'INSA de Rennes [2]. Le système a ensuite été remanié et étendu par Isabelle Puaut et Ivan Leplumey, puis par David Decotigny. C'est le résultat de ces différentes strates de modifications qui sera utilisé lors des TP et nommé NACHOS. Le fichier README, à la racine des sources, décrit, sans prétention d'exhaustivité, les modifications apportées au NACHOS d'origine.

1. Toutes les informations relatives à NACHOS pourront être trouvées sur la page d'accueil NACHOS à l'URL *http ://www.cs.berkeley.edu/~tea/nachos*

2. Le groupe était constitué de Matthieu Gabriac, Julien Gloaguen, Jérôme Le Dorze, Aurélien Letort, Antoine Mahé, Freddy Perraud, Anthony Remazeilles et Chloé Rispal et était encadré par Isabelle Puaut.

1.2 Présentation de la structure du système

Dans cette section, nous présentons les grandes composantes qui constituent le système NACHOS sans rentrer dans les détails de leur mise en œuvre.

1.2.1 Nachos est un processus Unix

L'objectif recherché lors de la construction de NACHOS était de permettre à des étudiants de se familiariser à investissement modéré avec les concepts de base des systèmes d'exploitation. Pour ce faire, NACHOS ne s'exécute pas directement au dessus du matériel de la machine hôte. À la place, il utilise un matériel (processeur, périphériques) *émulé*. L'ensemble formé par le matériel émulé, le noyau NACHOS et les applications, s'exécute au sein d'un même processus Unix (cf figure 1.1).

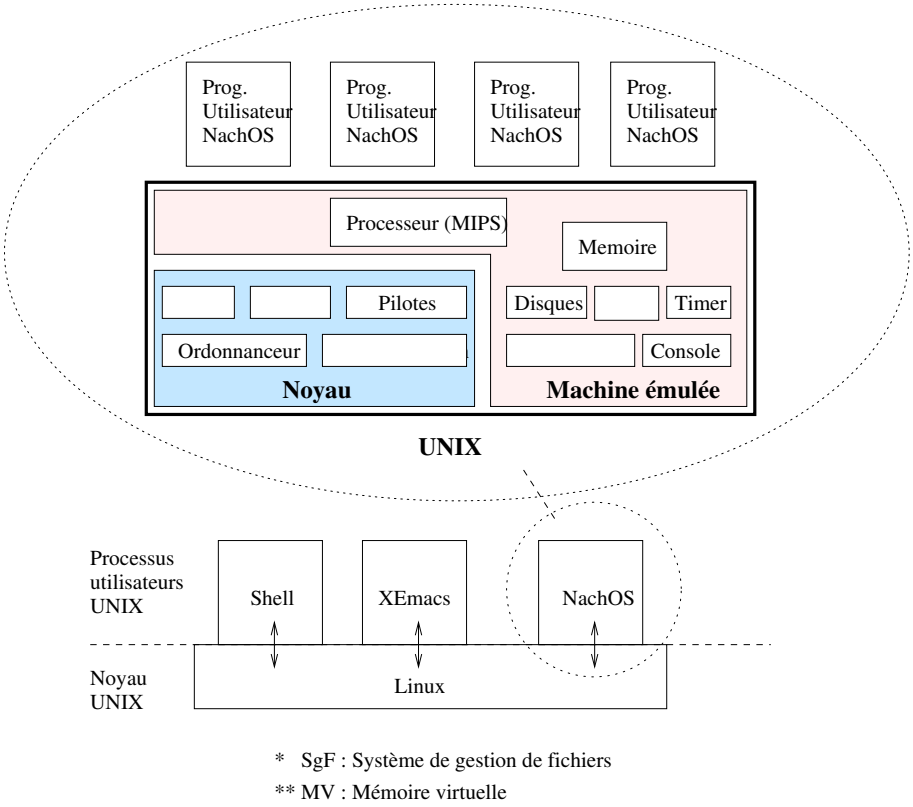


FIGURE 1.1 – Structure interne de NACHOS

L'intérêt d'une telle approche est double. D'une part, le développement du noyau est simplifié, car le matériel émulé est volontairement très simple. D'autre part, sa mise au point est simplifiée, puisque une erreur de programmation ne cause pas l'arrêt brutal de la machine hôte comme dans un système d'exploitation réel, et que les outils de mise au point classiques (par exemple *gdb*, cf annexe A) peuvent être utilisés.

NACHOS est développé en `C++`, en utilisant uniquement les caractéristiques de base du langage (encapsulation via les classes). Ceci permet de mieux structurer le code du noyau et

en faciliter la compréhension, sans nécessiter une connaissance approfondie du langage `C++` de la part des étudiants.

1.2.2 Émulation du matériel

Les principaux éléments matériels émulés sont un processeur MIPS sur lequel vont s'exécuter les programmes utilisateur, un timer, des disques, un coupleur série, une console et une unité de gestion de la mémoire (MMU - Memory Management Unit).

1.2.3 Modules principaux du noyau

La structure interne de Nachos met en évidence un découpage en modules, qui apparaît dans la structure des répertoires utilisée lors du développement. Les principaux modules sont :

- le *noyau* qui comprend les fonctionnalités vitales du système : taches légères (threads), primitives de synchronisation entre threads, ordonnancement ;
- les *pilotes* des périphériques émulés (disque, console)
- la gestion de la *mémoire virtuelle* ;
- la *gestion de fichiers*.

Les différentes classes `C++` du système sont réparties selon la structure de répertoires suivante :

- *machine* : classes d'émulation du matériel (ne doivent pas être modifiées) ;
- *kernel* : classes principales du noyau ;
- *vm* : classes de gestion de la mémoire virtuelle ;
- *drivers* : classes des pilotes de périphériques ;
- *filesys* : classes du système de gestion des fichiers ;
- *utility* : classes utilitaires (listes, statistiques, etc.) ;
- *userlib* : bibliothèque de fonctions pour les programmes utilisateur MIPS (à la libc) ;
- *test* : programmes utilisateur ;
- *doc* : documentation sur Nachos

Noyau de Nachos (répertoire *kernel*)

Le noyau de Nachos gère un ensemble de processus légers (threads) s'exécutant sur la machine MIPS émulée de manière quasi-parallèle. Nous nommons *processus* l'ensemble formé d'un espace d'adressage et des threads se le partageant.

L'ordonnancement des threads est réalisé selon une politique de type FIFO (First In, First Out), et ne gère aucune priorité. La synchronisation au sein du noyau se fait à l'aide de sémaphores, de variables de condition et de verrous. Par défaut, il n'y a pas de partage de temps (time-slicing).

Lorsqu'un thread est bloqué sur une primitive de synchronisation, un changement de contexte se produit et l'unité centrale est allouée au premier thread présent dans la file des prêts gérée par l'ordonnanceur (*readyList*). Si aucun thread n'est prêt, le système patiente jusqu'à l'arrivée d'une interruption.

Les différentes entités manipulées par Nachos sont identifiées par un type, codé par un entier (voir fichier *kernel/system.h*, type *ObjetTypeId*). Ce type est utilisé dans chaque appel système pour vérifier qu'il est bien appliqué sur le bon type d'objet.

Pilotes de périphériques (répertoire *drivers*)

Les pilotes de périphériques constituent la couche logicielle qui permet de faire fonctionner les composants matériels émulés. Ils permettent de passer des opérations asynchrones proposées par les émulations du matériel, à des opérations synchrones. Ceci se fait par l'utilisation des interruptions et des primitives de synchronisation (sémaphores et verrous). Sauf mention contraire explicite, les pilotes proposent des méthodes bloquantes qui rendent la main une fois que l'opération d'entrée/sortie est terminée.

Système de gestion des fichiers (répertoire *filesys*)

NACHOS dispose d'un système de gestion de fichiers, émulé au dessus du système de gestion de fichier Unix. Les fonctionnalités proposées sont :

— la création et suppression de répertoires ;
— la création et suppression de fichiers ;
— la lecture et l'écriture dans les fichiers.

Il est possible de copier un programme utilisateur, qui existe sous la forme d'un fichier Unix, dans le système de fichiers de NACHOS. Ce programme pourra alors être chargé puis exécuté par NACHOS. Le fichier de configuration NACHOS permet soit de conserver le contenu du disque entre deux lancements du système, soit de le ré-initialiser à chaque lancement.

Gestion de la mémoire virtuelle (répertoire *vm*)

NACHOS dispose d'une gestion complète de la mémoire virtuelle pour les programmes utilisateur MIPS : on utilise la traduction d'adresses d'une part, et le va-et-vient de pages entre la mémoire et le disque d'autre part.

La traduction d'adresses consiste à transformer à l'exécution, avec l'aide d'un support matériel (MMU), les adresses *virtuelles* manipulées par les programmes utilisateur MIPS en *adresses réelles* dans la mémoire de la machine MIPS. La traduction d'adresses utilise des tables stockées en mémoire sur la machine Unix. L'utilisation d'une zone de *swap* (ou zone d'échange) sur disque NACHOS permet, lorsque la mémoire est entièrement utilisée, de libérer des pages physiques en plaçant leur contenu sur disque dans la *zone d'échange*, ou *swap*[3]. Le défaut de page, mécanisme inverse, intervient pour charger une page présente dans la zone d'échange en mémoire réelle, ou pour signaler une erreur d'adressage.

1.2.4 Bibliothèque de fonctions pour les programmes utilisateur (répertoire *userlib*)

Les programmes utilisateur sont écrits en C. Des appels système, permettant de communiquer avec le noyau, ainsi que des routines diverses regroupées dans une bibliothèque liée aux programmes utilisateur, peuvent être utilisés. On ne dispose pas cependant de toutes les fonctionnalités des bibliothèques C classiques et encore moins de tous les appels système disponibles sous Unix (NACHOS reste un mini-système).

La conception du système, avec l'émulation d'une machine MIPS sur laquelle vont être exécutés les programmes utilisateur, impose à ces derniers d'être compilés en code MIPS. Ceci est réalisé par un compilateur croisé (cross-compilateur), en l'occurrence ici gcc, qui va générer du code MIPS exécutable sous NACHOS.

3. La zone de code, qui n'est pas modifiable, ne sera pas placée dans la zone de swap.

1.2.5 Paramétrage du système

Fichier de configuration (fichier nachos.cfg)

Il est possible de modifier certains paramètres de NACHOS et de la machine MIPS émulée sans avoir besoin de générer à nouveau l'exécutable `nachos`. Ceci est réalisé à l'aide d'un fichier de configuration qui comporte la valeur de paramètres tels que le nombre de pages de mémoire de la machine, le nom du programme MIPS à lancer au démarrage du système, le formattage du disque au démarrage, etc... Le format du fichier de configuration est détaillé dans le paragraphe 1.10.6.

1.2.6 Statistiques

Un module de statistiques permet de visualiser l'effet de certains paramètres du système sur ses performances. Les données sont collectées au cours du fonctionnement de NACHOS et portent sur les temps *virtuels* d'exécution des processus, sur les accès mémoires (nombre de défauts de page, nombre d'instructions exécutées), sur les accès au disque et à la console. Toutes les statistiques sont données processus par processus, les statistiques des différents threads du processus étant cumulées.

1.3 La machine émulée (répertoire *machine*)

1.3.1 Le processeur MIPS (fichiers *machine.cc, machine.h,* mipssim.cc, mipssim.h)

Le processeur émulé par NACHOS correspond à une architecture MIPS 32 bits. Le processeur dispose en interne de registres entiers, de registres flottants simple et double précision), et d'un registre de codes condition. L'objet *g_machine* de la classe *Machine* exporte plusieurs méthodes, permettant de lancer le processeur et d'inspecter son état (en particulier lecture et écriture des registres entiers et flottants) :

- **void Run()**, qui constitue la boucle de décodage et d'exécution des instructions du thread utilisateur en cours ;
- **int ReadIntRegister(int num)**, qui renvoie le contenu du registre entier *num* ;
- **void WriteIntRegister(int num, int value)**, qui écrit la valeur *value* dans le registre entier *num* ;
- **int ReadFPRegister(int num)**, qui renvoie la valeur du registre flottant *num* ;
- **void WriteFPRegister(int num, int value)**, qui écrit la valeur *value* dans le registre flottant *num* ;
- **char ReadCC(void)**, qui renvoie le contenu du code condition ;
- **void WriteCC(char val)**, qui initialise le contenu du code condition avec la valeur *val* ;

En outre, des constantes contiennent les numéros correspondants aux registres du processeur.

Certains registres entiers ont un rôle particulier lors des appels système pour le passage de paramètres (voir paragraphe 1.6).

L'objet g_machine renferme également les champs suivant :

- *mainMemory* : pointeur vers le tableau correspondant à la mémoire de la machine MIPS émulée ;

— un pointeur sur les différents composants de la machine : MMU (Memory Management Unit), ACIA (Asynchronous Communication Interface Adapter), contrôleur d'interruptions, disque et console.

1.3.2 Le contrôleur d'interruptions (fichiers *interrupt.cc, interrupt.h*)

Le contrôleur d'interruptions a pour rôle de définir si on doit ou non prendre en compte les interruptions qui sont générées par le matériel. L'objet global *interrupt* de la classe *Interrupt* correspond à ce contrôleur, et il exporte trois méthodes :

— **IntStatus SetStatus(IntStatus level)**, permet d'autoriser ou d'interdire les interruptions, et rend en résultat l'état précédent vis à vis des interruptions ;

— **IntStatus GetStatus()**, renvoie l'état courant vis à vis des interruptions.

Le type *IntStatus* est un type énuméré indiquant la (non) prise en compte des interruptions. Il contient les valeurs :

— *INTERRUPTS_OFF* : les interruptions sont interdites ;

— *INTERRUPTS_ON* : les interruptions sont autorisées.

1.3.3 La MMU (Memory Management Unit, fichiers *mmu.cc, mmu.h*)

Les adresses de données et d'instructions que le processeur MIPS émulé manipule sont *virtuelles*. Le rôle de l'unité de gestion de la mémoire est de lire et d'écrire les informations présentes à ces adresses virtuelles, en les traduisant en adresses physiques dans la mémoire physique (tableau *g_machine->mainMemory*, voir la section 1.3.1). La MMU utilise pour cela une *table de traduction d'adresses* (champ **translationTable** de la MMU).

La classe MMU propose les méthodes suivantes pour faire l'interface entre le système Nachos et le processeur MIPS simulé d'un côté, et la mémoire physique simulée de l'autre :

— **bool ReadMem(int vaddr, int size, int *value)**, qui remplit la donnée à l'adresse indiquée par *value* (de 1, 2 ou 4 octets de long), avec le contenu de ce qui se trouve à l'adresse virtuelle *vaddr* ;

— **bool WriteMem(int vaddr, int size, int *value)**, qui écrit la donnée à l'adresse indiquée par *value* (de 1, 2 ou 4 octets de long) vers l'adresse virtuelle *vaddr*.

Ces méthodes retournent *false* lorsque l'adresse virtuelle *vaddr* n'est associée à aucune adresse physique. Elles font appel à une méthode centrale : **ExceptionType Translate(int virtAddr, int* physAddr, int size, bool writing)**, dont le rôle est de récupérer l'adresse physique associée à une adresse virtuelle en fonction de la table de traduction d'adresse. Sommairement, il existe deux cas :

— La page associée à l'adresse recherchée n'est pas en mémoire physique, et la MMU déclenche un défaut de page. La routine prenant en charge ce défaut de page s'occupe d'aller charger la page désirée en mémoire physique ;

— La page associée à l'adresse est en mémoire physique, et la MMU retourne l'adresse physique correspondante.

La MMU participe à la gestion de la mémoire virtuelle. La compréhension de cet élément n'est pas nécessaire pour effectuer les premiers travaux pratiques. Son rôle est détaillé plus amplement dans la section 1.8.1, page 27.

1.3.4 Le coupleur série (fichiers *ACIA.h, ACIA.cc*)

Nachos possède une émulation de coupleur série qui permet d'envoyer ou de recevoir des

caractères via une liaison série. Pour simuler ce coupleur, on utilise les messages réseau UDP disponibles sur la machine Unix où est lancée NACHOS.

Le coupleur série est accessible au moyen de l'objet *acia*, instance de la classe *ACIA* (*Asynchronous Communication Interface Adapter*). Comme son nom l'indique, cet objet constitue l'interface du coupleur série.

Les registres de données

Le registre *outputRegister* (resp. *inputRegister*) constitue le registre de données en émission (resp. réception) et contient le caractère à envoyer (resp. recevoir). Les registres de données sont accessibles via les méthodes suivantes :

- **void PutChar(char)** place le caractère passé en paramètre, dans le registre *outputRegister* ;
- **char GetChar()** retourne le caractère contenu dans le registre *inputRegister* s'il est plein, 0 sinon.

Le coupleur série émet une interruption en réception lorsque le registre *inputRegister devient* plein et une interruption en émission lorsque le registre *outputRegister devient* vide.

Les registres d'état

Les registres *ouputStateRegister* et *inputStateRegister* indiquent l'état des registres de données. Ils sont accessibles en lecture uniquement, à travers les méthodes suivantes :

- **RegStatus GetOutputStateReg()** qui retourne le contenu du registre *outputStateRegister*, c'est-à-dire l'état (*i.e.* `EMPTY` ou `FULL`) du registre de données en émission *ouputRegister* ;
- **RegStatus GetInputStateReg()** qui retourne le contenu du registre *inputStateRegister*, c'est-à-dire l'état (*i.e.* `EMPTY` ou `FULL`) du registre de données en réception *inputRegister*.

Le registre de contrôle

Le registre de contrôle *mode* détermine le mode de fonctionnement du coupleur. Ce dernier peut fonctionner par test d'état/attente active, ou par interruptions. **void SetWorkingMode(int mod)**, méthode de la classe ACIA, permet de modifier le registre de contrôle *mode*, la méthode **int GetWorkingMode()** permet de le lire. Ce registre peut prendre les trois valeurs suivantes :

- *BUSY_WAITING* : pour fonctionner en attente active ;
- *SEND_INTERRUPT* : pour autoriser les interruptions en émission ;
- *REC_INTERRUPT* : pour autoriser les interruptions en réception.

Un masque peut être réalisé à l'aide de ces différentes constantes. Par exemple, pour autoriser simultanément les interruptions en émission et en réception, un ou bit à bit (opérateur "`|`" en `C`) est effectué entre *SEND_INTERRUPT* et *REC_INTERRUPT*.

1.3.5 Les disques (fichiers *disk.cc*, *disk.h*)

La machine utilise deux disques, l'un d'entre eux étant utilisé pour le système de gestion de fichiers, le deuxième pour la pagination à la demande (zone de swap).

Chaque disque (émulé) contient *NUM_TRACKS* pistes, chacune étant composée de *SECTORS_PER_TRACKS* secteurs. Chaque secteur a une taille fixe fournie au démarrage de NACHOS (directive de configuration **SectorSize**, voir la section 1.10.6). Deux méthodes permettent d'accéder au contenu d'un disque :

— **void ReadRequest(int sectorNumber, char* data)** lit un secteur sur le disque ;
— **void WriteRequest(int sectorNumber, char* data)** écrit un secteur sur le disque.

Ces deux méthodes retournent immédiatement, sans attendre la fin de l'entrée/sortie disque. La simulation du temps pris par la lecture ou écriture sur le disque, et la synchronisation pour attendre la fin du transfert, sont gérés par le pilote du disque (voir 1.4.3, page 17).

1.3.6 La console (fichiers *console.cc, console.h*)

La machine émulée contient une console, qui permet d'afficher des caractères à l'écran et de recevoir des caractères par le clavier. La console est gérée par son pilote (voir 1.4.2, page 17). La console émulée permet :

— l'affichage d'un caractère par la méthode **void PutChar(char ch)** ;
— la lecture d'un caractère au clavier par la méthode **char GetChar()**.

Lors d'un affichage, ou d'une écriture, de manière identique au disque, la fin de l'opération est spécifiée par une interruption demandée par la console.

1.4 Les pilotes de périphériques (répertoire *drivers*)

Les pilotes (drivers) de périphériques permettent au noyau de gérer les périphériques émulés par NACHOS (coupleur série, console, disque). Ils représentent les seuls moyens d'accès aux périphériques qui leur sont associés. De plus, les drivers assurent la synchronisation des entrées/sorties et gèrent le partage des périphériques en cadre multi-thread.

1.4.1 Le pilote du coupleur série (fichiers *drvACIA.cc, drvACIA.h*)

Ce pilote assure la synchronisation des envois et des réceptions de messages. Un objet global *g_acia_driver*, issu de la classe *DriverACIA* est créé lors du démarrage du système. Ce pilote peut être configuré vie le fichier de configuration NACHOS pour fonctionner soit en mode *attente active*, soit en mode *interruption* (en émission et en réception).

Description de la classe *DriverACIA*

Les membres de cette classe, qui seront utilisés exclusivement en mode interruption, sont :

— *send_buffer* et *receive_buffer* qui constituent respectivement les tampons en émission et en réception ;
— *send_sema* et *receive_sema* qui sont les sémaphores utilisés pour la synchronisation en émission et réception ;
— *ind_send* et *ind_rec* qui sont les indices de remplissage du tampon en émission et en réception.

Deux méthodes, accessibles via deux appels système, sont exportées :

— **int TtySend(char* buff)**, qui assure l'envoi de la chaîne de caractères passée en paramètre. En mode *interruption*, la primitive est *non bloquante* et se contente de

remplir le tampon en émission. En mode *attente active*, la primitive ne se termine que quand la chaîne de caractères a été intégralement transmise. La primitive renvoie le nombre de caractères envoyés ;

— **int TtyReceive(char* buff,int lg)**, qui assure la réception de chaînes de caractères via le tampon en réception et recopie la chaîne lue dans *buff*, de longueur maximale déterminée par *lg*. Cette primitive est *bloquante* et renvoie le nombre de caractères reçus et recopiés dans *buff*.

Ces deux méthodes renvoient le nombre de caractères effectivement envoyés (resp. reçus). Lors d'une utilisation en mode *interruption*, la taille des chaînes transmises est majorée par la taille des tampons (constante *MAXLINELEN*). Les routines de gestion d'interruptions sont les suivantes :

— **void InterruptSend()** est la routine de traitement d'interruption en émission responsable de l'envoi d'un caractère. Elle est systématiquement appelée lorsque le registre de données en émission de l'ACIA devient vide et les interruptions en émission sont autorisées ;

— **void InterruptReceive()** est la routine de traitement d'interruption en réception responsable de la réception d'un caractère. Elle est systématiquement appelée lorsque le registre de données en réception devient plein et les interruptions en réception sont autorisées.

Les routines de traitement d'interruptions détectent également la fin de l'envoi ou d'une réception de message.

Le choix du mode de fonctionnement du pilote (attente active ou interruption) est paramétrable au travers du fichier de configuration NACHOS (voir paragraphe 1.10.6) et est accessible par *cfg->ACIA*, qui peut prendre les valeurs *ACIA_BUSY_WAITING* ou *ACIA_INTERRUPT*.

1.4.2 Le pilote de la console (fichiers *drvConsole.cc, drvConsole.h*)

Il existe un pilote chargé de la synchronisation des routines d'écriture et de lecture via la console. L'objet *g_console_driver* issu de la classe *DriverConsole* assure ces opérations dans la console grâce à deux méthodes :

— **void PutString(char *buffer,int size)**

— **void GetString(char *buffer,int size)**

La console est un matériel asynchrone (une requête retourne immédiatement) ; le pilote rend l'utilisation de la console synchrone, en attendant la fin de l'opération désirée (signalée par une interruption). De plus, le pilote doit contrôler qu'il n'y a qu'une opération d'écriture à la fois, et une seule opération de lecture à la fois. Les méthodes du pilote de console sont appelées lors des appels système *Write(chaine*,longueur,ConsoleOutPut)* et *Read(chaine*,longueur,ConsoleInPut)* qui permettent respectivement d'écrire et de lire une chaîne de caractères sur la console. Il existe également dans la bibliothèque utilisateur de NACHOS (voir la section 1.10.1) des routines (*printf*, *Write*) permettant d'écrire vers la console.

1.4.3 Le pilote de disque (fichiers *drvDisk.cc, drvDisk.h*)

Cette classe permet de synchroniser les accès disque. Elle possède 3 attributs : un pointeur sur le disque dont les accès doivent être synchronisés, un sémaphore pour attendre la fin des entrées/sorties, un verrou pour assurer qu'une seule requête de lecture ou d'écriture ne peut être effectuée à la fois (le verrou est pris - respectivement relâché - au début - resp. à la fin -

des deux méthodes d'accès au disque). La classe définit les deux méthodes d'accès synchrones suivantes :

- **void ReadSector(int sectorNumber, char* data)**, lecture synchrone d'un secteur du disque ;
- **void WriteSector(int sectorNumber, char* data)**, écriture synchrone d'un secteur sur le disque.

Le caractère synchrone des méthodes repose sur deux routines de traitement d'interruption disque (*DiskRequestDone* pour le disque principal, *DiskSwapRequestDone* pour le disque de pagination).

1.5 Le noyau de Nachos (répertoire *kernel*)

1.5.1 Fonctionnement interne du noyau

NACHOS est un système d'exploitation pour des processus ayant des espaces d'adressage séparés, chaque processus étant parallèle (multi-thread). Quatre objets de NACHOS sont à la base de son fonctionnement :

- **Process** qui définit la notion de processus. Il s'agit d'une entité qui rassemble les ressources utilisées par une application du système. En particulier, un processus rassemble un ensemble de threads s'exécutant dans un espace d'adressage, associés aux objets suivants :
  - **Thread** qui contient les données nécessaires à la gestion des threads ;
  - **Addrspace** qui mémorise les données relatives à l'espace d'adressage.
- **Scheduler** qui gère l'ordonnancement des threads.

Objet Process (fichiers *process.cc, process.h*)

L'objet **Process** rassemble l'ensemble des ressources utilisées par chaque application du système :

- Champ **exec_file** : le fichier exécutable qui contient le code MIPS de l'application, et qui définit la structure de l'espace d'adressage (voir la section 1.5.1) ;
- Champ **addrspace** : l'espace d'adressage, commun à tous les threads du processus ;
- Champ **stat** : les statistiques d'utilisation des ressources par le processus (temps processeur consommé, nombre d'accès à la mémoire, ...).

Puisque chaque thread dispose d'une référence vers exactement un processus (voir ci-dessous), un objet processus rassemble donc implicitement un ensemble de threads.

Le constructeur de la classe **Process** a pour rôle d'initialiser un processus. On lui fournit le nom d'un chemin vers un fichier exécutable, et il construit l'espace d'adressage tel qu'il y est défini (voir ci-dessous).

Objet Thread (fichiers *thread.cc, thread.h*)

Un thread est un flot d'instructions exécuté par le processeur. Un objet de type *Thread* dans NACHOS correspond au contexte utile à l'exécution du flot d'instructions. En particulier, un tel contexte rassemble un pointeur de pile, un pointeur d'instruction, les autres registres du processeur, et une référence vers le processus auquel le thread appartient (pour définir l'espace d'adressage). Du fait de l'utilisation d'un processeur MIPS émulé et du fait que le

noyau s'exécute directement sur la machine hôte, le contexte d'un thread n'est pas limité au contexte de la machine MIPS émulée. À la place, son contexte est séparé en deux composantes :

— Le *contexte du thread*, constitué de l'état des registres de la machine MIPS : *thread_context.-int_registers* et *thread_context.float_registers*
— Le *contexte du simulateur*, constitué de l'état du simulateur MIPS, représenté par les variables d'état *simulator_context.buf* et le pointeur de pile *simulator_context.stackPointer*.

Cette notion de contexte du simulateur n'existerait pas si NACHOS s'exécutait directement sur un processeur MIPS. Cette notion a été introduite car NACHOS s'exécute sur un simulateur de processeur MIPS, réalisé par logiciel. Le contexte du simulateur sert alors à sauvegarder l'état d'exécution du simulateur.

Les principales méthodes de la classe *Thread* sont :

— **int Start(Process *owner, VoidFunctionPtr func, int arg)**, qui intègre le thread au processus **owner**. Cette méthode alloue et initialise les contextes du thread et du simulateur, et place le thread dans la file des prêts. Le pointeur *func* est un pointeur sur la fonction à exécuter (adresse de la première instruction du code du *Thread*) ;
— **Process* GetProcessOwner()** retourne le processus associé au thread ;
— **int Join(int Idthread)**, qui bloque le thread appelant jusqu'à la terminaison du thread *Idthread*. Renvoie -1 si *IdThread* est invalide ;
— **void Yield(void)**, qui met le thread appelant en queue de file des prêts. Utilisé pour relâcher volontairement le processeur au profit d'autres threads prêts ;
— **void Sleep(void)**, qui endort le thread appelant jusqu'à ce qu'on le réveille explicitement ;
— **void Finish(void)**, qui termine le thread appelant et planifie sa destruction ;
— **void SaveProcessorState(void)**, qui sauvegarde l'état des registres utilisateur (MIPS) ;
— **void RestoreProcessorState(void)**, qui restaure l'état des registres utilisateur (MIPS) ;
— **void InitSimulatorContext(int8_t *stackAddr, int stackSize)**, qui initialise le contexte noyau de l'objet **Thread**. Cette méthode est privée à la classe Thread et correspond essentiellement à l'initialisation de la pile pour l'exécution en contexte noyau ;
— **void InitThreadContext(int32_t initialPCREG,int32_t initialSP, int32_t arg)**, qui initialise le contexte de l'objet **Thread** (déclaration de la pile, de l'adresse de la première fonction MIPS à exécuter, et de l'adresse d'un paramètre à lui passer). Cette méthode est privée à la classe Thread.

Objet Addrspace (fichiers *addrspace.cc, addrspace.h*)

À chaque objet *Process* est associé un objet *Addrspace*, qui définit les adresses utilisables par tous les threads du processus. Le constructeur de l'objet *Addrspace* charge l'exécutable en mémoire et crée un espace mémoire associé (chargement du code, des données, ...). Ces notions seront vues au cours du second semestre et détaillées dans les TPs associés. Le format d'exécutable géré par NACHOS et chargé dans le constructeur de la classe *Addrspace* est le format ELF (Executable and Linkable Format).

Objet Scheduler (fichiers *scheduler.cc, scheduler.h*)

Une instance de la classe **Scheduler** est utilisée dans NACHOS. Elle sert à gérer la file des threads prêts (*readyList*) et à effectuer le changement de contexte entre les threads. Le

thread élu qui possède l'unité centrale ne fait pas partie de la file des prêts, mais est référencé par le pointeur global **g_current_thread**.

La classe Scheduler exporte trois méthodes :

— **void ReadyToRun(Thread *thread)** qui ajoute un thread en queue de liste des prêts (cette fonction suppose que les interruptions sont masquées) ;
— **Thread* FindNextToRun(void)** qui renvoie et enlève le premier thread de la liste des prêts ;
— **void SwitchTo(Thread *nextThread)** qui alloue le processeur au thread passé en paramètre (en général le résultat de **FindNextToRun**).

Méthode *SwitchTo*. Le changement de contexte entre deux threads (que les threads appartiennent ou non au même processus) est effectué dans la méthode *SwitchTo* du *Scheduler*. L'objectif du changement de contexte est de sauvegarder le contexte du thread appelant dans l'objet thread correspondant, et de restaurer celui du nouveau thread élu.

Dans un système d'exploitation s'exécutant sur machine nue, les applications et le système d'exploitation s'exécutent sur le même type d'architecture. De ce fait, seul le contexte relatif à cette architecture (registres du processeur essentiellement) doivent être sauvegardés. Ici, le fonctionnement est un peu différent, car les applications s'exécutent sur un processeur émulé (MIPS) alors que le noyau s'exécute directement sur la machine hôte. De ce fait, deux contextes doivent être gérés lors d'un changement de contexte : le contexte utilisateur (MIPS) et le contexte noyau.

La sauvegarde/restauration du contexte noyau vous est directement fournie (voir fonctions *getcontext/setcontext* appelées dans la méthode *SwitchTo*). Vous n'aurez pas à gérer le contexte noyau lors des travaux pratiques.

1.5.2 Outils de synchronisation (fichiers *synch.cc, synch.h*)

Trois types de synchronisation sont définis dans Nachos : les sémaphores, les verrous et les variables de condition. Toutes les méthodes relatives aux outils de synchronisation sont atomiques. Comme on est sur un système monoprocesseur, l'atomicité est mise en œuvre simplement en interdisant les interruptions.

Sémaphore

Contient un compteur et une file d'attente. Le fonctionnement est identique à celui des sémaphores à compteur vus en cours et en TD :

— **void P()** décrémente le compteur du sémaphore, puis fait attendre le thread appelant si ce compteur est devenu négatif ;
— **void V()** Incrémente le compteur du sémaphore, ce qui peut entraîner la libération d'un thread bloqué sur ce sémaphore.

Verrou

Ce mécanisme d'exclusion mutuelle possède un booléen et une file d'attente. Il est similaire aux sémaphores *binaires* (*i.e.* sémaphore dont le compteur est initialisé à 1), mais avec une notion de thread *propriétaire* du verrou :

— **void Acquire()** Le thread *s'approprie* le verrou s'il est libre, sinon il se place dans la file d'attente et s'endort ;

— **void Release()** Cette méthode est exécutable seulement par le *propriétaire* du verrou, pour relâcher le verrou. Elle regarde dans la file du verrou si un éventuel thread attend l'accès. Dans ce cas, elle retire ce thread de la file du verrou et le remet dans la file des prêts. Sinon elle positionne le verrou à "libre".

Variable de condition

Les variables de condition dans NACHOS sont des outils de synchronisation très simples. Un thread se met en état d'*attente* (primitive *Wait*) jusqu'à ce qu'un événement lui soit signalé (primitives *Signal* ou *Broadcast*). L'appel à *Signal* ou *Broadcast* si aucun thread n'est en attente est sans effet. Chaque variable de condition possède sa propre file d'attente.

— **void Wait()** Le thread appelant se place dans la file d'attente de la condition et s'endort ;
— **void Signal()** Le premier thread en attente de la condition est réveillé (oté de la file d'attente de la condition et mis dans la file des prêts). Dans le cas où aucun thread n'est en attente, la routine est sans effet ;
— **void Broadcast()** Même effet que signal mais sur *tous* les threads de l'ensemble d'attente.

1.6 Fonctionnement des appels système

1.6.1 Fonctionnement global

Les programmes utilisateur ont accès à une liste d'appels système qui sont invoqués par les programmes utilisateur en exécutant l'instruction MIPS *syscall*. Le rôle de cette instruction est de générer une exception. Celle-ci est récupérée par une routine de traitement d'exceptions située dans le fichier *exception.cc* du répertoire *kernel*. Cette routine va rediriger l'appel système, en fonction de son *type*, vers les pilotes de périphériques ou vers le noyau du système NACHOS.

Le *type* de l'appel système est fourni par les programmes utilisateur dans le registre *r2* de la machine MIPS, et les paramètres de l'appel sont stockés automatiquement par le compilateur dans les registres *r4*, *r5*, *r6* et *r7* selon leur nombre. La valeur de retour de l'appel, si il y en a une, est placée dans le registre *r2*.

La liste des appels système supportés par le noyau NACHOS est décrite dans le fichier *syscall.h* du répertoire *userlib*, et le code MIPS chargé de créer l'exception système se trouve dans le fichier *sys.s* du répertoire *userlib*. Ce dernier fichier assembleur (MIPS), une fois compilé, est lié avec les programmes utilisateur.

1.6.2 Exemple de déroulement d'un appel système

Nous détaillons ici le cheminement d'un appel système dans NACHOS. Prenons pour exemple un programme utilisateur dont l'effet est de créer un processus, et d'attendre la terminaison du thread de ce processus :

```
#include <userlib/syscall.h>
int main() {
        ThreadId newProc = Exec("/hello");
        Join(newProc);
   }
```

Soit, une fois compilé en assembleur MIPS (appel à *Join* seulement) :

```
lw      $4,16($fp)
jal     Join
```

Ce code généré pour l'appel à *Join* a placé la valeur située 16 octets plus haut que le cadre de pile (*i.e.* le contenu de la variable locale `newProc`), dans le registre *r4*.

Lors de l'édition de liens, *jal Join* effectue un branchement vers l'étiquette *Join* du fichier *sys.s* :

```
Join:
        addiu $2,$0,SC_Join
        syscall
```

Lors du décodage des instructions de l'application par la machine MIPS, l'instruction *syscall* provoque l'appel de la routine de traitement d'exceptions (*kernel/exception.cc*). Le type de l'exception système, ici *SC_Join*, permet à cette routine de savoir quelle est l'action à réaliser. Dans notre cas, nous allons demander au thread courant d'attendre la terminaison du thread passé en paramètre de l'appel système :

```
// Declaration de la variable qui va contenir l'identifiant
// passe en parametre
Thread* idThread;

// Recuperation de l'identificateur passe a l'appel systeme
idThread = (Thread*)g_machine->ReadIntRegister(4);

// Appel de la routine qui effectue l'operation Join
g_current_thread->Join(idThread);

// Écriture du code de retour dans le registre 2
g_machine->WriteIntRegister(2,0);
```

1.7 Système de gestion des fichiers (répertoire *filesys*)

Le système de gestion des fichiers propose des fonctions afin d'organiser un ensemble de fichiers dans une arborescence de style Unix. Chaque fichier contient un en-tête (*fileheader*), stocké sur disque, et décrivant où trouver les données du fichier sur le disque.

Le système de fichiers NACHOS possède sur disque : (i) une carte des secteurs libres du disque (classe BitMap) ; (ii) le répertoire racine du système de gestion de fichiers. Ces deux données sont elles-mêmes vues comme des fichiers, et sont stockées respectivement dans les secteurs 0 et 1 du disque. Tous les répertoires sont également vus comme des fichiers contenant le nom des fichiers et répertoires qu'ils regroupent, et les adresses disque des *fileheaders* de ces fichiers/répertoires. Lorsque des opérations sont effectuées sur l'un d'eux, les changements sont immédiatement enregistrés sur le disque si l'opération réussit, sinon aucun changement n'est enregistré.

Le système de gestion des fichiers gère les accès concurrents aux fichiers, organise la hiérarchie des répertoires et permet la création de gros fichiers et de fichiers de tailles extensibles. La figure 1.2 page 25 montre une vue synthétique de l'imbrication des différentes structures nécessaires au système de fichier en mémoire.

1.7.1 Classe *FileHeader* (fichiers *filehdr.cc, filehdr.h*)

Les objets du système de fichiers (répertoires et fichiers) possèdent une structure d'en-tête contenant les informations suivantes : taille de l'objet, nombre de blocs qu'il utilise, table des secteurs utilisés. La structure est soit enregistrée sur le disque, soit contenue dans une instance de la classe *FileHeader* quand le fichier/répertoire est ouvert. Les méthodes exportées par la classe *FileHeader* sont listées ci-dessous.

- **bool Allocate(BitMap *bitMap, int fileSize)** : initialise la structure d'en-tête de l'objet et alloue de la place sur le disque pour ses données. *bitMap* est la carte globale des secteurs libres du disque, qu'il faut avoir récupérée au préalable à partir du secteur 0 du disque (voir la classe *Bitmap*, section 1.9.3) ;
- **void Deallocate(BitMap *bitMap)** : désalloue les blocs de données de l'objet sur disque. *bitMap* est la carte globale des secteurs libres du disque, qu'il faut avoir récupérée au préalable à partir du secteur 0 du disque (voir la classe *Bitmap*, section 1.9.3) ;
- **void FetchFrom(int sectorNumber)** : initialise la structure d'en-tête d'un objet par lecture sur disque ;
- **void WriteBack(int sectorNumber)** : écrit sur le disque les modifications relatives à la structure d'en-tête,
- **int ByteToSector(int offset)** : renvoie le numéro de secteur correspondant à une donnée contenue au déplacement *offset* ;
- **int FileLength()** : renvoie la taille de l'objet en octets ;
- **void Print()** : affiche le contenu du fichier (pour le déboguage) ;
- **bool IsDir()** : permet de savoir si un objet est un répertoire ou non ;
- **void SetFile()** : mémorise qu'un objet est de type fichier ;
- **void SetDir()** : mémorise qu'un objet est de type répertoire.

1.7.2 Classe *FileSystem* (fichiers *filesys.cc, filesys.h*)

Ce module offre des routines pour initialiser le système de fichiers, pour créer, ouvrir ou effacer des fichiers et des répertoires :

- **FileSystem(bool format)**, initialise le système de fichiers. Si *format* est vrai, le disque est vidé et on initialise le système à un répertoire vide et la bitmap en conséquence.
- **int Create(char *name, int initialSize)**, crée un fichier dans le système. Cette méthode vérifie que le fichier n'existe pas déjà et que son nom est valide, alloue de la place sur le disque pour l'en-tête et pour les données du fichier, ajoute le fichier dans la table du répertoire concerné et met à jour les modifications effectuées sur le disque. Le paramètre *name* est le nom *absolu* du fichier (par exemple */toto*), car il n'y a pas de notion de répertoire de travail dans NACHOS.
- **OpenFile* Open(char *name)**, ouvre un fichier pour une lecture ou une écriture. On ramène l'en-tête concerné en mémoire.
- **int Remove(char *name)**, efface un fichier du système, retire le fichier du répertoire qui le contient, efface son en-tête et ses données, met à jour les changements au niveau de la bitmap et du répertoire.
- **void List()**, affiche les contenus de la bitmap, du répertoire de base et pour chacun de ses fichiers l'en-tête et les données.
- **int Mkdir(char *)**, crée un nouveau répertoire, vérifie que le nom est valide et que

le répertoire n'existe pas déjà, alloue de la place pour l'en-tête et pour les données et met à jour les changements sur le disque.
- **int Rmdir(char *)**, efface un répertoire après avoir vérifié qu'il existe et qu'il est vide.
- **bool decompname(char * name, char *head, char *tail)**, est utilisé pour gérer la hiérarchie des répertoires. *decompname* prend en argument un nom de fichier et le décompose en deux parties, retournées dans *head* et *tail*. Si le paramètre est *name=/dir1/dir2/fic* la fonction renvoie *dir1* dans head et */dir2/fic* dans *tail*. Retourne true en cas de succès, false sinon. Les deux pointeurs *name* et *tail* peuvent pointer vers la même chaîne, mais dans ce cas la chaîne en question est modifiée, puisque le marqueur de fin de chaîne (*i.e.* `\0`) y est inséré.
- **int FindDir(char *name)**, *name* est le nom absolu du fichier/répertoire à chercher et *FindDir* renvoie le secteur du dernier répertoire contenant le fichier/répertoire. Par exemple pour *name=/dir1/dir2/FicOuDir* on obtient le numéro de secteur de dir2 et `name` devient "FicOuDir". Cette méthode utilise la précédente et est appelée dans toutes les méthodes qui nécessitent de modifier le système de fichiers ou de vérifier la validité d'un nom. **Attention :** le contenu de la chaîne pointée par *name* est modifiée à l'issue de l'appel ! La fonction renvoie -1 lorsque le fichier/répertoire n'a pas été trouvé (mais `name` aura quand même été modifié !).
- **OpenFile *GetFreeMapFile()**, récupère la bitmap du disque.
- **OpenFile *GetDirFile()**, renvoie un *Openfile** sur le fichier du répertoire racine.

1.7.3 Classe *Directory* (fichiers *directory.cc, directory.h*)

La classe *Directory* est une table dont chaque entrée correspond à un fichier et contient le numéro de secteur sur lequel est sauvée sa structure d'en-tête. Cette classe contient les méthodes permettant de garder la cohérence du contenu des répertoires entre le disque et la mémoire.
- **void FetchFrom(OpenFile *file)**, lit le contenu du répertoire depuis le disque ;
- **void WriteBack(OpenFile *file)**, écrit les modifications concernant le répertoire sur le disque ;
- **int Find(char *name)**, retourne le numéro de secteur du *FileHeader* de *name*, ou -1 si *name* n'est pas trouvé ;
- **int Add(char *name, int newSector)**, rajoute un fichier de nom *name*, de secteur d'en-tête *newSector* au répertoire ;
- **int Remove(char *name)**, efface un fichier du répertoire ;
- **void List()**, liste tous les fichiers du répertoire ;
- **void Print()**, liste tous les fichiers du répertoire, l'emplacement de leurs structures d'en-tête et le contenu de chaque fichier ;
- **bool empty()**, retourne vrai si le répertoire est vide.

1.7.4 Classe *OpenFile* (fichiers *openfile.cc, openfile.h*)

Cette classe permet, pour un fichier ouvert, de mémoriser les informations sur son utilisation. Lorsqu'un fichier est ouvert, son *FileHeader* est situé en mémoire. Un *OpenFile* contient le nom du fichier qu'il représente, son *FileHeader*, ainsi que la position courante dans le fichier.

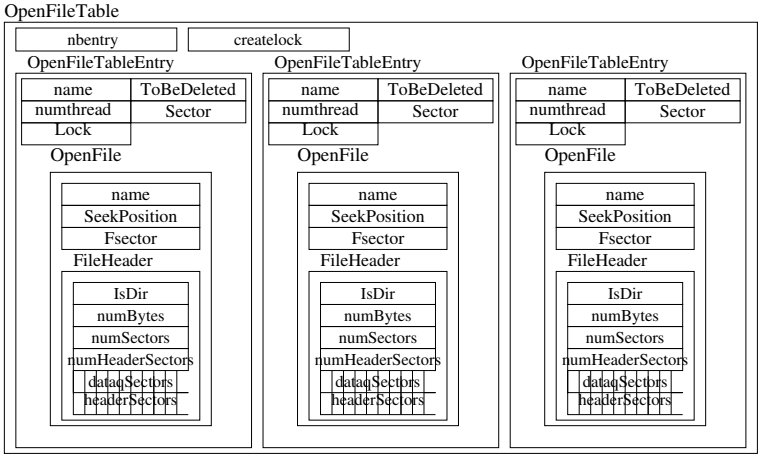


FIGURE 1.2 – Table des fichiers ouverts

— **OpenFile(int sector)** réalise l'ouverture du fichier correspondant au numéro de secteur passé en argument ;
— **~Openfile()** réalise la fermeture du fichier ;
— **void Seek(int position)** modifie la position courante dans le fichier ;
— **int Read(char *into, int numBytes)** réalise la lecture de *numBytes* octets à partir de la position courante, et les place dans le tampon *into*. Le contenu du fichier n'est pas conservé en mémoire (pas de gestion de cache disque), mais la position courante est incrémentée. Retourne le nombre d'octets effectivement lus ;
— **int Write(char *from, int numBytes)**, réalise l'écriture de *numBytes* octets à partir de la position courante depuis le tampon *from*. Le contenu de la zone mémoire écrite est transférée immédiatement sur le disque, de manière synchrone. La position courante est incrémentée. Retourne le nombre d'octets effectivement écrits ;
— **int ReadAt(char *into, int numBytes, int position)** réalise la lecture de *numBytes* octets à partir de *position* et les place dans le tampon *into*. Le contenu du fichier n'est pas conservé en mémoire (pas de gestion de cache disque). La position courante n'est pas modifiée. Retourne le nombre d'octets effectivement lus ;
— **int WriteAt(char *from, int numBytes, int position)**, réalise l'écriture de *numBytes* octets à partir de *position* depuis le tampon *from*. Le contenu de la zone mémoire écrite est transférée immédiatement sur le disque, de manière synchrone. La position courante n'est pas modifiée. Retourne le nombre d'octets effectivement écrits ;
— **int Length()** retourne la taille du fichier en octets ;
— **FileHeader* GetFileHeader()** retourne un pointeur vers la structure d'en-tête associée au fichier ;
— **char* GetName()** retourne le nom du fichier ;
— **void SetName(char*)** modifie le nom du fichier ;
— **bool IsDir()** retourne vrai si le fichier est un répertoire et faux sinon.

1.7.5 Classes *OpenFileTable* et *OpenFileTableEntry* (fichiers *oftable.cc, oftable.h*)

La classe *OpenFileTable* maintient une table des fichiers ouverts afin de gérer la synchronisation de leurs accès. Chaque fois qu'un thread ouvre un fichier, il faut vérifier dans la table qu'un autre thread ne l'a pas déjà ouvert pour gérer les accès concurrents de lecture/écriture. Chaque entrée (classe *OpenFileTableEntry*) dans la table contient le nom du fichier, un *OpenFile* du fichier, le nombre de threads qui ont ouvert ce fichier, un verrou pour la synchronisation, le numéro de secteur du *FileHeader*, ainsi qu'un booléen *ToBeDeleted* qui indique que le fichier doit être détruit quand tous les threads l'auront fermé. *OpenFileTable* possède les méthodes suivantes :

- **OpenFile * Open(char *name)**, crée une entrée dans la table pour un fichier qui n'est pas encore ouvert, ou sinon réutilise l'entrée correspondante quand elle est stockée dans la table. *name* correspond au nom du fichier depuis la racine NACHOS ;
- **void Close(char *name)**, est appelée lorsqu'un thread ferme le fichier, décrémente le nombre de threads qui ont le fichier ouvert. Si ce nombre devient nul, il retire le fichier de la table des fichiers ouverts ;
- **void FileLock(char *name)**, bloque le verrou du fichier afin d'effectuer une écriture exclusive ;
- **void FileRelease(char *name)**, relâche le verrou pour permettre des opérations de lecture ou d'écriture sur le fichier après avoir effectué les opérations sur le disque ;
- **int Remove(char *name)**, efface un fichier du répertoire et positionne *ToBeDeleted* à true (aucune nouvelle ouverture du fichier n'est permise, mais le fichier ne sera effectivement détruit que quand tous les threads l'auront fermé). Renvoie InexistFileError quand le fichier n'a pas pu être trouvé sur disque.

Pour gérer la table, les fonctions précédentes de la classe *OpenFileTable* reposent sur les méthodes suivantes :

- **int next_entry()**, pour obtenir l'index de la prochaine entrée disponible de la table ;
- **int findl(char *name)**, renvoie l'index de l'entrée dans la table correspondant au nom de fichier recherché, ou -1 si le fichier n'est pas présent dans la table.

1.8 Gestion de la mémoire virtuelle (répertoires *vm, machine*)

Comme sur la plupart des processeurs modernes, les adresses des instructions et des données que le processeur MIPS émulé manipule sont *virtuelles* : elles sont relatives à *l'espace d'adressage* courant, qui définit comment traduire ces adresses virtuelles (données, instructions du processeur MIPS) en adresses physiques (mémoire centrale du processeur MIPS, *i.e.* variable *g_machine->mainMemory*). Un thread ne peut s'exécuter que dans le cadre d'un espace d'adressage donné, défini sous NACHOS par le processus auquel le thread se rattache.

Le sous-système noyau de gestion de la *mémoire virtuelle* s'occupe de maintenir les tables nécessaires à ces traductions. Il permet au système de cloisonner les programmes entre eux (éviter qu'un programme puisse physiquement modifier les données d'un autre programme), de faciliter le chargement des programmes (ceux-ci seront toujours chargés aux mêmes adresses virtuelles, indépendamment des adresses physiques associées, et donc des autres programmes déjà chargés), d'autoriser à avoir une taille de mémoire virtuelle supérieure à la mémoire physique (en utilisant l'espace disponible sur les disques), ou de détecter les erreurs d'exécution

des programmes (accès à une adresse virtuelle associée à aucun support physique, disque ou mémoire physique).

1.8.1 Mécanisme de traduction d'adresses

On utilise le mécanisme de traduction d'adresse, qui introduit une séparation entre les adresses virtuelles, que manipule le processeur MIPS émulé lors de l'interprétation des programmes, et les adresses physiques qui sont transmises à la mémoire centrale du processeur MIPS émulé (voir la section 1.3.1).

L'unité de gestion de mémoire : Classe *MMU* (rép. *machine, fichiers* mmu.cc, mmu.h*)*

La classe *mmu* fournit deux méthodes d'accès à la mémoire virtuelle :
ReadMem(int addr, int size, int *value) :
— *addr* : adresse virtuelle de la lecture ;
— *size* : nombre d'octets à lire (1, 2 ou 4) ;
— *value* : valeur lue.
WriteMem(int addr, int size, int *value) :
— *addr* : adresse virtuelle de l'écriture ;
— *size* : nombre d'octets à écrire (1, 2 ou 4) ;
— *value* : valeur à écrire.
Ces deux fonctions font appel à la méthode :
ExceptionType Translate(int virtAddr, int* physAddr, int size, bool writing)
— *virtAdd* : adresse virtuelle à traduire ;
— *physAddr* : adresse physique correspondante ;
— *size* : nombre d'octets de l'accès (1, 2, ou 4) ;
— *writing* : type de l'accès (lecture/écriture).
pour transformer l'adresse virtuelle en adresse physique et ainsi pouvoir accéder aux valeurs voulues dans la mémoire de la machine.

Cette fonction calcule tout d'abord la page virtuelle associée à l'adresse virtuelle ainsi que le déplacement dans cette page. La fonction *Translate* effectue ensuite la traduction à partir de la table de traduction d'adresses (champ *translationTable, voir la section 1.8.1* de la *MMU*) en mettant à jour le cache. L'adresse réelle est alors obtenue grâce à la formule : *AdresseRéelle = NuméroPageRéelle * TaillePage + Déplacement*

En cours de traduction à l'aide de la table de traduction d'adresses, trois cas peuvent se présenter :

— Soit la page était déjà en mémoire, auquel cas la table contient l'adresse physique correspondante.
— Soit la page était sur disque, c'est-à-dire swappée ou dont les données sont contenues dans le fichier exécutable chargé, ou soit la page est *anonyme* (*i.e.* données non initialisées par le compilateur, ou pile d'un thread). Dans ce cas la MMU déclenche une exception de type *défaut de page*, qui est récupérée dans le noyau de Nachos par le gestionnaire de défaut de pages (*g_physical_page_manager.cc, pageFaultManager.h*, voir la section 1.8.2) qui charge la page en mémoire et met à jour la table de traduction avec le numéro de la page réelle qui lui est ainsi associée.

— Soit aucune traduction n'est disponible pour l'adresse virtuelle. Dans ce cas, une exception *AddressErrorException* est levée.

La fonction de traduction renvoie une exception correspondant au résultat de l'opération :

— *BusErrorException* : l'adresse virtuelle à traduire n'était pas alignée correctement ou la traduction a rendu une adresse physique invalide ;
— *AddressErrorException* : aucune traduction d'adresse n'est disponible pour cette adresse virtuelle ;
— *ReadOnlyException* : une écriture à été tentée sur une page protégée en écriture ;
— *NoException* : la traduction d'adresses s'est bien déroulée.

Table des pages virtuelles : Classes *TranslationTable*, *PageTableEntry*, (rép. *machine*, fichiers *translationtable.cc, translationtable.h*)

La table des pages virtuelles (classe *TranslationTable*) de chaque espace d'adressage est initialisée lors de la construction de l'espace d'adressage de chaque processus (constructeur de la classe *Addrspace*) en fonction de la structure du fichier exécutable lancé. Elle est ensuite modifiée au gré des *défauts de pages* et des *vols de pages* (voir la section 1.8.2).

La composition d'une entrée de cette table (classe *PageTableEntry*) est présentée figure 1.3. L'indexation de la table se fait à l'aide du numéro de page virtuelle.

physicalPage	addrDisk	valid	U	M	swap	io	readAllowed	writeAllowed

FIGURE 1.3 – Structure d'une entrée de la table des pages virtuelles

La signification des différents éléments qui composent une entrée est la suivante :

— *physicalPage* est le numéro de la page réelle correspondant à la page virtuelle ;
— *addrDisk* est la position sur le disque lorsque la page en question y est placée ;
— le bit *valid* détermine la validité de l'entrée dans la table (*i.e.* une page physique est associée à la page virtuelle) ;
— les bit *U* et *M* déterminent respectivement si la page a été référencée (respectivement modifiée). Ces bits sont positionnés par la MMU par matériel et remis à 0 par logiciel ;
— le bit *swap* permet de savoir, lorsque la page n'est pas valide (*valid* à 0), si la page est sur la zone de swap (zone modifiable) ou pas. Sa valeur est respectivement à 1 et 0 ;
— le bit *io* indique si la page est déjà concernée par une traduction d'adresse entraînant une entrée/sortie disque. Nécessaire pour synchroniser plusieurs défauts de page simultanés sur une même page ;
— les booléens *readAllowed* et *writeAllowed* permettent de savoir si la page est accessible respectivement en lecture et/ou en écriture. Lorsqu'ils valent tous les deux *faux*, alors aucune traduction pour la page virtuelle concernée n'existe : il n'y a ni page physique, ni page de *swap*, ni morceau de fichier exécutable qui contient les données de la page virtuelle.

Les méthodes d'accès aux données des tables des pages virtuelles font partie de la classe *TranslationTable* contenue dans les fichiers *translationtable.cc, translationtable.h*. Leurs noms commencent par le préfixe *get* ou *set* suivant que l'on veut lire ou écrire des attributs des entrées de tables et sont suivis du nom de la valeur à lire ou modifier. Par exemple, pour lire la valeur du bit *swap* on utilise la méthode **bool getBitSwap(int virtualPage)** et

FIGURE 1.4 – Structure d'une entrée dans la table des pages réelles

pour écrire une nouvelle valeur dans *addrDisk*, on utilise la méthode **void setAddrDisk(int virtualPage, int addrDisk)**.

Les tables des pages sont à un seul niveau. Elles sont structurées sous la forme d'un tableau de descripteurs de pages virtuelles. Les entrées (classe *PageTableEntry*) sont indexées directement en utilisant le numéro de la page virtuelle.

Table des pages réelles : Classe *PhysicalMemManager* (rép. *vm*, fichiers *physMem.cc, physMem.h*)

Le système maintient à jour un tableau – la table des pages réelles – indiquant l'état des pages physiques présentes sur la machine simulée. La figure 1.4 montre la structure d'une entrée dans cette table. La signification des différents éléments qui composent une entrée est la suivante :

- *virtualPage* est le numéro de la page virtuelle correspondant à cette page ;
- *owner* est un pointeur sur l'espace d'adressage du propriétaire de la page physique ;
- *locked* signale que la page est en cours de traitement (remplissage lors d'un défaut de page, recopie lors d'un remplacement de page). Le bit *locked* reste positionné pendant toute la durée d'un défaut de page ou remplacement de page ;
- *free* indique si la page est libre ou non.

C'est grâce à cette table qu'on peut demander une page physique libre et l'associer à un espace d'adressage donné (méthode **int AddPhysicalToVirtualMapping(AddrSpace* owner, int virtualPage)**), quitte à invoquer le voleur de pages quand aucune page n'est disponible. Cette méthode et le voleur de pages reposent respectivement sur **int FindFreePage** et **void RemovePhysicalToVirtualMapping(long numPage)**, dont le rôle est de gérer la liste des pages physiques non allouées (resp. en en retirant une si la liste n'est pas vide, et en en ajoutant une à la liste).

1.8.2 Mécanisme de swap

Voleur de pages : Classe PhysicalMemManager (rép. *vm*, fichiers *physMem.cc, physMem.h*)

Le voleur de pages est implanté dans la méthode **int EvictPage()**. Il est appelé lorsqu'on souhaite allouer une nouvelle page mais qu'aucune page libre n'est présente dans la mémoire réelle (méthode **int AddPhysicalToVirtualMapping(AddrSpace* owner, long virtualPage)**). Le principe consiste alors à réquisitionner la page d'un autre processus, à la placer dans la zone d'échange sur le disque et à donner la page libérée au processus demandeur.

Pour choisir une page, le voleur fait le tour des entrées de la table des pages réelles de manière circulaire (algorithme de l'horloge). La page réquisitionnée est choisie suivant certains critères. Elle ne doit pas avoir été référencée récemment (bit U à 0) et ne doit pas

être verrouillée (bit *locked* à 0). Pour pouvoir faire la différence entre les pages récemment référencées et les autres, l'algorithme remet à zéro le bit de référence (bit U). Ainsi, les pages non utilisées entre deux tours de table du voleur de page pourront être réquisitionnées puisqu'elles apparaîtront comme non référencées. Lorsque le voleur de page réquisitionne une page modifiée, la page est recopiée sur disque avant réquisition.

Le principe de fonctionnement du voleur de pages semble simple mais il est nécessaire de faire attention à certains problèmes dûs au parallélisme :

— Quand une page modifiée est réquisitionnée, la recopie de la page sur disque provoque un blocage du processus demandeur. Pendant la durée du blocage, un autre processus peut déclencher un défaut de page et avoir besoin à son tour de réquisitionner une page.
— Lorsque le voleur de page est amené à se bloquer, l'état des pages qu'il était en train d'examiner peut avoir changé pendant la durée du blocage.
— Toutes les pages réelles sont verrouillées (bit *locked* à 1). Il faut dans ce cas suspendre le processus appelant tant que la situation n'évolue pas.

Routine de traitement de défauts de page : Classe PageFaultManager (rép. *vm*, fichiers *pageFaultManager.cc, pageFaultManager.h*)

La routine de traitement de défauts de page est appelée par la MMU quand une demande d'accès à une page virtuelle provoque un défaut de page (*i.e.* la page n'est pas en mémoire physique). L'unique méthode de cet objet est : **Exception PageFault(int vpn)** où *vpn* est la page virtuelle en cause.

Cette méthode consulte d'abord le bit *swap* de cette page, pour savoir où est le contenu de la page à charger :

— bit *swap*=1 : la page est dans la zone de swap (page de pile ou de données). *addrDisk* contient alors le **numéro de la page** à récupérer dans le swap (numéro du secteur sur disque). Si ce champ a été positionné à -1, alors c'est qu'un voleur de page est en train de l'initialiser, il faut donc attendre que ce champ soit positionné à une autre valeur avant de charger la page ;
— bit *swap*=0 et *addrDisk*=-1 : la page n'existe pas encore en mémoire, ne correspond à aucune zone de disque (fichier ou *swap*), mais est quand même à allouer. Elle correspond au premier accès à une page dite "anonyme". Les pages de ce type correspondent à la pile d'un contexte utilisateur d'un thread, ou à une zone de données de l'exécutable non initialisée par le compilateur (section *.bss* du format binaire ELF, voir la section 1.10.3) ;
— bit *swap*=0 et *addrDisk*≠-1 : la page est à charger dans l'exécutable (page de code ou page de données pas encore chargée). *addrDisk* contient alors la position de la page à récupérer dans le fichier, en nombre d'octets depuis le début du fichier.

Dans le premier cas, on demande au gestionnaire de swap de charger cette page. Le chargement s'effectue vers une page temporaire pour des raisons de synchronisation. Ensuite, on demande au gestionnaire de mémoire réelle de donner une page réelle libre où l'on transfère alors le contenu de la page temporaire.

Dans les deux autres cas, les opérations à effectuer sont identiques sauf pour l'initialisation de la page :

— page "anonyme" (pile du contexte utilisateur, zone de données du programme non initialisées par le compilateur) : la page est remplie de 0 (page vierge) ;

— page de code ou de données : initialisation par chargement depuis l'exécutable.

La gestion de la zone de swap est effectuée par le gestionnaire de swap, de type *SwapManager* (fichiers *swapManager.cc, swapManager.h*). Le gestionnaire de swap exporte les méthodes **void GetPageSwap(int numSector,char* SwapPage)** et **int PutPageSwap(int numSector, char* SwapPage)** pour respectivement lire et écrire dans la zone de swap. Lorsque **PutPageSwap** est appelée avec une valeur de **numSector** de **-1**, le gestionnaire de swap se charge d'allouer une nouvelle page dans la zone de swap et retourne son numéro. Les lectures/écritures depuis un fichier exécutable utiliseront les méthodes **ReadAt/WriteAt**.

Dans tous les cas, la table des pages est mise à jour et la méthode renvoie une exception rendant compte du résultat de l'opération (*NoException* pour indiquer que tout s'est bien déroulé).

1.8.3 Format des fichiers exécutables

Les fichiers sources sont compilés en code MIPS par un compilateur croisé qui génère le format d'exécutable ELF (*Executable and Linking Format*). Ce format de fichier est décrit dans *elf32.h* du répertoire *kernel*. C'est le constructeur de l'espace d'adressage (classe *Addrspace*) qui s'occupe de l'exploiter afin d'initialiser les tables de traduction d'adresses conformément au plan mémoire qui est décrit dans l'exécutable (voir la section 1.10.5).

1.9 Les fichiers utilitaires (répertoire *utility*)

Nachos possède des outils de déboguage, de gestion de listes et de gestion de bitmaps. Cette partie décrit les routines et les accès à ces outils.

1.9.1 Les routines de déboguage (fichiers *utility.h, utility.cc*)

Nachos offre certaines routines permettant d'afficher des messages facilitant le déboguage des fonctions et méthodes système lors de leur implantation. Il est possible de sélectionner le 'type' de message de déboguage que l'on désire systématiquement afficher. Chaque type de message est identifié par un drapeau (flag). Une chaîne de caractères interne au système recense tous les 'flags' des messages à afficher. Cette chaîne est construite à partir de la ligne de commande lors du lancement de Nachos. Voici la liste des flags prédéfinis dans Nachos :

— '+' – tous types de messages ;
— 'a' – espaces d'adressage ;
— 'd' – drivers de périphériques ;
— 'e' – exceptions (notamment appels système) ;
— 'f' – système de gestion de fichiers ;
— 'h' – émulation de la machine (périphériques, notamment disque) ;
— 'i' – gestionnaires d'interruptions ;
— 'm' – émulation de la machine (processeur) ;
— 's' – outils de synchronisation ;
— 't' – threads et processus ;
— 'u' – utilitaires ;
— 'v' – gestion de mémoire virtuelle (VM).

Cette liste peut être complétée par d'autres flags suivant vos besoins. Les routines d'aide au déboguage sont les suivantes :

— **void DebugInit(char* flaglist)** initialise la liste des flags positionnés à partir de la chaîne de caractères *flaglist*. Appelée au démarrage de NACHOS a partir des options passées sur la ligne de commande.
— **bool DebugIsEnabled(char flag)** retourne vrai si *flag* appartient à la chaîne de caractères contenant les 'flags' des messages autorisés à être affichés.
— **void DEBUG(char flag, char *format, ...)** permet d'afficher un message via la console si *flag* correspond à un flag autorisé. La chaîne de caractères passée en paramètre doit être au format standard d'un printf. Les paramètres à afficher sont passés en paramètre à la fonction DEBUG.
— **ASSERT(condition)** permet d'afficher un message d'erreur lorsque la condition passée en paramètre n'est pas vérifiée. Ce message contient la ligne et le fichier depuis lesquels ASSERT a été appelé.

1.9.2 Les listes (fichier *list.h*)

NACHOS propose une structure de listes ainsi que les accès classiques aux listes et à leurs éléments. Ces listes sont définies dans la classe modèle *List*. Les objets issus de *List* ne chaînent pas directement entre eux les éléments de la liste. Chaque élément est associé à un descripteur d'élément défini dans la classe modèle *ListElement*. Un descripteur contient les informations nécessaires au chaînage, un pointeur vers l'élément associé, et sa priorité dans la liste. Dans le cas d'une liste triée, la priorité de l'élément détermine sa place dans la liste, les éléments les plus prioritaires (*i.e.* valeur de priorité la plus élevée) étant placés en tête de liste. Les constructeurs, destructeurs et accès liste disponibles sont décris ci-dessous :

— **List()** est le constructeur de la classe *List* et permet d'initialiser une liste a vide ;
— **~List()** est le destructeur de la classe *List*. Il efface les descripteurs d'éléments mais pas les éléments eux-mêmes. En effet, chaque élément peut être chaîné dans des listes distinctes ; il n'est donc pas souhaitable de les effacer de la mémoire lors de la destruction d'une liste ;
— **void Prepend(void *item)** insère l'élément pointé par *item* en tête de liste ;
— **void Append(void *item)** insère l'élément pointé par *item* en queue de liste ;
— **void *Remove()** efface le descripteur de tête de la liste, et retourne cet élément en résultat ;
— **void Mapcar(VoidFunctionPtr func)** applique la fonction pointée par *func* à chaque élément de la liste ;
— **bool IsEmpty()** retourne vrai si la liste est vide, faux sinon ;
— **void SortedInsert(void *item, Priority sortKey)** insère un élément dans la liste par ordre de priorité croissante ;
— **void *SortedRemove(Priority *keyPtr)** permet d'effacer l'élément de tête d'une liste triée. La valeur de la clé de l'élément effacé est contenu à l'adresse *keyPtr* passée en paramètre ;
— **bool Search(void *item)** retourne vrai si l'élément pointé par *item* est contenu dans la liste ;
— **void RemoveItem(void *item)** efface l'élément pointé par *item* s'il est contenu dans la liste.

Le type de la clé des éléments d'une liste n'est pas prédéfini. La classe *ListElement* est une classe *template*. Il faut donc préciser le type de la clé lors de son instanciation ou définir de nouveaux types de listes de la façon suivante : `typedef List<int> Listint;` pour définir

une liste donc la clé sera un entier.

1.9.3 Les bitmaps : Classe BitMap (fichiers *bitmap.cc, bitmap.h*)

NACHOS contient également des objets bitmap définis dans la classe *BitMap*. Ce sont des tableaux de bits pouvant être indépendamment à 0 ou à 1. Ils sont utilisés dans la gestion du disque. Les accès aux objets *BitMap* sont les suivants :

- **BitMap(int nitems)**, le constructeur, alloue l'espace mémoire nécessaire pour le stockage des bits. Cette allocation se fait par bloc de 32 bits. Tous les bits sont initialisés à 0 ;
- **void Mark(int which)** met le bit de rang *which* à 1 ;
- **void Clear(int which)** met le bit de rang *which* à 0 ;
- **bool Test(int which)** retourne vrai si le bit de rang *which* est à 1, faux sinon ;
- **int Find()** retourne le rang du premier bit à 0 et l'affecte à 1. En d'autres termes, cette fonction trouve et alloue un bit. Si aucun bit n'est à 0, retourne -1 ;
- **int NumClear()** retourne le nombre de bits à 0 ;
- **void Print()** affiche le contenu d'un bitmap ;
- **void FetchFrom(OpenFile *file)** initialise le contenu d'un bitmap à partir d'un fichier NACHOS. *file* est le pointeur vers le fichier ;
- **void WriteBack(OpenFile *file)** écrit le contenu d'un bitmap vers un fichier NACHOS. *file* est le pointeur vers le fichier.

1.10 Mon premier programme Nachos

Cette section présente tout d'abord les appels système disponibles dans NACHOS et les fonctions de la librairie NACHOS. Nous présentons ensuite comment développer un programme utilisateur, le compiler et l'exécuter sous NACHOS.

1.10.1 Appels système disponibles (répertoire *userlib*, fichiers *sys.s*)

L'ensemble des appels système disponibles depuis un programme utilisateur dans NACHOS est déclaré dans le fichier *syscall.h* du répertoire *kernel*. L'ensemble des routines utilisateur provoquant ces appels système est défini dans le fichier *sys.s* du répertoire *userlib*. Sauf mention contraire, en cas d'échec, chacune de ces primitives système renvoie -1 et NoError (0) en cas de succès. En cas d'erreur, un code d'erreur est conservé en interne à NACHOS (voir fichiers *kernel/msgerror.h/.cc*). Un appel système (*PError*) permet d'afficher sur la console le message d'erreur correspondant.

Gestion d'erreurs.

- **void PError(char *mess)**. Affiche sur la console le message d'erreur correspondant à la dernière erreur rencontrée lors de l'exécution d'un appel système (ou la chaîne *"no error"* si le dernier appel système s'est déroulé normalement). Le message affiché sur la console est préfixée par le paramètre *mess* pour personaliser les messages d'erreurs à la convenance de l'utilisateur.

Gestion de processus légers (threads).

- **void Halt()** : arrête NACHOS.
- **void SysTime(Nachos_Time *t)** : renvoie le temps passé dans NACHOS au moment de l'appel.
- **void Exit(int status)** : sort du programme utilisateur (*status* = 0 signifie une sortie normale)
- **ThreadId Exec(char *filename)** : créé un nouveau processus pour l'exécutable NACHOS passé en paramètre, et renvoie l'identificateur du thread créé dans ce nouveau processus (-1 en cas d'erreur).
- **ThreadId newThread(char * debug_name, VoidFunctionPtr func, int arg)** : crée un nouveau thread de nom *debug_name* (pour le déboguage), qui exécutera la procédure *func* avec l'argument *arg*. Le thread est créé dans le même processus que le thread courant. Cette fonction renvoie l'identificateur du thread créé. **Attention :** Cet appel système ne doit pas être appelé directement par les programmes utilisateur, car il ne gère pas les programmes se terminant sans appeler *Exit*. Les programme utilisateur doivent appeler à la place la fonction de bibliothèque utilisateur *threadCreate* (voir ci-dessous), qui gère la fin de processus que ce dernier appelle ou pas *Exit*.
- **int Join(ThreadId id)** : attend la fin du thread *id*.
- **void Yield()** : commute du thread courant vers un autre thread prêt s'il y en a (relâchement volontaire du processeur).
- **void PError(char *mess)** : affiche la dernière erreur rencontrée dans l'exécution d'une primitive système, avec en entête le message personnalisé *mess*.

Accès aux fichiers.

- **int Create(char *name,int size)** : crée le fichier NACHOS *name* de taille *size*. Renvoie 0 en cas de succès.
- **OpenFileId Open(char *name)** : ouvre le fichier *name* et renvoie l'identificateur associé à ce fichier.
- **int Write(char *buffer, int size, OpenFileId id)** : écrit *size* octets depuis *buffer* vers le fichier de descripteur *id*, et renvoie le nombre d'octets effectivement écrits. Si *id* vaut *ConsoleOutput*, alors la chaîne est envoyée sur la console.
- **int Read(char *buffer, int size, OpenFileId id)** : lit *size* octets depuis le fichier *id* vers *buffer* et renvoie le nombre de caractères lus, qui peut être inférieur au nombre demandé dans le cas d'un fichier trop petit. Si *id* vaut *ConsoleInput*, alors la chaîne est saisie depuis la console.
- **int Close(OpenFileId id)** : ferme le fichier identifié par *id*. Renvoie 0 en cas de succès.
- **int Remove(char* name)** : efface le fichier de nom *name*. Renvoie 0 en cas de succès.
- **int Mkdir(char* name)** : crée un nouveau répertoire de nom *name*. Renvoie 0 en cas de succès.
- **int Rmdir(char* name)** : détruit le répertoire de nom *name*. Renvoie 0 en cas de succès.
- **int Mmap(OpenFileId f, int size)** : rend accessible *size* octets du fichier *f* dans l'espace d'adressage du processus appelant, à partir du début du fichier. Le résultat est l'adresse virtuelle à laquelle le fichier pourra être accédé, ou *-1* en cas d'erreur.

f est un descripteur de fichier (le fichier doit être ouvert avant l'appel à *Mmap*). La taille demandée est arrondie à un nombre entier de pages.

Synchronisation.

- **SemId SemCreate(char *debug_name, int count)** : crée un sémaphore de nom *debug_name* (deboguage uniquement) initialisé à la valeur *count*, et retourne son identificateur.
- **int SemDestroy(SemId sema)** : détruit le sémaphore spécifié par l'identificateur *sema*. Renvoie 0 en cas de succès.
- **int V(SemId sema)** : effectue l'opération V sur le sémaphore *sema*. Renvoie 0 si le sémaphore est valide.
- **int P(SemId sema)** : effectue l'opération P sur le sémaphore *sema*. Renvoie 0 si le sémaphore est valide.
- **LockId LockCreate(char *debug_name)** : crée un verrou de nom *debug_name* et retourne son identificateur.
- **int LockDestroy(LockId id)** : détruit le verrou spécifié par l'identificateur *id*. Renvoie 0 en cas de succès.
- **int LockAcquire(LockId id)** : acquisition du verrou d'identificateur *id*. Renvoie 0 si le verrou est valide.
- **int LockRelease(LockId id)** : relâche le verrou d'identificateur *id*. Renvoie 0 si le verrou est valide.
- **CondId CondCreate(char *debug_name)** : crée une nouvelle variable de condition de nom *debug_name*, et retourne son identificateur.
- **int CondDestroy(CondId id)** : détruit la variable de condition d'identificateur *id*. Renvoie 0 en cas de succès.
- **int CondWait(CondId cond)** : met dans l'ensemble d'attente associée de la condition *cond* le thread courant. Renvoie 0 si la condition est valide.
- **int CondSignal(CondId cond)** : le premier thread bloqué sur la condition est réveillé. Renvoie 0 si la condition est valide.
- **int CondBroadcast(CondId cond)** : idem mais sur *tous* les threads de l'ensemble d'attente. Renvoie 0 si la condition est valide.

Accès au coupleur série.

- **int TtySend(char *mess)** : envoie le message *mess*, terminé par le caractère nul, via la ligne série. La valeur de retour indique le nombre de caractères effectivement envoyés.
- **int TtyReceive(char *mess, int length)** : reçoit un message *mess* de longueur maximale *length* par la ligne série. La valeur de retour spécifie le nombre de caractères effectivement reçus.

1.10.2 Fonctions de la bibliothèque Nachos (répertoire *userlib*, fichiers *libnachos.cc, libnachos.h*)

L'ensemble de la bibliothèque NACHOS est décrite dans les fichiers *libnachos.c* et *libnachos.h*. Ces fonctions sont directement accessibles depuis les programmes utilisateur et offrent des facilités supplémentaires pour la programmation par rapport aux seuls appels système.

Les noms de fonctions sont préfixés par *"n_"* (par exemple *"n_printf"*) pour les différencier plus facilement des fonctions standard de la libc. Les fonctions à disposition sont les suivantes :

Opérations sur les processus légers :

— **ThreadId threadCreate(char * debug_name, VoidNoArgFunctionPtr func)** : crée un nouveau thread de nom *debug_name* (pour le déboguage) qui exécutera la procédure *func* (sans lui passer d'argument) dans le même espace d'adressage que celui du thread courant. Cette fonction fait appel à l'appel système *newThread* en gérant correctement la terminaison du thread (appel automatique à *Exit* lorsque *func* se termine). Elle retourne l'identificateur du thread ainsi créé. C'est cette fonction (et pas *newThread*) qui devra être appelée par les programmes utilisateur.

Opérations d'entrées/sortie :

— **void n_printf(char*format,...)** : affiche sur la console la chaîne *format*, qui peut comprendre des références à des variables : %c pour un caractère, %s pour une chaîne, %d ou %i pour un entier, %x pour affichage d'un nombre en hexa, %f pour un flottant. Les variables sont spécifiées dans les paramètres suivants du printf. Enfin, la chaîne formatée peut contenir des caractères spéciaux comme \n (retour à la ligne) ou \t (tabulation).

Opérations sur les chaînes de caractères :

— **int n_strcmp(const char *s1, const char *s2)** : compare les 2 chaînes *s1* et *s2* et renvoie un entier supérieur, égal ou inférieur à zéro suivant que *s1* est supérieur, égal ou inférieur à *s2*. Ces comparaisons sont effectuées suivant l'ordre lexicographique.
— **char* n_strcpy(char *dst, const char *src)** : copie la chaîne *src* (terminée par un \0) dans *dst*, et renvoie *dst* si la copie a marché. **Attention** aux risques de débordement de chaîne *dst* si elle est insuffisamment grande pour contenir *s1* (\0 terminal compris).
— **size_t n_strlen(const char *s)** : renvoie la taille de la chaîne *s* sous le format *size_t*, qui est équivalent au type *int*. Le résultat ne prend pas en compte la présence du \0 terminal.
— **char* n_strcat(char *dst, const char *src)** : concatène la chaîne *src* à la fin de la chaîne *dst*. Renvoie *dst* en cas de succès. **Attention** à ce que *dst* soit suffisamment grande pour contenir le résultat.
— **int n_toupper(int c)** : renvoie la majuscule correspondant au caractère (casté en *int*) passé en paramètre.
— **int n_tolower(int c)** : renvoie la minuscule correspondant au caractère (casté en *int*) passé en paramètre.
— **int n_atoi(const char *str)** : convertit la chaîne *str* (par exemple “1024”) en entier (sur cet exemple 1024).
— **int n_read_int()** : lecture d'un entier sur l'entrée standard (en l'attente d'une fonction plus élaborée...).

Opérations sur les emplacements mémoire :

- **void* n_memcpy(void *b1, const void *b2, size_t n)** : copie les n premiers octets de *b2* vers *b1*. Renvoie *b1* si la fonction a marché.
- **int n_memcmp(const void *b1, const void *b2, size_t n)** : compare les *n* premiers octets de 2 zones mémoire. La valeur retournée est égale à zéro si les deux zones sont identiques, -1 si la zone *b1* est inférieure à la zone *b2*, 1 sinon.
- **void* n_memset(void *b, int c, size_t n)** : affecte la valeur *c* aux *n* premiers octets à partir de l'adresse *b*.

1.10.3 Compilation d'un programme utilisateur

Un programme utilisateur, écrit en langage C, est compilé au moyen d'un compilateur croisé pour obtenir du code MIPS. Le fichier exécutable obtenu est sous format *ELF*.

Il est possible d'effectuer cette compilation automatiquement, au moyen du *Makefile* du répertoire *test*. Pour cela, il suffit d'enregistrer le programme (appelons-le *prog.c*) dans ce répertoire, et de modifier la ligne suivante :

```
PROGRAMS = halt hello shell matmult sort prog
```

Au chargement d'un nouveau processus, NACHOS affiche l'organisation du nouvel espace d'adressage, tel qu'il est spécifié dans l'exécutable chargé grâce au format ELF :

```
****  Loading file /shell :
        - Section .sys : file offset 0x1000, size 0x240, VM addr 0x2000, R/X
        - Section .text : file offset 0x2000, size 0x1e30, VM addr 0x4000, R/X
        - Section .rodata : file offset 0x4000, size 0x54, VM addr 0x8000, R
        - Program start address : 0x4000
```

Un exécutable au format *ELF* est découpé en *sections*, correspondant aux différentes entités nécessaires à l'exécution du programme. Chaque section possède un nom : *.sys* pour les routines d'appels système en assembleur, *.text* pour le code, *.bss* pour les données non initialisées par le compilateur, *.data* et *.rodata* pour les données initialisées par le compilateur, etc. Toutes ou partie de ces sections peuvent être présentes dans un exécutable *ELF*. Un affichage du type précédent indique l'emplacement de chaque section dans le fichier exécutable (*file offset*), leur taille (*size*), là où l'exécutable demande qu'elles soient chargées dans l'espace d'adressage du processus (*VM addr*), et les droits associés (*R* pour lecture, *W* pour écriture, *X* pour exécution [4]). Il indique également l'adresse de la première fonction à exécuter (*Program start address*).

1.10.4 Compilation de Nachos

Pour compiler NACHOS, il suffit d'utiliser le fichier *Makefile* qui se trouve à la racine des sources. Si vous changez l'ensemble des fichiers à compiler, ou seulement leur nom ou leur emplacement, le fichier *Makefile* du répertoire où a lieu le changement (*kernel*, *machine*, ...) doit être modifié en conséquence. La compilation **avec *gmake* obligatoirement** génère un fichier exécutable nommé *nachos* à la racine de l'arborescence des sources NACHOS. À la modification d'un fichier d'en-tête ou d'un fichier source, seuls les fichiers qui en dépendent sont recompilés automatiquement au prochain *gmake*. Pour faire du ménage et effacer tous

4. Sous NACHOS, le droit en exécution n'est pas pris en compte, il est affiché ici à titre purement indicatif.

les résultats des compilations passées, faire *gmake clean* (ou simplement make si gmake est sélectionné par défaut).

1.10.5 Exécution d'un programme Nachos (répertoire *kernel*, fichiers *main.cc, system.cc*)

L'exécutable du système d'exploitation, qui se nomme *nachos*, a la caractéristique de proposer à l'utilisateur un ensemble de paramètres qui permettent d'initialiser le système. Ces directives d'exécution se partagent en deux classes : certaines peuvent être fournies sur la ligne de commande, en même temps que l'exécutable NACHOS, d'autres sont définies dans un fichier de configuration.

Les directives fournies sur la ligne de commande sont les suivantes :

-d : permet d'afficher à l'écran les messages de déboguage, dont le type est spécifié. Ainsi `nachos -d f` affiche ceux concernant le système de fichiers. Si cette directive est suivie de *+*, ou bien si elle n'est pas suivie par un type de message, tous les messages de déboguage sont affichés. L'ensemble des types de messages sont définis dans la section 1.9.1, page 31.

-s : déclenche une exécution "pas à pas" d'un programme utilisateur. Entre chaque instruction est alors affiché le contenu de tous les registres de la machine, ainsi que les interruptions en attente. Appuyer sur entrée entraîne l'exécution d'une nouvelle instruction.

-x : permet de lancer dès l'initialisation du système un programme utilisateur, qui est dans le système de fichiers NACHOS. Il faut pour cela donner le nom relatif de ce programme par rapport au répertoire racine du disque NACHOS.

-z : affiche un message de copyright de NACHOS.

-f <nomfich> : utilise le fichier de configuration <nomfich> à la place du fichier de configuration par défaut *nachos.cfg*.

Il est évidemment possible de combiner plusieurs directives lors du lancement.

1.10.6 Configuration de Nachos (répertoire *utility*, fichiers *config.cc, config.h*)

NACHOS possède aussi un fichier de configuration, dont l'intérêt est de permettre de modifier des paramètres du système au lancement de l´exécutable, sans avoir besoin de recompiler tout le système. Ce fichier de configuration est le fichier *nachos.cfg* par défaut (il doit se trouver à l'endroit à partir duquel on lance NACHOS). Un fichier de configuration par défaut vous est fourni à la racine des sources. Le nom de ce fichier peut être modifié en utilisant l'option -f. Il y est ainsi possible de déterminer la structure du système et d'effectuer des opérations, avant le lancement du système, sur le système de fichiers. Pour spécifier un paramètre, la syntaxe est la suivante :

```
<nom de paramètre> = <valeur du paramètre>
```

Structure interne de Nachos

Les différents éléments qui peuvent être spécifiés par l'utilisateur sont :

```
PrintStat = [ 0 | 1 ]
```

Permet d'affiche des statistiques d'utilisation du processeur, de la mémoire, et du disque à la terminaison de NACHOS.

`NumPhysPages = <entier>`

Le nombre de pages physiques dans la mémoire de la machine émulée.

`MaxVirtPages = <entier>`

La nombre maximal de pages virtuelles dans un espace d'adressage (voir la section 1.8.1).

`UserStackSize = <entier>`

La taille (en octets) des piles utilisateur.

`ProcessorFrequency = <entier>`

Précise la fréquence du processeur MIPS émulé (en MHz). Sert dans les statistiques à regarder (approximativement) l'impact de la fréquence du processeur sur les performances des applications, à performances de périphériques équivalentes.

`SectorSize = <entier>`

Précise la taille d'un secteur du disque (en octets). Doit être une puissance de deux.

`PageSize = <entier>`

Précise la taille d'une page en mémoire physique (en octets). Doit être une puissance de deux. Dans la mise en œuvre actuelle du système de pagination, la taille d'une page doit être identique à la taille d'un secteur.

`UseACIA = [ None | BusyWaiting | Interrupt]`

Indique si le coupleur série et son pilote sont inexistants (valeur *None*), présents avec une utilisation du coupleur en mode attente active (valeur *BusyWaiting*) ou en mode interruption (valeur *Interrupt*). Mettez ce paramètre à *None* tant que le coupleur série n'est pas utilisé, la simulation de ligne série étant très consommatrice en temps processeur.

`TargetMachineName = <nom>`

Le nom de la machine cible, lors de la communication via le coupleur série.

`NumPortLoc = <entier>`

Le numéro de port local (>1024) d'enregistrement du service réseau pour la simulation de la communication par le port série. Ce paramètre est ignoré lorsque *UseACIA* est à *None*.

`NumPortDist = <entier>`

Le numéro de port distant (>1024) du service réseau pour la simulation de la communication par le port série. Ce paramètre est ignoré lorsque *UseACIA* est à *None*. Les deux numéros de ports doivent être croisés dans les fichiers de configuration des deux machines qui communiquent.

Gestion du système de fichiers

Les opérations sur le système de fichiers sont effectuées avant le lancement du système, et de ce fait ces instructions ne sont pas prises en compte dans les statistiques.

`ProgramToRun = <nom>`

Le nom du programme utilisateur à lancer. Il faut indiquer le chemin dans le disque NACHOS. Équivalent à l'option `-x` sur la ligne de commande.

`FormatDisk = [ 0 | 1 ]`

Permet d'effectuer le formattage du disque au démarrage.

`ListDir = [ 0 | 1 ]`

Affiche toute l'arborescence du système de fichiers, avec la position des fichiers.

`PrintFileSyst = [ 0 | 1 ]`

Affiche tout le contenu de tous les fichiers du système de fichiers, octet par octet.

`FileToPrint = <chemin Nachos>`

Affiche le contenu d'un fichier du disque NACHOS.

`FileToCopy = <chemin Unix> <chemin Nachos>`

Copie un fichier Unix dans le système de fichier NACHOS. Les deux noms à donner sont les noms relatifs, respectivement par rapport au répertoire courant Unix, et par rapport à la racine du système NACHOS. Plusieurs directives de ce type peuvent être fournies, ce qui provoque la copie de plusieurs fichiers.

`FileToRemove = <chemin Nachos>`

Retire le fichier spécifié du système. Plusieurs directives de ce type peuvent être fournies, ce qui provoque la suppression de plusieurs fichiers.

`NumDirEntries = <entier>`

Correspond au nombre d'entrées par répertoire.

`DirToMake = <chemin Nachos>`

Crée un répertoire dans l'arborescence de NACHOS. Plusieurs directives de ce type peuvent être fournies, ce qui provoque la création de plusieurs répertoires.

`DirToRemove = <chemin Nachos>`

Efface un répertoire de NACHOS. Plusieurs directives de ce type peuvent être fournies, ce qui provoque la suppression de plusieurs répertoires.

Tous ces paramètres ont une valeur par défaut (voir le constructeur de la classe *Config*). Dans le noyau NACHOS, les champs de l'objet glocal *g_cfg* instanciant la classe *Config* permettent de récupérer la valeur associée à ces paramètres. Par exemple :

```
g_cfg->PageSize;
```

permet de récupérer la taille d'une page.

Chapitre 2

Travaux pratiques du module SGP (18h)

Les trois premiers TP de SGP (consistent à utiliser les appels système de gestion du parallélisme et de la synchronisation entre processus sous Linux. Tous les appels système ne sont pas intégralement documentés dans les sujets de TP. Pour la documentation intégrale, *man nom_appel* ou *man -s 2 nom_appel*.

Merci de respecter les conventions de nommage des fichiers indiquées dans les sujets.

2.1 TP1 (Linux) - Utilisation de fork, exec, wait, pipes

2.1.1 Utilisation des primitives fork, exec et wait

La primitive *fork* est la primitive qui permet sous Unix de créer des processus. Le nouveau processus créé (appelé processus fils), hérite d'un certain nombre d'attributs du processus père :

- le même code ;
- une copie de sa zone de données ;
- les variables d'environnement ;
- les descripteurs de fichiers ouverts ;
- son état d'exécution courant (valeur des registres, dont compteur ordinal).

Le seul moyen de distinguer le processus fils du processus père est que la valeur de retour de la fonction *fork* est 0 dans le processus fils créé et le numéro du processus fils dans le processus père.

La primitive exec permet de recouvrir le code initial d'un processus (généré par *fork*) par le code d'un programme dont le nom est passé en paramètre. Cette primitive possède plusieurs formes d'appel ; nous utiliserons la fonction execl.

La primitive *wait* provoque la suspension du processus appelant jusqu'à ce qu'un de ses processus fils se termine.

Exercice. Ecrire un programme *partie1.c* qui crée un processus fils (*fork*). Ce dernier recouvre son code initial par un code généré à partir de *partie2.c* qui affiche un message (par exemple *"Je suis le fils"*). Le processus père attend alors la terminaison de son fils et affiche le code de retour renvoyé par ce dernier.

2.1.2 Utilisation des primitives de communication entre processus

Le tube

Le tube (en anglais *pipe*) est un mécanisme caractéristique de communication du système Unix. Il permet de mémoriser des informations et se comporte comme une file FIFO. Un tube est de taille limitée ; le système se charge de suspendre tout processus qui essaie d'écrire dans un tube plein. Un tube correspond à deux descripteurs de fichiers : le premier permet de lire dans le tube ; le second permet d'y écrire. Un tube est créé par appel à la primitive *pipe*. La lecture et l'écriture dans un tube se font de manière classique en utilisant les primitives de lecture-écriture fichier (*read*, *write*).

Le système assure aux utilisateurs un certain nombre de sécurités :

- un processus qui tente d'écrire dans un tube alors que plus aucun processus n'est en mesure d'y lire, est tué ;
- la primitive read rend 0 en résultat lorsque le tube est vide et que tous les processus susceptibles d'y écrire ont fermé le tube en écriture.

Exercice. Utiliser les tubes pour écrire un système producteur-consommateur. Le producteur lit au clavier une suite de chaînes de caractères frappée au clavier. Il s'arrête à la saisie d'une chaîne vide. Le consommateur récupère ces chaînes par le biais du tube et les affiche à l'écran. Les deux processus seront dans le même fichier source *prodcons.c*. Le producteur créera le consommateur (*fork*). Pour la lecture des chaînes au clavier, on utilisera la fonction *fgets*.

Le tube nommé

Contrairement aux tubes classiques, un tube nommé possède un identificateur dans le système de gestion de fichiers, et peut donc être connu par l'ensemble des processus. Pour créer un tube nommé, on appelle la primitive *mknod* en lui passant dans le paramètre mode la valeur S_IFIFO | mode d'accès (par exemple S_IFIFO | 0666). L'utilisation d'un tube nommé est alors identique à celle d'un fichier classique (*open, read, write, close*).

Il est intéressant de savoir que :

- Si un processus essaie d'ouvrir un tube nommé en lecture, le processus est suspendu si aucun processus ne l'a ouvert en écriture ; la suspension dure jusqu'à ce qu'un processus l'ouvre en écriture.
- De façon similaire, un processus tentant d'écrire sera suspendu tant qu'il n'existe pas de lecteur.

Exercice. Reprendre l'exercice précédent en utilisant maintenant des tubes nommés. Les processus producteur et consommateur sont indépendants (fichiers source *prod.c* et *cons.c*). Le producteur crée le tube nommé et le consommateur le détruit.

2.1.3 Interface des appels systèmes

— #include <sys/types.h>
#include <unistd.h>
pid_t fork(void) ;

Cette fonction crée un nouveau processus. Le nouveau processus (processus fils) est une copie exacte du processus appelant (processus père). Le processus fils hérite d'un ensemble de caractéristiques du processus père (variables d'environnement, descripteurs de fichier ouverts, dispositions vis-à-vis des signaux, masque de création des fichiers, ...). La fonction retourne -1 en cas d'erreur, 0 chez le processus fils et l'identificateur du processus fils (pid) dans le processus père.

— #include <unistd.h>
int execl(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/) ;

La fonction *execl* recouvre l'image d'un processus avec celle d'un nouveau processus, qui est obtenue à partir d'un fichier exécutable. *path* est le chemin d'accès (absolu ou relatif) à cet exécutable, nom de l'exécutable inclus. *arg0, .. argn* est une liste de chaines de caractères terminée par le pointeur NULL. Ce sont les les arguments passés au programme exécutable (*arg0* est le nom de l'exécutable, récupérable par *argv[0]*). La fonction *execl* retourne -1 en cas d'erreur.

— #include <stdlib.h>
void exit(int status) ;

Termine le processus appelant. Le paramètre *status* est la valeur de retour du processus (par convention, 0 dénote une terminaison sans erreur). La valeur de retour peut être récupérée par le processus père via un appel à *wait*.

— #include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc) ;

La fonction *wait* suspend l'exécution du processus appelant jusqu'à ce qu'un de ses fils termine, par un appel à *exit* ou la réception d'un signal. La fonction retourne l'identificateur du processus fils qui se termine, ou -1 en cas d'erreur. Si *stat_loc* n'est pas NULL, l'entier mis à jour par la fonction précise la cause de la terminaison du processus fils et le cas échéant le code de retour passé à *exit* (voir l'aide en ligne pour plus de détails, macros *WIFEXITED* et *WEXITSTATUS*).

— #include <unistd.h>
int pipe(int fd[2]) ; /* fd[0] et fd[1] sont les descripteurs de fichiers permettant l'accès au tube */

La fonction *pipe* crée un tube. Cette fonction remplit les descripteurs de fichiers *fd[0]* et *fd[1]*. *fd[0]* est ouvert par la fonction pipe pour lire dans le tube, alors que *fd[1]* est ouvert pour y écrire. *pipe* retourne 0 en cas de succès, et -1 en cas d'erreur.

— #include <fcntl.*h* >
#include <unistd.h>
#include <sys/stat.h>
int mknod(const char *path, mode_t mode, dev_t dev) ;

La fonction *mknod* permet (entre autres) de créer des tubes nommés, le nom étant le paramètre *path*. mode contient les droits d'accès sur le tube (par exemple S_IFIFO | 0666 pour un tube accessible en lecture et écriture par tous). Le paramètre *dev* n'est pas utilisé pour la création des tubes nommés (*mknod* peut être appelé pour créer d'autres objets que des tubes). *mknod* retourne 0 en cas de succès, et -1 en cas d'erreur.

— #include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char * nom_fichier, int oflag) ;

Ouvre le fichier de nom *nom_fichier* avec le mode d'ouverture spécifié par *oflag* (O_RDONLY : lecture, O_WRONLY : écriture). Retourne le descripteur du fichier ouvert (-1 en cas d'erreur)

— #include <unistd.h>
int close(int descr) ;

Ferme le fichier qui correspond au descripteur *descr*.

— #include <unistd.h>
int read (int descr, void *buf, size_t nombre) ;

Lit *nombre* octets dans le fichier identifié par *descr* et place le résultat en mémoire à l'adresse *buf*. Retourne le nombre d'octets lus, 0 en cas de fin de fichier, ou -1 en cas d'erreur.

— #include <unistd.h>
int write(int descr, const void * buf, size_t nombre) ;

Ecrit *nombre* octets dans le fichier identifié par *descr* à partir de la zone mémoire d'adresse *buf*. Retourne le nombre d'octets écrits, ou -1 en cas d'erreur.

— #include <stdio.h>
char *fgets(char *buf, int size, FILE *stream) ;

Lit une chaîne de caractères dans le fichier ouvert identifié par *stream* (mettre *stdin* pour l'entrée standard) et la stocke dans le tampon *buf*. L'adresse de buf est retournée par la fonction, sauf en cas d'erreur, où la valeur *NULL* est retournée.

— #include <ctype.h>
int toupper(int c) ;

Si le paramètre de *toupper* est une lettre minuscule, la fonction retourne la majuscule correspondante.

— #include <stdio.h>
void perror(const char *s) ;

Tous les appels système qui retournent la valeur -1 positionnent la variable globale entière *errno* qui indique la source de l'erreur. La fonction de librairie C *perror* affiche à l'écran une chaîne de caractères qui correspond à l'erreur, après avoir imprimé la chaîne passée en paramètre. Par exemple, l'appel *perror("Erreur : ") ;* après une tentative d'ouverture d'un fichier inexistant affichera à l'écran le message : *Erreur : No such file or directory.*

— #include <unistd.h>
int unlink(const char *path) ;

La fonction *unlink* détruit un lien sur le fichier de nom path. Si le nombre de liens sur le fichier devient nul et qu'aucun processus n'utilise le fichier, ce dernier est détruit.

2.2 TP2 (Linux) - Utilisation des signaux et des fonctions setjmp, longjmp et ptrace

2.2.1 Utilisation des signaux

Les signaux sont des mécanismes de communication entre processus du système Unix. C'est par ce mécanisme que le système communique avec les processus des utilisateurs en cas d'erreur (violation mémoire, division par zéro,...), ou à la demande de l'utilisateur lui-même. Les signaux sont identifiés par des nombres entiers. La liste des principaux signaux est donnée ci-dessous, en indiquant leur nom, leur valeur, l'action par défaut qui est déroulée lors de l'envoi du signal, et leur rôle. La liste complète des signaux figure dans le fichier d'inclusion *<signal.h>* (voir aussi *man -s 7 signal*).

Nom	Numéro	Action	Rôle
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe : write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped

Un comportement par défaut est déroulé lors de la réception d'un signal par un processus. Ainsi, la réception de la plupart des signaux par un processus provoque sa terminaison. La réception de certains signaux provoque la génération de fichiers *core*.

Tout processus a la possibilité d'émettre un signal à un autre processus, et ceci en utilisant l'appel système *kill*.

Le programmeur a la possibilité de choisir la fonction exécutée lors de la réception d'un signal, en appelant la fonction système *signal* ou la fonction système *sigaction*. Toutefois, les signaux *SIGKILL* et *SIGSTOP* ne peuvent pas être déroutés. Si le traitement du signal ne provoque pas la terminaison du processus, son exécution est reprise à l'emplacement où le signal l'a interrompue (dans le cas d'une division par zéro, l'instruction qui a déclenché l'émission du signal est relancée).

Par ailleurs, tout processus a un masque de signaux qui définit l'ensemble des signaux dont la livraison au processus est momentanément suspendue (voir fonction système *sigprocmask*).

Exercice. Utiliser la primitive *signal* pour gérer les tentatives de divisions par zéro (un signal *SIGFPE* est généré lors d'une tentative de division par une valeur entière nulle). Le fichier de test sera nommé *div0.c*

Exercice. Ecrire un programme formé d'une boucle de calcul qui affiche toutes les secondes son propre état d'avancement. Pour cela, on utilisera la primitive *alarm*, qui envoie le signal *SIGALRM* au processus qui appelle la primitive, au terme d'un délai passé en paramètre de la primitive ; la primitive *alarm* n'est pas cyclique : un seul signal *SIGALRM* est envoyé. Le fichier de test sera nommé *tictac.c*.

2.2.2 Utilisation des fonctions setjmp et longjmp

Les fonctions de la bibliothèque C standard *setjmp* et *longjmp* permettent respectivement de sauvegarder l'état d'exécution d'un programme et demander la reprise du programme au point sauvegardé.

Exercice. Modifier le programme écrit dans l'exercice précédent pour qu'à chaque fois que l'on frappe le caractère d'interruption du programme (contrôle C), le calcul reprenne au début. On prendra par exemple un produit de matrice. Si la routine de traitement de signal comporte un *longjmp*, la routine ne termine pas et le signal est bloqué indéfiniment. On devra dans ce cas utiliser la fonction système *sigprocmask* pour débloquer le signal. Le fichier de test sera nommé *tictacforever.c*.

2.2.3 Utilisation de la fonction ptrace

La fonction *ptrace* est une fonction Unix qui permet à un processus père de contrôler l'exécution d'un de ses processus fils. Cette fonction est utilisée lors de la réalisation de metteurs au points pour la pose de points d'arrêt ou la visualisation de variables. Le processus fils se comporte normalement, jusqu'à ce qu'il rencontre un signal, ce qui provoque l'arrêt du processus fils. Le processus père en est alors averti par l'intermédiaire de la fonction *wait* ; le processus père peut alors examiner ou modifier l'image mémoire du processus fils, ainsi que forcer sa reprise ou sa terminaison.

Exercice. Ecrire un processus fils composé d'une boucle avec un indice de boucle. L'exécution de ce processus fils sera contrôlée par un processus père qui affichera périodiquement l'état d'avancement du fils (affichage de la valeur de l'indice de boucle). Pour éviter les problèmes d'obtention de l'adresse de l'indice de boucle, les deux processus seront obtenus par clonage (appel à *fork()*. Le fichier de test sera nommé *monitor.c*.

2.2.4 Interface des appels système

— #include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig) ;

Envoie le signal *sig* au processus d'identificateur *pid*. Retourne la valeur 0 si l'envoi est effectué,-1 en cas d'erreur.

— #include <signal.h>
void (*signal(int sig, void(*func)(int))) (int) ;

Initialise la fonction de traitement du signal *sig* à la fonction *func*. Les valeurs de func *SIG_DFL* et *SIG_IGN* permettent respectivement de reprendre la fonction de traitement par défaut et d'ignorer le signal. La fonction *signal* retourne -1 en cas d'erreur, ou l'adresse de la fonction de traitement du signal précédente. Lors d'un signal :
- la fonction de traitement du signal est positionnée à *SIG_DFL* (traitement par défaut) avant d'exécuter le traitement du signal (la connexion du traitement est effectuée pour une seule occurrence du signal) ;
- le signal faisant l'objet du traitement est bloqué pendant l'exécution de la routine de traitement du signal.

Lorsque *sigset* est utilisée, lors d'un signal :
- le signal est ajouté au masque des signaux du processus concerné par le signal ;
- le traitement du signal est appelé ;
- le signal est oté du masque des signaux du processus. Attention : si le traitement du signal ne se termine pas (ex : déroutement par *longjmp*), le masque de signal reste modifié. On peut le restaurer en utilisant la fonction système *sigprocmask*.

— #include <unistd.h>
unsigned alarm(unsigned sec) ;

Envoi du signal *SIGALRM* au processus appelant au terme d'un délai de *sec* secondes.

— #include <setjmp.h>
int setjmp(jmp_buf env) ; /* Structure de données contenant l'état du processus */

Sauvegarde dans *env* l'état d'exécution du processus appelant. Retourne la valeur zéro, sauf si l'on reprend l'exécution après un appel à *longjmp*. Dans ce cas, *setjmp* retourne la valeur de retour passée en paramètre à la fonction *longjmp*.

— #include <setjmp.h>
void longjmp(jmp_buf env, int val) ;

Restaure l'état du processus appelant avec la valeur *env* sauvegardée au préalable par l'appel à la primitive *setjmp* ; *setjmp* retourne alors la valeur *val*.

— #include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data) ;

L'argument *request* détermine l'action à effectuer par *ptrace*, et peut prendre les valeurs suivantes (voir l'aide en ligne pour quelques requêtes supplémentaires) :
- PTRACE_TRACEME : cette requête doit être effectuée par le processus fils pour être contrôlé par son père. L'effet de cette requête est de stopper le processus fils lors de l'occurrence de chaque signal. Les arguments *pid*, *addr* et *data* sont ignorés par cette requête, et sa valeur de retour n'est pas initialisée.
- PTRACE_PEEKTEXT, PTRACE_PEEKDATA : cette requête est exécutée par le processus père pour lire un mot respectivement dans l'espace de code et de données du processus fils, identifié par *pid*. *addr* désigne l'adresse du mot à lire ; *data* est ignoré. Un code de retour de -1 est retourné en cas d'erreur. La valeur lue est retournée en retour de la fonction.
- PTRACE_POKETEXT, PTRACE_POKEDATA : cette requête est symétrique à la précédente. *data* contient la valeur à écrire.
- PTRACE_CONT : cette requête permet au processus père de reprendre l'exécution du

processus fils. Si *data* vaut 0, la liste les signaux du processus fils est nettoyée. Si *data* correspond à un numéro de signal valide, ce signal est envoyé au processus fils lors de sa reprise. *addr* doit être initialisé à 1. La requête retourne -1 en cas d'erreur.

— PTRACE_SINGLESTEP : cette requête relance l'exécution du processus fils pour une seule instruction machine.

2.3 TP3 (Linux) - Utilisation de threads Posix (pthread)

2.3.1 Présentation de la librairie de processus légers

Notion de processus léger

Un processus léger (en Anglais *lightweight process*, ou *thread*) est un fil de contrôle plus léger qu'un processus Unix standard, dans le sens où il n'est pas lié à un espace d'adressage. Plusieurs processus légers s'exécutent en concurrence au sein du même espace d'adressage, et partagent les données accessibles à partir de ce dernier. Le contexte d'un processus léger est de petite taille : il est limité aux registres de la machine. Ainsi, la création et le changement de contexte entre processus légers est moins coûteux que leurs analogues pour des processus Unix.

La notion de processus léger est particulièrement bien adaptée aux applications devant réagir à des événements asynchrones (par exemple pour la construction de serveurs) et aux architectures multiprocesseur à mémoire partagée.

Librairie de threads Posix (pthreads)

La librairie de processus légers Posix (*pthreads*) est disponible sur tout système Unix. Elle permet l'exécution de fils de contrôle parallèles (*threads*) au sein d'un même processus Unix.

La librairie *pthreads* fournit, entre autres, des primitives pour :

- créer et détruire des processus légers ;
- contrôler l'ordonnancement des processus légers ;
- synchroniser les processus légers entre eux. Les outils de synchronisation disponibles sont :
  - les sémaphores d'exclusion mutuelle ;
  - les variables de condition ;
  - les sémaphores lecteur/rédacteur.

Les principales primitives offertes par la librairie de processus légers sont données ci-dessous. On ne donne pas l'interface complète des primitives, très nombreuses (pour plus de précisions, consulter l'aide en ligne grâce à la commande *man pthreads*). Toutes les primitives retournent 0 en cas de succès et un code d'erreur dans le cas contraire. Pour les utiliser inclure *pthread.h* lors de la compilation, et utiliser la bibliothèque *pthread* lors de l'édition de liens (*-lpthread*).

2.3.2 Travail à réaliser

Écrire un programme parallèle de produit de matrices, dont le programme source sera nommé *matmult.c*. Les matrices source M1 et M2 et la matrice résultat M seront des matrices 64x64 de valeurs flottantes. On rappelle la formule du produit de deux matrices M1 et M2 (les indices d'accès aux matrices commencent à 0) :

$$M[i,j] = \sum_{0}^{63} M1[i,k] * M2[k,j]$$

On réalise le produit de matrice par 8 processus légers esclaves, et un processus léger maître, chargé de la création des processus esclaves. La ligne i sera calculée par le processus esclave *i modulo 8*).

Nom	Rôle
pthread_create	Creates a new thread of execution
pthread_detach	Marks a thread for deletion
pthread_exit	Terminates the calling thread
pthread_join	Causes the calling thread to wait for the termination
pthread_self	Returns the thread ID of the calling thread
pthread_attr_setschedparam	Set the scheduling parameter attribute in a thread attributes object
pthread_attr_setschedpolicy	Set the scheduling policy attribute in a thread attributes object
pthread_mutex_init	Initialize a mutex with specified attributes
pthread_mutex_lock	Lock a mutex and block until it becomes available
pthread_mutex_unlock	Unlock a mutex
pthread_cond_broadcast	Unblock all threads currently blocked on the specified condition variable
pthread_cond_init	Initialize a condition variable with specified attributes
pthread_cond_signal	Unblock at least one of the threads blocked on the specified condition variable
pthread_cond_wait	Wait for a condition and lock the specified mutex
pthread_rwlock_init	Initialize a read/write lock object
pthread_rwlock_rdlock	Lock a read/write lock for reading, blocking until the lock can be acquired
pthread_rwlock_unlock	Unlock a read/write lock
pthread_rwlock_wrlock	Lock a read/write lock for writing, blocking until the lock can be acquired

2.4 TP4 (Nachos) - prise en main de Nachos (travail à la maison)

2.4.1 Consignes lors de la réalisation des TPs au dessus de Nachos

— Pour vous y retrouver, et permettre aux enseignants de repérer votre code au milieu du code existant, encadrez votre code par les directives *ifdef/endif* suivantes, en n'oubliant pas d'encadrer *aussi* les zones de commentaires :

```
#ifndef ETUDIANTS_TP
  <le vieux code que vous allez changer pendant le TP>
#endif
#ifdef ETUDIANTS_TP
  <le beau code que vous voulez ajouter a la place>
#endif
```

Cette consigne est **IMPÉRATIVE**. Elle servira aux enseignants à récupérer automatiquement votre code parmi le code de NACHOS, et à vous même pour repérer le code modifié en cas de problème.

— Vous ne devez pas modifier le répertoire *machine*, ni les parties gérant le contexte noyau des threads.

— Pour mettre au point votre système, vous disposez d'un système d'affichage de traces. N'hésitez pas non plus à utiliser un débogueur, comme par exemple *gdb* (voir l'annexe A).

Les durées d'exécution des TPs sont données à titre indicatif. Il est prévu que vous consacriez 1 à 2h de travail personnel quand 2h sont prévues en TP.

2.4.2 Questionnaire d'exploration de Nachos

Sont consignées ici quelques questions pour vous aider à rentrer dans le code de NACHOS. Les réponses aux questions s'obtiennent en lisant (et relisant) ce document et le code source de NACHOS, et les pages *html* générées automatiquement à partir du code source de NACHOS. Le moyen le plus simple pour explorer le source est d'utiliser la commande *grep* qui recherche une chaîne dans un fichier ou ensemble de fichiers (faire *man grep*).

Mécanisme d'appel système

Lister, en expliquant, les fichiers et fonctions/méthodes impliqués dans l'exécution d'un programme utilisateur qui nécessite un appel au système, en mode utilisateur et système. On pourra par exemple regarder le fichier *test/hello.c*, qui se contente d'afficher un message sur la console.

Lister les principales fonctions et méthodes appelées, en mode utilisateur et noyau, en indiquant leur rôle.

Gestion de threads et de processus

1. Indiquer ce qui doit être sauvegardé lors d'un changement de contexte.
2. Quelle variable est utilisée pour mémoriser la liste des threads prêts à s'exécuter ? Est-ce que le thread élu appartient à cette liste ? Comment accéder à ce thread ?
3. A quoi sert la variable *g_alive* ? Quelle est la différence avec le champ *readyList* de l'objet g_scheduler ?
4. Comment se comportent les routines de gestion de listes vis à vis de l'allocation de mémoire ? Est-ce qu'elles se chargent d'allouer/désallouer les objets chaînés dans la liste ? Pourquoi ?
5. A quel endroit est placé un objet thread quand il est bloqué sur un sémaphore ?
6. Comment faire en sorte qu'on ne soit pas interrompu lors de la manipulation des structures de données du noyau ?
7. A quoi sert la méthode *SwitchTo* de l'objet g_scheduler ? Quel est le rôle des variables *thread_context* et *simulator_context* de l'objet thread ? Que font les méthodes *SaveSimulatorState* et *RestoreSimulatorState* ? Que devront (à terme) faire les méthodes *SaveProcessorState* et *RestoreProcessorState* de l'objet thread ?
8. Expliquer l'utilité du champ *typeId* de tous les objets manipulés par le noyau (sémaphores, tâches, threads, etc.).

Environnement de développement

1. Lister les outils offerts par NACHOS pour la mise au point des programmes utilisateur. Comment par exemple visualiser toutes les opérations effectuées par la machine MIPS émulée ?
2. Peut-on utiliser l'utilitaire *gdb* pour mettre au point le code de NACHOS ? Le lancer et visualiser le contenu de différentes variables du noyau.
3. Peut-on utiliser *gdb* pour mettre au point les programmes utilisateur ? Expliquer.

2.4.3 Quoi rendre aux enseignants

Pour le questionnaire Nachos, vous rendrez aux enseignants une réponse aux questionnaire donné ci-dessus. Pour les TP proprement dits, vous devez envoyer à l'enseignant le code source (fichiers modifiés seulement, programmes de test inclus), et un court document expliquant ce que vous avez réalisé, comment ça a été testé, le résultat des tests (avec copie d'écran à l'appui), et le cas échéant ce qui n'a pas pu être terminé et le diagnostic associé.

2.5 TP 5 (Nachos) : Ordonnancement et synchronisation (8h encadrées)

Ce TP nécessite d'analyser uniquement le contenu des répertoires *kernel* et *userlib* et de comprendre la documentation associée.

Il est important de souligner qu'avec la version de NACHOS qui vous est livrée au départ, bien qu'il soit possible de compiler le noyau, il n'est pas encore possible d'exécuter des programmes. Ce sera possible (pour des programmes non multithreadés, comme par exemple le programme *hello*) d'exécuter des programmes seulement quand les outils de synchronisation et les threads seront développés.

2.5.1 Outils de synchronisation

L'objectif ici est d'implanter des outils de synchronisation (voir § 1.5.2). Le travail demandé est le suivant :

1. Compléter le fichier *synch.cc* pour écrire le code des sémaphore, des verrous (locks), et des variables de condition. On veillera à assurer l'atomicité des primitives de synchronisation. Le code des variables de condition n'est pas indispensable au démarrage du noyau, il pourra être écrit/testé plus tard.
2. Créer le code des appels système de synchronisation. Pour ce faire, consulter le paragraphe 1.6 qui décrit le fonctionnement des appels système sous NACHOS. Les fichiers concernés par les appels système sont les suivants :
   — fichier assembleur MIPS *sys.s*, qui est lié avec les programmes utilisateur et utilise l'instruction MIPS *syscall* pour exécuter un point d'entrée du noyau (voir paragraphe 1.6)
   — fichier *exception.cc*, qui intercepte tous les appels au noyau et, selon le numéro de l'appel (contenu du registre MIPS *r2*), appelle la méthode mettant en œuvre l'appel.

   Seul le fichier *exception.cc* est à compléter, le fichier *sys.s* intégrant déjà l'ensemble des appels système. On veillera à vérifier, avant appel des méthodes des objets de synchronisation, *(i)* que les éléments passés en paramètres correspondent à des objets précédemment créés par le noyau (variable globale *objectIDs*), et *(ii)* que les objets sont du type attendu (champ *typeId*).

2.5.2 Gestion de threads

Dans la version des sources qui vous a été livrée, un seul processus utilisateur s'exécute à un moment donné, et il est composé d'un seul flot d'exécution (thread). L'objectif ici est de supporter plusieurs threads. Pour cela, vous devrez écrire ou compléter les méthodes suivantes :

— Les méthodes *SaveProcessorState* et *RestoreProcessorState* de la classe Thread, dont le rôle est de sauvegarder (resp. restaurer) le contexte utilisateur du thread depuis (resp. vers) le processeur MIPS émulé.
— La méthode *Start* de la classe Thread, dont l'objectif est d'initialiser tous les éléments d'un thread (piles noyau et utilisateur), de l'associer au processus indiqué en paramètre, et de marquer le thread comme étant prêt à être exécuté. On utilisera pour cela la méthode *StackAllocate* de la classe *AddrSpace* pour allouer une nouvelle pile utilisateur,

la fonction *AllocBoundedArray* (voir *machine/sysdep.h*) pour allouer une nouvelle pile noyau. La taille de la pile utilisateur est spécifiée dans le fichier de configuration. La taille de la pile noyau est constante (constante *SIMULATORSTACKSIZE* dans le fichier *kernel/thread.h*).
On utilisera également les méthodes *InitSimulatorContext* et *InitThreadContext* de la classe Thread pour initialiser son contenu (contexte noyau et utilisateur). On indiquera également au processus spécifié que le nombre de threads a été augmenté. Enfin, le nouveau thread sera inséré dans la file des threads vivants (*g_alive*) et aussi dans celle des threads prêts.

— La méthode *Finish* de la classe Thread, dont l'objectif est de marquer le thread comme étant détruit (champ *g_thread_to_be_destroyed*) et de l'enlever de la file des prêts (appel à la méthode *Sleep*). La destruction proprement dite n'est pas effectuée tout de suite, car on est encore en train d'utiliser sa pile ; elle sera effectuée dans l'ordonnanceur (méthode *SwitchTo*) après le changement de contexte.

Programmes de test. Tester de manière intensive votre code via l'écriture de programmes de test :

— des sémaphores (rendez-vous entre 3 threads, synchronisation producteur/consommateur, etc)
— des verrous et variables de condition.

Bonus (pour les courageux). Ajout à NACHOS d'un des points suivants :

— gestion des threads par priorité ;
— ajout de partage de temps à l'ordonnançeur ;
— ajout de barrières comme outil de synchronisation supplémentaire ;
— ajout de paramètres aux threads créés par *threadCreate* ;
— ajout de paramètres aux programmes utilisateur.

2.6 TP 6 (Nachos) : Gestion d'entrées/sorties caractères (4 h encadrées)

Ce TP nécessitera une intervention au niveau du répertoire *drivers* (fichiers *drvACIA.h* et *drvACIA.cc*).

2.6.1 Construction d'un pilote série

L'objectif de ce TP est de gérer l'envoi et la réception de messages via une liaison série. Il s'agit de créer un pilote (ou driver) de coupleur série, capable d'émettre et de recevoir de manière asynchrone des lignes de caractères. Pour cela nous utiliserons le coupleur série simulé. La synchronisation sera effectuée en utilisant le noyau de synchronisation écrit dans le TP précédent.

2.6.2 Description du pilote de coupleur série

Un pilote de coupleur série est un dispositif capable d'échanger des messages. Un message est une séquence d'au plus *lmax* caractères ASCII, terminée par le caractère de code ASCII 0. Les opérations offertes par le pilote permettront d'émettre et de recevoir un message (voir § 1.4.1). Le pilote gérera le coupleur par attente active dans un premier temps, puis par interruptions dans un second temps. Le pilote est représenté par l'objet global *g_acia_driver*, instanciation de la classe *DriverACIA*.

2.6.3 Les procédures d'émission et de réception

Avec attente active. La procédure d'émission gère l'envoi du tampon passé en paramètre, caractère par caractère, en procédant à une attente active sur le registre de données en émission.

La procédure de réception gère la réception du message, caractère par caractère, en procédant à une attente active sur le registre de données en réception.

Sous interruptions. La procédure d'émission remplit le tampon d'émission (si ce dernier est libre), puis demande les interruptions en émission. C'est ensuite la routine de traitement d'interruptions en émission qui entretient le transfert du message. Lorsque ce dernier a transféré le dernier caractère, il effectue un V sur le sémaphore d'accès au tampon d'émission pour le libérer et permettre l'exécution de la requête en émission suivante.

En réception, la routine de traitement d'interruptions stocke tous les caractères reçus dans le tampon de réception (si ce dernier est libre) jusqu'à la fin du message. À la réception du dernier caractère du message, la routine de traitement d'interruption fait un V sur le sémaphore d'accès au tampon de réception. La procédure de réception vide alors le contenu du tampon de réception pour le transmettre à l'appelant. Enfin elle libère le tampon en réception pour permettre l'exécution de la requête en réception suivante.

2.6.4 Travail demandé

Le travail demandé est le suivant :

1. Implémenter dans le fichier *drivers/drvACIA.cc* :

   - le constructeur de la classe *DriverACIA* qui réalise l'initialisation du coupleur ;
   - les méthodes **TtySend** et **TtyReceive** de manière à fonctionner en attente active, en prévoyant le partage du coupleur série entre plusieurs threads.
2. Ecrire un programme de test avec un émetteur s'exécutant sur une machine et un consommateur s'exécutant sur une autre machine.
3. Les exécuter sur deux machines distantes.
4. Reprendre la première partie de manière à fonctionner sous interruptions. Implémenter les méthodes **interruptSend** et **interruptReceive**.
5. Reprendre la troisième étape et comparer les deux modes de fonctionnement à l'aide des statistiques obtenues.

Le fichier de configuration. Veillez bien à mettre à jour les champs *useACIA* et *TargetMachineName* dans le fichier de configuration *nachos.cfg*. Deux fichiers de configuration seront nécessaires : un pour l'émetteur, l'autre pour le récepteur. Les numéros de port de communication devront être inversés dans les deux fichiers. Attention ! l'utilisation du coupleur série induit une nette diminution de la vitesse d'exécution du système. Il est judicieux de copier les fichiers test sans que le système utilise le coupleur série, puis de redémarrer le système en mettant à jour les champs *useACIA*, et *FormatDisk*.

Chapitre 3

Travaux pratiques du module SGM (18h)

3.1 TP1 (Nachos) - Gestion de mémoire virtuelle (10h encadrées)

L'objectif de ce TP est de réaliser les deux éléments centraux d'un système de gestion de mémoire virtuelle :

— la routine de traitement des défauts de page qui charge en mémoire à la demande les pages depuis le disque ;

— un algorithme de remplacement de page, appelé également *voleur de page*.

On étudiera pour ce TP le contenu du répertoire *vm*. Le système de gestion de mémoire virtuelle doit correspondre à la description donnée au paragraphe 1.8. Il est recommandé de tester votre système progressivement, lorsque vous aurez réalisé chacune des trois étapes suivantes.

3.1.1 Espaces d'adressage séparés

Dans la version de NACHOS mise en place lors du premier TP, un seul processus (multithreadé si vous avez terminé la première partie de ce TP) s'exécute. Ce qui est demandé ici est de permettre l'exécution de plusieurs processus ayant des espaces d'adressage séparés (*i.e.* ayant chacun leur table des pages privée). Pour ce faire, étudier et modifier si nécessaire les méthodes *SaveProcessorState* et *RestoreProcessorState* de la classe *Thread* (gestion du champs *translationTable* de la MMU.

Dès que vous aurez terminé cette partie du TP, vous pourrez utiliser le *shell* fourni avec NACHOS.

3.1.2 Chargement des programmes à la demande

Jusqu'à présent, le code et les données des programmes étaient chargés en mémoire dès leur lancement. Il s'agit ici de changer le chargement de l'exécutable et l'allocation de la pile utilisateur de manière à ne pas allouer les pages en mémoire dès le chargement, mais plutôt de déclencher un défaut de page lors de leur premier accès, en mettant le bit *valid* de la table de traduction d'adresses à *false*. Le code concerné est le constructeur de la classe *AddrSpace* et la méthode *StackAllocate* de la classe *AddrSpace*.

La routine de traitement des défauts de page s'occupera alors d'allouer une page physique, et de la remplir avec le contenu de la page virtuelle demandée, avant de redonner la main au thread ayant provoqué le défaut de page.

Routine traitement des défauts de pages

Écrire la routine de traitement des défauts de page (méthode *PageFault* de la classe *PageFaultManager*). Dans un premier temps, on supposera qu'il existe toujours une page de libre en mémoire réelle, et qu'il n'y a qu'un seul thread qui s'exécute dans le programme que l'on exécute. Les actions à réaliser par la routine sont les suivantes :

1. charger la page manquante depuis le disque (le numéro de secteur sera trouvé dans la table de traduction d'adresses, et aura été initialisée au chargement du programme en mémoire) dans une page temporaire. Dans le cas de pages anonymes (voir la section 1.8.2), aucun chargement depuis le disque n'est à faire : il suffit de remplir la page temporaire avec des 0.
2. recherche d'une page libre en mémoire réelle (on utilisera pour cela le gestionnaire de mémoire physique - classe *PhysicalMemManager* du fichier *physMem.cc*). Copie du contenu de la page temporaire vers cet emplacement.
3. modifier la table de traduction d'adresses et la table des pages réelles en conséquence, en particuler dans la méthode *AddPhysicalToVirtualMapping*.

Les structures de données et constantes utilisées par la routine de traitement des défauts de page sont :

— la table des pages réelles ;
— le nombre de pages réelles, et la taille d'une page, qui sont accessibles via l'objet de configuration global *g_cfg* (champs *g_cfg->NumPhysPages* et *g_cfg->PageSize*) ;
— la table des pages du processus courant ;
— les objets modélisant le gestionnaire de swap (objet *g_swap_manager*) ;
— la mémoire de la machine, accessible via l'objet global g_machine (champ *g_machine->mainMemory*).

3.1.3 Algorithme de remplacement de page

Lorsque la mémoire physique est pleine et qu'un processus veut charger en mémoire une page qui est absente de la mémoire physique, la routine de traitement des défauts de page appelle un algorithme de remplacement de page, qui réquisitionne une page à un processus, en la recopiant sur disque si nécessaire. On vous demande ici d'implanter l'algorithme de remplacement de page qui a été présenté dans le paragraphe 1.8. Ceci sera réalisé en complétant la méthode *EvictPage* de la classe *PhysicalMemManager*.

3.1.4 Trucs et astuces

Nous énumérons ci-dessous quelques trucs vous permettant d'éviter des problèmes lors de la mise au point de votre système de gestion de la mémoire virtuelle :

— Un processus P1 qui recopie une page réelle X sur disque perd la main sur l'entrée/sortie disque. Pour éviter qu'un autre processus P2 veuille aussi recopier cette même page X sur disque suite à son propre appel au voleur de page, on a introduit le bit *locked* associé à chaque page en mémoire réelle.

— Soient deux threads *T1* et *T2*. Si *T2* déclenche un défaut de page pendant que *T1* est en train de résoudre ce même défaut de page, il faut que *T2* se bloque pendant la résolution du défaut de page. Pour ce faire, on utilisera le bit *IO* présent dans la table de traduction d'adresses.
— Quand un processus P1 perd la main sur une E/S disque, le processus P2 qui prend la main peut accéder au voleur de page : il faut donc faire attention à ce que P1 retrouve son contexte quand il reprendra la main. Pour cela, utiliser une *variable locale local_i_clock* pour le parcours de la table des pages réelles dans l'algorithme de remplacement de page par chaque processus, tout en maintenant une variable globale *i_clock*.

3.1.5 Bonus

On pourra implanter une des fonctionnalités suivantes :
— Ajout d'un TLB en plus de la MMU. Examiner l'impact sur les performances en modifiant les statistiques.
— Table des pages hiérarchiques ou inversées à la place des tables des pages linéaires telles qu'elles sont prévues dans Nachos
— Segments de mémoire partagés par plusieurs processus
— Un mécanisme de copy-on-write

3.2 TP 2 (Nachos) : Introduction de fichiers mappés (4h encadrées)

Dans ce TP, nous vous demandons de mettre en œuvre des fichiers mappés (voir le cours pour une description du principe de fonctionnement). On procédera de la manière suivante :

- Ajout d'un nouvel appel système, nommé *Mmap*. La partie assembleur est déjà écrite, il vous suffira de récupérer l'exception dans le fichier *kernel/exception.cc*.
- Remplissage d'une nouvelle méthode nommée *Mmap* dans la classe *AddrSpace* pour réaliser ce nouvel appel système. La méthode réservera un ensemble de pages consécutives dans l'espace d'adressage du processus (méthode *Alloc* de la classe *AddrSpace*) et y associera les adresses disques dans le fichier que l'on désire mapper (déplacement dans le fichier mappé).
  L'adresse disque utilisée sera un déplacement dans le fichier mappé. La routine de défaut de page lira le contenu du fichier en utilisant la méthode *ReadAt* . La routine de défaut de page devra être modifiée, car elle est prévue au départ pour lire des secteurs dans les fichiers exécutables uniquement. Pour ce faire, on maintiendra une liste de fichiers mappés par processus, contenant pour chaque fichier mappé : la première page mappée, le nombre de pages mappées, le descripteur de fichier correspondant (*OpenFile**).
- Ecrire un programme de test des fichiers mappés, par exemple un programme de tri des éléments d'un tableau d'entiers, stockés au préalable dans un fichier.
- Gérer le cas de l'éviction de la mémoire d'une page correspondant à un fichier mappé, ainsi que la fin d'un processus ayant mappé un fichier. Dans ces deux cas, il est nécessaire de recopier la page dans le fichier mappé si elle a été modifiée, afin de ne pas perdre les modifications.

Par souci de simplification, on ne recyclera jamais les pages virtuelles qui ont été un jour mappées dans un fichier (on ne réalisera pas de méthode *Munmap*).

3.3 TP3 (Linux) - Utilisation des segments de mémoire partagée, sémaphores, files de messages

3.3.1 Utilisation des sémaphores et des segments de mémoire partagée

Sémaphores

L'utilisation de sémaphores est effectuée à travers 3 fonctions :

— *semget* : cette primitive crée un ensemble de n sémaphores ;
— *semctl* : cette primitive modifie les paramètres des sémaphores (valeur initiale, destruction) ;
— *semop* : cette primitive réalise les fonctions P et V.

— #include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, /* Voir ci-dessous */
int nsems, /* nsems représente le nombre de sémaphores à allouer */
int semflg) ; /* semflg sera positionné à 0660 (droits d'accès) */

La fonction rend un entier qui est l'identificateur du groupe de sémaphores alloués (*nsems* est le nombre d'éléments de ce groupe). Les sémaphores alloués sont numérotés à partir de 0. L'identificateur rendu par *semget* est nécessaire pour l'appel des deux autres fonctions. On mettra dans le paramètre *key* la valeur *IPC_PRIVATE* si le groupe de sémaphores est créé avant un appel à fork (sinon, regarder la page de manuel de *semget()*). Le paramètre *semflg* représente les droits d'accès sur le groupe de sémaphores, et sera positionné à *0660*. En cas d'erreur la fonction rend -1.

— int semctl(int semid, int semnum, int cmd, union semun arg) ;

Cette fonction permet d'appliquer une action, définie par les paramètre *semid* et *semnum* (groupes de sémaphores et numéro du sémaphore dans le groupe). L'action à effectuer est définie par le paramètre *cmd* (SETVAL : initialisation, IPC_RMID : destruction). *arg* est un argument optionnel à l'action. Lorsque cette fonction est appelée dans l'objectif de détruire des sémaphores, le numéro du sémaphore dans le groupe de sémaphore n'est pas exploité et tous les sémaphores du groupe sont détruits. Lorsque *cmd* est initialisé à *SETVAL*, on doit initialiser le paramètre *arg.val* à la valeur du compteur du sémaphore. A noter que le type union *semun* et la variable de ce type doivent être déclarés par l'utilisateur. En cas d'erreur la fonction rend la valeur -1.

— int semop(int semid, struct sembuf *sops, int nsops) ;

Cette fonction permet d'effectuer un P ou un V sur un groupe de sémaphores identifié par *semid*. Le paramètre *sops* est un tableau d'actions, la taille de ce tableau étant définie par le paramètre *nsops*. En cas d'erreur la fonction rend la valeur -1.

La structure *sembuf* est composée de trois champs :
— short sem_num ; /* Numéro du sémaphore concerné dans le groupe*/
— short sem_op ; /* Type de l'opération : P : -1 ou V : 1 */
— short sem_flg ; /* sem_flg sera initialisé à 0 */

Segments de mémoire partagée

L'utilisation de segments de mémoire partagée est effectuée à travers 4 fonctions :

— *shmget* : cette primitive crée un segment de mémoire partagée
— *shmctl* : cette primitive modifie les paramètres des segments de mémoire partagée (destruction)
— *shmat* : cette primitive attache un segment de mémoire partagée au processus
— *shmdt* : cette primitive détache un segment de mémoire partagée du processus

— #include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg) ;

La fonction crée un segment de mémoire partagée. *key* définit la visibilité du segment de mémoire partagée, et sera fixé à *IPC_PRIVATE* si le segment de mémoire partagée est créé avant un appel à *fork()* (sinon, consulter la page de manuel). *size* définit la taille du segment de mémoire partagée, et *shmflags* définit les droits d'accès sur ce segment (on utilisera la valeur *0660*). La fonction rend un entier qui est l'identificateur du segment de mémoire partagée. Cet identificateur est nécessaire pour l'appel des trois autres fonctions. En cas d'erreur la fonction rend la valeur -1.

— int shmctl(int shmid, int cmd, struct shmid_ds *buf) ;

Effectue une action sur le segment d'identificateur *shmid*, cette action étant définie par *cmd*. Pour détruire un segment, *cmd* doit être initialisé à 0 et *buf* à 0. En cas d'erreur la fonction rend la valeur -1.

— void *shmat(int shmid, const void*shmaddr, int shmflg) ;

Cette fonction attache un segment de mémoire partagées dans l'espace d'adressage du processus appelant. Si le paramètre *shmaddr* est 0, le système choisit l'adresse de projection du segment et la retourne en résultat. Le paramètre *shmflg* définit le mode d'attachement du segment et est un ou bit à bit des valeurs suivantes : SHM_RND pour un choix d'adresse du segment par le système, SHM_RDONLY pour un segment en lecture seule. La fonction retourne l'adresse du segment de mémoire partagée ou -1 en cas d'erreur.

— int shmdt(void*shmaddr) ;

Détache un segment de mémoire partagée. En cas d'erreur la fonction rend la valeur -1.

Exercice. Réaliser un mécanisme de producteur-consommateur par le biais d'un segment de mémoire partagée. La synchronisation entre les deux processus (père et fils) utilisera les sémaphores. Pour mieux vous faire comprendre ce que l'on peut faire et ne pas faire avec des segments de mémoire partagée, les deux processus se partageant le segment de mémoire partagée n'auront aucun lien de filiation, et le segment de mémoire partagée contiendra une liste chainée de 5 entiers. On pourra utiliser la fonction *nice* pour diminuer la priorité du processus consommateur et ainsi observer les différents entrelacements possibles entre productions et consommations.

Attention. Les outils sémaphore (et par la suite segments de mémoire partagée et files de messages) ne sont pas détruits automatiquement lors de la terminaison du processus qui les a créés. Par conséquent, en cas de terminaison anormale d'un processus à cause d'une erreur de programmation, les sémaphores que vous auriez du détruire à la fin de l'exécution du processus sont conservés. La commande *ipcs* permet de visualiser les sémaphores, segments de mémoire partagée et files de messages qui restent actifs. La commande *ipcrm* permet de les détruire. Le nombre d'outils de communications par machine étant limité, vérifiez avant de vous déconnecter (et de temps en temps en cours de session) que vous les avez bien tous détruits.

3.3.2 Utilisation de files de messages

L'utilisation de files de messages est effectuée à travers quatre fonctions :
- *msgget* : cette primitive crée une file de messages ;
- *msgctl* : cette primitive modifie les paramètres associés à une file de messages (destruction) ;
- *msgsnd* : cette primitive met un message dans une file de messages ;
- *msgrcv* : cette primitive retire un message d'une file de messages.

- #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/msg.h>
  int msgget(key_t key, int msgflg) ;

  La fonction rend un entier qui est l'identificateur de la file de messages créée. Les paramètres sont similaires à la création de segments de mémoire partagée. En cas d'erreur la fonction rend la valeur -1.

- int msgctl(int msqid, int cmd, struct msqid_ds *buf) ;

  Pour détruire une file de message, mettre *cmd* à IPC_RMID et *buf* à 0. En cas d'erreur la fonction rend la valeur -1.

- int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg) ;

  Envoi d'un message situé à l'adresse msgp, de taille *msgsiz* en octets, sur la file *msqid*. Le paramètre *msflfg* sera positionné à 0. En cas d'erreur la fonction rend la valeur -1.

- int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg) ;

  Réception d'un message sur la file *msqid*. Les paramètres *msgp* et *msgsz* définissent l'adresse et la taille en octets de la zone dans laquelle le message sera recopié. Le paramètres *msgtyp* sera initialisé à 0 (tous les types de messages seront acceptés). Le paramètre *msgflg* sera initialisé à 0. En cas d'erreur la fonction rend la valeur -1.

Dans les routines *msgsnd* et *msgrcv*, le tampon utilisateur *msgp* doit débuter par un entier long identifiant le type du message (valeur non nulle) et se poursuivre par le corps du message. L'allocation du tampon est à la charge de l'utilisateur de ces deux routines.

On utilisera la fonction *strcpy* pour recopier les chaînes de caractères.

Exercice. Utiliser les files de message pour faire communiquer deux processus cycliques : un processus producteur, qui envoie une suite de messages, et un processus consommateur, qui reçoit et affiche le contenu des messages. Les fichiers source seront nommés *emetteur.c* et *recepteur.c*.

3.4 TP4 (Linux) - Fonctions Unix mprotect, mmap, munmap

L'objectif de ce TP est d'utiliser trois fonctions de gestion de la mémoire, les fonctions *mmap*, *munmap* et *mprotect* (par curiosité, on pourra regarder également les routines *mlock*, *munlock*, *memcntl*).

3.4.1 Fonction mprotect

La fonction *mprotect* sert à changer les attributs de protection d'une zone de l'espace d'adressage virtuel d'un processus, cette zone devant être cadrée sur un multiple de pages. La taille d'une page pourra être obtenue par la routine *getpagesize*.

#include <sys/mman.h>
int mprotect(void *addr, size_t len, int prot) ;

La fonction *mprotect* change les attributs de protection de la zone *[addr, addr + len)*, en alignant si nécessaire la fin de la zone sur une frontière de page. prot est la nouvelle protection sur les pages de la zone, et est construit comme un ou bit-à-bit entre les valeurs suivantes :
— PROT_READ : droits en lecture ;
— PROT_WRITE : droits en écriture ;
— PROT_EXEC : droits en exécution ;
— PROT_NONE : aucun droit d'accès.
La fonction retourne 0 en cas de succès et -1 en cas d'erreur, le code d'erreur étant indiqué dans la variable *errno*.

3.4.2 Fonctions mmap et munmap

La fonction *mmap* permet d'établir une correspondance entre un fichier et une région de l'espace d'adressage d'un processus (fichier mappé). Une fois mappé, le fichier n'est plus accédé par des fonctions d'entrée/sortie (*read*, *write*) mais directement par des accès mémoire.

#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off) ;
— *addr* : adresse à laquelle on veut établir la correspondance (0 si on laisse le système choisir)
— *len* : taille (en octets) de la région d'espace d'adressage concernée
— *prot* : détermine les accès permis sur la région (lecture, écriture, exécution). Ce paramètre doit être soit PROT_NONE ou un ou bit à bit des valeurs PROT_READ, PROT_WRITE ou PROT_EXEC. prot doit être en accord avec les droits autorisés sur le segment mémoire
— *flags* : options pour établir la correspondance. Ce paramètre sera positionné à MAP_SHARED pour signifier que les écritures devront être reportées dans le fichier.
— *filedes* : descripteur de fichier concerné (obtenu par open)
— *off* : offset (en octets) concerné par la routine mmap (la mise en correspondance concerne len octets à partir de l'offset *off* dans le fichier). Doit être aligné sur une frontière de page.
— valeur de retour : adresse à laquelle la correspondance est établie en cas de succès, -1 en cas d'erreur

#include <sys/mman.h>
int munmap(void addr, size_t len) ;
— supprime la correspondance qui existait à l'adresse virtuelle *addr* sur len octets. *addr* doit correspondre à une adresse ayant été retournée au préalable par *mmap*.
— la fonction retourne 0 en cas de succès, -1 en cas d'erreur

3.4.3 Exercice

On considère un programme P qui utilise une structure de données (par exemple une grosse matrice M) qui sera stockée dans un fichier. Par exemple, le programme pourra afficher dix valeurs aléatoires dans le fichier.

Utiliser la routine *mprotect* pour compter et afficher à l'écran le pourcentage des pages de M accédées par le programme (par rapport au nombre total de pages de M). On nommera le fichier source *matrice.c*.

Le principe retenu sera le suivant. On travaillera directement sur M en mémoire, en utilisant la fonction *mmap*. Pour simplifier, la taille de M sera directement un multiple de la taille d'une page (voir la fonction *getpagesize*).

On protégera les pages de M contre tout type d'accès. Le signal *SIGSEGV* sera détourné pour détecter les violations de protection et comptabiliser le pourcentage de pages accédées. Pour détourner le signal, on utilisera la fonction *sigaction* plutôt que la fonction *signal*, car *sigaction* permet de récupérer non seulement le numéro de signal mais aussi des informations sur chaque type de signal (par exemple dans le cas d'un *SIGSEGV* l'adresse en cause).

#include <signal.h>
int sigaction(int sig, const struct sigaction *act,struct sigaction *oact) ;
— sig : numéro de signal
— act : adresse d'une structure définissant l'action à connecter au signal
— ocat : adresse d'une structure permettant de récupérer l'ancienne action
Les champs suivants des deux structures devront être initialisés :
— void (*sa_sigaction)(int, siginfo_t *, void *) ; : adresse de l'action associée au signal
— int sa_flags ; type de connexion : ici uniquement *SA_SIGINFO*, indiquant que la fonction de traitement du signal recevra non seulement le numéro de signal sig, mais aussi deux paramètres supplémentaires, dont *sip*, de type *siginfo_t*. sip->si_addr dans le cas d'un signal *SIGSEGV* contient l'adresse en cause.

Annexe A

Notes sur l'utilisation de gdb

Le GNU Debugger (`gdb`) permet d'étudier l'état d'exécution d'un programme qui a *planté*, ou d'exécuter le programme *pas-à-pas*, afin de remédier plus facilement aux éventuels bugs qui y sont disséminés. C'est une puissante alternative aux *printf*, notamment dans le cas où un programme a planté, pour repérer l'emplacement du plantage...

Pour les inconditionnels des interfaces graphiques *ddd* ajoute une sur-couche graphique rudimentaire à *gdb*.

A.1 Compilation du programme à déboguer

Pour qu'un programme `C` ou `C++` puisse être passé au crible de gdb, il doit être compilé avec l'option `-g` (ou `-ggdb`) de gcc :

```
shell> cat toto.c
#include <stdlib.h>

void met_a_zero(int * tab, int len)
{
  int i;
  for (i = 0 ; i < len ; i++)
        tab[i] = 0;
}

int main()
{
        int taille = 5678;
        int * mon_tableau = (int*) malloc(taille);
        met_a_zero(mon_tableau, taille);
        return 0;
}
shell> gcc -g -c toto.c
shell> gcc -o toto toto.o
shell> ./toto
bash: segmentation fault (core dumped)  ./toto
```

Par défaut, NACHOS est déjà compilé avec l'option `-g`.

Le dernier message n'est pas une insulte ! C'est une précieuse aide que vous rend le système : il génère un fichier **core** qui contient l'image de la mémoire occupée par le programme au moment où il a planté. C'est grâce à une image de ce type que gdb parvient à inspecter l'état et les variables de votre programme (voir la section A.2.3).

A.2 Utilisation de gdb

A.2.1 Lancement

Le debugger se lance à l'aide de la commande `gdb`. Une fois lancé, on obtient le prompt à partir duquel on contrôle le debugger :

```
shell> gdb
GNU gdb (GDB) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
(gdb)
```

Quelques commandes importantes sont présentées ci-dessous. À tout moment, vous pouvez avoir la documentation sur les (autres) commandes disponibles en tapant la commande *help* au prompt de `gdb`.

A.2.2 Charger un programme et en afficher le code

Pour charger le programme à debugger, ici, `toto`, on tape `file toto`. Pour en afficher le code, il suffit de taper `list`. Le debugger affiche alors les premières lignes du code. Pour voir les suivantes, il suffit de retaper `list` ou bien d'appuyer directement sur la touche `entrée`. En effet, lorsqu'aucune commande n'est spécifiée, gdb exécute à nouveau la commande précédente.

```
(gdb) file toto
Reading symbols from toto...done.
(gdb) list
3       void met_a_zero(int * tab, int len)
4       {
5         int i;
6         for (i = 0 ; i < len ; i++)
7               tab[i] = 0;
8       }
9
10      int main()
11      {
12              int taille = 5678;
```

A.2.3 Repérer l'endroit où le programme a *planté*

Il y a deux méthodes pour récupérer l'état du programme lors de son plantage :

— Par exécution directe depuis gdb ;

— Par analyse après plantage, grâce au fichier *core*.

Exécution directe du programme depuis gdb

La commande `run`, permet de lancer le programme directement depuis gdb. Lorsque le programme *plante*, gdb affiche l'endroit et les circonstances de la faute :

```
(gdb) run
Starting program: /udd/puaut/toto

Program received signal SIGSEGV, Segmentation fault.
0x105a0 in met_a_zero (tab=0x20800, len=5678) at toto.c:7
7               tab[i] = 0;
```

Utilisation du fichier *core*

Si on a oublié de lancer gdb avant exécution pour analyser un plantage, le fichier **core** généré [1] peut servir à faire une analyse *post-mortem* du plantage. Il suffit alors de charger le programme comme en A.2.2, puis d'utiliser la commande *core* de *gdb*, en lui fournissant le chemin vers le fichier **core** :

```
(gdb) core /udd/puaut/core
Core was generated by `./toto'.
Program terminated with signal 11, Segmentation Fault.
Reading symbols from /usr/lib/libc.so.1...done.
Reading symbols from /usr/lib/libdl.so.1...done.
Reading symbols from /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1...done.
#0  0x105a0 in met_a_zero (tab=0x20800, len=5678) at toto.c:7
7               tab[i] = 0;
```

A.2.4 Exécuter le code pas-à-pas

Si on choisit de lancer le programme directement depuis gdb, il est possible d'observer son état avant l'apparition de la faute : c'est le rôle des points d'arrêt, ou *breakpoint*, qu'on peut mettre ça et là dans le programme pour l'obliger à s'interrompre aux endroits du code qui nous intéressent.

Ici, nous allons nous attarder sur la fonction `main`. Pour cela, on place un *breakpoint* à son début en tapant `break main`. Il est aussi possible de placer un point d'arrêt à une ligne particulière en spécifiant son numéro : `break 12`. Si plusieurs fichiers sont impliqués, on écrit : `break toto.c:main` ou `break toto.c:12` afin de lever toute ambiguïté :

```
(gdb) break toto.c:main
Breakpoint 1 at 0x105c4: file toto.c, line 12.
```

Notre premier *breakpoint* est placé, on peut à présent lancer le programme avec `run`. Le debugger interrompt le programme dès l'entrée dans la fonction `main` :

```
(gdb) run
Starting program: /udd/puaut/toto
Breakpoint 1, main () at toto.c:12
12              int taille = 5678;
```

1. Si un tel fichier n'est jamais généré, faites `man limit` ou `man ulimit`, suivant votre *shell*, pour modifier le paramètre système `coredumpsize`.

Il est alors possible d'exécuter les instructions une à une grâce à la commande `next`. Cette dernière ne permet pas de suivre un appel de fonction. La fonction est exécutée directement. Lorsqu'on veut explorer la fonction appelée, on utilise `step` :

```
(gdb) next
13              int * mon_tableau = (int*) malloc(taille);
(gdb) next
14              met_a_zero(mon_tableau, taille);
(gdb) step
met_a_zero (tab=0x20800, len=5678) at toto.c:6
6         for (i = 0 ; i < len ; i++)
```

Pour continuer l'exécution jusqu'à un hypothétique prochain *breakpoint*, on utilise la commande `cont` :

```
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x105a0 in met_a_zero (tab=0x20800, len=5678) at toto.c:7
7               tab[i] = 0;
```

Dans cet exemple, aucun autre *breakpoint* n'avait été posé. Le programme a donc continué jusqu'à la même erreur que précédemment.

A.2.5 Identifier la pile d'appels qui a conduit au *plantage*

La commande `run` ou l'utilisation d'un fichier **core** nous indique que l'erreur a eu lieu dans la fonction `met_a_zero()`. On utilise la commande `backtrace` pour avoir une idée de la pile d'appels de fonctions qui a conduit à cette erreur :

```
(gdb) backtrace
#0  0x105a0 in met_a_zero (tab=0x20800, len=5678) at toto.c:7
#1  0x105f0 in main () at toto.c:14
```

Ceci s'interprète en remontant : notre fonction `main()` a appelé notre fonction `met_a_zero()`, et c'est l'endroit où s'est produit l'erreur. Il est alors possible d'afficher les valeurs des différentes variables (par exemple celle de `i`, voir la section suivante), en plus des arguments qui sont affichés par défaut (ici `tab` et `len`).

A.2.6 Afficher le contenu d'une variable

Exécuter le code pas-à-pas, ou constater un cas d'erreur (comme ici) est relativement peu intéressant si l'on ne peut pas consulter les données manipulées par le programme. Il est possible de le faire en utilisant la commande `print` suivie du nom de la variable :

```
(gdb) print i
$1 = 3586
```

Ceci indique que le programme a *planté* au moment où le 3587ème élément du tableau était écrit.

On peut demander d'afficher des expressions plus compliquées que `i` ou `taille` : il suffit de reprendre la syntaxe du C. Ainsi :

```
(gdb) print tab[12]
$2 = 0
(gdb) print tab
$3 = (int *) 0x20800
(gdb) print *tab
$4 = 0
(gdb) print &tab
$5 = (int **) 0xffbee464
(gdb) print &tab[12]
$6 = (int *) 0x20830
```

A.2.7 Changer le cadre de pile courant

La commande `print` permet d'afficher la valeur des variables locales (ici `i`) de la fonction courante (ici : `met_a_zero()`). Il est possible d'afficher les variables locales des fonctions qui sont en amont sur la pile d'appels, grâce à la commande `frame`. On fournit à cette commande l'identifiant de la fonction indiqué par la commande `backtrace` sous la forme "`#identifiant adresse in nom_fonction()...`". Par exemple, pour afficher la valeur de la variable `taille` locale à la fonction `main()` alors qu'on est déjà dans `met_a_zero()`, on peut faire :

```
(gdb) backtrace
#0  0x105a0 in met_a_zero (tab=0x20800, len=5678) at toto.c:7
#1  0x105f0 in main () at toto.c:14
(gdb) frame 1
#1  0x105f0 in main () at toto.c:14
14              met_a_zero(mon_tableau, taille);
(gdb) print taille
$7 = 5678
```

A.2.8 Interpréter l'erreur

gdb vous permet de réunir tous les indices afin de trouver l'origine de l'erreur. Mais réunir des indices ne suffit pas pour mener une enquête à son terme : il faut à la fois réunir les *bons* indices, et les interpréter (ce qui va de paire), pour essayer d'en deduire le scenario qui a mené à l'erreur. Cela, vous seuls pouvez le faire, car aucun outil n'est encore assez intelligent pour ce genre d'enquête.

Dans notre exemple simple, on remarque que le tableau `mon_tableau` avait été alloué avec une taille de 5678 octets (variable `taille` du `main()`, voir section A.2.7). Or, pour i=3586 (dans `met_a_zero()`, voir section A.2.6), on est à 14344 octets (*ie* 3586 * `sizeof(int)`) du début du tableau, soit 8666 octets plus loin que la fin de la zone allouée pour le tableau... Une fois le scénario de l'erreur identifié, la correction est en général "toute bête", comme c'est le cas dans l'exemple (correction laissée à titre d'exercice).

A.2.9 Quitter gdb

Pour sortir du debugger, on utilise la commande `quit` :

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

A.3 Liste non exhaustive des commandes principales

help : donne la liste des rubriques d'aide et commandes disponibles sous gdb.

A.3.1 Contrôle de l'exécution

break [ligne—fonction] : place un point d'arrêt au début de la fonction ou de la ligne spécifiée. Le debugger renvoie le numéro de *breakpoint*.

delete [num] : retire un point d'arrêt spécifié.

run : lance ou relance le programme en cours.

cont : reprend l'exécution du programme (par exemple, après un avoir atteint un *breakpoint*).

next : exécute de la prochaine instruction.

step : exécute la prochaine instruction et suit les appels de fonction.

A.3.2 Visualisation

print [expression] : affiche la valeur d'une expression (variable, adresse, contenu d'un tableau, ...).

display [expression] : affiche la valeur d'une expression après chaque instruction exécutée.

backtrace : affiche la pile d'appels jusqu'à la fonction courante

A.3.3 Divers

frame [index_backtrace] : se place dans le contexte d'une fonction appelante pour en visualiser les variables locales.

pwd : affiche le répertoire courant.

cd [répertoire] : modifie le répertoire courant.

quit : quitte gdb.

file [exécutable] : charge un fichier à debugger.

list : affiche le code du programme.

list [fichier :num] : affiche le code de `fichier` à la ligne `num`.

help info : donne la liste des informations disponibles (commande **info**) sous gdb.

Annexe B

Foire aux Questions (FAQ)

— **Puis-je installer** NACHOS **sur ma machine Linux ?**
— Oui. En entrant *make*, le noyau se compile correctement. Ca ne permet pas de compiler de nouveaux programme utilisateur (il faut pour celà installer un compilateur croisé *mips*), mais ça permet s'exécuter les programmes existants récupérés avec l'archive NACHOS. Pour compiler les programmes utilisateur, deux possibilités : (i) recopier les binaires du compilateur croisé *mips* de l'ISTIC sur votre machine (en créant des liens symboliques pour que l'emplacement dans l'arborescence soit le même) ou (ii) installer un compilateur croisé *mips*.
— **Mon Linux est 64-bits, est-ce que ça marche ?**
— Oui, depuis 2013-2014. Dans une version antérieure, ce n'était pas le cas au préalable à cause de problèmes de passage de paramètres aux appel système entre les applications *mips* (32 bits) et le noyau (64-bits). Le mode de passage de paramètre a été modifié pour passer à un linux 64-bits.
— **Des problèmes d'installation ?**
— Les enseignants vous garantissent une installation correcte sur les machines de l'ISTIC, le reste est de votre responsabilité. Si vous résolvez vos problèmes d'installation sur d'autres plate-formes que celles de l'ISTIC, les informations peuvent être intégrées à cette FAQ, à la page web du module ou au code source lui-même.
— **Est-ce que** NACHOS **s'exécute sur d'autres environnements que Linux ?**
— Il ne s'exécute pas actuellement sur Cygwin. Il a été testé par contre sur macOS 10.8 et 10.9. Certains versions antérieures de macOS ne sont pas supportées, car elle ne dispoeant pas des fonctions *makecontext/swapcontext* nécessaires à la mise en place des changements de contexte). Pour exécuter sur macOS, ajouter dans le HOST_CPPFLAGS du Makefile.config -D_XOPEN_SOURCE.
— **Que doit-on savoir de C++ pour réaliser les TPs Nachos ?**
— Pas grand chose en réalité. Les TPs utilisent uniquement des classes C++, sans héritage. L'interface des classes est déjà écrite dans les fichiers d'inclusion (.h). Vous aurez uniquement à modifier le corps des classes (fichiers .cc). Si vous savez programmer en C, vous n'aurez pas de problème.
— **J'ai réussi à installer Nachos chez moi, est-ce que ça me dispense de venir en TP ?**
— Non, sauf dispense d'assiduité signée par le responsable du module, l'assiduité en TP est obligatoire. Une absence non justifiée à *deux* TP entraîne la note 0.