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

En cours de développement

- **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 à 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. Cette fonction doit absolument être utiliser à la fin de l'exécution d'un thread (mise à part les utilisateurs avancés, 2.3

- **unsigned int ThreadId();**

  Retourne l'identifiant du thread courant.

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

  Attente passive de la terminaison du thread d'identifiant **CID**. Retourne une adresse correspondant à l'objet donné par le thread enfant lors de son exit. Si cette adresse est équivalente ¡trouver une valeur¿, alors l'identifiant **CID** n'as pas été reconnu ou le thread n'est pas enfant. Cette fonction détruit le thread système. Il faut donc veiller à l'utiliser.

2.3 Threads mode avancé

- **void WaitForAllChildExited();**

  Met le thread en attente active. Le thread sort de l'attente quand plus aucun de ses threads enfants ne sont en cours d'execution

- **int StopChild(int CID)**

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

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

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

  Réveille 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**

- **void makeAllChildSurvive();**

  Met tous les threads enfant en **mode survivor**

2.4 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

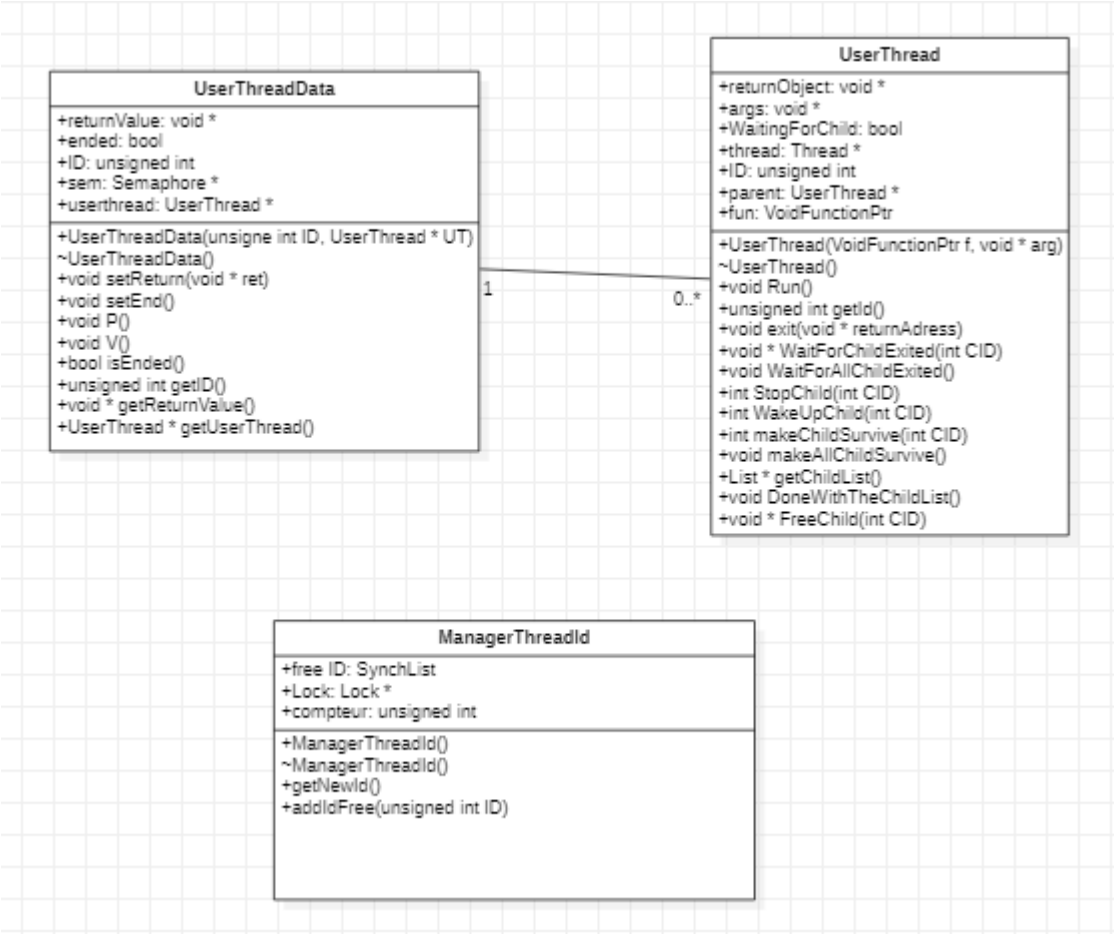
Ici je veux bien que t'explique vite faire que la CI execute ton script, ce que fait ton script. J'écrirais les tests pour step2 et threads.

4 Implémentation

4.1 Threads

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

- UserThread
- ManagerUserThreadID
- UserThreadData



Ainsi, chaque thread utilisateur sera associé à un thread kernel grâce au stockage de l'adresse d'un UserThread dans la classe Thread du Kernel.

Lorsqu'un appel système se déclenche, via le pointeur machine-¿currentThread où se trouve un pointeur vers son UserThread, on pas alors utiliser les UserThread pour y appliquer les différentes opérations nécessaires.

L'assignation des identifiants d'UserThread se fait via le UserThreadManager. Le userThreadManager stocke une variable compteur qui est incrémentée à chaque demande d'identifiant. UserThread étant stocké sur des "unsigned int", on peut se voir limiter en nombre de thread parallèle par la limite de taille d'un unsigned int. Cependant, les identifiants sont recyclés. À chaque destruction de UserThread, on réinjecte alors son identifiant dans la liste du ThreadManager, il est alors réutilisable. Néanmoins cela implique une plus grosse structure de stockage et une éventuelle perte de temps par rapport à d'autres structures (exemple : bitmap)

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, ...