Pré-Rapport Projet NachOS
M1 Informatique

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

Janvier 2015

Table des matières

1 Fonctionnalités du noyau

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

— appel system : toutes ces fonction on été porté 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

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

— mémoire virtuel : work in progress (ce sera peut-etre fini d'ici la date de rendue, mais je ne sais pas si nous auront le temps d'en parler ici)

Nous somme actuellement entrain de finr de débugger dette dernière partie.

2 Spécification des fonctions utilisateur

2.1 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 RENVOYÉE : 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 RENVOYÉE : 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`.

2.2 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 RENVOYÉE : 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 RENVOYÉE : `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.

2.3 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 RENVOYÉE : 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 Test

3.1 Organisation

Pour automatiser les tests, nous avons décidé de créer un script qui va lancer nos fichier de test à travers du projet. Nous avons essayer de faire suffisamment de tests pour tester tous les défaut possible du programme, ils sont répartit en une série de fichiers séparé dans des dossiers. Tous nos programme de test son situé dans le dossier code/test. Ils sont classé par étape avec un dossier par étape.

3.2 Utilisation

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

3.3 Tests effectués et comportements

3.3.1 Appel système via synchconsole

— 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.

3.3.2 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

3.3.3 Virtual memory

—

4 Choix d'implémentation

4.1 Appels système

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. sécurisation des lecture/écriture pour le multithreading *à décrire*

4.2 Threads

4.2.1 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

4.2.2 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. Pour stocker ces identifiants, nous avons utilisé un tableau d'identifiants de pile où chaque case contient un booléen représentant si l'espace mémoire associé dans la pile est utilisé par un thread (true) ou pas (false). Un thread demande un espace dans la pile à sa création, et le libérera lors de l'appel à do_UserThreadExit. L'identifiant en pile d'un thread sera le numéro de la case qui lui sera attribuée dans ce tableau. Il est possible de calculer l'emplacement de la pile d'un thread dans l'espace d'adressage à partir de son identifiant en pile, et un tableau dont la clé est l'identifiant unique, et le contenu l'identifiant en pile + 2 (x) où si.

— x=0 veut dire qu'aucun thread portant cet identifiant unique n'a jamais été créé.
— x=1 veut dire qu'un thread portant cet identifiant unique a été créé, mais s'est terminé.
— x=2+ veut dire que le thread ayant cet identifiant unique est en cours d'exécution, et sa pile est dans le bloc (x - 2)

4.2.3 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

4.3 Mémoire virtuelle

5 Organisation du travail

5.1 Fonctionement du groupe

5.2 Planning