TACOS

Julien ALAIMO, Hugo FEYDEL, Olivier HUREAU

2019-2020

1 Fonctionnalités intéressante/importante

TACOS (The Amazing Control Operating System) est un système d'exploitation incomplet mais robuste : il dispose d'un système de multithreading complet et d'un système de fichier opérationnel. Il est également livré avec les fonctionnalités basiques d'entrées/sorties (cf. 2.1).

1.1 Le Multithreading

Le multiThreading permet à un utilisateur de faire tourner plusieurs instructions en parallèle. Similaire au processus dans un environement linux, les threads partagent le même espace mémoire.

Notre réalisation permet une utilisation simpliste et permissive du multithreading : si certaines mécaniques de fermeture sont oublié, le système le fera par lui même.

1.2 Le Système de fichier

Le système de fichier fourni permet à l'utilisateur de se déplacer à sa guise dans l'arborescence, de créer/supprimer des répertoires et des fichiers, d'ouvrir des fichiers, de lire leur contenu ou le modifier.

Il est possible d'utiliser des noms de chemins (absolus comme relatifs) et les répertoires spéciaux "." (répertoire courant), ".." (répertoire parent) et "/" (répertoire racine).

Attention, ce système de fichier opère sur des fichiers de tailles fixes uniquement.

2 Spécifications

2.1 Entrées/Sorties

- **void PutChar(char c);**

  Ecrit le caractère **c** sur la sortie standard

- **void PutString(char * string);**

  Ecrit la chaîne de caractères **string** sur la sortie standard. La chaîne de caractères doit finir par '\0'. La taille maximale de la chaîne de caractères est définie par la constante **MAX_STRING_SIZE** (cf. "system.h").

  L'appel système **PutString** est moins coûteux que plusieurs appels système **PutChar**.

- **int GetChar();**

  Retourne la valeur ascii d'un caractère rentré dans l'entrée standard. La fonction attend qu'un caractère soit disponible, elle est bloquante.

- **void GetString(char * string, int taille);**

  Ecrit la chaîne de caractères rentrée dans l'entrée standard à l'adresse de la chaîne de caractères passée en paramètre **string**. La taille de cette chaîne de caractère sera inférieure ou égale au paramètre **taille**. Si celle de la chaîne rentrée lui est supérieure, la chaîne obtenue en sera une troncature.

- **void GetInt(int * n);**

  Ecrit l'entier rentré dans l'entrée standard à l'adresse pointée par le paramètre **n**. L'entier peut être positif comme négatif. La valeur absolue de l'entier ne doit pas être supérieure à

  $$10^{10} - 1$$

  (valeur absolue). Sinon une erreur est levée.

- **void PutInt(int n);**

  Ecrit l'entier en paramètre **int n** sur la sortie standard L'entier peut être positif comme négatif. La valeur absolue de l'entier ne doit pas être supérieure à

  $$10^{10} - 1$$

  (valeur absolue). Sinon une erreur est levée.

2.2 Threads

Le programmation multithreadé permet à l'utilisateur d'éxecuter plusieurs threads simultanément. Les threads partagent la mémoire virtuelle et l'utilisation du coeur. Chaque thread possède néanmoins son propre fil d'exécution, ses propres registres (dont le pointeur d'instrcution PC) et sa pile.

Tout les threads sont identifié par des identifiants unique stocké sur des unsigned int. A la mort de chaque thread, lors de la mort de son parent, l'identifiant du thread enfant est retransmis au système de gestion des identifiants afin qu'il soit recyclé. Le thread principal ayant pour identifiant '1', si la mémoire le permet, il est possible d'utiliser jusqu'à $2^{32} - 1$ thread simultanément mais aussi possible d'avoir une infinité de thread ayant était éxecuté sur la machine d'un point de vue temporel.

Dans le système tacOS il existe un lien de parenté entre tout les threads. Chaque thread créé est enfant du thread créateur. A la fin de l'exécution d'un thread dis ***parent*** , celui-ci va attendre la fin de l'exécution de tout ses fils.

Cependant, un mode ***survivant*** est mis en place tel que lorsqu'un un thread est dis ***survivant***, il ne seras pas attendu pas son thread père met par son grand père.

Voici les différentes procédures liéé aux threads :

- **unsigned int CreateThread(void * f (void * arg), void * arg;**

  Crée un thread utilisateur et lance la routine **f**. Le deuxième argument **void * arg** est passé en argument a la fonction f. Retourne l'identifiant du thread.

- **void ExitThread(void * obj)**

  Termine le thread courant et injecte l'adresse de l'objet dans les structures l'adresse **void * obj** afin qu'un thread parent puisse l'utiliser. Doit être appelé lorsque l'on désire terminer un thread "fils". *Le système possède néanmoins un mécanisme permettant de rattraper cet oubli*

- **unsigned int ThreadId();**

  Retourne l'identifiant du thread courant.

- **void * WaitForChildExited(int CID);**

  Attente passif que le thread d'identifiant **CID** termine. Retourne une adresse correspondant à l'objet donné par le thread enfant lors de son exit. Si cette adresse est équivalente à **(void *) -1**, alors l'identifiant **CID** n'as pas était reconnu où le thread n'est pas enfant.

- **void WaitForAllChildExited();**

  Met le thread en attente actif. Le thread sort de l'attente quand plus aucun de ses threads enfants sont en cours d'execution (y compris les thread en mode survivant)

- **int StopChild(int CID)**

  Met en pause le thread enfant d'identifiant **CID**. La fonction retourne :

  - **0** Si succès
  - **1** Si l'arrêt à échoué
  - **2** Si l'identifiant n'est pas celui d'un de ses enfants

  Si le thread est en zone critique (Ecriture /lecture) dans la conssole, le thread s'arretera après cette lecture/ecriture.

- **int WakeUpChild(int CID);**

  Reveil le thread enfant d'identifiant **CID**. La fonction retourne :

  - **0** Si succès
  - **1** Si l'identifiant n'est pas celui d'un de ses enfants
  - **2** Si le thread n'est pas en pause

- **int makeChildSurvive(int CID);**

  Met le thread enfant d'identifiant **CID** en **mode survivor** jusqu'à la mort du père .

- **void makeAllChildSurvive();**

  Met tout les threads enfant en **mode survivor** jusqu'à la mort du père courant.

2.3 Système de fichiers

- Fonctions de navigation/création/suppression sur l'arborescence :

  Les fonctions **Create()** et **Remove()** opèrent depuis le répertoire courant.

  - **bool Create(const char *name, int initialSize, File_type type = f);**
    Crée un fichier ou un répertoire selon le type **type** (f pour fichier, d pour répertoire, fixé à f par défaut), de nom **name** et de taille **initialSize**. Retourne vrai si succès, faux si le fichier/répertoire existe déjà ou si l'espace disponible est insuffisant.
  - **bool Remove(const char *name, unsigned int tid = 0);**
    Supprime un fichier de nom **name** du thread d'identifiant **tid**.
    Retourne vrai si succès, faux si le fichier n'existe pas.

  Les fonctions **OpenFromPathName()**, **MkdirFromPathName()** et **RmdirFromPathName()** utilisent la fonction **CdFromPathName()** afin de se placer dans le bon répertoire, effectuer leurs opérations, et enfin de se replacer dans le répertoire avant les changements.

- **path_parse_t * CdFromPathName(const char *path_name, unsigned int tid = 0, int truncate = 0);**

  Change le répertoire courant avec celui de nom du dernier élément de **path_name** du thread d'identifiant **tid** et retourne un pointeur vers une structure **path_parse_t**.

  Exemple :

  CdFromPathName("/folder1/folder2") vous placera dans "folder2" si la racine contient "folder1", NULL sinon.

- **bool MkdirFromPathName(const char* path_name, unsigned int tid = 0);**

  Crée un répertoire de nom du dernier élément de **path_name** du thread d'identifiant **tid** dans le répertoire de nom de l'avant dernier élément de **path_name**.

  Retourne vrai si succès, faux si chemin incorrect ou si le répertoire existe déjà ou si l'espace disponible est insuffisant.

  Exemples :

  MkdirFromPathName("folder1/folder2/file1") renverra false.

  MkdirFromPathName("folder2/folder3") créera "folder3" dans "folder2" en admettant que le répertoire courant le contienne.

  MkdirFromPathName("/folder3") créera "folder3" à la racine.

- **bool RmdirFromPathName(const char* path_name, unsigned int tid = 0);**

  Supprime un répertoire de nom du dernier élément de **path_name** du thread d'identifiant **tid** dans le répertoire de nom de l'avant dernier élément de **path_name**.

  Retourne vrai si succès, faux si le répertoire n'existe pas ou si le répertoire n'est pas vide.

- Fonctions utilisateurs :

  - **int UserOpen(const char *name, unsigned int tid);**

    Ouvre un fichier de nom du dernier élément de **name** du thread d'identifiant **tid** et retourne l'entier du descripteur de fichier associé (-1 si échec).

  - **int UserRead(int fileDescriptor, char *into, int numBytes, unsigned int tid);**

    Lit **numBytes** octets depuis le fichier décrit par **fileDescriptor** et les écrit dans **into**.

  - **int UserWrite(int fileDescriptor, char *from, int numBytes, unsigned int tid);**

    Ecrit **numBytes** octets depuis **from** dans le fichier décrit par **fileDescriptor**.

– **void UserSetSeek(int fileDescriptor, int position, unsigned int tid);**
  Change la position du curseur dans l'entête du fichier décrit par **fileDescriptor** par **position**.
– **int UserCloseFile(int fileDescriptor, unsigned int tid);**
  Ferme le fichier décrit par **fileDescriptor**.

3 Tests Utilisateurs

une partie "tests utilisateurs" décrivant les programmes de test que vous avez réalisés, ce qu'ils montrent, ...

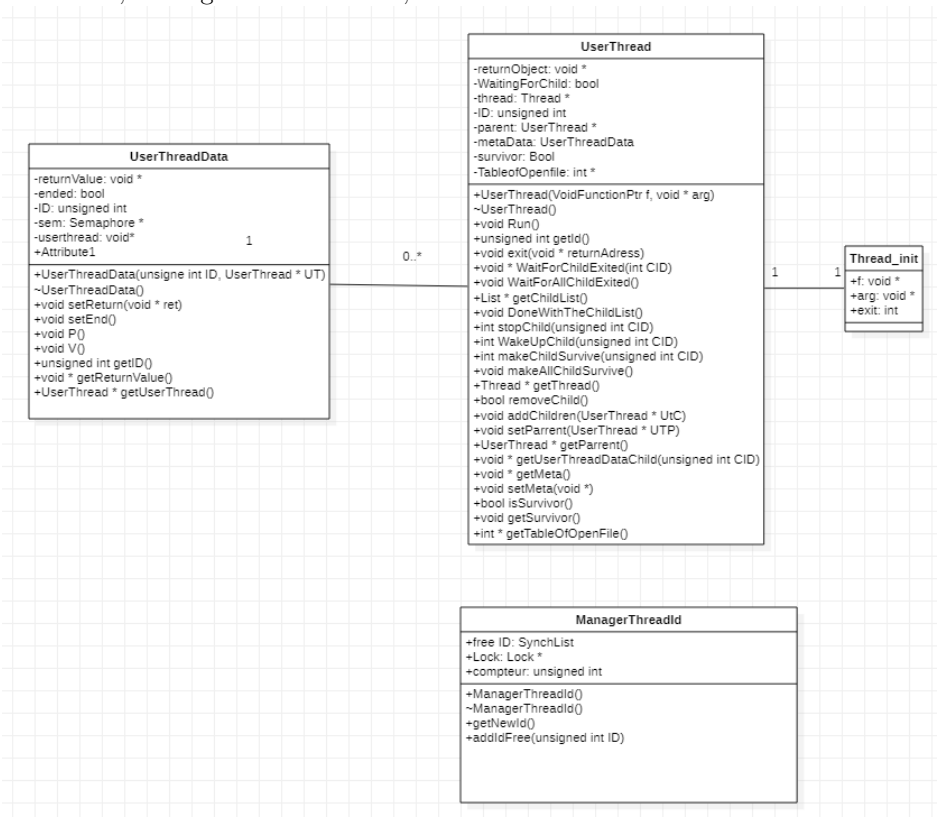
3.1 Threads

- Le test de la class ***List*** *(que nous avons modifié)* se lance avec l'argument -tl. Ce test va vérifier si : Un liste retourne le bon boolean lors de l'appel de IsEmpty(), les éléments ont bien était ajouté dans le bon ordre, les éléments ont bien était supprimé dans le bon ordre, la fonction ajouté get(index) fonctionne correctement. *voir testImplem.cc*
- Le test de la class **UserThreadManagerId** se lance avec l'argument -utmmono. Ce test va vérifier si : Le manager commence au bon endroit, les identifiants sont recyclés, un identifiant recyclé est donné en priorité par rapport à un nouvel indentifiant. *voir testImplem.cc*
- **CreateAndReturn.c** : Ce test vérifie qu'un Thread se crée correctement et que sa valeur de retour est juste.
- **ExitWithoutStatement.c** : Ce test vérifie que même lorsqu'un thread n'utilise pas d'appel de retour (return ou ExitThread), celui-ci se termine correctement.
- **MultiGetChar.c** : Ce test vérifie la spécification de la console and mode multithreadé : Plusieurs getString sont lancé en même temps, on vérifie que l'execution de celle-ci est bien de manière séquentiel.
- **Survive.c** : Ce test vérifie que l'implémentation du mode survivor fonctionne correctement. Le fils doit écrire sur le terminal avec que le petit fils ait finis d'écrire.
- **WaitForThread.c** : Ce test vérifie que les parents attendent bien que leurs fils se terminent (un seul père, 2 fils)
- **WaitForThreadMulti.c** : Ce test vérifie que les parents attendent bien que leurs fils se terminent (un seul père, 1 fils, 1 petit fils)
- **Sleep.c** : Ce test vérifie qu'un thread père peux bien mettre en attente un fils : Le fils est programmé pour ecrire en continu sur la console, le StopThread() arrète donc bien l'enfant de parler.
- **Wake.c** : Ce test vérifie père peux bien réveiller son fils.

4 Implémentation

4.1 Threads

Pour implémenter les threads, nous avons choisis d'utiliser les 3 classes suivantes : UserThread, ManagerUserThreadID,UserThreadData



L'adresse d'un UserThread est stocké dans la class Thread, Ainsi, le UserThread du currentThread est acessible via currentThread->getAdressOfUserThread(). En ayant accès au UserThread, on a donc les informations nécessaire sur le thread tel que : Les UserThreadData de tout ses fils, le UserThread du père, l'identifiant est toutes autres informations nécessaire.

Nous avons choisis de créer une nouvelle classe afin de gagner en lisibilité au niveau du code. De plus, cet implémentation est certes gourmandes en mémoire mais permet d'effectuer des tâches plus rapidement. Il aurais était possible d'utiliser un arbre de UserThread mais la recherche du UserThread correspondant à chaque thread aurais était beaucoup trop longue.

La création du Thread utilisateur se fait alors par le biai du constructeur de la classe UserThread. Lors de la construction un "new Thread()" est appelé.

Le terminaison des threads sans fonction de retour ou ExitThread() se fait par l'appel du syscall ExitThread stocké dans le registre RA (31) de chaque thread dès son initialisation.

La destruction des classes Thread se fait lors de l'appel d'un WaitChildExited(unsigned int CID). Lors de cette appel, le père essaye de prendre un sémaphore du UserThread fils (token rajouté lors du ExitThread() , récupèrer l'adresse de retour du fils et détruire les différentes classes.

L'assignation des identifiants d'UserThread se fait via le UserThreadManager.

4.2 Système de fichiers

- Le Système de fichiers est organisé de la manière suivante :
  - Une table globale des fichiers ouverts sous forme de liste chaînée (cf. **global_file_table_t**).
  - Une table des fichiers ouverts par thread sous forme de liste chaînée (cf **file_table_t**). Les 2 premiers élément de l'OpenFileTable de chaque thread sont dédiés aux répertoires racine (indice 0) et courant (indice 1).

```
typedef struct global_file_table {
    OpenFile *openFile;
    struct global_file_table *next;
} global_file_table_t;
```

```
typedef struct file_table {
    unsigned int tid;
    char *path;
    OpenFile** OpenFileTable;
    struct file_table *next;
} file_table_t;
```

- La fonction "CdFromPathName" découpe le nom de chemin par le séparateur "/" en tableau de chaînes de caractères (stockée dans **pathSplit**) afin de le parcourir répertoire par répertoire (en changeant le répertoire courant) jusqu'à (**size - truncate**) et retourne un pointeur vers une structure **path_parse_t** (cf. ci-après).

```
  typedef struct path_parse {
    char** pathSplit;
    int size;
} path_parse_t;
```

  Exemple : En admettant que "folder1" existe dans le répertoire courant, CdFromPathName("folder1/folder2/file1", 1) vous placera dans "folder2"

et renverra un pointeur vers une structure suivante avec les champs suivants : pathSplit = [[folder1] [folder2]] ; size = 2.

5 Scolaire

une partie plus "scolaire" où vous décrivez l'organisation de votre travail (planning, ...), commentaires constructifs sur le déroulement du projet, ...