Rapport Projet NachOS
M1 Informatique

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

Janvier 2015

1 questionnement sur le contenu/la forme du rapport

-> utilité de dire que nous aons dût modifier les fonctions de lecture/écriture en mémoire.
-> possibilité d'un process qui se crée, mais le dernier process se termine avant que le compteur est été incrémenté ? -> le process en cour de lancement ne se lancera jamais ?
-> faire des schema, notament pour la mémoire virtuelle et les id des threads
-> test de la partie process

Table des matières

2 Fonctionnalités du noyau

Voici les partie que nous avons réussi à implémenter jusque là :

— (aprtie 2) appel system : toutes les fonction d'écriture et de lecture ont été portées au niveau utilisateur et sont 'thread safe'. Nous avons aussi utilisé des sémaphores afin de permettre aux fonction PutString et GetString d'afficher/lire une chaine entière avant qu'une autre thread puisse écrire/lire à l'écran. Il n'y a pas de blocage entre la lecture et l'écriture.

— (partie 3) threads : toutes les fonctions permettant de créer, détruire et gérer des threads on été porté au niveau utilisateur.

— (partie 4) mémoire virtuel : le système de mémoire virtuel à été implémenté. La fonction ForkExec a été portée au niveau utilisateur.

— (partie 5) système de fichier : Il est maintenant possible de créer/supprimer des fichiers et des dossiers et de se déplacer dans l'architecture de dossier au niveau utilisateur. L'architecture est sauvegarder d'une éxécution à la suivante.

— (partie 6) réseaux :

pour toutes ces parties, nos tests n'ont révélé aucun problème. Les tests effectué sont détaillé dans la section qui leur dédiés.

3 Spécification des fonctions utilisateur

3.1 Console

SIGNATURE : `char GetChar()`

DESCRIPTION : Lis un caractère depuis l'entrée standard et le retourne sous forme de `char`. Retourne `EOF` dans la 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 chaine de caractères lue vers le buffer pointé par `s`. La longueur maximale d'une chaine 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 chaine présente dans le buffer pointé par `s` ainsi qu'un saut de ligne (`\n`) sur la sortie standard. L'écriture de la chaine 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 chaine 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 régèles 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, et est considéré comme "fils" de celui-ci.
Lorsqu'un thread termine (même le thread `main`), il attend la terminaison de tous ses "fils".

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'éxecution et libère son emplacement dans la pile. Elle attend la fin de tous les threads fils de ce thread avant de terminer son exécution. Si c'est le dernier thread du processus, ce dernier se termine aussi 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 revient 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 le l'identifiant du thread en cours d'exécution. Cet identifiant n'est unique que dans le contexte de son processus.

3.3 Mémoire virtuelle

SIGNATURE : `int ForkExec(char *path)`

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

VALEUR DE RETOUR : Identifiant du processus créé 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`, `-4` : Impossible de créer un environnement d'exécution.
- `-3` : Pas assez de pages libres en mémoire pour créer l'espace d'adressage du processus.

SIGNATURE : `int GetPid()`

DESCRIPTION : Renvoie le l'identifiant du processus en cours d'exécution dans le contexte de l'exécution en cours.

3.4 Système de fichiers

SIGNATURE : `int mkdir(..)`

DESCRIPTION :

VALEUR DE RETOUR :

`-1` :

3.5 Réseau

SIGNATURE : `int Send(..)`

DESCRIPTION :

VALEUR DE RETOUR :

`-1` :

`-2` :

`-3` :

4 Test

4.1 Organisation

Pour automatiser les tests, nous avons décidé de créer un script qui va lancer nos fichier de test au travers du projet. Nous avons essayer de faire suffisamment de tests pour tester tous les défauts possible du programme, ils sont répartis en une série de fichiers séparé par étape du projet dans des dossiers correspondant au numéro de l'étape. Tous ces dossiers sont situé dans code/test.

4.2 Utilisation

Pour lancer les tests, ils suffit d'exécuter le script testpart.sh situer dans le dossier code/test

4.3 Tests effectués et comportements

4.3.1 Console

- une chaine de 1 caractères validé : écrit le caractère correctement à l'écran ou bien dans le fichier de destination
- utilisation de la console validé : la console réécrit tout ce qui est tappé correctement
- ecriture/lecture parallèle validé : possibilité de faire des caractère entrelacé avec putchar ou bien des string continue avec putstring
- trop de caractères validé : sépare la chaine écrire en plusieurs chaines de taille équivalante à taillemax (situer dans le fichier .cc)
- pas de caractère validé : si rien n'est entrée, rien ne se passe et si on entre juste le caractère retour chariot, il est traité comme n'importe quelle lettre
- arrêt avec EOF validé : arrete la lecture de fichier.

4.3.2 Thread

- création d'un thread :
  test de création courante validé : le thread se crée
  tester toutes les possibilités d'échecs validé : retourne la bonne erreur
  très grand nombre de threads validé : crée le threads normalement jusqu'au moment ou l'on dépasse le nombre maximum de threads. A partir ce cette limite, la fonction retourne une erreur.

— arreter un thread avec Exit :
vérifier les fuites mémoires validé : toutes les structures allouées sont bien supprimées
plus dans l'ordonanceur validé : l'ordonnanceur n'essaie plus de donner la main à la thread
état du parent validé : le parent n'a plus le thread courant dans sa liste et continue de touner correctement
état des enfants validé : on attend la fin des enfants avant de supprimer la thread courante. Ce choix est détaillé dans la partie "choix d'implémentation".
— lancement d'une fonction validé : testé avec les fonctions disponible, elles sont effectuées correcetement
— retours d'erreurs validé : quand les mauvais argument sont passé à la fonction ou bien qu'il n'y a plus assez de place pour créer un thread, la fonction renvoie bien le bon code d'erreur.

4.3.3 Mémoire virtuelle

— créer un processus : validé : le processus est créé correctement et la fonction passée en paramètre démarre normalement.
— stress de la mémoire : validé : les fichiers matmult et sort s'éxécutent sans problèmes.
— allocation et libération de pages : validé : les pages sont alloué et libérée correctement. les pages ne sont pas donnée alors qu'elles sont utilisé et ne sont libérée qu'à la fin des processus.
— caractères entrelacé avec processus : ? ? ?
— retours d'erreurs : validé : les bonnes valeur sont retourné en cas de mauvais arguments ou de manque de place.

4.3.4 système de fichier

créer à partir d'appel système ! ! !
— créer un dossier/fichier : validé : Le dossier/fichier est créer et est accessible.
— déplacement vers un dossier parent/fils : validé : On peux se déplacer à volonter dans les dossiers.
— suppression dossier/fichier : validé : Le dossier/fichier est supprimé correctement. Les pages sont libéré et il ne réapparait pas lors de la prochaine lecture du disque.
— enregistrement de la structure : validé : La structure s'enregistre correctement et est rechargée sans erreur notable au début de l'éxécution de la machine.

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, nous avons due créer 2 appel systèmes et quelques fonctions utilisateurs. Les appels système appelent les descriptions des fonctions utilisateurs. Cette section étant très guidée dans le sujet, nous n'avons eu aucun choix d'implémentation particulier. Nous avons sécurisé des lecture/écriture pour le multithreading. les fonctions de lectures (GetInt, GetChar et GetString) ne peuvent plus s'intérompre les une les autres et les fonctions d'écritures (PutChar, PutInt et PutString) ne s'intérompent plus les unes les autre non plus. Mais les fonctions d'écriture peuvent intérompre les fonction de lecture et inversement.

5.2 Threads

5.2.1 La structure thread

— id : son id, permetant d'acceder à son adresse mémoire
— liste de tout ses fils : utilisé pour faire une join sur les fils au moment de UserThreadExit

5.2.2 Gestion des identifiants de threads et de la pile

Deux identifiants :
— un identifiant unique : unique parmi tous les threads du processus.

— un identifiant en pile : pour les processus actifs.

La pile est divisée en un nombre de blocs déterminé par le nombre de pages par thread. 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 : N étant le bloc ayant l'adresse de pile la plus petite, et 0 le bloc ayant l'adresse la plus grande. Un thread demande un bloc en pile à sa création, et le libère lorsque 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 dernier n'est pas alloué (négatif, -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.

5.2.3 Implementation du join

structures : vecteur séquentiel contenant la liste des threads en attente et l'identifiant des threads qu'ils attend

— élement who : Thread qui attend

— élement forId : identifiant du thread que who doit attendre

dès l'appel à join du thread d'id x, on ajoute la paire (thread courant, x) à la liste des threads en attente et le thread courant appel la fonction Sleep. Elle l'enlevera de la liste d'attente du scheduler pour ne pas le réveiller inutilement, c'est le thread sur lequel le join a été effectué qui reveillera ce thread lors de son appel a UserThreadExit

5.3 Mémoire virtuelle

5.3.1 Pagination mémoire

le frame provider, utilise une bitmap et toutes les fonction implémentées pour celle-ci. La fonction de recherche d'un bit vide nous est utilisé pour aller chercher une page vide et son adresse. Et la fonction retournant le nombre de bit a 0 est utilisé pour avoir le nombre de page libre. La fonction libérant les pages utilise la fonction des bitmaps pour libérer le bit correspondant dans la bitmap.

nous avons pu remarqué qu'avec la façon dont nous avons implémenté notre table de page : ppn (physical page number) = vpn (virtual page number) + 128 (taille d'une page) -> ceci est une faille de sécurité car elle permet de faire d'aller écrire à des endroit où l'on ne devrait pas pouvoir. -> cette implémntation permet tout de même d'avoir tous les autres avantages des pages virtuelles

Lorsque quelque chose demande de la mémoire (création d'un process, enregistrement d'un fichier...), la bitmap nous donne une adresse puis l'adresse de la page est donée au programme pour qu'il puisse y inscrire ce dont il a besoin.

Lors de la fin du programme, la(les) page(s) est déclarée(s) vide et le bit correspondant est remit à 0.

5.3.2 Implémentation de Fork et ForkExec

Ces deux fonctions sont inspiré de StartProcess. Elles font appel à l'appel system `do_UserProcessCreate`. ce dernier va ouvrir l'executable, lui allouer de la mémoire (un addrspace). Il va ensuite créer le thread et y attacher la mémoire allouée. Il va enfin mettre à jour le nombre de processus créés.

/!/ForkExec le thread créé sur StartUserProcess/!/

La différence entre ForkExec et Fork est que ForkExec ne mets pas la mémoire à la même adresse que celle du thread en cours.

Avant de se fermer, la machine attend la fin du dernier process. Pour savoir si le processus actuel est le dernier, la machine garde un compteur du nombre de processus en cours de fonctionement, qui est incrémenté quand un

process est créer et qui est décrémenté quand un process se termine. Le dernier process est donc celui qui met le compteur a 0 en se terminant.

5.4 système de fichier

la position dans les dossier n'est pas sauvegarder entre deux éxécutions

5.5 réseaux

Une tentative d'envoi ne se termine que lorsqu'il un message reçoit son acquittement ou qu'il ait été renvoyé (TEMPO*MAXREEMISSIONS) fois. ON ne peu attendre de message d'acquitement que pour un seul message à la fois. Pour ce faire, on utikisse un deamon qui est bloqué au départ, puis lancé dès que l'on tente d'envoyer un message puis rebloqué lorsque la tentative se fini. le type du message est précisé dans son header. Il peux s'agir soit d'un message normal (MSG), soit d'un message d'acquitement (ACK) la faiblesse de ce système est que les sygnaux d'acquitement peuvent être perdu facilement. Par exemple si une machine s'éteint en ayant envoyé un ACK qui sera perdu, l'expéditeur retentera d'envoyer le message à l'infini. pour transferer les fichiers, un premier paquet est envoyé contenant la taille du fichier à envoyer. puis l'envoie est fragmenté en plusieurs paquets de MaxStringSize si la taille du fichier supérieure à cette limite utilisateur. Le récépteur met tous les paquet dans un buffer dont la taille est définie par le contenu du premier paquet, pusi écrit le contenu du buffer dans un fichier.

6 Organisation du travail

6.1 Fonctionement du groupe

Malik : code/debug/docs
Amine : code/docs
Sebastien : code/docs
Jean-Yves : organisation/docs/tests

6.2 Planning

— semaine 1 :
  mise en place du projet
  nous avons commencé par faire la partie 1 chacun de notre coté,
  puis nous avons tous travaillé ensemble sur les parties 2 puis 3.
— semaine 2 :
  cette semaine, nous avons commencer par finir la partie 3
  puis nous avons enchainé sur la partie 4.
— semaine 3 :
  Malik : partie 5
  Amine : partie 6
  Sebastien : débug partie 4 et 5
  Jean-Yves : travail de mise en page et avancement du rapport
— semaine 4 :
  lundi/mardi : fin des partie 5 et 6 et debug du reste
  mercredi : fin du rapport (mise en page + schema + correction de contenu) et préparation du contenu de la présentation
  jeudi : entrainement pour la présentation
  vendredi : D-day