Rapport Projet Nachos
M1 Informatique/MOSIG
Groupe G

Amine Aït-Mouloud
Sébastien Avril
Jean-Yves Bottraud
El Hadji Malick Diagne

Janvier 2015

Table des matières

1 Introduction

Ce document représente notre rapport final dans le cadre du projet Nachos proposé pour les parcours M1 Informatique et MOSIG. Il abordera les principales fonctionnalités du noyau, la documentation des fonctions proposées à l'utilisateur, l'organisation de l'équipe et celle des tests, ainsi que les différents partis pris.

2 Fonctionnalités du noyau

Voici les parties que nous avons réussi à implémenter jusque là :
— Appels système de lecture et d'écriture : Caractères, chaînes de caractères, et entiers.
— Multithreading : Création, destruction d'un thread, ainsi que l'attente d'un autre thread. Partage de la pile d'exécution entre threads.
— Multi-processus s'exécutant en parallèle.
— Adressage virtuel via une table de pages.
— Un système de fichiers : Création et suppression de fichiers/dossiers, ainsi que déplacement dans hiérarchie.
— Transmission synchrone sur le réseau avec attente d'acquittement avant prochaine transmission.
— API d'envoi et de réception de messages et de fichiers par réseau.

3 Documentation des fonctions utilisateur

3.1 Console synchrone

SIGNATURE : `char GetChar()`

DESCRIPTION : Lis un caractère depuis l'entrée standard et le retourne sous forme de `char`. Retourne `EOF` dans le cas de la fin de fichier.

VALEUR DE RETOUR : Renvoie un `char` qui représente le caractère lu, ou `EOF` en cas de fin de fichier ou d'erreur.

SIGNATURE : `void GetString(char *s, int n)`

DESCRIPTION : Lis au maximum `n` caractères depuis l'entrée standard jusqu'à rencontrer un caractère dit "bloquant", c'est à dire : un retour chariot (`\r`), un saut de ligne (`\n`), ou un caractère de fin (`\0` ou `EOF`), et copie la chaîne de caractères lue vers le buffer pointé par `s`. La longueur maximale d'une chaîne récupérée est définie par la constante `MaxStringSize`.
Si un caractère dit "bloquant" est rencontré, il est remplacé par l'octet nul (`\0`), sinon ce dernier est placé dans l'emplacement mémoire pointé par `s+n`.

SIGNATURE : `int GetInt()`

DESCRIPTION : Lis une chaîne de caractères depuis l'entrée standard selon les mêmes règles que `GetString`, la convertit en entier naturel, et retourne ce dernier.

VALEUR DE RETOUR : Entier naturel lu depuis l'entrée standard.

SIGNATURE : `void PutChar(const char ch)`

DECRITPION : Écrit le caractère `ch` sur la sortie standard.

SIGNATURE : `void PutString(const char *s)`

DESCRIPTION : Écrit la chaîne présente dans le buffer pointé par `s` ainsi qu'un saut de ligne (`\n`) sur la sortie standard. L'écriture de la chaîne sur la sortie standard se poursuit jusqu'à la rencontre d'un caractère dit "bloquant", c'est à dire : un retour chariot (`\r`), un saut de ligne (`\n`), ou un caractère de fin (`\0` ou `EOF`). La longueur maximale de la chaîne à afficher est définie par la constante `MaxStringSize`.

SIGNATURE : `void PutInt(int i)`

DESCRIPTION : Convertit l'entier naturel `i` en chaîne de caractères et l'affiche sur la sortie standard selon les mêmes règles que `PutString`.

3.2 Threads

SIGNATURE : `int UserThreadCreate(void *f(void*), void* arg)`

DESCRIPTION : Permet de créer un thread dans le même espace d'adressage que le thread en cours. Le thread créé lancera la fonction f donnée en paramètre avec l'argument `arg`. L'identifiant du thread est unique tout au long de l'exécution du processus.
Le thread créé est *de facto* dans le même processus que le thread dans lequel il a été créé.
Si l'un des threads d'un processus appelle `exit`, la gestion de l'attente des threads non-terminés est gérée par la terminaison du processus.

VALEUR DE RETOUR : Identifiant du thread en cas de succès, code d'erreur négatif sinon :

- `-1` : Raison inconnue.
- `-2` : Pas assez d'espace dans la pile du processus.

SIGNATURE : `void UserThreadExit()`

DESCRIPTION : Termine le thread en cours d'exécution et libère son emplacement dans la pile. Si c'est le dernier thread du processus, ce dernier se termine aussi après avoir attendu ses threads toujours en cours d'exécution et les ressources qu'il utilise sont libérées.

SIGNATURE : `int UserThreadJoin(int id)`

DESCRIPTION : Attend que le thread ayant pour identifiant `id` se termine, et si le thread s'est déjà terminé la fonction retourne de suite. Plusieurs threads peuvent attendre un même thread étant donné que l'identifiant des threads est unique dans le contexte du processus.

VALEUR DE RETOUR : `0` en cas de succès, code d'erreur négatif sinon :

- `-1`, `-2` : Identifiant fourni introuvable (jamais créé ou négatif).
- `-3` : Tentative d'attendre le thread en cours d'exécution.

SIGNATURE : `int GetTid()`

DESCRIPTION : Renvoie l'identifiant du thread en cours d'exécution. Cet identifiant n'est unique que dans le contexte de son processus.

VALEUR DE RETOUR : Identifiant du thread en cours d'exécution.

3.3 Mémoire virtuelle

SIGNATURE : `int ForkExec(char *path)`

DESCRIPTION : Crée un nouveau processus dans un nouvel environnement d'exécution, et y exécute le fichier exécutable dont le nom est pointé par `path`.

VALEUR DE RETOUR : `0` en cas de succès, code d'erreur négatif sinon :

- `-1` : Impossible d'ouvrir le fichier présent à l'emplacement décrit dans le buffer pointé par `path`.
- `-2` : Impossible de créer un environnement d'exécution.
- `-3` : Pas assez de pages libres en mémoire pour créer l'espace d'adressage du processus.

3.4 Système de fichiers

SIGNATURE : `int mkdir(char *name)`

DESCRIPTION : Crée un répertoire portant le nom `name` dans le répertoire courant.

VALEUR DE RETOUR :

`0` : Le chemin spécifié n'existe pas, ou le dossier en cours contient déjà 8 fichiers/dossiers.

`1` : Succès.

SIGNATURE : `int rmdir(char *name)`

DESCRIPTION : Supprime le répertoire portant le nom `name` du répertoire courant. Si le dossier n'est pas vide, cette fonction retournera une erreur.

VALEUR DE RETOUR :

`0` : Le dossier à supprimer est introuvable ou bien le répertoire n'est pas vide.

`1` : Succès.

SIGNATURE : `int mkfile(char *name, int initialsize)`

DESCRIPTION :

VALEUR DE RETOUR :

`0` : Le dossier contient déjà 8 fichiers/dossiers, un fichier du même nom existe déjà, ou bien il n'y a plus de place sur le disque.

`1` : Succès.

SIGNATURE : `int rmfile(char *name)`

DESCRIPTION :

VALEUR DE RETOUR :

`0` : Il n'y a pas de fichier de ce nom dans le répertoire courant.

`1` : Succès.

SIGNATURE : `int cd(char *name)`

DESCRIPTION :

VALEUR DE RETOUR :

`0` : Le dossier de destination est introuvable ou bien il ne peut être ouvert.

`1` : Succès.

3.5 Réseau

SIGNATURE : `unsigned Send(char *tosend, unsigned size, int localPort, int to, int remotePort)`

DESCRIPTION : Envoie des données de taille `size` présentes dans le buffer pointé par `tosend` à la machine présente en adresse `to` sur le port distant `remotePort`. Les acquittements sont reçus sur le port local `localPort`.

VALEUR DE RETOUR : Nombre total d'octets envoyés acquittés.

SIGNATURE : `void Receive(int localPort, char *got, unsigned size)`

DESCRIPTION : Attend la réception de données de taille `size` sur le port local `localPort`, et les met dans le buffer pointé par `got`.

SIGNATURE : `int SendFile(char *path, int localPort, int to, int remotePort)`

DESCRIPTION : Envoie le fichier présent à l'emplacement `path` vers la machine ayant l'adresse `to` sur le port distant `remotePort`. Les acquittements pour les paquets sont reçus sur le port `localPort`.

VALEUR DE RETOUR : `0` en cas de succès et tous les octets acquittés, code d'erreur négatif sinon :

`-1` : Impossible d'ouvrir le fichier présent à l'emplacement décrit dans le buffer pointé par `path`.

`-2` : Taille du fichier invalide.

`-3` : Succès partiel, mais pas tous les octets envoyés ont été acquittés.

SIGNATURE : `int ReceiveFile(int localPort, char *path)`

DESCRIPTION : Attend la réception d'un fichier sur le port local `localPort`, et les met dans un fichier à l'emplacement décrit dans le buffer pointé par `path`.

VALEUR DE RETOUR : `0` en cas de succès et tous les octets acquittés, code d'erreur négatif sinon :

`-1` : Taille du fichier reçue invalide.

`-2` : Impossible de créer ou d'écraser le fichier présent à l'emplacement décrit dans le buffer pointé par `path`.

`-3` : Impossible d'ouvrir le fichier présent à l'emplacement décrit dans le buffer pointé par `path`.

4 Tests

4.1 Organisation

Pour automatiser les tests, nous avons décidé de créer un script (`testparts.sh`) qui va lancer les fichiers de test du projet. Nous avons essayé de faire suffisamment de tests afin tester tous les défauts possibles du programme, ils sont répartis en une série de fichiers séparés par étape du projet dans les dossiers correspondant à l'étape, par exemple, les tests présents dans le dossier `etape4` correspondent à la partie sur la gestion de la mémoire virtuelle. Tous ces dossiers sont situés dans `code/test`.

Les fichiers `.sh` se basent sur les programmes de test utilisateur (`.c`), unitaires pour la plupart.

4.2 Utilisation

Pour lancer les tests, ils suffit d'exécuter le script `testparts.sh` situé dans le dossier `code/test`.

4.3 Tests effectués et comportements

4.3.1 Console

- Écriture d'une chaîne de 1 caractère : Écrit le caractère correctement sur la console
- Écriture/lecture parallèle : Les chaînes de caractères sont écrites/lues d'un coup quelque soit l'ordonnancement, les lectures/écritures de caractères seuls peuvent être entrelacés selon l'ordonnancement.
- Dépassement de la taille maximale d'une chaîne : Chaîne tronquée.
- Arrêt avec `Ctrl+D` en début de ligne : Arrête la console, n'est pas confondu avec ÿ.

4.3.2 Multithreading

- Création d'un thread :

  Création d'un thread : OK dans les bonnes conditions et retourne le bon code d'erreur dans les mauvaises.

  Création d'un très grand nombre de threads : Crée le thread normalement jusqu'au moment où ce n'est plus possible (pile pleine, ou nombre maximal de threads atteint). Gestion d'erreur fonctionnelle.

— Terminaison d'un thread :

Vérification des fuites mémoires : Toutes les structures allouées sont bien supprimées.

Terminaison automatique : Tous les threads se terminent bien à la terminaison de leur processus même s'ils ne se sont pas explicitement terminés par `UserThreadExit`.

— Lancement d'une fonction : Testé avec les fonctions disponibles, elles se sont déroulées correctement.

4.3.3 Mémoire virtuelle

— Création d'un processus : OK dans les bonnes conditions et retourne le bon code d'erreur dans les mauvaises.

L'exécutable fourni est bien exécuté.

— Stress de la mémoire

Les fichiers `matmult` et `sort` s'exécutent sans problèmes.

Création d'un grand nombre de processus en même temps se passe bien dans les limites de la mémoire disponible. Au-delà, création de processus rejetée.

4.3.4 Système de fichiers

— Création de dossier/fichier : Le dossier/fichier est créé et accessible.
— Déplacement vers un dossier parent/fils : On peut se déplacer à volonté dans les dossiers.
— Suppression dossier/fichier : Le dossier/fichier est supprimé correctement. Il ne réapparaît pas lors de la prochaine lecture du disque.

4.3.5 Réseau

— Test en anneau : La machine 1 envoie un message à la machine 2 qui l'envoie à la machine 3 .... qui l'envoie à la machine n, qui renvoie le message à la machine 1.

Les données sont bien reçues et bien renvoyées, même avec une perte de paquets significative.

— Envoi/réception de fichier : Fichier envoyé et reçu puis enregistré correctement.

5 Choix d'implémentation

5.1 Console

Le but de cette section est la mise en place des entrées/sorties de niveau utilisateur. Pour cela, les appels système de lecture et d'écriture ont été créés. Cette section étant très guidée dans le sujet, aucun choix d'implémentation particulier ne s'est posé.

La lecture et l'écriture ont été sécurisées avec un sémaphore chacun. Les fonctions de lectures (`GetInt`, `GetChar` et `GetString`) ne peuvent pas s'interrompre les unes les autres et les fonctions d'écritures (`PutChar`, `PutInt` et `PutString`) ne s'interrompent pas les unes les autres non plus. Mais les fonctions d'écriture peuvent interrompre les fonctions de lecture et inversement.

`GetString` et `PutString` appelant respectivement `GetChar` et `PutChar`, des versions internes "*sans sémaphore*" de ces deux dernières routines ont été implémentées pour pouvoir les appeler depuis les deux premières sans récursion de sémaphores.

5.2 Threads

5.2.1 Gestion des identifiants de threads

— un identifiant séquentiel unique parmi tous les threads du processus.
— un identifiant en pile qui représente l'emplacement en pile réservé au thread.

La pile est divisée en un nombre de blocs déterminé par le nombre de pages par thread + un *padding* afin d'éviter les débordements. L'identifiant en pile représente le numéro du bloc de la pile qui est alloué au thread. Les blocs de pile sont numérotés de `0` à `N-1`. `N-1` étant le bloc ayant l'adresse en mémoire la plus petite, et `0`

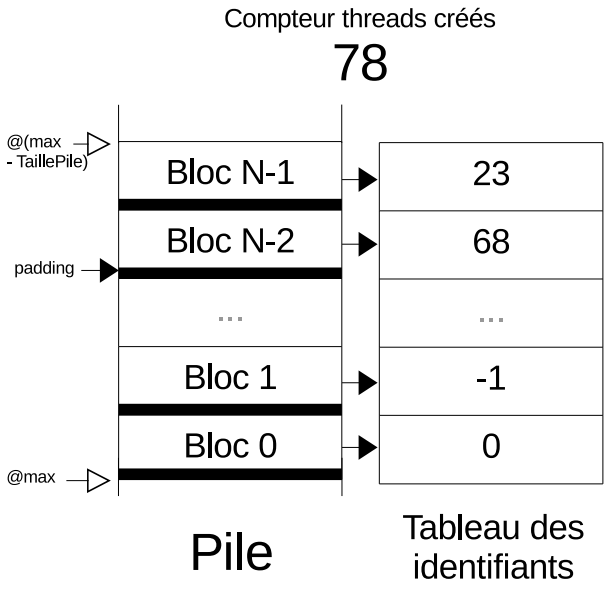

FIGURE 1 – Exemple d'un état des structures de gestion de l'identifiant de thread.

le bloc ayant l'adresse la plus grande. Un thread demande un bloc en pile à sa création, et le libère lorsqu'il se termine.

Afin de stocker les identifiants uniques et les identifiants en pile des threads, on utilise un tableau de taille `N` (nombre maximal de threads dans un processus), où chaque numéro de case représente un bloc en pile, et le contenu de la case représente l'identifiant unique du thread qui est alloué actuellement dans ce bloc en pile (si positif ou nul), ou que ce bloc en pile n'est pas alloué (négatif, -1). Voir Figure 1.

Afin de tester l'état d'un thread ayant pour identifiant unique `x`, on utilise les algorithmes suivants :

— `x` est supérieur au nombre de threads créés : Le thread n'a jamais été créé ;
— `x` est inférieur au nombre de threads créés, mais n'est pas présent dans le tableau : Le thread a été créé, mais s'est terminé ;
— `x` est inférieur au nombre de threads créés, et est présent dans le tableau à la case `i` : Le thread a été créé, et est en cours d'exécution, et le bloc en pile numéro `i` lui est alloué.

Il n'existe aucune hiérarchie entre threads, tous ceux créés dans un même processus sont au même niveau.

5.2.2 Implémentation du join

Un vecteur contenant la liste des threads en attente et l'identifiant des threads qui les attends.
— élément `who` : Thread qui attend.
— élément `forId` : Identifiant du thread que `who` doit attendre.

Dès l'appel à `Join` pour le thread ayant pour identifiant `x` et si le thread x est en cours d'exécution on ajoute la paire (thread courant, x) à la liste des threads en attente et le thread courant se met en pause.

Lorsque le thread ayant pour identifiant `x` se termine, il réveillera le thread qui s'était mis en attente pour qu'il continue son exécution.

Dans le cas où le thread `x` est déjà terminé, la fonction retourne immédiatement.

L'identifiant du thread étant unique dans un processus, il est possible de faire autant de `Join` que possible sur tous les identifiants.

5.3 Mémoire virtuelle

5.3.1 Pagination mémoire

Toutes les demandes de mémoire se font via le `FrameProvider` qui garde une `Bitmap` des pages vides ou non en mémoire physique.

Lors de la création d'un processus, une vérification de la disponibilité d'assez de pages physiques est faite afin de s'assurer que le processus peut s'exécuter. Ensuite, un *mapping* entre les pages virtuelles du processus et les pages physiques disponibles est fait.

Lors de la terminaison d'un processus, les pages physiques utilisées par ce dernier sont toutes libérées.

5.3.2 Multiprocessus

La création d'un processus déclenche la création d'un thread noyau avec un nouvel espace d'adressage ainsi que l'incrémentation d'un compteur des processus actifs.

Lors de la terminaison d'un processus, tous les threads créés dans celui-ci et ne s'étant pas terminé sont attendus, puis si le processus en question est le dernier processus de la machine, terminer la machine. Dans le cas contraire, le processus sera détruit par un des processus encore actif.

Seul un compteur des threads actifs est gardé car il n'existe pas hiérarchie entre processus.

5.3.3 Shell

Le *shell* implémenté permet de lancer un seul programme utilisateur à la fois. La valeur de retour de l'appel système `ForkExec` permet de déterminer si la commande est valide.

Lorsque le nouveau processus est lancé, le *shell* fait un appel à `Yield` tant que le *shell* n'est pas le seul processus actif.

5.4 Système de fichiers

Chaque dossier est composé de 8 fichiers/dossiers maximum (en plus des dossiers spéciaux `"."` et `".."`. Les noms des fichiers ne dépassent pas les 9 caractères.

Si un fichier n'est pas trouvé dans le répertoire en cours, il sera cherché dans `"/System"`, à l'image d'un `"/usr/bin"` pour Linux.

Concernant le déplacement dans l'arborescence de répertoires, nouveau `OpenFile` vers le répertoire courant a été ajouté. Celui-ci est modifié uniquement par l'appel système `cd`.

Les chemins sont pris en compte par la fonction `MoveTo` qui navigue dans les répertoires jusqu'à celui demandé. on peut spécifier un chemin absolu en rajoutant `"/"` devant le chemin, sinon le chemin est relatif.

5.5 Réseaux

5.5.1 En-tête du message

En plus des informations déjà présentes dans l'en-tête d'un paquet, son type (message de données (`MSG`) ou message d'acquittement (`ACK`), ainsi que son identifiant séquentiel unique parmi les messages envoyés de la machine en cours, sont ajoutés dans cet en-tête.

5.5.2 Démons

Le démon `"postal worker"` déjà implémenté, et dont la tâche était simplement d'attendre la réception des paquets, et de les mettre sur le bon port, se voit ajouter la tâche de renvoyer des acquittements pour les paquets de type `MSG`, ainsi que d'acquitter les paquets selon les identifiants des paquets de type `ACK` reçus.

Un second démon nommé `"postal sender"` tourne aussi en fond, et s'occupe de retransmettre toutes les `"TEMPO"` secondes le paquet, un maximum de "MAXREEMISSIONS" fois, tant que le `"postal worker"` n'aura pas signifié qu'un acquittement a été reçu pour le message en attente d'acquittement.

5.5.3 Transmission fiable d'un seul paquet

Lors de l'envoi, un paquet `MSG` est envoyé à la bonne destination via la couche réseau physique simulée par Nachos, et on déclenche le démon `"postal sender"` qui s'occupe de retransmettre périodiquement le paquet tout en attendant un acquittement.

5.5.4 Transmission sans limite de taille

Les données sont fragmentées en plusieurs paquets d'une taille maximale de `MaxMailSize` et transmises séquentiellement à l'aide de la couche de transmission fiable.

La taille maximale pour une transmission de données est la taille maximale d'une zone mémoire utilisateur, c'est à dire `MaxStringSize`.

5.5.5 Transmission de fichiers

À l'aide de la couche de transmission sans limite de taille, la taille du fichier est envoyée, puis le contenu du fichier par parties de `MaxStringSize`.

À la réception, dès qu'une partie du fichier source est reçue, elle est écrite par fragments de `MaxStringSize` dans le fichier destination. La taille du fichier source reçue en premier sert à initialiser la taille du fichier destination, ainsi qu'à connaître la quantité de données attendues.

6 Organisation du travail

L'équipe s'est réunie presque tous les jours de semaine à l'université afin de connaître le reste de l'équipe, et adapter le planning quand il le fallait.

Avant de démarrer chaque étape, chaque membre a bien lu le sujet de celle-ci. S'en suivait une discussion qui permettait de répartir les tâches pour cette partie entre les différents membres de l'équipe.

L'utilisation d'un *repository* git a permit l'échange rapide et intuitif de code entre les membres de l'équipe, ainsi qu'un retour à des versions plus stables lorsque cela a été nécessaire.

Afin de valider une implémentation, les tests unitaires devaient tous réussir.

Des outils tels *gdb* ou *valgrind* ont permis une détection beaucoup plus rapide des bugs et des fuites mémoire.

Sur les quatre premières étapes, le travail s'effectuait par des équipes de deux personnes, travaillant chacune sur une partie de l'étape en parallèle, ce qui a permis une progression assez rapide sur ces étapes.

En ce qui concerne les deux dernières parties (5 et 6), elles ont été réalisées en parallèle en fin de semaine 3 et début de semaine 4. Durant cette période, le *repository* a été divisé en deux branches pour permettre le travail en parallèle des deux équipes.

Les derniers jours ont été exclusivement réservés à la fusion des deux branches du *repository*, la complétion du rapport, la préparation de soutenance, et celle de la démo.