Université Grenoble Alpes

Master 1 - Informatique

Rapport de projet

NachOS : Projet de programmation système

Auteurs (Groupe E) :
Stephen Abello
Paul Carretero
Florent Chastagner
Clément Didier

Responsables :
Vincent Danjean
Vania Marangozova

25 janvier 2017

Table des matières

1 Introduction **2**
1.1 Système d'exploitation NachOS 2
1.2 Notre objectif 2

2 Notre programme, les fonctionnalités et l'utilisation de notre noyau NachOS **2**
2.1 Gestion d'une console utilisateur 2
2.2 Gestion des Threads Utilisateurs et Processus 2
2.3 Interface du Système de fichiers 3
2.4 Interface Réseau (mode noyau) 3
2.5 Fonctionnalités utilisateurs additionnelles 3

3 Spécifications des Appels systèmes **3**
3.1 Console 3
3.2 Assert et Sémaphores Utilisateurs 4
3.3 Threads utilisateurs 5
3.4 Gestion des processus et adressage virtuel 6
3.5 Système de fichiers 6

4 Réalisations des tests utilisateurs **8**
4.1 Tests sur la console 8
4.2 Tests sur les threads Utilisateurs 8
4.3 Tests sur les structures de synchronisation 9
4.4 Tests sur la gestion des processus et de l'adressage virtuel 9
4.5 Tests sur le système de fichiers 9

5 Choix d'implémentation **9**
5.1 Console 9
5.2 Threads utilisateurs 9
5.3 Processus et adressage virtuel 10
5.4 Système de fichiers 10
5.5 Réseaux 10

6 Organisation **12**
6.1 Répartition du travail 12
6.2 Organisation et Planning 12

7 Conclusion **12**

Annexes : Documentation générale des fonctions créées **12**

1 Introduction

1.1 Système d'exploitation NachOS

NachoOS est un système d'exploitation à visée pédagogique. L'ensemble des opérations basiques exécutables par un système d'exploitation sont réalisable à travers NachOS. Un squelette de code et des primitives sont données pour pouvoir implémenter des fonctions permettant l'exécution de programme utilisateur de plus haut niveau.

1.2 Notre objectif

L'objectif de ce projet est de créer un ensemble de fonctionnalités utilisateurs pour NachOS à partir des primitives existantes. À travers ce projet nous avons notamment pu développer une console utilisable par des programmes utilisateurs, une gestion des threads et processus. De plus notre noyau offre des primitives de plus haut niveau pour la gestion de fichiers et l'utilisation d'un réseau.

2 Notre programme, les fonctionnalités et l'utilisation de notre noyau NachOS

Voici les 2 flavors NachOS que nous utilisons :

nachos-complete charge tous les modules et le nouveau système de fichier.

nachos-classic ne charge pas le nouveau système de fichiers (utilise fs_stub) ni la partie réseau. Utile notamment pour tester des programmes de plus de 3.75Ko.

2.1 Gestion d'une console utilisateur

Notre implémentation propose une console utilisateur proposant des primitives pour afficher ou récupérer des données. Notre console affiche les chaînes de caractères de manière contiguë. Plusieurs options sont proposées pour récupérer une entrée (sous forme d'entier, de caractère ou de chaînes de caractères) ou l'afficher (un entier, une chaîne de caractères ou un caractère).

2.2 Gestion des Threads Utilisateurs et Processus

Notre implémentation permet la création de threads utilisateurs par un appel système. Le programme utilisateur récupère le TID du thread créé (ou -1 si le thread n'a pas pu être créé). Il est possible pour un programme utilisateur d'attendre un autre thread. Deux threads peuvent attendre le même thread. Il n'est pas nécessaire que le thread que l'on souhaite attendre existe encore ou ai existé par le passé. De manière générale lorsqu'une demande d'attente de threads invalide est effectué alors la fonction se termine et permet au thread de continuer.
Un programme utilisateur à la possibilité de créer un nouveau processus dans un autre espace d'adressage.

Nos threads utilisateurs se terminent automatiquement à la fin de la fonction passée en paramètre par l'utilisateur. Il est toutefois possible de demander la fin du thread courant à n'importe quel moment de son exécution. De plus, lorsque le dernier processus se termine alors la machine NachOS se termine. De la même manière il est possible de terminer le système NachOS par un appel système.

2.3 Interface du Système de fichiers

Notre système de fichiers permet des opérations simples (création ou suppression de fichiers ou de répertoires). Un appel système permet de changer de répertoire courant en prenant en compte un chemin de fichier absolu (commençant par '/') ou relatif (commençant par un répertoire). Pour naviguer et afficher plus facilement dans l'arborescence des répertoire un shell (fsshell) utilisateur a été créé et reprend les fonctions énoncées précédemment.

Il est possible d'ouvrir jusqu'à 10 fichiers simultanément. De plus, notre système de fichiers supporte les accès concurrent. Seul un thread peut accéder à un fichier.

2.4 Interface Réseau (mode noyau)

Notre réseau permet une intercommunication entre plusieurs machines NachOS. Il est par exemple possible d'envoyer ou recevoir des messages.

2.5 Fonctionnalités utilisateurs additionnelles

2.5.1 Sémaphores

Notre programme propose aux utilisateurs l'utilisation de Sémaphore. Il est possible d'en créer (avec le nombre de jeton initial), de prendre un jeton (ou attendre s'il n'y en a plus), ou encore de libérer un jeton et détruire la sémaphore utilisateur.

2.5.2 Assert Utilisateur

Notre noyau NachOS permet également aux programmes utilisateurs l'utilisation d'*Assert* avancés. Ces asserts utilisateurs permettent d'afficher le résultat d'un test à valider sans pour autant terminer le programme.

3 Spécifications des Appels systèmes

Nous ne détaillerons ici que les appels systèmes que nous avons implémenté. Une spécification plus exhaustive de l'ensemble des fonctions que nous avons implémenté (appels système et hors appels systèmes) est disponible en annexe. De plus, la partie réseau de notre implémentation ne propose pas d'appels systèmes.

3.1 Console

3.1.1 GetChar

Usage Obtient un caractère depuis l'entrée de la console standard et le retourne. La fonction est bloquante, le thread attend une entrée utilisateur avant de poursuivre

Retour Retourne un char, caractère entré dans la console

3.1.2 GetInt

Usage Obtient un entier sur 8bits depuis l'entrée de la console standard. Ne permet d'obtenir que des entiers positifs et inférieur ou égaux à 255. La fonction est bloquante, le thread attend une entrée utilisateur avant de poursuivre.

Arguments n Entier codé sur 8 bits. n pointera vers la valeur lue.

Vérification et exception Invalide un Assert si l'entier entrée n'est pas comprise entre 0 et 255.

3.1.3 GetCharInt

Usage Obtient un caractère depuis l'entrée de la console standard et retourne l'entier (table ASCII) associé. La fonction est bloquante, le thread attend une entrée utilisateur avant de poursuivre.

Retour Retourne un int représentant le caractère entré dans la console de la table ASCII.

3.1.4 GetString

Usage Obtient une chaîne de caractère de taille maximale définie depuis l'entrée de la console standard. L'utilisateur choisit la taille du buffer de réception. Il choisit donc la taille de la chaîne à récupérer. La fonction est bloquante, le thread attend une entrée utilisateur avant de poursuivre.

Arguments s Buffer dans lequel les données lues seront écrites, il doit être initialisé et doit pouvoir contenir n place. n (entier) Correspond à la taille maximale de la chaîne de caractères (Il doit être fixé par l'utilisateur).

3.1.5 PutInt

Usage Écrit un entier sur la console standard.

Arguments n Entier a écrire sur la console

3.1.6 PutChar

Usage Écrit un caractère sur la console standard. La fonction garantit un affichage synchrone d'un caractère.

Arguments c doit être de type char.

3.1.7 PutString

Usage Écrit une chaîne de caractères sur la console standard.

Arguments s Chaîne de caractères à entrer. De taille maximum MAX_STRING_SIZE.

Vérification et exception Cette fonction ne permet pas la concaténation dans l'appel de la fonction (eg PutString("ab" + "bc") interdit).

3.2 Assert et Sémaphores Utilisateurs

```
1 void AssertFull(int res, const char* condition, const int lineNumber, const
      char* functionName);
2 void Assert(res)
```

3.2.1 Assert

Usage version simplifiée du Assert utilisateur, ne termine pas le programme et affiche des informations sur l'évaluation de l'Assert, permet de spécifier un test boolean, la description, le numéro de ligne et le nom de la fonction sont renseignés automatiquement.

Arguments res : entier boolean a vérifier (1 vrai, 0 sinon).

3.2.2 AssertFull

Usage version complète de Assert, permet de modifier tous les détails sur tous les paramètres affichables par iziassert.

Arguments res (int) : entier boolean. condition (const char *) : description du test. lineNumber (int) : numéro de la ligne. functionName (const char *) : nom de la fonction appelant AssertFull.

```
1 sem_t * UserSemCreate(char * name, int nb);
2 void UserSemP(sem_t * sem);
3 void UserSemV(sem_t * sem);
4 void UserSemDelete(sem_t * sem);
```

3.2.3 UserSemCreate

Usage Créer une sémaphore de type sem_t et renvoie un pointeur vers elle. Il est nécessaire de passer en paramètre un nom ainsi que le nombre de tokens que possède cette sémaphore. Cette sémaphore devra ensuite être détruite par UserSemDelete.

Arguments name nom de la sémaphore, ne doit pas dépasser MAX_SEM_NAME_SIZE caractères.
nb Nombre de token a passer a la sémaphore, doit être égal ou supérieur à zéro

Retour Renvoie un pointeur vers la sémaphore sem_t

3.2.4 UserSemP

Usage Consomme un token de la sémaphore ou attend qu'un token soit entré s'il n'y en a pas de disponible. Un pointeur vers une sémaphore crée par UserSemCreate doit être passé en paramètre.

Arguments sem Sémaphore qui doit recevoir l'opération P(), doit crée par UserSemCreate

3.2.5 UserSemV

Usage Poste un token dans la sémaphore sem. Un pointeur vers une sémaphore crée par UserSemCreate doit être passé en paramètre.

Arguments sem Sémaphore qui doit recevoir l'opération V(), doit crée par UserSemCreate.

3.2.6 UserSemDelete

Usage Permet de détruire une sémaphore dont le pointeur est passé en paramètre, il doit pointer vers une sémaphore créer par UserSemCreate.

Arguments sem Sémaphore qui doit être détruite, doit être crée par UserSemCreate.

3.3 Threads utilisateurs

```
1 int UserThreadCreate(void f(void *arg), void *arg);
2 int UserThreadJoin(int tid);
3 void UserThreadExit();
```

3.3.1 UserThreadCreate

Usage Demande la création d'un thread utilisateur au système qui exécute la fonction dont le pointeur sera passé en paramètre et qui transmettra à cette fonction l'argument passé en 2nd paramètre (un void *) qui devra être récupéré comme un void *, et casté dans son type d'origine par exemple.

Arguments f Pointeur (non NULL) vers une fonction utilisateur qui sera exécutée au début du thread. La fonction ne doit prendre qu'un seul argument de type void*
arg Argument (non NULL) à passer a la fonction f, à caster en void*

Retour Retourne le TID du thread créé ou -1 si la création a échouée (Manque de place dans la mémoire etc). TID unique dans la machine (deux processus ne peuvent pas avoir de threads avec des TID identiques).

Vérification et exception Invalide un Assert si une erreur c'est produite lors de l'attribution de l'espace mémoire pour le thread utilisateur.

3.3.2 UserThreadJoin

Usage Attend la terminaison du thread dont le TID est passé en paramètre, si ce thread n'existe pas ou plus (Déjà terminé, ou TID n'appartenant a aucun thread), la fonction laisse passer l'utilisateur. Si un thread essaye de s'attendre lui même, il passe aussi. Si l'utilisateur essaye de rejoindre un TID impossible (< 1, donc soit le thread TID passé en paramètre existe et est en cours d'exécution ou si d'autre thread attende ce thread, la fonction place le thread courant en attente jusqu'à ce que le thread soit terminé.

Arguments TID Thread ID du thread a attendre.

Retour Retourne -1 si le TID est inférieur (strict) à 1.
Retourne 1 si la fonction a attendu le thread ayant le TID passé en paramètre (le thread #TID était en exécution). Retourne 2 si aucun thread n'avait le TID passé en paramètre. Retourne 3 si l'on tente de s'attendre soit même.

3.3.3 UserThreadExit

Usage Demande la terminaison du thread utilisateur actuel, elle peut être retardée si d'autres processus sont en création/terminaison, les données du processus seront détruites et son espace mis à disposition pour de nouveaux threads. Si d'autres threads attendent la fin de l'exécution de ce thread (UserThreadJoin), ils reprendront peu après l'appel de cette fonction.

3.4 Gestion des processus et adressage virtuel

```
1 void Halt();
2 void Exit(int statut);
3 int ForkExec(char *s);
4 void SimpleShellProcJoin();
```

3.4.1 Halt

Usage Permet à l'utilisateur d'arrêter la machine. Attendra la fin de l'exécution de tous les threads (noyau) avant de s'exécuter. Est automatiquement appelé à la fin d'un thread (noyau).

3.4.2 Exit

Usage Permet à l'utilisateur de terminer le thread.

Arguments statut : Code de retour, 0 généralement si aucune erreur.

3.4.3 ForkExec

Usage Créer un thread système dans un nouveau processus et lance le programme dont le nom est passé en paramètre.

Arguments s Le nom du fichier exécutable à lancer.

Retour Retourne le PID du processus créé, -1 si échec.

3.4.4 SimpleShellProcJoin

Usage Utilisé par le shell. Permet d'attendre la fin d'un unique processus lancé avec forkexec. permet de continuer si un processus a été créé par forkexec puis s'est terminé.

Note : Ne permet pas d'effectuer des synchronisations complexes entre les processus. Si le processus attendu lance un autre processus alors la fonction permet de continuer. Dans un tel cas il sera nécessaire d'attendre la fin de tout les processus lancés avec forkexec avant de pouvoir de nouveau faire cet appel système (appel non bloquant, donc inutile, sinon).

Vérification et exception : Une terminaison anormale d'un processus entraînera un verrou mortel.

3.5 Système de fichiers

```
1 void Mkdir (char *name);
2 void RmDir (char *name);
3 void Remove (char *name);
4 int ChangeDirectory (char *name);
5 OpenFileId Open (char *name);
6 void List ();
7 void Close (char * name);
8 void Open (char * name);
9 int Write (char *buffer, int size, OpenFileId id);
```

```
10 int Read (char *buffer, int size, OpenFileId id);
11 void SetCursor(OpenFileId id, int pos);
```

3.5.1 Mkdir

Usage Créer un dossier du nom passé en paramètre. Ne fait rien en cas d'impossibilité (trop de dossier par exemple).

Arguments name nom du dossier à créer, name a une taille maximum de 9.

3.5.2 RmDir

Usage Supprime un dossier de nom passé en paramètre Ne fait rien en cas d'impossibilité (mauvais nom par exemple).

Arguments name nom du dossier à supprimer de taille maximum 9.

Vérification et exception Le système de fichiers NachOS FILESYS doit être lancer (sans effet sinon).

3.5.3 Remove

Usage Supprime un fichier de nom passé en paramètre. Ne fait rien en cas d'impossibilité (mauvais nom par exemple).

Arguments name nom du fichier à supprimer de taille maximum 9.

Vérification et exception Le système de fichiers NachOS FILESYS doit être lancer (sans effet sinon).

3.5.4 ChangeDirectory

Usage permet de changer de répertoire courant.

Arguments Chemin du fichier dans lequel basculer. Le chemin peut être absolu (commence par '/') ou relatif. De taille maximum MAX_STRING_SIZE.

Retour retourne 1 en cas du changement réussi, 0 sinon. Si on a pas pu changer de dossier alors on retourne au dossier racine.

Vérification et exception Le système de fichiers NachOS FILESYS doit être lancer (sans effet sinon).

3.5.5 List

Usage Affiche la liste des fichier et répertoire contenu dans le répertoire courant.

Vérification et exception Le système de fichiers NachOS FILESYS doit être lancer (sans effet sinon).

3.5.6 Close

Usage Ferme le fichier passé en paramètre une fois les opérations de lectures/écritures terminées.

Arguments name nom du fichier à fermer. Le fichier name doit être ouvert et dans le répertoire courant.

3.5.7 Open

Usage Ouvre le fichier passé en paramètre.

Arguments name nom du fichier à ouvrir. Le fichier name doit être dans le répertoire courant.

3.5.8 Create

Usage Créer un fichier du nom passé en paramètre à l'emplacement actuel dans le système fichier.

Arguments name chaîne de caractères de longueur maximum 9 correspondant au nom du fichier à créer.

Vérification et exception Ne créera pas le fichier en cas d'impossibilité (plus de place ou nom déjà pris).

3.5.9 Write

Usage Écrit le buffer passé en paramètre (de taille définit) sur le fichier ouvert.

Arguments Buffer, un pointeur vers le buffer de taille minimum size. size la taille des données à écrire. id un pointeur vers le descripteur de fichier OpenFileId.

Retour retourne le nombre de caractères écrit.

3.5.10 Read

Usage Écrit dans le buffer passé en paramètre (de taille définit) les données présentes sur fichier ouvert (pour lecture par le programme). à partir de l'emplacement actuel du curseur.

Arguments Buffer, un pointeur vers le buffer de taille minimum size. size la taille des données à lire. id un pointeur vers le descripteur de fichier OpenFileId.

Retour retourne le nombre de byte lus.

3.5.11 SetCursor

Usage replace le curseur à l'emplacement indiqué.

Arguments id l'identifiant du fichier ouvert OpenFileId. pos une position (int).

Vérification et exception Aucune vérification n'est effectué sur la validité de la position pos.

4 Réalisations des tests utilisateurs

4.1 Tests sur la console

asynchstringtest.c : lance 10 threads utilisateurs qui affiche une première ou une seconde chaîne de caractères de façon contiguë (pas d'entrelacement).

inttest.c : demande à l'utilisateur d'entrer un int et l'affiche, permet de tester les appels systèmes sur la console getCHarInt, GetInt et PutInt

stringtest.c : permet de vérifier le bon fonctionnement de la console en affichant un train en mouvement. Un terminal de largeur d'au moins 200 caractères est recommandé.

putchar.c : affiche une série de 5 caractères puis demande un caractère et l'affiche. Permet de tester les appels systèmes getchar et putchar. putchartest.c permet de tester les appels système Halt et PutChar (affiche les caractères de la table ascii).

4.2 Tests sur les threads Utilisateurs

userthreadtest.c : Illustre le fonctionnement des threads utilisateurs. Par groupe de 3, des threads utilisateurs sont lancés (ils affichent 10 lignes chacun) exécutés puis attendus (par groupe de 3) avant de poursuivre. Ce test permet de vérifier les appels systèmes associés aux threads.

simpleusertheadstest.c : Créer 3 threads, vérifie à l'aide de Assert que ces threads sont créés correctement et vérifie que la variable passée en paramètre s'incrémente à chaque thread.

tetris.c Ce test permet de tester la plupart des fonctionnalités de la console et de la gestion des threads [1].

1. Permet aussi de jouer au tetris sur NachOS (note : il est possible d'entrer jusqu'à 2 instructions, par exemple 1+2+enter pour faire aller la pièce à gauche puis la faire pivoter)

4.3 Tests sur les structures de synchronisation

semaphore.c : Exemple d'un programme utilisateur utilisant des sémaphores pour créer un scénario de consommateurs/producteurs (parcours la table ASCII).

4.4 Tests sur la gestion des processus et de l'adressage virtuel

forktest.c Lance 10 programmes utilisateurs dans de nouveaux processus. Ces programmes utilisateurs (userpage0 et userpage1) affiche une chaine de caractères ou lance d'autre thread utilisateur par exemple.

shell.c Affiche un shell utilisateur permettant de lancer un autre processus, de l'attendre et de répéter l'opération. Affiche également deux nachos.

4.5 Tests sur le système de fichiers

fsshell.c Une fois compiler ce programme doit être copier sur le système de fichiers NachOS. Il permet d'exécuter des opérations de bases sur notre système de fichier, notamment se déplacer (avec prise en compte des chemins de fichier), de créer ou supprimer fichier ou répertoire.

5 Choix d'implémentation

5.1 Console

Nous utilisons plusieurs sémaphores pour contrôler l'accès aux sections critiques de la console. Nous ne devons acquérir la sémaphore "semPutChar" avant de pouvoir écrire un caractère. De cette manière nous sommes assurés que deux accès concurrent ne peuvent pas être réalisés pour écrire un caractère sur la console. Nous attendons également que la console libère un jeton associé à une écriture correcte ou à une lecture possible.
Nous utilisons également une sémaphore semRead et semWrite pour assurer des lectures et écritures contiguë.
Il est également nécessaire de considérer les limitations MIPS puisque chaque entier est codé est 8 bit seulement et donc ne permet qu'un maximum de 255.

5.2 Threads utilisateurs

Les threads utilisateurs sont créés dans l'espace d'adressage du thread noyau (processus) courant. Nous initialisons une structure pour passer en paramètre les informations requises pour lancer le thread (nom de la fonction à exécuter et arguments). Nous recherchons pour cela un espace libre dans l'espace d'adressage du thread noyau courant grâce à une bitmap. Si cela est possible alors on réserve l'espace sur la bitmap. Dans un second temps la fonction StartUserThread initialise les registre notamment pour indiquer l'emplacement de la fonction et l'adresse de la fonction à appeler lors de la fin du thread (ceci permet d'éviter d'avoir à appeler userthreadexit).
La création d'un thread entraîne l'acquisition d'un verrou de jointure (Lock) unique à ce thread et à l'inscription de ce thread (pointeur vers l'objet) dans le tableau des threads courants (le nombre max de threads par processus est connu).
On ajoute ajoute une structure avec le tid du thead, le verrou de jointure et le nombre de threads en attente (au début à 1) dans le tableau "d'attente".

Lors de la terminaison d'un thread nous supprimons les données associées à ce thread et libérons l'espace dans la bitmap. On libère (sans supprimer) le verrou de jointure. On se supprime du tableau des threads actifs. On décrémente le nombre de threads en attente dans le tableau des threads en attente. Le garbage collector parcours ce tableau et y supprime les entrée avec 0 thread en attente.

Lorsqu'un thread tente d'en attendre un autre qui n'est pas dans la liste des threads actif, alors on ne fait rien de particulier (ce thread n'existe plus ou n'a jamais exister).
Si Le thread est dans le tableau des threads actif alors on incrémente le nombre de threads qui l'attende dans le tableau des threads en attente. Une fois le verrou obtenu on décrémente le nombre de threads en attente et on poursuit (en libérant le verrou et en appelant le garbage collector pour libérer les ressources).

5.3 Processus et adressage virtuel

Lors de leur création les processus se voient associé un PID, celui-ci est représenté par un entier sur 32 bit et est incrémenté à chaque création de processus.

Aucune réutilisation de PID n'est possible, le nombre de processus pouvant donc être créé sur une exécution de la machine est de 2^{32}. Une fois le PID associé au processus celui-ci est ajouté la liste des processus en cours. Lorsque le processus se termine son PID est alors retiré de la liste. Si le processus est le dernier de la liste la machine est arrêtée (Halt).

5.4 Système de fichiers

Notre système de fichiers implémente les fonctions de création et de suppression de dossier, ainsi que le changement de dossier. Il permet aussi de garder une liste des fichiers ouverts et gère les accès concurrents aux fichiers. Enfin il permet d'utiliser des *pathname* afin de se déplacer dans les dossiers.

Accès concurrent des fichiers Pour limiter l'accès à un fichier à un seul Thread à la fois, nous avons créé une classe $FileMap$ qui simule tableau de *map* C++ de deux int. Nous aurions pu utiliser tableau de *map* C++ si la technologie de NachOS nous l'avait permis. Lors de l'appel des fonctions *Open* et *Close* de filesys, les méthodes de $FileMap$ seront appelées afin de vérifier si le fichier est déjà ouvert, si oui par quel Thread, et si le nombre de fichier ouvert maximum n'est pas atteint.

Analyse du chemin de fichier Notre implémentation permet l'analyser les chemins de fichier. La fonction $ChangeDirPath$ parcourt un chemin passé en paramètre, en copiant dans un buffer temporaire les noms de fichiers rencontrés et appel la fonction $ChangeDir$ à chaque '/' rencontré avec ses données copiées. Si le premier caractère est un '/' alors on revient sur le dossier racine. De cette manière nous permettons un déplacement avec un nom de chemin absolu ou relatif (avec prise en comte des . et ..).

5.5 Réseaux

5.5.1 Notre implémentation actuelle

Le réseau fiable avec messages de taille fixe n'est pas totalement terminé, il ne prend actuellement pas en compte le début et la fin de connexion avec la machine distante. Son fonctionnement se résume par la création d'un acquittement pour chaque message

reçu, afin d'attester de la bonne réception du message auprès de l'émetteur. Dans le cas inverse, l'envoi d'un message est effectué toutes les TEMP0 secondes, tant qu'un message d'acquittement n'ai été reçu et que le nombre maximum de ré-émission n'est pas dépassé.

FIGURE 1 – Paquet réseau simple

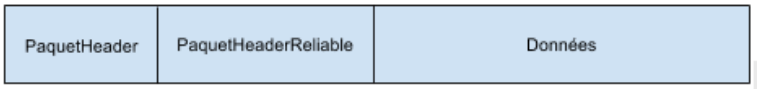

FIGURE 2 – Paquet réseau fiable

5.5.2 Choix d'implémentation du réseau fiable

Chaque message correspond à un paquet network disposant d'un entête PaquetHeader et d'une section de données. La couche ReliableNet réalisée rajoute quand à elle un entête PaquetHeaderReliable au début de la section de données et augmente la taille contenu dans le PaquetHeader.

5.5.3 Pseudo Code

```
1  Si ReceptionMessage Alors
2      LectureEntetes(Message)
3      Si Type(Message) = MSG Alors
4          Si Message != MessagePrecedant Alors
5              MettreAjour(numReception)
6          FinSi
7          //Envoi un ACK meme dans le cas de la reception d un message
       identique
8  Envoyer(ACK, MessageIndex)
9      Sinon Si Type(Message) = ACK Et estACKduMessagePrecedementEnvoye Alors
10         MettreAJour(numAcquittement)
11     FinSi
12 FinSi
13
14 Si EnvoiMessage Alors
15      // Creation d un thread executant les instructions suivantes
16     EcritureEntete(NouveauMessage)
17     Pour nombre d envois MAXREEMISSIONS Faire
18         Si Type(NouveauMessage) = MSG Et NouveauMessageDejaAcquitte Alors
19             Sortir du Pour
20         Sinon Si Type(NouveauMessage) = ACK Et MessageRecuDejaAcquitte
       Alors
21             Sortir du Pour
22 FinSi
23 Envoi(NouveauMessage)
24 Attendre(TEMP0)
25     FinPour
26     Si MAXREEMISSIONS atteint Alors
27         Fermeture
```

```
28      FinSi
29  FinSi
```

5.5.4 Limites actuelles

Lorsque tous les messages ont étés acquittés, les machines distantes ne renvoient plus de nouveaux messages (MSG), ainsi elle ne savent pas si leur ACK a été reçu. Elles continuent donc d'envoyer MAXREEMISSIONS fois un message ACK. (Problème lié à la non implémentation de la fin de la communication (type de message END)).

6 Organisation

6.1 Répartition du travail

Nous avons réussi à répartir le travail de manière homogène tout au long de notre projet. De manière générale nous travaillons par groupe de deux personnes sur deux parties différentes en parallèle. Si l'un des deux groupes était en avance, nous en profitions généralement pour réaliser des parties optionnelles ou finaliser le code (commentaires etc.) L'utilisation de Git avec plusieurs branche pour chaque partie ou sous partie nous a permis de mettre en œuvre une tel organisation.

6.2 Organisation et Planning

Malgré quelques journées banalisées durant les 4 semaines de projet, nous avons pût, de manière générale, suivre le planning proposé. Toutefois, vers la fin de la période nous avons manqué de temps pour fournir une implémentation complète et finalisée de la fiabilité des réseaux (étape 6).

7 Conclusion

Bien que de nombreuses parties optionnelles et bonus n'ont pût être implémenté dans les délais impartis, nous avons pu produire une implémentation complète répondant aux objectifs. Nous avons pris soin, au cours du développement, à conserver de bonnes pratiques de programmation (commentaire sur les fonctions créées, libérations des ressources utilisées etc.). Par ailleurs notre implémentation propose plusieurs parties bonus permettant une utilisation plus agréable de NachOS pour l'utilisateur.

Du fait des limitations "matérielles" de NachOS, nous avons également pût constater la nécessité de produire du code optimisé et adapté au support visé (notamment éviter un trop grand nombre d'appels systèmes entraînant un sur-coût important lors des changements de contexte). Par exemple même si le résultat est identique, il sera plus agréable de jouer au tétris sur un processeur rapide. Dans le cas contraire on risque d'observer un affichage saccadé et inconfortable...