Damien Caplet
Benjamin Demaziere
Enver Keka
Thibaud Humbert

Groupe P

Projet Système - NachOS

Rapport Final

Master 1 Informatique 2019 / 2020

UFR IM²AG - Janvier 2020

Table des matières

Les fonctionnalités et leurs limites

NachOS est un pseudo système d'exploitation (OS) : c'est un programme qui tourne sur un OS classique (ici un système basé sur Unix) et simule un OS basique. Cet OS possède les fonctionnalités suivantes :

- Console interactive
- Gestion multiprocessus et multithreads
- Virtualisation de la mémoire
- Système de fichiers et répertoires
- Communication via le réseau

De par sa nature d'OS à visée pédagogique, Nachos a des limites techniques très basses, que l'on peut facilement rencontrer. Voici la liste de ces limites, par domaine.

Console

Taille max d'une chaîne de caractères : 256 caractères dont le caractère de fin de chaîne, soit 255 positions disponibles

Threads utilisateurs

Nombre max de threads en parallèle : $2^{32} - 1$

Mémoire virtuelle

Taille d'une page mémoire : 128 octets
Nombre de pages mémoire : 512
Mémoire théorique totale : 64 Kbo

Système de fichiers

Taille d'un secteur : 128 octets
Nombre de plateaux : 32
Nombre de secteurs par plateaux : 32
Taille max d'un fichier : 3,75 Kbo
Nombre max de fichiers ouverts en parallèle : 10

Réseau

Nombre de réémissions : 100
Durée d'une tentative de réémission : 10000 ticks d'horloge
Taille max d'un paquet échangé : 32 octets
Nombre max de sockets sur le système : 10
Nombre de ports disponibles : 10, numérotés de 0 à 9.

Documentation

Ci-dessous, vous trouverez une liste des appels système disponibles, par domaine.

Console

La console regroupe des fonctions de lecture et d'écriture afin de pouvoir rendre les programmes utilisateurs interactifs. Toutes ces fonctions sont « thread-safe ».

char GetChar()
Rôle : renvoie un caractère lu sur l'entrée clavier
Valeur de retour : le caractère lu

void PutChar (char ch)
Rôle : affiche un caractère sur la sortie écran
Paramètres : ch => le caractère à afficher

char * GetString ()
Rôle : lit une chaîne de caractères sur l'entrée clavier, la place dans un buffer et en renvoie l'adresse dans l'espace utilisateur.
Valeur de retour : l'adresse de la chaîne dans l'espace mémoire du programme utilisateur
Attention : Cette chaîne sera toujours terminée par le caractère de fin de chaîne : '\0'. De plus, la taille maximal d'une chaîne étant fixée à 256, il reste 255 positions disponibles.

void PutString (const char string[])
Rôle : affiche une chaîne de caractères sur l'entrée clavier.
Paramètres : string => la chaîne à afficher.
Attention : Il n'y a pas d'obligation que cette chaîne soit terminée par le caractère de fin de chaîne : '\0'. Seuls les 255 premiers caractères seront affichés.

int GetInt()
Rôle : lit un entier sur l'entrée clavier et le renvoie
Valeur de retour : la valeur lue, dans un int

void PutInt (const int value)
Rôle : affiche un entier sur la sortie écran
Paramètre : value => int à afficher

Threads utilisateur

int UserThreadCreate (void * f (void *), void * arg) :
Rôle : créer un nouveau thread utilisateur.
Paramètres : f => une fonction, arg => l'argument à passer à cette fonction
Valeur de retour : Renvoie un entier qui correspond à l'identifiant du thread ou -1 si la création du thread a échoué. L'identifiant renvoyé est un nombre entre 1 et 2^{32} - 1. Il ne faut donc pas

appeler UserThreadCreate plus de 2^{32}-1 fois sinon il y a un risque d'avoir plusieurs threads avec le même id.
Attention : la taille de la pile est faible, des structures locales trop importantes peuvent faire déborder la pile.
Précision : Cet appel système peut être fait depuis le thread principal ou depuis des threads déjà créés par cet appel.

void UserThreadExit() :
Rôle : Termine le thread appelant. Puisque cet appel est fait automatiquement à la fin de la fonction passée à UserThreadCreate, il n'est pas obligatoire de le faire.
Précision : En cas d'appel par le thread principal, cette fonction ne fait rien.

void UserThreadJoin (int idT) :
Rôle : Permet à un thread d'attendre la terminaison d'un autre thread créé au sein du même processus. Un thread peut attendre la terminaison de plusieurs threads et un thread peut être attendu par plusieurs threads.
Paramètre : idT => l'identifiant du thread à joindre, fourni par UserThreadCreate.
Précision : Si le thread idT n'existe pas ou s'est déjà terminé le thread appelant continue de s'exécuter.
Attention : Aucun thread ne peut attendre le thread principal, ou attendre sa propre terminaison. L'appel système ne vérifie aucun inter-blocage.

Synchronisation : Pour gérer les problématiques de synchronisation entre les threads, les sémaphores sont disponibles pour les programmes utilisateur. Pour gérer ces sémaphores, un nouveau type est défini : sem_t pour manipuler un sémaphore via un pointeur). Voici les différentes fonctions associées à ces sémaphores.

void SemaphoreInit (sem_t * sem, int val) :
Rôle : Créé un sémaphore en précisant son nombre de jetons initial.
Paramètres : sem => le sémaphore à initialiser, val => le nombre de jetons

void SemaphoreP (sem_t * sem) :
Rôle : Prend un jeton sur un sémaphore
Paramètres : sem => le sémaphore sur lequel on souhaite prendre un jeton

void SemaphoreV (sem_t * sem) :
Rôle : Rend un jeton sur un sémaphore
Paramètres : sem => le sémaphore sur lequel on souhaite rendre un jeton

void SemaphoreFree (sem_t * sem) :
Rôle : Détruit un sémaphore, libère les ressources utilisées par ce sémaphore
Paramètres : sem => le sémaphore à détruire

Remarque générale : Le thread principal (celui qui lance la fonction « main ») attend la terminaison de tous ses threads avant de s'arrêter.

Mémoire virtuelle

int ForkExec (char * string) :
Rôle : crée un nouveau processus et lui fait exécuter un programme donné en paramètre
Paramètres : string => nom du programme à lancer par le nouveau processus
Valeur de retour : 0 si tout s'est bien passé

Système de fichiers

int UserMkdir (char * dirName) :
Rôle : permet de créer un nouveau répertoire en précisant son nom
Paramètre : dirName => une chaîne de caractères contenant le nom du répertorie à créer
Valeur de retour : 0 en cas de succès, 1 sinon

int UserChdir (char * dirName) :
Rôle : déplace le répertoire courant dans un de ses sous-répertoires actuellement existants, en précisant son nom
Paramètre : dirName => une chaîne de caractères contenant le nom du sous-répertoire cible
Valeur de retour : 0 en cas de succès, 1 sinon

int UserRmdir (char * dirName) :
Rôle : supprime un répertoire s'il existe dans le répertorie courant, en précisant son nom
Paramètre : dirName => une chaîne de caractères contenant le nom du répertorie à supprimer
Valeur de retour : 0 en cas de succès, 1 sinon

void UserListdir() :
Rôle : Affiche le contenu du répertoire courant

int UserMkFile (char * fileName, int size) :
Rôle : crée un nouveau fichier, en précisant son nom et sa taille
Paramètres : fileName => une chaîne de caractères contenant le nom du fichier à créer, qui ne doit pas exister dans le répertoire courant, size => la taille du fichier à créer
Valeur de retour : 0 en cas de succès, 1 sinon

int UserRmFile (char * fileName) :
Rôle : supprime un fichier dans le répertoire courant, en précisant son nom
Paramètres : fileName => une chaîne de caractères contenant le nom du fichier à supprimer, existant dans le répertoire courant.
Valeur de retour : 0 en cas de succès, 1 sinon

int UserOpenFile (char * fileName) :
Rôle : ouvre un fichier présent dans le répertoire courant, en précisant son nom
Paramètres : fileName => une chaîne de caractères contenant le nom du fichier à ouvrir, existant dans le répertoire courant.
Valeur de retour : l'identifiant du descripteur de fichier qu'on vient d'ouvrir, 1 sinon
Attention : Il doit rester de la place dans la table de fichiers du système

void UserCloseFile (int fd) :
Rôle : ferme un fichier actuellement ouvert, en précisant son descripteur de fichier associé
Paramètres : fd => le descripteur de fichier qu'on souhaite fermer.

Réseau

int SocketCreate (socket_t * socket) :
Rôle : création d'un nouveau socket client
Paramètres : socket => un socket vide
Valeur de retour : 0 si la création a échoué, 1 sinon

int SocketConnect (socket_t * socket, int adr, int port) :
Rôle : connexion d'un socket client à un socket serveur
Paramètres : socket => un socket rempli par SocketCreate, adr => l'adresse du serveur, port => le port d'écoute du serveur.
Valeur de retour : 0 si la connexion a échoué, 1 sinon

int SocketServerCreate (socket_t * socket, int port) :
Rôle : créer une socket serveur
Paramètres : socket => un socket vide, port => le port local d'écoute
Valeur de retour : 0 si la création a échoué, 1 sinon

int SocketAccept (socket_t * socket, socket_t * clientResponse) :
Rôle : met le socket serveur donné en état d'accepter la connexion d'un socket client
Paramètres : socket => un socket serveur rempli par SocketServerCreate, clientResponse => un socket vide
Valeur de retour : 0 si la création a échoué, 1 sinon
Attention : Ne supporte pas plusieurs demandes de connexion en parallèle

int SocketSend (socket_t * socket, char * data, int size) :
Rôle : le thread appelant tente d'envoyer des données
Paramètre : socket => un socket rempli par SocketAccept ou SocketCreate, data => un buffer contenant les données à envoyer, size => le nombre max d'octets à envoyer
Valeur de retour : 0 si l'envoi a échoué, taille des données envoyées sinon

int SocketReceive (socket_t * socket, char * data, int size) :
Rôle : le thread appelant tente de recevoir des données
Paramètres : socket => un socket rempli par SocketAccept ou SocketCreate, data => un buffer pour y placer les données reçues, size => le nombre max d'octets à recevoir
Valeur de retour : 0 si la réception a echoué, taille des données récupérées sinon

void SocketClose (socket_t * socket) :
Rôle : termine une connexion sur un socket
Paramètres : socket => socket rempli par SocketAccept, SocketCreate ou SocketServerCreate

Tests fonctionnels

Voici les fichiers de test montrant le fonctionnement de chaque fonctionnalité, par domaine.

Console

Un fichier pour chaque fonctionnalité : putchar.c, getchar.c, putstring.c, getstring.c, getint.c, putint.c
Cas particulier : putint.c remonte un warning à la compilation car on l'appelle avec une valeur qui dépasse ce qu'on peut mettre dans un int.

Threads utilisateur

Voici les tests montrant l'utilisation des threads utilisateurs, à lancer avec l'option -rs <x> :
makethreads.c : création de 5 threads en parallèle
threadsjoin.c & thread_join2.c : création de 2 threads supplémentaires et attente de leur terminaison
thread_pile.c : création de 2 threads qui vont remplir la pile (attention : le comportement de l'application n'est pas garantie quand la pile déborde)
thread_creer_thread.c : création d'un thread qui va en créer un autre à son tour, synchronisation par jointure depuis le main
thread_sansExit.c : création d'un thread qui se termine sans appeler UserThreadExit.

Mémoire virtuelle

forkexecsimple.c : création de deux processus mono-thread
forkexecmultiple.c : création de 10 processus créant chacun 5 threads qui écrivent tous en même temps dans la console

Système de fichiers

fstest.c : teste les appels systèmes de base des répertoires
10filestest.c : création et tentative d'ouverture de plus de 10 pour tester la table des fichiers

Réseau

simple<Client||Serveur>.c : création d'un socket, envoi d'un paquet puis fermeture du socket
plusieursPaquets<Client||Serveur>.c : échange de plusieurs paquets de données entre le serveur et le client
mult<Client||Serveur>.c : demande de connexions successives vers un serveur et échange de données.

Choix d'implémentation

Console

La console est en fait une synchconsole basée sur la console de base. La spécification suit celle fournie pour la réalisation de l'étape 2 du projet. Cependant, les fonctions de lecture GetString et GetInt adoptent une approche similaire au GetChar. Cela nous a paru être une approche plus cohérente et plus intuitive.

Threads utilisateur

On garde une liste de threads actifs dans l'espace d'adressage (addrspace).
Pour réaliser UserThreadCreate, on ajoute dans la liste le thread qui vient d'être créé. Puis on alloue trois pages pour sa pile, pour cela on dispose d'un bitmap qui garde l'état d'utilisation des pages et on récupère trois pages consécutives. On met la valeur du registre RetAddrReg à l'adresse de UserThreadExit, cela permet d'appeler UserThreadExit à la fin de l'exécution du thread sans que l'utilisateur ait à le faire.

Pour réaliser UserThreadJoin, chaque thread dispose d'une liste de sémaphores. On récupère le thread dans la liste à partir de l'idT en paramètre. S'il existe, on crée un sémaphore, le thread appelant bloque sur le sémaphore via la procédure P() et on ajoute ce sémaphore à la liste du thread idT. Quand le thread idT appelle UserThreadExit, il fait V() sur tous les sémaphores de sa liste, et permet aux threads appelant de continuer leur exécution.

Pour réaliser UserThreadExit, on desalloue le thread dans l'addrspace. Pour que le thread main termine quand il a fini ses opérations et que tous les autres threads sont terminés, on a un sémaphore dans l'addrspace, quand il ne reste plus que le thread main on fait V() sur le sémaphore, le thread main peut donc se terminer.

Mémoire virtuelle

C'est une variable globale qui compte le nombre de processus lancés sur la machine. Lorsqu'un processus veut arrêter son exécution, il se met en attente sur une barrière de synchronisation en incrémentant une variable comptant le nombre de processus en attente et en désallouant ses ressources. La barrière est levée lorsque le nombre de processus en attentes sur la barrière est égal aux nombre de processus lancés sur la machine, c'est à dire lorsque tous les processus ont terminé leur exécution, et que l'on peut donc éteindre la machine sans risque.

Système de fichiers

Nous avons rajouté dans chacun des répertoires des entrées «.» et «..» leur permettant d'accéder à leur père et à eux-mêmes en permanence. Le répertoire courant est sauvegardé dans une variable du système de fichiers. Une table des fichiers ouverts a été ajouté pour gérer les différents fichiers ouverts en parallèle.
Pour augmenter la taille maximale des fichiers, nous nous sommes inspirés du système d'Unix et ses inodes. Nous avons fait le choix de placer dans les FileHeader un tableau de 28 entrées dont : 20 sont des allocations directes, 6 sont doublements indirectes, et 2 sont des allocations avec triple indirection.

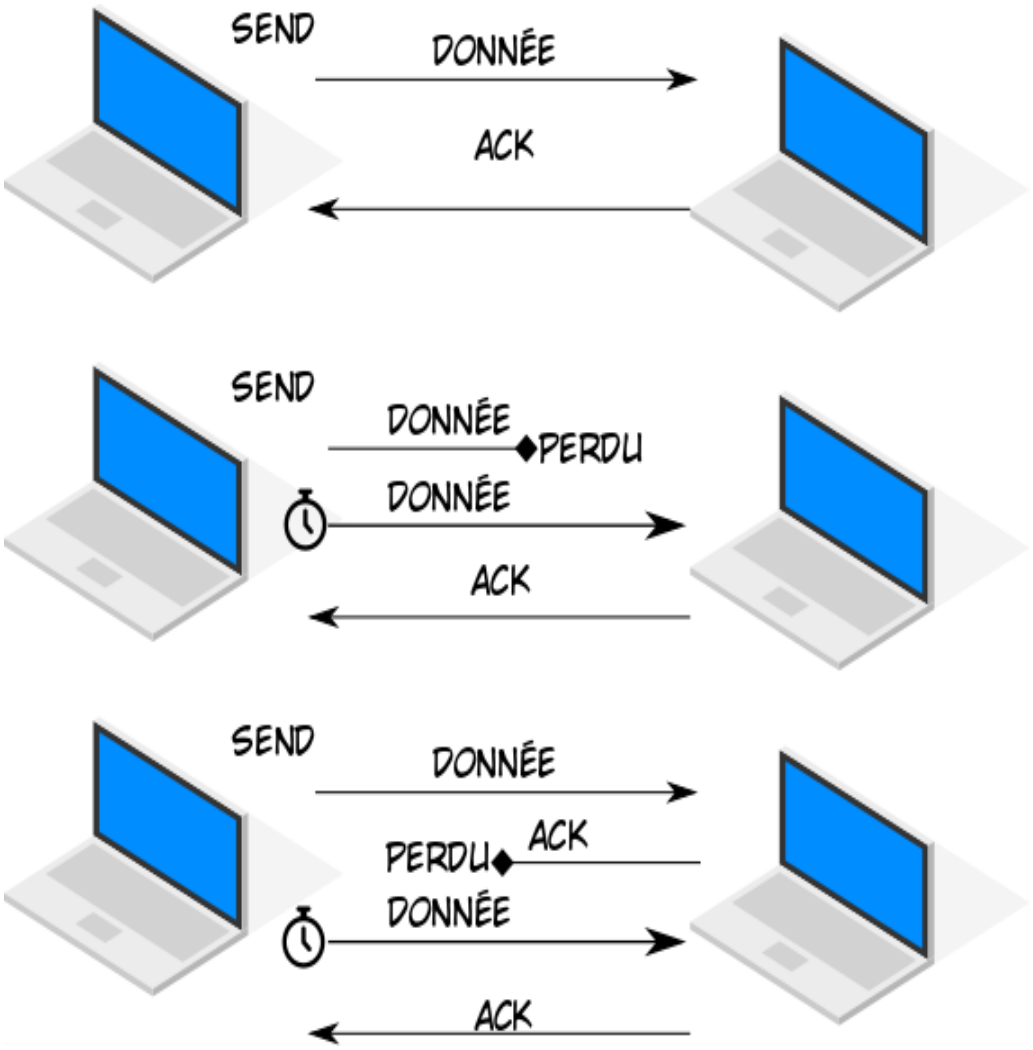
Les chemins d'accès ont également été implémentés : ceux-ci nous déplacent cependant directement dans le répertoire visé.

Réseau

On a créé une surcouche Socket sur la classe PostOffice. Elle comporte deux classes :

- SocketClient : permet de se connecter au serveur, d'envoyer et recevoir des données et de fermer la connexion.
- SocketServer : permet d'attendre des connexions entrantes et crée une SocketClient quand connexion est ouverte.

A chaque envoi, on attend un acquittement de la part du destinataire, si on ne le reçoit pas on renvoie le paquet.



Organisation

Planning – Historique

06/01 : Début de l'étape 1. Installation Nachos et création des dépôts Git.
07/01 : Suite et fin de l'étape 1.
08/01 : Début de l'étape 2.
09/01 : Suite et fin de l'étape 2.
10/01 : Début de l'étape 3 partie 1.
13/01 : Suite de l'étape 3 partie 1.
14/01 : Fin de l'étape 3 partie 1, début partie 2. Début de l'étape 4 partie 1, en parallèle.
15/01 : Suite de l'étape 3 partie 2. Fin de l'étape 4 partie 1, début partie 2. Début des tests
16/01 : Fin de l'étape 3. Début d'écriture du Pre-Rapport.
17/01 : Fin de l'étape 4. Rendu Pre-Rapport et consolidation du code.
20/01 : Début des étapes 5 partie 1 et 6 partie 1, en parallèle.
21/01 : Suite des étapes 5 et 6 partie 1. Consolidation des tests.
22/01 : Fin des étapes 5 et 6 partie 1, début partie 2.
23/01 : Début du travail sur la démo et la présentation.
24/01 : Fin de l'étape 5 partie 6. Début de rédaction du Rapport à partir du Pre-Rapport.
27/01 : Suite de la rédaction du Rapport. Début de la rédaction de la présentation.
28/01 : Fin des étapes 5 et 6 partie 2.
29/01 : Consolidation finale du code. Suite de la rédaction Rapport et présentation.
30/01 : Fin de la rédaction et impression du Rapport. Fin de rédaction de la présentation
31/01 : Préparation et passage de la Présentation finale.

Répartition des tâches

Le développement a été fait en commun jusqu'à la fin de l'étape 2. A partir de là, Damien a été chargé de la seconde partie de l'étape 3 pendant que Thibaud et Enver s'occupaient de la première partie de l'étape 3 puis de l'étape 4. Benjamin a pris en charge la rédaction des différents écrits ainsi que de la mise en place des tests permettant de prouver le bon fonctionnement de ce qui est implémenté au fur et à mesure. La même répartition a été reconduite pour la réalisation en parallèle des étapes 5 (Thibaud et Enver) et 6 (Damien). Pour la présentation finale, le choix le plus logique était de faire échoir à une personne la présentation de ce qu'il avait fait. La rédaction de la présentation a donc été éclatée dans un premier temps puis réunifiée au final.