Carole PLASSON
Jeremy WAMBECKE
Jeremy SEGUIN
Gregory CANO
Michael OMER
Groupe B

Projet système 2014

Livret utilisateur

Contenu

Introduction :

NachOS est un système d'exploitation multi tâches offrant une interface d'entrées/sorties en ligne de commande, permettant aux programmes d'interagir avec l'utilisateur. Il comporte un système de fichiers permettant de gérer une arborescence de fichiers. Il est également possible de faire communiquer deux machines entre elles via le réseau.

1. Fonctionnalités

1.1 Noyau

Le système permet de créer des threads partageant la même mémoire mais ayant chacun leur pile d'exécution. Chaque thread exécute une fonction donnée qui peut prendre un argument. Il est possible de les synchroniser à l'aide de sémaphores pour réaliser des attentes efficaces ou des sections critiques. Grâce à son implémentation sans hiérarchie, un thread peut attendre la terminaison de tout autre thread et récupérer son code de terminaison défini par l'utilisateur.

Ce système supporte l'exécution simultanée de plusieurs processus de façon indépendante via une gestion de mémoire virtuelle. Ainsi, une erreur dans un programme ne peut altérer l'intégrité du système. Un processus a la possibilité d'en créer un autre exécutant un programme donné. Comme il n'existe pas de hiérarchie entre processus, l'un d'entre eux peut attendre la terminaison d'un processus précis. De plus, la mémoire d'un processus est gérée de manière transparente par le noyau mais il est possible d'en allouer une partie supplémentaire sur demande dans un programme.

Le système de fichier de NachOS s'organise en hiérarchie de répertoires, dans laquelle il est possible de se déplacer ou manipuler des éléments. Ce système gère les chemins de fichiers, qu'ils soient absolus ou relatifs.

Le réseau de NachOS permet de transmettre des informations entre plusieurs machines et ainsi de faire communiquer des programmes situés sur des machines différentes.

1.2 Espace utilisateur

Le système intègre un Shell permettant de lancer simplement des programmes utilisateur et d'accéder aux fonctionnalités du système.

Afin de réaliser des programmes fonctionnant sur NachOS, il existe plusieurs bibliothèques de fonctions prêtes à l'emploi. Des fonctions d'entrées-sorties formatées permettent d'interagir avec l'utilisateur, tandis que d'autres permettent la manipulation de chaînes de caractères, y compris des conversions entre entier et chaîne et inversement. Enfin si vous voulez réaliser des programmes communiquant entre eux à l'aide du réseau, vous pouvez utiliser une bibliothèque spécialement conçue dans cette optique.

2. Spécifications

2.1 Appels système

2.1.1 Threads

Les threads NachOS sont entièrement gérés par le noyau. Lors de sa création, un thread se voit attribuer une pile de 2048 octets ainsi qu'un identifiant unique, le TID. Celui-ci n'est pas réutilisé au sein d'un même processus, le nombre de threads pouvant être lancés lors de l'exécution d'un programme est limité à INT_MAX. S'il n'y a plus d'espace disponible (dans la mémoire virtuelle du processus ou en mémoire physique) le thread ne peut être créé. Un thread se termine avec un appel à UserThreadExit ou lorsqu'il termine l'exécution de la fonction qui lui a été assignée. Si un thread ne fait pas d'appel à UserThreadExit pour se terminer, sa valeur de retour sera 0, sinon elle vaudra l'argument de UserThreadExit.

- **int UserThreadCreate(void f(void *arg), void *arg)**

Crée un nouveau thread pour le processus en cours d'exécution. Le thread exécutera la fonction *f* en lui passant en paramètre *arg*, cette exécution n'est pas nécessairement immédiate. *arg* vaut 0 en cas d'absence de paramètre. Renvoie -1 s'il n'y a plus de ressources disponibles pour créer le thread ou si l'adresse de f ou de arg correspond à une adresse invalide. Autrement la création du thread est réussie et la fonction renvoie son TID.

- **int GetTid()**

Renvoie le TID du thread appelant.

- **void UserThreadExit(int code)**

Termine le thread appelant. Cette fonction ne retourne jamais, le thread est toujours détruit. L'entier passé en paramètre correspond à la valeur de terminaison du thread et peut être récupéré grâce à un appel à UserThreadJoin dans un autre thread du même processus. Lorsque le dernier thread en cours d'exécution d'un processus appelle cette fonction, le processus se termine.

- **int UserThreadJoin(int tid, int* ptr_return)**

Le thread courant attend la terminaison du thread situé dans le même processus dont le tid est passé en argument. Si ce dernier est déjà terminé, la fonction retourne immédiatement.

Le code de terminaison du thread attendu, qui correspond à l'argument passé à UserThreadExit, est placé à l'emplacement mémoire ptr_return si ce dernier ne vaut pas 0. Renvoie -1 dans le cas d'un tid non attribué à un thread du processus ou si le thread est déjà attendu, 0 sinon.

2.1.2 Processus

Un processus est identifié par son PID, qui est unique et non réutilisé par le système. De plus, le PID du premier processus lancé par NachOS vaut 0. Le nombre de processus pouvant être lancés lors d'une exécution de NachOS est donc limité à la valeur INT_MAX. Leur création dépend de la mémoire physique disponible sur la machine et échoue si celle-ci est insuffisante. Un processus se termine lorsqu'il exécute une instruction return dans la fonction main ou qu'un de ses threads appelle la fonction Exit. Le programme ne se termine que lorsque tous ses threads se sont terminés.

- **int ForkExec(char* executable)**

Cette fonction permet la création d'un processus exécutant un programme donné. La chaîne de caractères *executable* correspond au chemin du fichier contenant le programme. Renvoie le PID du processus si la création a réussi, ou -1 en cas d'echec.

- **void Exit(int returnCode)**

Termine le processus appelant en affichant le code de retour passé en paramètre. Cette fonction ne retourne jamais.

- **void* Mmap(int length)**

Permet d'allouer une portion de mémoire virtuelle d'un processus. L'argument *length* représente la taille de mémoire nécessaire en octets. Renvoie un pointeur sur le début de la mémoire allouée ou 0 en cas d'échec d'allocation.

- **int Unmap(void* addr)**

Permet de désallouer une portion de mémoire virtuelle obtenue par un appel à Mmap. *addr* correspond au premier octet de celle-ci.

- **int GetPid()**

Renvoie le PID du processus courant.

- **int WaitPid (int pid)**

Fonction permettant au processus courant d'attendre la terminaison du processus identifié par *pid*. Un processus ne peut pas s'attendre lui-même, mais peut attendre n'importe quel autre processus. Renvoie le PID du processus terminé si réussite ou -1 si le pid est celui du processus courant, si le PID n'est pas valide ou s'il est déjà attendu.

- **int GetListProcess(int* list)**

Écrit à l'adresse *list* les PID des processus actifs. *list* doit être alloué suivant le nombre de processus récupéré avec la fonction GetNbProcess() et multiplié par la taille d'un int.

- **int GetNbProcess()**

Renvoie le nombre de processus en cours d'exécution.

2.1.3 Entrées / Sorties

- **void PutChar(char c)**

Écrit un caractère sur la sortie standard.

- **int GetChar()**

Fonction récupérant un caractère sur l'entrée standard. Renvoie un entier, celui correspondant au caractère lu ou -1 si une fin de fichier est détectée.

- **int PutString(char* s)**

Écrit sur la sortie standard la chaîne de caractères située à l'adresse *s*. L'écriture est limitée à *MAX_STRING_SIZE* caractères. Renvoie le nombre de caractères écrits, ou -1 si l'adresse de la chaîne n'est pas valide.

- **int GetString(char* s, int n)**

Récupère sur l'entrée standard une chaîne de caractères de taille inférieure ou égale à *n* et la stocke à l'adresse *s*. Cette fonction s'arrête en cas de saut de ligne ou de fin de fichier. En cas de saut de ligne, ce caractère est également écrit. Le caractère '\0' est écrit à la fin de la chaîne. Renvoie le nombre de caractères lus (sans le '\0') ou -1 si l'adresse de la chaîne à écrire n'est pas valide.

- **int PutInt(int i)**

Écrit l'entier *i* sur l'entrée standard. Renvoie le nombre de caractère écrits.

- **int Random(int nbMax)**

Renvoie un nombre choisi aléatoirement dans l'intervalle [0; nbMax[

2.1.4 Sémaphores

- **int SemInit(sem_t * adrSem, int initValue)**

Fonction initialisant le sémaphore situé à l'adresse *adrSem* à la valeur initiale *initValue*.
Chaque sémaphore possède un identifiant unique, compris entre 0 et INT_MAX. Au plus INT_MAX sémaphores peuvent donc être créés pour un processus. Renvoie 0 si la création a réussi, -1 si INT_MAX sémaphores ont déjà été créés ou si adrSem est invalide.

- **int SemWait(sem_t *adr)**

Fonction permettant au sémaphore situé à l'adresse *adr* d'ajouter le thread courant dans la liste d'attente de la ressource. Tant que la valeur du sémaphore est inférieure ou égale à 0, l'appel est bloquant. Sinon la fonction retourne immédiatement et la valeur du sémaphore est décrémentée. Renvoie 0 si l'appel a réussi ou -1 si l'adresse du sémaphore est invalide.

- **int SemPost(sem_t *adr)**

Fonction incrémentant la valeur du sémaphore et permettant à un thread en attente de se réveiller. Renvoie 0 si l'appel a réussi ou -1 si l'adresse du sémaphore est invalide.

- **int SemDestroy(sem_t *adr)**

Fonction permettant la destruction du sémaphore situé à l'adresse *adr*. Renvoie 0 si l'appel a réussi, -1 si l'adresse du sémaphore est invalide.

2.1.5 Arguments

- **arg_list arg_start()**

Crée et initialise la liste des arguments variables de la fonction appelante. Permet de gérer des arguments dont le nombre est variable. La fonction doit avoir au moins un premier argument fixe, celui-ci ne sera pas présent dans la liste. La fonction ne peut pas avoir plus de 4 arguments. Renvoie la liste des arguments de la fonction.

Ex : f (int n, …) "..." peut-être 0, 1, 2 ou 3 arguments.

- **int arg_arg(arg_list arg)**

Renvoie le prochain argument dans la liste *arg*.

Restriction : pas de contrôle sur le nombre d'appels, si plus d'appels que le nombre de paramètres sont effectués, le comportement sera inconnu.

- **int arg_end(arg_list arg)**

Detruit la liste *arg*. Renvoie 0 en cas de réussite, -1 sinon.

2.1.6 Système de fichiers

- **int cd(char* path)**
Si *path* correspond à un chemin de répertoire valide, celui-ci devient le répertoire courant. Renvoie 0 en cas de réussite, -1 sinon.

- **char* pwd()**
Copie au plus 128 caractères du chemin du répertoire courant dans *buffer,* qui doit être alloué à 128 caractères.

- **int mkdir(const char *path)**
Crée un nouveau répertoire vide dont le chemin d'accès est *path*. Le dernier nom du path est alors le nom de ce nouveau répertoire. Les répertoires précédant ce nom doivent exister. Renvoie 0 en cas de succès, -1 si le *path* (sans le nom du répertoire) est invalide ou si le répertoire existe déjà.

- **void ls()**
Affiche le contenu du répertoire courant.

- **int rmdir(char *path)**
Supprime le dossier dont le chemin d'accès est *path*. Renvoie 0 en cas de succès, -1 si le repertoire n'est pas vide ou si le chemin d'accès n'est pas valide.

- **int pathExists(const char* path)**
Vérifie si le chemin *path* est valide. Renvoie 1 si path existe, 0 s'il ne l'est pas et -1 en cas d'erreur.

- **int rm(char* path)**
Supprime le fichier de chemin d'accès *path*. Renvoie 0 en cas de succès, -1 sinon.

- **int Create(const char *path, int initialSize)**
Crée un fichier de taille *initialSize* et de nom *path*. Renvoie 0 en cas de succès, -1 sinon.

- **int Cat(char *name)**
Affiche le contenu du fichier de chemin *name*. Renvoie 0 en cas de succès, -1 sinon.

- **OpenFileId Open (char *name);**
Permet l'ouverture en lecture et en écriture du fichier dont le chemin est *name*. Renvoie l'identifiant du fichier, ou -1 si le fichier n'existe pas ou que le processus tente d'ouvrir deux fois le même fichier.

- **int Close (OpenFileId id)**
Permet la fermeture du fichier dont l'identifiant est passé en paramètre. Renvoie 0 en cas de réussite, -1 si le fichier n'existe pas ou a déjà été fermé.

- **int Write (char *buffer, int size, OpenFileId id)**
Permet l'écriture d'au plus *size* caractères du *buffer* dans le fichier dont l'identifiant est passé en paramètre. Renvoie le nombre de caractères écrits, ou -1 en cas d'erreur.

- **int Read (char *buffer, int size, OpenFileId id)**
Lit au plus *size* caractères du fichier d'identifiant *id* puis les stockent dans *buffer.* La lecture se fait à partir du curseur associé à ce fichier. Renvoie le nombre de caractère lus, ou -1 en cas d'erreur.

- **int Seek(int position, int idFile)**
Modifie la position du curseur dans le fichier d'identifiant *idFile.* Renvoie 0 si le déplacement a pu être effectué, -1 si le fichier n'est pas ouvert.

- **void copy(const char *from, const char *to)**
Copie le fichier UNIX de chemin *from* dans NachOS avec le chemin *to.*

2.1.7 Réseau

- **void Sleep(int tempo)**
  Endort le thread d'une durée de *tempo* secondes.

- **int IniSocket(int to)**
  Permet d'initialiser le socket réseau. Renvoie le numéro du socket créé, -1 en cas d'erreur.

- **int Send(sock_t socket, char* message)**
  Envoie la chaîne de caractères *message* sur le réseau en utilisant le socket donné en paramètre. Retourne le nombre de caractères envoyés, -1 si le socket n'est pas valide.

- **int Receive(sock_t socket, char* message)**
  Attend la réception d'une chaîne de caractères depuis le réseau en utilisant le socket donné en paramètre. La chaîne de caractères est stockée dans *message*. Retourne le nombre de caractères reçus, -1 si le socket n'est pas valide.

- **int GetHostname()**
  Renvoie l'adresse réseau de la machine.

2.2 Bibliothèques utilisateur

2.2.1 Mémoire

- **void* malloc(size_t taille)**
  Alloue *taille* octets de mémoire. Renvoie l'adresse de la partie allouée ou 0 si l'allocation échoue.

- **void* realloc(void *ptr, size_t taille)**
  Permet de changer la taille de la zone mémoire pointée par *ptr* obtenue par une allocation. *taille* représente la nouvelle taille de la zone. Les données présentes dans la zone réallouée sont conservées. Renvoie l'adresse de la nouvelle zone mémoire ou 0 en cas d'échec de l'allocation.

- **void* calloc(size_t count, size_t size)**
  Alloue de la mémoire pour un vecteur de *count* éléments de *size* octets. Renvoie un pointeur sur la zone allouée ou 0 si l'allocation échoue.

- **void free(void* ptr)**
  Libère la zone mémoire d'adresse *ptr* précédemment allouée à une variable.

2.2.2 Entrée/sorties formatées

- **void Printf(char* message, ...)**
  Affiche le message contenant jusqu'à trois variables. Avec un maximum de 256 caractères(*MAX_LENGTH*). Les arguments possibles pour *message* sont %d %i (int), %c (char) et %s (char*).

- **void Scanf(char* typeVariable, ...)**
  Récupère jusqu'à trois variables entrées dans la console. Avec un maximum de 256 caractères (*MAX_LENGTH*). Les arguments possibles pour *typeVariable* sont %d %i (entiers), %c (char) et %s (char*).

2.2.3 Chaîne de caractères

- **char* StrCpy(char* source, char* destination)**
  Copie la chaîne *source* à l'adresse mémoire *destination*. Renvoie l'adresse de la chaîne *destination*.

- **char* StrNCpy(char* source, char* destination, int taille)**
  Copie les *taille* premiers caractères de la chaîne *source* à l'adresse mémoire *destination*. Renvoie l'adresse de la chaîne *destination*.

- **char* StrNDCpy(char* source,char* destination, int debut, int taille)**

Copie les *taille* caractères de la chaîne *source* à partir de la position *debut*, à l'adresse *destination*. Renvoie l'adresse de la chaîne *destination*.

- **int StrCmp(char* element1,char* element2)**
  Renvoie 1 si les deux chaînes passées en paramètre sont identiques, -1 sinon.

- **int ChrCmp(char element1,char element2)**
  Renvoie 1 si les deux caractères passés en paramètre sont identiques, -1 sinon.
- **int StrLen (char * chaîne)**
  Renvoie le nombre de caractères présents dans la chaîne passée en paramètre sans compter le caractère de fin de chaîne.

- **char* StrCat(char *chaine1,char *chaine2)**
  Concatène *chaine1* à la fin de *chaine2*. Renvoie *chaine2*.

- **char* StrNCat(char * chaine1, char* chaine2, int taille)**
  Concatène au plus *taille* caractères de *chaine1* à la fin de *chaine2*. Renvoie *chaine2*.

- **char StrChr(char* chaine, char c)**
  Si *c*'est présent dans la chaîne de caractères, cette fonction renvoie son adresse dans la chaîne. Sinon, elle renvoie 0.

2.2.4 Utilitaires

- **int Atoi(char* element)**
  Transforme la chaîne de caractères *element* en entier. Si la chaîne contient des caractères qui ne sont pas des chiffres, le comportement est indéfini.

- **char* Itoa(int element)**
  Transforme l'entier *element* en chaîne de caractères. La chaîne est allouée dynamiquement par la fonction et doit être libérée.

2.2.5 Réseau

- **int creerSocket(int to)**
  Initialise la socket réseau avec la machine cible d'adresse *to*. Renvoie le numéro du socket créé ou -1 si le nombre socket dépasse INT_MAX.

- **int envoyerMessage(sock_t socket, char* message)**
  Envoie la chaine de caractères *message* à la machine liée par *socket*. Renvoie le nombre de caractères émis ou -1 si la socket n'existe pas ou si *message* n'est pas une adresse valide.

- **int recevoirMessage(sock_t socket, char* message)**
  Attend une chaine de caractères venant de la machine liée par *socket* et la stocke dans *message*. Renvoie le nombre de caractères reçus ou -1 si la socket n'existe pas ou si *message* n'est pas une adresse valide.

3. Tests utilisateurs

3.1 Etape 3

semProdConso : Création de 2 threads qui utilisent le principe producteur/consommateur avec une synchronisation à base de sémaphores utilisateur. Permet de tester ces sémaphores.
semSupporter : Création de plusieurs threads représentant des supporters de 2 équipes. Ces supporters affichent un message un certain nombre de fois. On utilise une synchronisation pour l'afficher sans entrelacement. Inspiré d'un TP.
testJoinAttenteRec : Le thread main crée 3 threads : h, g, f. Le main attend f, qui attend g, qui attend lui-même h.
esMulti : Crée des threads effectuant des affichages, pour tester la synchronisation des entrées/sorties.

3.2 Etape 4

userProcMulti : Ce programme permet de lancer 120 processus qui exécutent le programme userThreadMulti
userThreadMulti : Ce programme crée 10 threads qui affichent abcd.
waitPidAll : Ce programme teste tous les cas d'executions/d'erreurs de WaitPid
StackOverFlowMulti : Ce programme crée un processus exécutant le programme userThreadMulti puis lance un thread qui entraîne l'erreur "StackOverFlow". On remarque que l'autre processus n'est pas affecté par l'erreur et poursuit son exécution normalement.

3.3 Bibliothèque utilisateur

test_lib : Récupère une chaîne au clavier, la convertit en entier et l'affiche. Convertit ensuite l'entier 42 en chaïne de caractères et l'affiche. Permet de tester *Atoi*, *Itoa*, *Printf*, *Malloc* et *Free*.

3.4 Système de fichiers

testFile : Teste les fonctions de base applicables aux fichiers. (Création, ouverture, lecture, écriture et fermeture).
pendu : Implémentation du jeu du pendu, inspiré d'un exercice du site openClassRoom (*http://fr.openclassrooms.com/*).. Utilise les fonctions pour le système de fichiers (*open*, *read*, *close*, *Seek*), les fonctions d'entrées/sorties (*Printf*, *Scanf*, *PutString*, *PutChar*), les fonctions *malloc*, *free* et *random*.

3.5 Réseau

./nachos -co : (exécuter NachOS avec l'option -co) : Ce programme exécute un schéma de lecteur/rédacteur pour tester le fonctionnement des conditions noyaux.

InetAnneau : Ce programme nécessite les options -m et -o définissant le numéro de l'hôte (-m) et de cible (-o)et doit être exécuté en parallèle sur plusieurs machines distinctes. Il transmet un message sur plusieurs et s'arrête lorsqu'il revient à la machine source.

4. Implémentation

4.1 Détection de fin de fichier

La console a été modifiée pour que l'appel système *getchar* puisse renvoyer un entier et détecter la fin de fichier sans confondre avec le caractère de valeur 255. La console ne gère désormais que des entiers. En cas de fin de fichier, -1 est alors renvoyé à la place du caractère de valeur 255.

4.2 Traitement d'erreur

Lorsqu'un processus lève une exception, il faut l'arrêter sans qu'il puisse affecter les autres. Tous ses threads doivent être terminés, un appel à Exit ne suffit donc pas car il ne termine pas les autres threads du processus. Nous avons donc ajouté une fonction *KillProcess* permettant de terminer un processus sans affecter les autres. Cette fonction libère les ressources du processus puis retire de la liste de l'ordonnanceur tous ses threads en cours d'exécution. Le thread actuel du processus passe alors la main ou, si le processus était le dernier en cours d'exécution, le noyau est arrêté.

Lorsqu'un appel système lève une exception, il doit renvoyer -1 mais le programme ne doit pas s'arrêter. Pour cela, nous avons ajouté un booléen à l'objet Thread (isSyscall dans thread.cc) permettant de savoir qu'on exécutait un appel système lorsqu'une exception est levée. Elle est alors ignorée et la fonction ayant provoqué l'exception (par exemple ReadMem de Machine) peut alors renvoyer false et l'appel système -1.

4.3 Processus

Une classe Process a été ajoutée pour gérer les informations d'un processus, y compris son *AddrSpace* qui gère son espace d'adressage. Elle contient également un gestionnaire de threads, qui contient tous les threads créés par le processus, ainsi que la liste des sémaphores utilisateur créés au sein du processus.

4.3.1 Gestion de mémoire

Dans l'espace d'adressage d'un processus, le code et les données d'un programme sont chargées en mémoire à partir de la seconde page, la première page étant interdite pour détecter plus facilement des fautes de pages. La pile du thread principal, le premier créé, est placée à la suite du programme.

Un allocateur de mémoire virtuelle, utilisant la méthode d'allocation first fit, a été implémenté et permet d'allouer des portions de l'espace d'adressage du processus. Ce système permet de gérer au mieux l'allocation de la mémoire virtuelle des processus en considérant à la fois les demandes de mémoire de l'utilisateur et les piles des threads créés. Une page interdite peut également être placée au début de la zone allouée pour détecter un débordement de pile.

Une fonction *map* a été implémentée dans *AddrSpace* pour associer une partie de la mémoire virtuelle du processus à de la mémoire physique. Cette fonction utilise l'allocateur de mémoire virtuelle et permet à l'utilisateur d'allouer des portions de mémoire virtuelle sur demande, et à la bibliothèque *malloc* de créer le tas.

4.3.2 Création de processus

AddrSpace a été modifié, le chargement du code en mémoire n'est plus effectué dans le constructeur mais dans une fonction annexe, *loadInitialSections*, afin de pouvoir tester la réussite du chargement. Le nombre de processus est limité par le nombre de frames physiques pouvant être allouées, un nouveau processus ne pouvant être créé uniquement s'il reste assez de frames pour placer son code, ses données et la pile du thread principal.

Une fois créé, le thread principal appelle la fonction *UserStartProcess* lorsqu'il prend la main la première fois. Cette fonction initialise les registres de la machine pour le nouveau processus et lance l'exécution du programme associé.

4.3.3 Récupération de la valeur de retour du *main*

Dans le fichier *start.S*, la fonction *__start* correspond au point d'entrée du programme et redirige vers la fonction main. Comme c'est *__start* qui appelle main, à la fin de l'exécution de cette dernière la fin de *__start* sera exécutée, si un appel à *Exit* n'a pas été fait explicitement dans le programme.

En exécutant l'instruction *return*, le compilateur place la valeur de retour dans le registre r2. Il suffit alors de placer cette valeur dans r4 et d'appeler *Exit*, ce qui équivaut à l'appeler avec la valeur de retour du *main* en paramètre. *Exit* est ainsi appelé dans tous les cas, explicitement ou implicitement, avec la valeur de retour du programme.

4.3.4 Attente de processus

Un processus peut attendre n'importe quel processus sauf lui-même. L'attente est réalisée à l'aide de sémaphores : chaque processus créé possède un sémaphore initialisé à 0 qu'il relâche à sa terminaison. Dans le WaitPid(), le processus qui l'a appelé essaie de prendre ce sémaphore et est donc bloqué jusqu'à ce qu'il soit terminé. L'attente est donc passive et ne consomme pas de ressources.

4.4 Threads

4.4.1 Création de threads

Lorsqu'il est créé, un thread copie la référence du processus du thread actuel, il appartiendra ainsi à ce processus et pourra accéder à ses informations.

Lorsqu'il prend la main pour la première fois, le nouveau thread créé exécute la fonction *StartUserThread.* Celle-ci commence par initialiser les registres du thread. Le registre PC est initialisé avec l'adresse de *__startThread*, point d'entrée des threads. Le pointeur de pile est initialisé avec l'adresse de pile récupérée au moment de la création. Pour pouvoir récupérer les adresses de la fonction f et de son argument dans *__startThread*, on les copie dans les registres R26 et R27, qui sont dédiés au système d'exploitation d'après la documentation MIPS.

4.4.2 Terminaison automatique d'un thread

De même que pour l'appel à exit après le main de la fonction *__start* de Start.s, on crée une fonction *__startThread* dans Start.s qui sera le point d'entrée des threads. Cette fonction est placée à l'adresse 12, car elle est située à la suite de *__start* qui comporte 3 instructions. La fonction *__startThread* copie alors l'argument, contenu dans R27, dans R4 et saute à l'adresse contenue dans R26 correspondant au début de la fonction à exécuter. Au retour de celle-ci, *UserThreadExit* sera exécutée si elle n'est pas appelée explicitement auparavant.

4.4.3 Attente d'un thread à l'aide de la fonction UserThreadJoin

Pour pouvoir mettre en place l'appel système *UserThreadJoin*, il faut conserver le thread jusqu'à la fin du programme. Pour cela, lors de la terminaison d'un thread, l'objet de type Thread associé n'est pas supprimé immédiatement comme cela est fait de base dans *NachOS*. Ses ressources (par exemple sa pile) sont toutefois libérées.

Un sémaphore *s_join* est ajouté à chaque thread. Lors de la création d'un thread, ce sémaphore est bloqué etest relâché lors de sa terminaison. De cette manière, lors de l'appel à la fonction *join(tid, ptr_return)*, un parcours de la liste des threads du processus est réalisé afin de trouver le thread ayant le TID recherché comme identifiant. Une fois trouvé, celui qui l'attend appelle la fonction P() sur le sémaphore du thread récupéré, ce qui a pour effet de le bloquer jusqu'à ce que le thread se termine et le relâche.
La variable *thread_return* a aussi dû être ajouté dans thread.h afin de stocker le code de retour d'un thread. Si *ptr_return* n'est pas null, c'est la valeur de cette variable qu'il contiendra.

4.5 Sémaphores utilisateurs

Pour créer un sémaphore utilisateur, il faut utiliser l'appel système *SemInit* qui prend une adresse de *sem_t* en paramètre. *do_SemInit* demande la création d'un sémaphore noyau qui est lié par un identifiant unique au *sem_t*. L'utilisateur se sert ainsi de ce sémaphore de type *sem_t* permettant de faire le lien avec le sémaphore noyau. Les appels systèmes SemWait et SemPost correspondent respectivement aux fonctions P() et V() des sémaphores noyaux, ils demandent une adresse de *sem_t* et s'en servent pour récupérer l'identifiant du sémaphore dans la mémoire du MIPS.

4.6 Conditions

Une condition contient une liste de sémaphores qui va permettre d'endormir et de réveiller les threads appelants. Quand un thread appelle Wait(Lock*), il doit avoir pris le lock qu'il donne en paramètre. Cette fonction crée un sémaphore initialisé à 0, l'ajoute dans sa liste de sémaphore et le verrouille pour attendre. Lors d'un appel à Signal(Lock*), un sémaphore de la liste est relâché et le thread associé peut continuer son exécution. Broadcast(Lock*) fait la même chose que Signal() mais pour tous les sémaphores de la liste.

4.7 Système de fichiers

4.7.1 Nom des chemins

Les noms des chemins contenant des caractères autres qu'alphanumériques (a-z, A-Z et 0-9) et certains caractères spéciaux (. / _ et -) seront considérés comme invalides. De plus les noms de dossier/fichier ne contenant que des '.' sont interdits, seuls les dossiers . et .. sont autorisés. Il en va de même pour les noms vides.

4.7.2 Manipulation

Le parcours du système de fichier et l'affichage du contenu des dossiers se fait à l'aide de fonctions système à partir d'un dossier courant. Toutes les actions sont faites à partir du dossier courant sauf si un path donné en précise un autre. Des fonctions système permettent de créer, supprimer ou vérifier l'existence de fichiers et de dossiers.

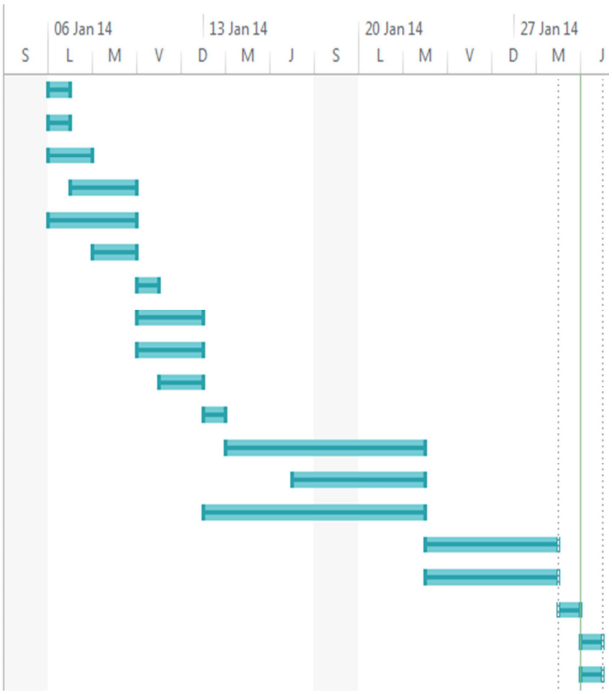
4.7.3 Niveau d'indirection

Nous avons rajouté un niveau d'indirection dans les en-têtes de fichiers. Chaque en-tête possède un tableau contenant les numéros des secteurs d'indirections. Ces secteurs d'indirection contiennent le nombre de secteurs alloués pour les données du fichier suivi des numéros de ces secteurs.

Ce système d'indirection simple permet d'avoir des fichiers de plus grande taille (120ko).

Organisation du travail

Nom de la tâche	Durée	Début	Fin	Préd
Organisation du groupe	1 jour	Lun 06/01/14	Lun 06/01/14	
Installation des outils, nachos et compilation	1 jour	Lun 06/01/14	Lun 06/01/14	
Etape1	2 jours	Lun 06/01/14	Mar 07/01/14	
Etape2	3 jours	Mar 07/01/14	Jeu 09/01/14	
rapport etapes 1 et 2	4 jours	Lun 06/01/14	Jeu 09/01/14	
réalisation des tests etape 2	2 jours	Mer 08/01/14	Jeu 09/01/14	
Réunion pour la gestion des threads	1 jour	Ven 10/01/14	Ven 10/01/14	
Etape3	2 jours	Ven 10/01/14	Dim 12/01/14	
Rapport etape 3	2 jours	Ven 10/01/14	Dim 12/01/14	
Réalisation des tests etape 3	2 jours	Sam 11/01/14	Dim 12/01/14	
Réunion pour préparer l'etape 4	1 jour	Lun 13/01/14	Lun 13/01/14	
Etape 4	7 jours	Mar 14/01/14	Mer 22/01/14	
réalisation des tests etape 4	4 jours	Ven 17/01/14	Mer 22/01/14	
Rapport etape 4	8 jours	Lun 13/01/14	Mer 22/01/14	
Etape Système de fichiers	4 jours	Jeu 23/01/14	Mar 28/01/14	
Etape Réseau	4 jours	Jeu 23/01/14	Mar 28/01/14	
Synthèse du rapport final	1 jour	Mer 29/01/14	Mer 29/01/14	
Fusion finale du projet	1 jour	Jeu 30/01/14	Jeu 30/01/14	
Réalisation de la soutenance et entrainement	1 jour	Jeu 30/01/14	Jeu 30/01/14	



Planning 1

Nous avons effectué des réunions en groupe au début de chaque étape, afin d'avoir les points de vue de chacun et d'en faire une synthèse. Le travail s'est effectué dans la même pièce avec une mise au point, chaque jour, matin et soir. Nous avons tenu un log afin de suivre l'avancement du projet. Le rapport et les tests ont été réalisés au fur et à mesure, une partie des tests ayant été effectuée avant l'implémentation. Les parties 1 à 4 ont été effectuées en groupe pour une bonne compréhension du système NachOS. Les parties système de fichiers et réseau ont quant à elles été effectuées en parallèle, ainsi trois personnes ont travaillé sur le système de fichiers et les deux autres sur le réseau.

Nous avons rencontré quelques problèmes avec la partie 4. Notre choix de ne terminer que le processus provoquant une erreur nous a pris plus de temps que prévu. En effet, l'ordonnancement implémenté de base dans NachOS nous a posé des problèmes de compréhension et a rendu difficile le debug. De plus, impémenter une gestion de mémoire virtuelle robuste et efficace a été longue et compliquée à mettre en place.

Pour le système de fichiers, nous avons manqué de temps pour mettre en place la gestion des accès concurrents aux fichiers. Le système implémenté rencontre des problèmes lors de l'accès de plusieurs threads. Lancer un exécutable multi-threadé ou créer un processus via le système de fichiers n'est pas possible, nous aurions voulu avoir le temps de corriger ce problème.

Pour le réseau nous avons eu des problèmes de compréhension avec les mailbox et l'utilisation des adresses hôtes et distantes. Nous pensions qu'il fallait renseigner les numéros de box, alors qu'il fallait les fixer par défaut. Nous avons passé du temps à implémenter les conditions, une fois ceci fait nous avons pu utiliser les sockets.

Conclusion

Ce projet nous a permis d'améliorer notre gestion du temps et notre organisation du travail. Travailler en groupe est intéressant mais pas toujours simple, la gestion d'un groupe nécessite du temps.

Nous avons pu approcher en pratique les concepts vus dans l'UE de CSE et exécuter certains programmes réalisés au cours des TP. Ce projet nous a par ailleurs permis une certaine liberté d'implémentation et, malgré quelques regrets, nous avons pu réaliser la majorité de nos idées.