Travaux pratiques U.E. SEA
(Systèmes d'Exploitation Avancés)

Première année de master informatique

Année universitaire 2025-2026

Isabelle Puaut

Table des matières

Introduction

Ce document contient les éléments de compréhension et les sujets de travaux pratiques du module SEA (Systèmes d'Exploitation Avancés).

Objectifs des travaux pratiques

Les travaux pratiques visent à réaliser une partie de noyau de système d'exploitation. Pour ce faire, les noyaux de systèmes réels étant trop volumineux et complexes, les travaux pratiques se dérouleront sur un petit noyau de système d'exploitation, nommé NACHOS (*Not Another Completely Heuristic Operating System*), qui possède les bonnes propriétés suivantes :

- inclure des fonctionnalités présentes dans tout système d'exploitation moderne, comme par exemple la pagination à la demande ;
- être relativement simple et peu volumineux, permettant à des étudiants de comprendre son code puis le modifier après un investissement raisonnable ;
- de s'exécuter au dessus d'une machine émulée par logiciel, ce qui facilite sa mise au point, autorisant notamment l'utilisateur de débogueurs ;

Le code d'origine de NACHOS [1] a été remanié aux fils des ans, à l'INSA de Rennes puis à l'Université de Rennes 1. C'est le résultat de ces différentes strates de modifications qui sera utilisé lors des TP et nommé NACHOS.

Organisation du document

Le document est organisé comme suit. Une description détaillée de NACHOS est présentée dans le chapitre 1. Les sujets de TP sont donnés dans le chapitre 2. 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 à NACHOS, merci de le signaler (e-mail : puaut@irisa.fr).

1. *http ://www.cs.berkeley.edu/~tea/nachos*

Chapitre 1

Description de Nachos

Ce chapitre est consacré à une description du fonctionnement et de la structure interne de NACHOS. Le texte décrit le résultat *attendu* suite à la réalisation des travaux pratiques. Certaines parties détaillées ici ne sont pas dans les fichiers fournis, et seront à réaliser pendant les travaux pratiques. Certaines parties de NACHOS ne sont pas ou très peu décrites car elles ne seront pas utilisées lors des travaux pratiques.

Nous présentons dans le paragraphe 1.1 l'organisation générale du système, chacun de ses modules étant détaillé dans la suite. Le paragraphe 1.2 est consacré aux composants matériels. Les paragraphes suivants présentent les composants logiciels de NACHOS : pilotes de périphériques (paragraphe 1.3), cœur du noyau (gestion des processus légers et de leur synchronisation, paragraphe 1.4), gestion de mémoire virtuelle (paragraphe 1.5), et quelques fonctions utilitaires (paragraphe 1.6). Le fonctionnement des appels système est détaillé dans le paragraphe 1.7. Ce chapitre se termine par une description des étapes de développement, compilation et exécution d'un programme utilisateur avec NACHOS.

1.1 Structure de Nachos

1.1.1 Nachos en bref

L'objectif de NACHOS est 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* par logiciel. 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).

Les principaux éléments matériels émulés sont un processeur RISC-V 64-bits sur lequel vont s'exécuter les programmes utilisateur, un timer, des disques, une console et une unité de gestion de la mémoire (MMU - Memory Management Unit).

L'intérêt d'utiliser du matériel émulé est double. D'une part, le développement du noyau est simplifié, car le matériel émulé (processeur, périphériques), 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 pour mettre au point le code du noyau.

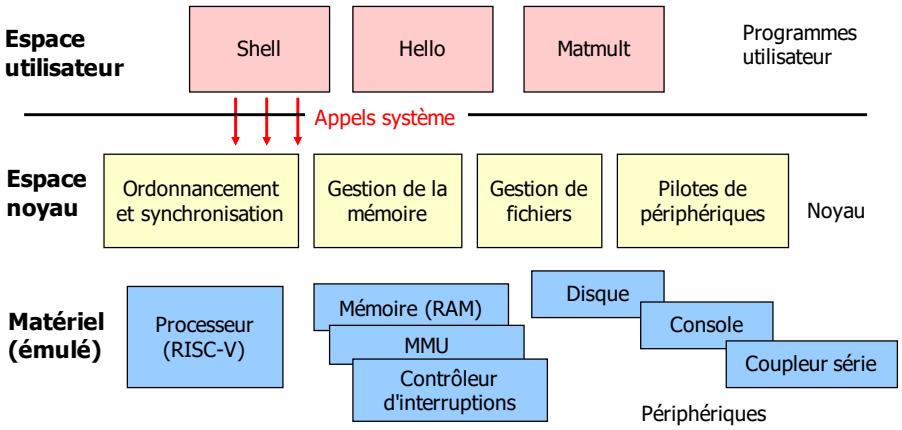

FIGURE 1.1 – Structure interne de NACHOS

1.1.2 Architecture logicielle de Nachos

NACHOS est développé en `C++`, en utilisant uniquement les caractéristiques de base du langage (encapsulation). Ceci permet de mieux structurer le code du noyau et en faciliter la compréhension, sans nécessiter une connaissance approfondie du langage `C++`. Les classes présentes dans NACHOS sont regroupés en différentes catégories, qui apparaissent également dans la structure des répertoires :

- le *noyau* qui comprend les fonctionnalités vitales du système : processus légers (threads), espaces d'adressages, 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 (mini libc) ;
- *test* : programmes utilisateur ;
- *doc* : documentation

Noyau de Nachos (répertoire *kernel*)

Le noyau de NACHOS gère un ensemble de processus légers (threads) s'exécutant de manière pseudo-parallèle. Nous nommons *processus lourd* 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), sans priorité, non-préemptif. 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.

Lorsqu'un thread est bloqué sur une primitive de synchronisation, un changement de contexte se produit, et le processeur est alloué au premier thread présent dans la file des prêts. Si aucun thread n'est prêt, le système patiente jusqu'à l'arrivée d'une interruption.

Les différents objets manipulés par NACHOS sont identifiés 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 (les paramètres des appels systèmes sont transmis dans des registres, donc ne peuvent pas être typés).

Pilotes de périphériques (répertoire *drivers*)

Les pilotes de périphériques sont la couche logicielle permettant d'effectuer des entrées/sorties sur les périphériques. Ils permettent de passer des opérations asynchrones proposées par le matériel, à des opérations en général synchrones (bloquantes). Ceci se fait par l'utilisation des interruptions et des primitives de synchronisation (sémaphores et verrous).

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 fichier Unix dans le système de fichiers de NACHOS lors du démarrage de ce dernier. Le fichier de configuration NACHOS permet soit de conserver le contenu de son disque entre deux lancements du système, soit de le réinitialiser à chaque lancement.

Gestion de la mémoire virtuelle (répertoire *vm*)

NACHOS dispose d'un système de gestion de mémoire virtuelle : 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 en *adresses réelles* dans la mémoire de la machine. La traduction d'adresses utilise des tables des pages (une par espace d'adressage) stockées en mémoire. 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 cette zone [1]. Le défaut de page, mécanisme inverse, intervient pour charger en mémoire réelle un page anciennement sur disque.

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

Les programmes utilisateur seront écrits en C. Des appels système ainsi que des routines diverses regroupées dans une bibliothèque liée statiquement aux programmes utilisateur (*libc*),

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

peuvent être utilisés. On ne dispose pas cependant de toutes les fonctionnalités de la *libc* classique, et encore moins de tous les appels système disponibles sous Unix (NACHOS reste un mini-système).

NACHOS s'exécutant au dessus d'un processeur émulé RISC-V, les programmes utilisateur doivent être compilés en utilisant un compilateur croisé, en l'occurrence ici *gcc*, qui va générer du code RISC-V exécutable sous NACHOS.

1.1.4 Paramétrage du système

Fichier de configuration (fichier nachos.cfg)

Il est possible de modifier certains paramètres de NACHOS et de la machine 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 à lancer au démarrage du système, le formattage et l'initialisation du disque au démarrage, etc... Le format du fichier de configuration est détaillé dans le paragraphe 1.8.6.

1.1.5 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 d'exécution des processus (nombre de cycles), 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.

1.2 La machine émulée (répertoire *machine*)

1.2.1 Le processeur RISC-V (fichiers *machine.cc, machine.h,* instruction.cc, instruction.h)

Le processeur émulé par NACHOS est un processeur RISC-V. Le processeur dispose en interne de registres entiers, de registres flottants, et d'un registre contenant le *program counter* (pc). L'objet global *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* ;

En outre, des constantes, définies dans le fichier machine.h, 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 1.7).

L'objet *g_machine* contient également les champs suivants :

— *mainMemory* : pointeur vers le tableau correspondant à la mémoire de la machine ;
— un pointeur sur les différents composants de la machine : MMU (Memory Management Unit), ACIA (Asynchronous Communication Interface Adapter), contrôleur d'interruptions, disque et console.

1.2.2 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. Le champ *interrupt* de l'objet global *g_machine*, 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.

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

Les adresses de données et d'instructions que le processeur RISC-V 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 1.2.1). La MMU utilise pour cela une *table de traduction d'adresses* (champ *translationTable* de la MMU). 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 1.5.1, page 16.

1.2.4 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 1.8.6). 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 1.3.2, page 12).

1.2.5 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 1.3.1, page 12). 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 indiquée par une interruption.

1.3 Les pilotes de périphériques (répertoire *drivers*)

Les pilotes (drivers) de périphériques permettent au noyau de gérer les périphériques émulés par NACHOS (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.

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

Il existe un pilote permettant d'effectuer des entrées/sorties sur 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 contrôle 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 1.8.1) des routines (*n_printf*, *Write*) permettant d'écrire vers la console.

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

Ce pilote est en charges des entrées/sorties disque. Elle possède trois 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 est 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 synchrones (bloquantes) 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).

1.4 Le noyau de Nachos (répertoire *kernel*)

1.4.1 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 classes de NACHOS sont à la base de son fonctionnement :

- *Process* qui définit la notion de processus (lourd). Un processus rassemble les ressources utilisées par une application, en particulier un ensemble de threads et un espaca 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.

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

La classe *Process* rassemble l'ensemble des ressources utilisées par chaque application :

- Champ *exec_file* : le fichier exécutable qui contient le code de l'application ;
- 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, nombre de défauts de page et d'accès disque, ...).

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 associé.

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

Un thread (processus léger) est un flot d'instructions exécuté par le processeur. Un objet de type *Thread* dans NACHOS contient le 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 émulé et du fait que le noyau s'exécute directement sur la machine hôte, le contexte d'un thread n'est pas strictement limité au contexte de la machine émulée. À la place, son contexte est séparé en deux composantes :

- Le *contexte du thread (thread_context)*, constitué de l'état des registres du processeur RISC-V : *thread_context.int_registers* et *thread_context.float_registers*
- Le *contexte du simulateur (simulator_context)*, constitué de l'état du simulateur RISC-V, 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 réel (non simulé). 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 simu-

lateur, 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 ERROR 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 RISC-V ;
— *void RestoreProcessorState(void)*, qui restaure l'état des registres RISC-V ;
— *void InitSimulatorContext(int8_t *stackAddr, int stackSize)*, qui initialise le contexte du simulateur de l'objet *Thread* ;
— *void InitThreadContext(int64_t initialPCREG,int64_t initialSP, int64_t arg)*, qui initialise le contexte de l'objet *Thread* (pointeur de pile, adresse de la instruction à exécuter, paramètre à passer).

Classe 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, ...). Le format d'exécutable géré par NACHOS et chargé dans le constructeur de la classe *Addrspace* est le format ELF (Executable and Linkable Format) 64 bits.

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

La classe *Scheduler* gère la file des threads prêts (*readyList*) et réalise le changement de contexte entre les threads. Le thread élu (qui s'exécute sur le processeur) 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) doivent être sauvegardés. Ici, le fonctionnement est un peu différent, car les applications s'exécutent sur un processeur émulé (RISC-V) 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 du processeur simulé (RISC-V) et le contexte du simulateur. Vous n'aurez pas à gérer le contexte du simulateur lors des travaux pratiques.

1.4.2 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 mono-processeur, 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 nécessaire ;
— *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 (ôté 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.

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

Les adresses des instructions et des données que le processeur RISC-V manipule sont des adresses *virtuelles* : elles sont relatives à *l'espace d'adressage* courant, qui définit comment traduire ces adresses virtuelles (données, instructions) en adresses physiques (mémoire centrale de la machine, *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 gestionnaire de 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).

1.5.1 Mécanisme de traduction d'adresses

Les paragraphes suivants décrivent les principales classes mettant en œuvre le mécanisme de traduction d'adresses de la machine.

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(uint64_t addr, int size, uint64_t *value)* :
— *addr* : adresse virtuelle de la lecture ;
— *size* : nombre d'octets à lire (1, 2, 4 ou 8) ;
— *value* : valeur lue.
WriteMem(uint64_t addr, int size, uint64_t value) :
— *addr* : adresse virtuelle de l'écriture ;
— *size* : nombre d'octets à écrire (1, 2, 4 ou 8) ;
— *value* : valeur à écrire.
Ces deux fonctions font appel à la méthode :
*ExceptionType Translate(uint32_t virtAddr, uint32_t *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 1.5.1* de la *MMU*). 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 1.5.2) 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 (numéro de page virtuelle hors borne). Dans ce cas, une exception *AddressErrorException* est levée.

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

— *BUSERROR_EXCEPTION* : l'adresse virtuelle à traduire n'était pas alignée correctement ou la traduction a rendu une adresse physique invalide ;
— *ADRESSERROR_EXCEPTION* : aucune traduction d'adresse n'est disponible pour cette adresse virtuelle ;
— *READONLY_EXCEPTION* : une écriture à été tentée sur une page protégée en écriture ;
— *NO_EXCEPTION* : 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 une table des pages *linéaire* (simplement indexée par le numéro de page virtuelle), à *un seul niveau* (sans arborescence de tables), et possédant un nombre maximal de pages virtuelles par processus.

La table des pages 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 1.5.2).

La composition d'une entrée de cette table (classe *PageTableEntry*) est présentée Figure 1.2. L'indexation de la table se fait à l'aide du numéro de page virtuelle.

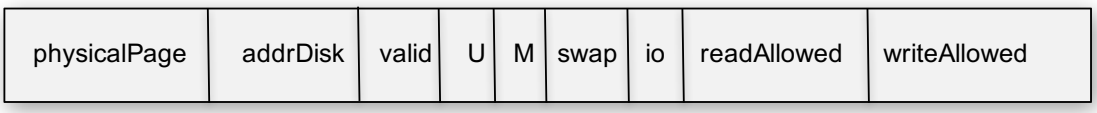

FIGURE 1.2 – Structure d'une entrée de la table des pages virtuelles

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 ;

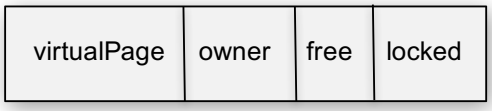

FIGURE 1.3 – Structure d'une entrée dans la table des pages réelles

— le bit *valid* (ou V) 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 mis à jour par la MMU par matériel à chaque accès mémoire et remis à 0 par logiciel (algorithme de remplacement de page et de recopie de pages) ;
— le bit *swap* permet de savoir, lorsque la page n'est pas valide (*valid* à 0), si la page est dans 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. Ce bit est 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.

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(uint64_t virtualPage)* et pour écrire une nouvelle valeur dans *addrDisk*, on utilise la méthode *void setAddrDisk(uint64_t virtualPage, int addrDisk)*.

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. La figure 1.3 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, la verrouiller, changer le processus propriétaire, la déverrouiller ou la libérer.

1.5.2 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. Le principe consiste alors à réquisitionner la page d'un autre processus, à la placer dans la zone de swap 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, aussi appelé algorithme de la seconde chance). 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.
- 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 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(uint64_t 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).
- bit *swap*=0 et *addrDisk*=INVALID_SECTOR : 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 ou à une zone de données non initialisée (section *.bss* du format binaire ELF, voir la section 1.8.3) ;
- bit *swap*=0 et *addrDisk*≠INVALID_SECTOR : 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. Les opérations à effectuer dans les deux autres cas sont les suivantes :

— page “anonyme” : la page est remplie de 0 (page vierge), par exemple en utilisant la fonction de la *libc bzero* ;

— 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 INVALID_SECTOR, 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 (*NO_EXCEPTION* pour indiquer que tout s'est bien déroulé).

1.5.3 Format des fichiers exécutables

Les fichiers sources sont compilés en code RISC-V par un compilateur croisé qui génère des exécutables au format ELF (*Executable and Linking Format*) avec un format d'entête 64 bits. Ce format de fichier est brièvement décrit dans le fichier *elf.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 1.8.5).

1.6 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.

1.6.1 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 ;

— 'f' – système de gestion de fichiers ;

— 'h' – émulation de la machine (périphériques, notamment disque) ;

— 'i' – gestionnaires d'interruptions ;

— 'm' – émulation de la machine ;

— 's' – outils de synchronisation ;

— 't' – threads et processus ;
— 'u' – utilitaires ;
— 'v' – gestion de mémoire virtuelle.

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.

1.6.2 Les listes (fichier *list.h*)

NACHOS propose une structure de listes à simple chaînage, ainsi que les accès classiques aux listes et à leurs éléments. Ces listes sont définies dans la classe générique *List*. Les objets instances de la classe *List* sont les chaînons (classe *ListElement*), assurant la mise en liste des éléments sans se soucier de l'allocation et libération de la mémoire qu'ils occupent (un objet de la classe *ListElement* contient un pointeur sur l'élément mis en liste). Chaque objet de la classe *ListElement* contient également 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 décrits 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

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6-7	t1-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments/return values	Caller
f12-17	fa2-7	FP arguments	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

TABLE 1.1 – Convention d'utilisation des registres

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.

1.7 Fonctionnement des appels système

1.7.1 Fonctionnement global

Les programmes utilisateur ont accès à une liste d'appels système qui sont appelés par les programmes utilisateur en exécutant l'instruction RISC-V *ecall*. 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 le noyau du système NACHOS.

Le *type* de l'appel système est fourni par les programmes utilisateur dans le registre *a7 (x17)* du RISC-V, et les paramètres de l'appel sont stockés automatiquement par le compilateur dans les registres *a0*, *a1*, *a2* et *a3* (*x10* à *x13*) selon leur nombre. La valeur de retour de l'appel, si il y en a une, est placée dans le registre *a0* (*x10*). Le tableau 1.7.1, issu de la documentation RISC-V, précise les conventions d'utilisation des registres :

La liste des appels système supportés par le noyau NACHOS est donnée dans le fichier

syscall.h du répertoire *userlib*, et le code RISC-V réalisant l'appel système se trouve dans le fichier *sys.s* du répertoire *userlib*. Ce fichier est écrit en assembleur, et une fois compilé, est lié statiquement avec les programmes utilisateur.

1.7.2 Exemple de déroulement d'un appel système

Nous détaillons ici le cheminement d'un appel système. 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);
}
```

Le code assembleur corresponsant est le suivant (appel à *Join* seulement) :

```
  ld      a0,-24(s0)
  auipc   ra,0xffc03
  jalr    -1548(ra) #Join
```

Ce code généré pour l'appel à *Join* a récupéré la valeur d'une variable locale (ligne 1) pour la passer en paramère lors de l'appel à l'appel de la fonction *Join* (ligne 3).

Le code de la fonction *Join*, définie dans le fichier *sys.s*, est le suivant. Il se contente d'effectuer un appel système, en mettant le numéro associé dans le registre *a7* et de retourner à l'appelant.

```
Join:
  addi a7,zero,SC_JOIN
  ecall
  jr ra
```

Lors du décodage des instructions de l'application par le processeur RISC-V simulé, l'instruction d'appel système *ecall* 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 :

```
// Recuperation de l'identificateur passe a l'appel systeme
int64_t tid = g_machine->ReadIntRegister(10);
Thread* t = (Thread *)g_object_addrs->SearchObject(tid);

// Appel de la routine qui effectue l'operation Join
g_current_thread->Join(t);

// Écriture du code de retour dans le registre 2
g_machine->WriteIntRegister(10,NO_ERROR);
```

À noter que le code donné ci-dessus est légèrement simplifié (ne gère pas les cas d'erreur, et ne teste pas si le contenu du registre *x10* correspond bien à un thread valide).

A noter dans ce code que pour vérifier les paramaètres passés aux appels système, c'est un identificateur de *Thread* qui est passé à l'appel système. Une structure de données du noyau (*map*, contenue dans la variable globale *g_object_addrs*) fait la correspondance, quand elle existe, entre l'identifiant de *Thread* et le pointeur sur l'objet *Thread* dans le noyau.

1.8 Mon premier programme Nachos

Cette section présente tout d'abord de manière exhaustive 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.

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

L'ensemble des numéros des appels système est déclaré dans le fichier *syscall.h* du répertoire *userlib*. L'ensemble des routines utilisateur correspondant est défini dans le fichier *sys.s* du répertoire *userlib*. Sauf mention contraire, en cas d'échec, chacune de ces primitives système renvoie un résultat de type t_error, valant ERROR (-1) en cas d'erreur ou NO_ERROR (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 au dernier appel système appelé.

Ne pas confondre la routine permettant d'appeler un appel système et la méthode système NACHOS correspondante, qui ne peut pas être appelée directement à partir d'un programme.

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 personnaliser les messages d'erreurs à la convenance de l'utilisateur.

Gestion de processus légers (threads).

— *void Halt()* : arrête NACHOS (shutdown).
— *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)* : crée un nouveau processus pour l'exécutable NACHOS passé en paramètre, et renvoie l'identificateur du thread créé dans ce nouveau processus (ERROR 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 explicitement *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* (bloque tant que le thread n'est pas terminé).
— *void Yield()* : commute du thread courant vers un autre thread prêt s'il y en a (relâchement volontaire du processeur).

Accès aux fichiers.

- *t_error Create(char *name,int size)* : crée le fichier NACHOS *name* de taille *size*.
- *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 *CONSOLE_OUTPUT*, 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 *CONSOLE_INPUT*, alors la chaîne est saisie depuis la console.
- *t_error Close(OpenFileId id)* : ferme le fichier identifié par *id*.
- *t_error Remove(char* name)* : efface le fichier de nom *name*.
- *t_error Mkdir(char* name)* : crée un nouveau répertoire de nom *name*.
- *t_error Rmdir(char* name)* : détruit le répertoire de nom *name*.
- *uint64_t 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 INVALID_ADDRESS en cas d'erreur. *f* est un descripteur de fichier (le fichier doit être ouvert avant l'appel à *Mmap*). La taille demandée, exprimée en octets, 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, ou ERROR en cas d'échec.
- *t_error SemDestroy(SemId sema)* : détruit le sémaphore spécifié par l'identificateur *sema*. Cet appel échoue si un processus est bloqué en attente du sémaphore.
- *t_error V(SemId sema)* : effectue l'opération V sur le sémaphore *sema*.
- *t_error P(SemId sema)* : effectue l'opération P sur le sémaphore *sema*.
- *LockId LockCreate(char *debug_name)* : crée un verrou de nom *debug_name* et retourne son identificateur, ou ERROR en cas d'échec.
- *t_error LockDestroy(LockId id)* : détruit le verrou spécifié par l'identificateur *id*. Cet appel échoue si un processus est bloqué en attente du verrou.
- *intt_error LockAcquire(LockId id)* : acquisition du verrou d'identificateur *id*.
- *t_error LockRelease(LockId id)* : relâche le verrou d'identificateur *id*.
- *CondId CondCreate(char *debug_name)* : crée une nouvelle variable de condition de nom *debug_name*, et retourne son identificateur, ou ERROR en cas d'échec.
- *t_error CondDestroy(CondId id)* : détruit la variable de condition d'identificateur *id*. Cet appel échoue si un processus est bloqué en attente sur la condition.
- *int CondWait(CondId cond)* : met dans l'ensemble d'attente associée de la condition *cond* le thread courant.
- *t_error CondSignal(CondId cond)* : le premier thread bloqué sur la condition est réveillé.
- *t_error CondBroadcast(CondId cond)* : idem mais sur *tous* les threads de l'ensemble d'attente.

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

La mini bibliothèque C de NACHOS est décrite dans les fichiers *libnachos.c* et *libnachos.h*. Ses 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 l'affichage d'un nombre en hexadécimal, %f pour un flottant (%ld, %li, %lx peuvent être utilisés pour les entiers longs). 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). Cette fonction est similaire à la fonction *printf* avec des options de formattage beaucoup plus pauvres.

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 fonctionné. *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 copie a fonctionné.
— *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*.

1.8.3 Compilation d'un programme utilisateur

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

Il est possible d'effectuer cette compilation automatiquement, au moyen du *Makefile* du répertoire *test*, qui contient des règles de compilation pour compiler un source unique en un binaire RISC-V. Pour cela, il suffit d'enregistrer le programme (appelons le *prog.c*) dans ce répertoire, et de modifier la ligne PROGRAMS pour y ajouter la cible *prog* :

```
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*. 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 parties 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 virtuel d'adressage du processus (*VM addr*), et les droits associés (*R* pour lecture, *W* pour écriture, *X* pour exécution [2]). Il indique également l'adresse de la première fonction à exécuter (*Program start address*).

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

1.8.4 Compilation de Nachos

Pour compiler NACHOS, il suffit d'utiliser le fichier *Makefile* qui se trouve à la racine des sources (sauf bien entendu si vous ajoutez de nouveaux fichiers, ce qui n'est pas nécessaire dans le cadre des travaux pratiques, sauf si vous réalisez les exercices bonus). La compilation génère un fichier exécutable nommé *nachos* à la racine de l'arborescence des sources. À la modification d'un fichier d'en-tête ou d'un fichier source, seuls les fichiers qui en dépendent sont recompilés. Pour faire du ménage et effacer tous les résultats des compilations passées, faire *make clean*.

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

L'exécutable du système d'exploitation, qui se nomme *nachos* est paramétrable, via des paramètres fournis dans la ligne de commande, et via un fichier de configuration.

Les paramètres fournis sur la ligne de commande sont les suivants :

-d : permet d'afficher à l'écran les messages de déboguage, dont le type est spécifié. Ainsi `nachos -d fv` affiche ceux concernant le système de fichiers et la mémoire virtuelle. Si ce paramètre est suivi de *+*, ou bien s'il n'est pas suivi d'un type de message, tous les messages de déboguage sont affichés. L'ensemble des types de messages sont définis dans la section 1.6.1, page 20.

-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.

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

Le fichier de configuration de NACHOS permet de modifier des paramètres du système et de l'architecture au lancement de l´exécutable, sans avoir besoin de recompiler le système. Le fichier de configuration par défaut utilisé est le fichier *nachos.cfg*, qui 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. Pour spécifier la valeur d'un paramètre dans le fichier *nachos.cfg*, la syntaxe est la suivante :

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

1.8.6 Fichier de configuration 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 1.5.1).

`UserStackSize = <entier>`

La taille (en octets) de la pile de chaque thread.

`ProcessorFrequency = <entier>`

Précise la fréquence du processeur 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. La taille d'une page doit être identique à la taille d'un secteur.

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 global *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 dans le code du noyau.

Chapitre 2

Sujets de travaux pratiques (18h)

2.1 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 (commentaires inclus) par les directives *ifdef/endif* suivantes :

```
#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
```

Par ailleurs :

— Vous ne devez pas modifier le répertoire *machine*[1] ;

— 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.

Quoi rendre aux enseignants ?

Chacun des deux TPs sera testé en séance en présence de l'enseignant, pour valider (ou pas) son fonctionnement, et montrer à l'enseignant que les membres du binôme maitrisent le code qui a été écrit. Tout TP non validé avec l'enseignant sera considéré comme non fonctionnel.

En complément du code montré en séance, vous devez envoyer à l'enseignant à chaque étape le code source des fichiers modifiés (programmes de test inclus), et un court document, synthétique et honnête, expliquant ce que vous avez réalisé, comment ça a été testé, et le cas échéant ce qui n'a pas pu être terminé et le diagnostic associé.

1. Sauf si vous faites un des exercices bonus nécessitant de modifier le matériel.

Si vous invitez l'enseignant à votre repository *git* ISTIC, utilisez le tag TP-1 et TP-2 pour chacun des deux TP.

2.2 Questionnaire d'exploration de Nachos (travail à la maison)

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, le code source de NACHOS, et les pages *html* générées automatiquement à partir du code source de NACHOS. Un moyen 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 de manière exhaustive, en expliquant, les fichiers et fonctions/méthodes impliqués dans l'exécution d'un programme utilisateur qui nécessite un appel au système :

1. en mode utilisateur (avant d'entrer en mode noyau) ;
2. en mode noyau (quand on s'exécute en mode noyau).

On pourra par exemple regarder le programme utilisateur *test/hello.c*, qui se contente d'afficher un message sur la console.

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 (actif) 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 *type* 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 RISC-V émulée ?
2. Peut-on utiliser l'utilitaire *gdb* pour mettre au point le code de NACHOS ?
3. Peut-on utiliser *gdb* pour mettre au point les programmes utilisateur ?

2.3 TP 1 - Ordonnancement et synchronisation (8 heures encadrées)

Ce TP nécessite d'analyser uniquement le contenu des répertoires *kernel* et *userlib*.

Il est à noter que la NACHOS qui vous est livrée au départ compile mais ne s'exécute pas (encore) correctement. Cela provient du fait que les drivers utilisent les outils de synchronisation, qui ne sont pas encore développés. Il vous sera possible (pour des programmes mono-thread, comme par exemple le programme *hello*) d'exécuter des programmes une fois les outils de synchronisation développés.

2.3.1 Outils de synchronisation

L'objectif ici est d'implanter des outils de synchronisation (voir § 1.4.2). Le travail demandé est le suivant :

1. Compléter le fichier *synch.cc* pour écrire le code des sémaphores, 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. Pour les sémaphores, on choisira la réalisation des sémaphores dans laquelle les compteurs de sémaphores passent en négatif (deuxième réalisation du cours).
2. Créer le code des appels système de synchronisation. Pour ce faire, consulter le paragraphe 1.7 qui décrit le fonctionnement des appels système sous NACHOS. Les fichiers concernés par les appels système sont les suivants :
   — fichier assembleur *sys.s*, qui est lié statiquement avec les programmes utilisateur et utilise l'instruction RISC-V *ecall* pour exécuter un appel système (voir paragraphe 1.7)
   — fichier *exception.cc*, qui centralise tous les appels au noyau et, selon le numéro de l'appel (contenu du registre RISC-V *r10*), 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, que les objets sont du type attendu (champ *type*).

Certains appels système prennent en argument des pointeurs (par exemple des chaînes de caractères). Pour récupérer le contenu d'une chaîne de caractères, on appellera la fonction *GetStringParam* (voir un exemple d'utilisation dans l'appel système *PError*).

2.3.2 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 du thread depuis (resp. vers) le processeur RISC-V émulé.
- La méthode *Start* de la classe Thread, dont l'objectif est d'initialiser tous les éléments d'un thread, 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 pour le simulateur RISC-V. 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 utilisateur et contexte du simulateur). 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 (variable globale *g_thread_to_be_destroyed*) et de l'ôter des différentes structures de données (liste des threads existants, liste des objets) et le bloquer.
- La méthode *SwitchTo* de la classe Scheduler, qui effectue la destruction proprement dite de manière différée quand the thread n'est plus actif. Attention, la méthode *SwitchTo* étant celle qui commute de contexte (en particulier change le pointeur de pile) il est délicat d'utiliser des variables locales dans cette méthode.

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

- des sémaphores (rendez-vous, synchronisation producteur/consommateur, lecteur/rédacteur, client/serveur, etc)
- des verrous et variables de condition.

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

- Ajout d'aun appel système TryP() aux sémaphores (prise du sémaphore quand celà est possible, retour d'un code d'erreur dans le cas contraire)
- Ajout de partage de temps à l'ordonnanceur ;
- Gestion des threads par priorité ;
- Ajout de barrières (aussi appelées rendez-vous) comme outil de synchronisation supplémentaire (on passera en paramètre à la fonction de création de barrière le nombre de threads impliqués) ;
- Ajout de paramètres aux threads créés par *threadCreate* ;
- Ajout de paramètres aux programmes utilisateur (argc et argv).

2.4 TP 2 - Gestion de mémoire virtuelle et de fichiers mappés (10 heures encadrées - dont deux sur fichiers mappés)

L'objectif de ce TP est de réaliser les éléments suivants 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* ;
- des fichiers dits *mappés*, permettant d'accéder aux fichiers via des adresses virtuelles à la place des fonctions de lecture et d'écriture classiques dans les fichiers.

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 1.5. Il est recommandé de tester votre système progressivement, lorsque vous aurez réalisé chacune des trois étapes suivantes.

2.4.1 Espaces d'adressage séparés

Dans la version de NACHOS mise en place lors du premier TP, un seul processus (multi-thread 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, de manière à changer le pointeur sur la table des pages d'un processus à chaque changement de contexte.

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

2.4.2 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 (entre autres) 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*.

À l'issue de cette seconde étape, le premier accès réalisé par un programme déclenchera un défaut de page.

Routine traitement des défauts de pages

La routine de traitement des défauts de page est responsable de l'allocation d'une page physique, de son remplissage, avant de redonner le processeur au thread ayant provoqué le défaut de page.

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 (si ce n'est pas le cas, augmenter la valeur de *NumPhysPages* dans

le fichier de configuration), 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. recherche d'une page libre en mémoire réelle (on utilisera pour cela le gestionnaire de mémoire physique - classe *PhysicalMemManager* du fichier *physMem.cc*)
2. remplissage de cette page. Le remplissage s'effectuera par lecture depuis le disque ou par simple initialisation avec des 0 dans le cas de pages anonymes (voir la section 1.5.2).
3. modifier la table de traduction d'adresses et la table des pages réelles en conséquence.

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*).

2.4.3 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 1.5. Ceci sera réalisé en complétant la méthode *EvictPage* de la classe *PhysicalMemManager*.

2.4.4 Trucs et astuces pour pagination en contexte multi-threads

Nous énumérons ci-dessous quelques cas à gérer lors du test de votre système de pagination à la demande en contexte multi-thread :

— Un processus P1, qui recopie une page réelle x sur disque perd le processeur lors de l'entrée/sortie disque. Pour gérer ce cas, il est nécessaire de gérer correctement le bit *locked*, pour éviter que la page x soit réquisitionnée avant la fin de la recopie.
— 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.

2.4.5 Introduction de fichiers mappés

Nous vous demandons ici de mettre en œuvre des fichiers mappés (voir le cours pour une description du principe de fonctionnement). On procédera de la manière suivante :

— Remplissage d'une nouvelle méthode nommée *Mmap* dans la classe *AddrSpace* pour réaliser l'appel système (déjà géré dans le fichier *exception.cc*). 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 : l'adresse virtuelle de début du fichier mappé, la taille associée en octets, le descripteur de fichier correspondant (*OpenFile**).

— Écrire 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*).

2.4.6 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.
— Mise en place de segments de mémoire partagés par plusieurs processus.
— Mise en place d'un allocateur dynamique de mémoire (*malloc*).

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 *RISC-V*), mais ça permet s'exécuter les programmes existants récupérés avec l'archive Nachos. Pour compiler les programmes utilisateur, il faut installer un compilateur croisé RISC-V.
— **Comment installer un compilateur croisé RISC-V sur ma machine Linux ?**
— A la connaissance des enseignants, le paquet d'installation automatique du compilateur RISC-V sur Ubuntu n'est pas fonctionnel. Il faut donc installer le compilateur croisé depuis les sources (lire le documentation sur *https ://github.com/riscv-collab/riscv-gnu-toolchain*), soit sur Ubuntu les commandes suivantes (avec /opt/riscv le répertoire d'installation) :

```
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
./configure --prefix=/opt/riscv --with-arch=rv64im --with-abi=lp64
make
```

— **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 jusqu'à la version 14.6. 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.