

TACOS

Julien ALAIMO, Hugo FEYDEL, Olivier HUREAU

2019-2020

1 Fonctionnalités intéressante/importante

Le système a été pensé simple d'utilisation mais avec certaines possibilités pour aller plus loin. En effet, il nous tenait à cœur d'avoir un environnement de développement permissif où certaines erreurs habituelles n'arrêteront pas l'exécution du processus. Nous avons alors décidé que le système fonctionnera permissivement pour un utilisateur non avancé. Cependant pour un utilisateur avancé, il est possible d'utiliser autrement le système d'exploitation, à ses risques et périls.

1.1 Le Multithreading

D'après notre motivation, nous avons décidé de penser autrement les threads : un mode classique et un mode avancé. Dans le mode classique on retrouvera les mécanismes classiques des threads (création de threads, exit de threads et récupération d'adresse pour "retourner" un objet.

Le mode avancé n'est pas recommandée pour les personnes ne maîtrisant pas bien le langage tacOS. Il peut engendrer des deadlocks ou fuites de mémoires. Ces outils restent néanmoins à disposition pour les utilisateurs avancés

Initialement, lors de l'exit d'un thread parent une interruption système va être enclenchée pour attendre que tous les threads enfants soient finis afin d'éviter des deadlocks ou autres. Il sera néanmoins possible par la suite de ne pas suivre le système. En effet, il est possible de transformer des threads en "**mode survivor**".

Lorsque le thread est en "**mode survivor**", celui-ci ne va pas mourir avant son père mais survivre. Il va par la suite être adopté par le thread parent de son ancien parent. Cette manipulation est impossible lorsque le thread parent est le thread principal du programme.

2 Spécifications

une partie "spécifications" listant ce qui est disponible pour les programmes utilisateurs. Il faut mettre ici le genre d'information que vous trouvez dans les pages man. On doit donc trouver tous les appels systèmes implémentés avec leur prototype, la description des arguments, la description du fonctionnement (fonctionnalités utilisateurs, pas implémentation) de l'appel système, de la valeur de retour éventuelle, la signalisation des erreurs, ... Si vous avez également une bibliothèque utilisateur, vous devez décrire ses fonctions de la même manière que les appels systèmes.

2.1 Entrées/Sorties

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

Ecris le caractère "char c" sur la sortie standard

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

Ecris la chaîne de caractère "char * string" sur la sortie standard. La chaîne de caractère doit finir par '\0'. La taille maximale de la chaîne de caractère est de **MODIFIE ICI STP**. 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. Attention il peut y avoir blocage.

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

Ecris la chaîne de caractère rentrée dans l'entrée standard à l'adresse de la chaîne de caractère en paramètre **char * string**. La taille de cette chaîne de caractère sera inférieure ou égale au paramètre **int taille**. **JULIEN MODIFIE ICI, c'est peut être faux.**

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

Ecris l'entier rentrée 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);**

Ecris 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 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 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 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 trouver une valeur, alors l'identifiant **CID** n'as pas était reconnu où 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 actif. Le thread sort de l'attente quand plus aucun de ses threads enfants sont en cours d'exécution
- **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
- **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**
- **void makeAllChildSurvive();**
Met tout les threads enfant en **mode survivor**

2.4 Système de fichiers

JULIEN AND HUGO

3 Tests Utilisateurs

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

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.

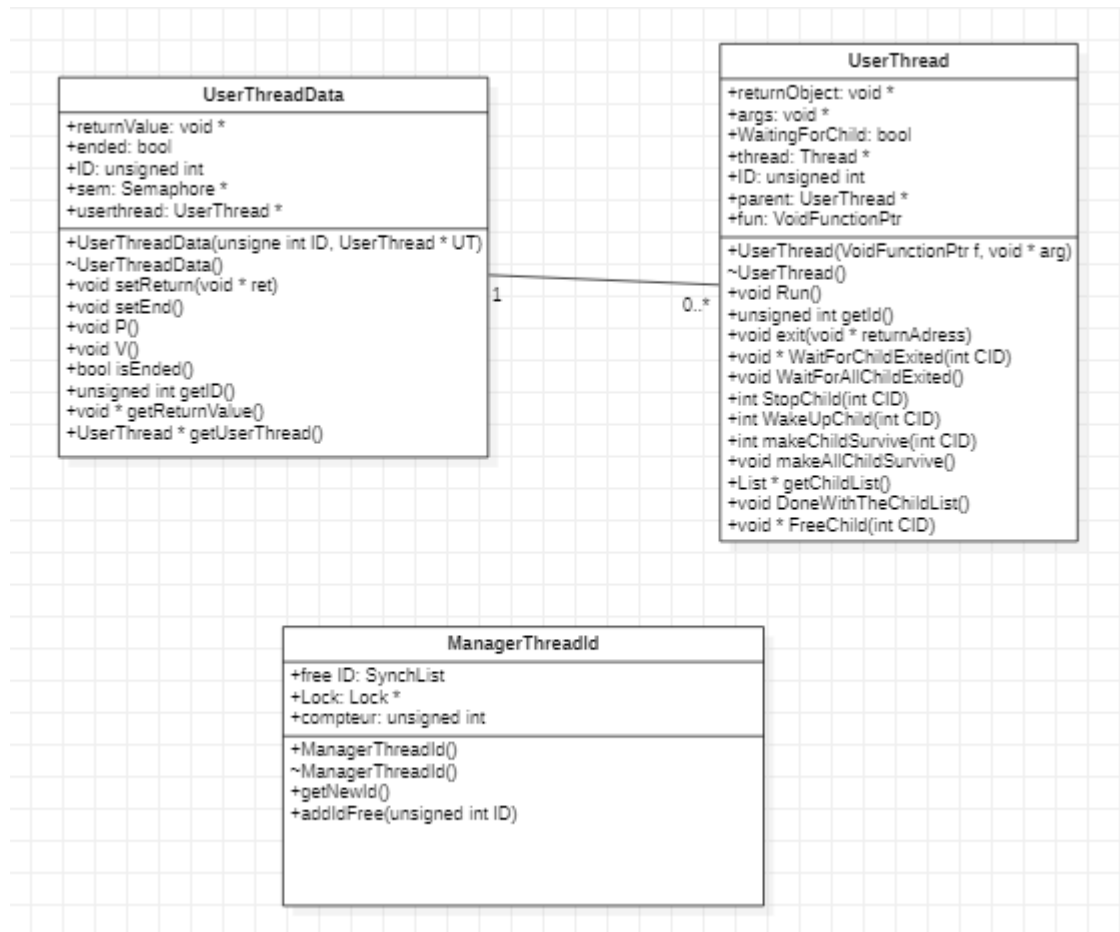
4 Implémentation

une partie "implémentation" qui explique les points importants de votre implémentation. C'est donc la seule partie qui parle du détail du code que vous avez écrit. Expliquez vos choix d'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` ou se trouve un pointeur vers son UserThread, on ne peut 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. C'est 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. Cela permet alors d'être sûr qu'un programme à longue utilisation ne se retrouve jamais à court d'identifiant. Néanmoins cela implique une plus grosse structure de stockage et une éventuelle perte de temps par rapport à d'autres structures (exemple : bitmap)

5 Scolaire

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