TACOS

Julien ALAIMO, Hugo FEYDEL, Olivier HUREAU

2019-2020

1 Fonctionnalités intéressante/importante

Le système a était pensé simple d'utilisation mais avec certaines possibilités pour aller plus loin. En effet, il nous tenais à cœur d'avoir un environnement de développement permissif où certaines erreurs habituels n'arrêterons pas l'exécution du processus. Nous avons alors décidé que le système fonctionnerai 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

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 du multithreading d'une façon permissive : si certaines mécaniques de fermeture sont oublié, le système le fera lui même. A contrario, il est possible pour un utilisateur avancé de manipuler plus en détail ses structures.

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 charactère "char c" sur la sortie standard

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

  Ecris le chaine de caractère "char * string" sur la sortie standard. La chaine de caractère doit finir par '\0'. La taille maximal de la chaine de caractère est de **MODIFIE ICI STP**. L'apel 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 charactère soit disponnible. Attention il peux y avoir blocage.

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

  Ecris la chaine de caractère rentrée dans l'entrée standart à l'adresse de la chaine de caractère en paramètre **char * string**. La taille de cette chaîne de caractère sera inférieur ou égal au paramètre **int taille**. **JULIEN MODIFIE ICI, c'est peut etre faux**.

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

  Ecris l'entier rentrée dans l'entrée standart à l'adresse pointé par le paramètre **n**. L'entier peux être positif comme négatif. La valeur absolue de lentier ne dois pas êre supérieur a

  $$10^{10} - 1$$

  (valeur absolue). Sinon une erreur est levée.

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

  Ecris l'entier en paramètre **int n** sur la sortie standard L'entier peux être positif comme négatif. La valeur absolue de l'entier ne dois pas êre supérieur a

  $$10^{10} - 1$$

  (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'execution 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 ¡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'execution

- **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.

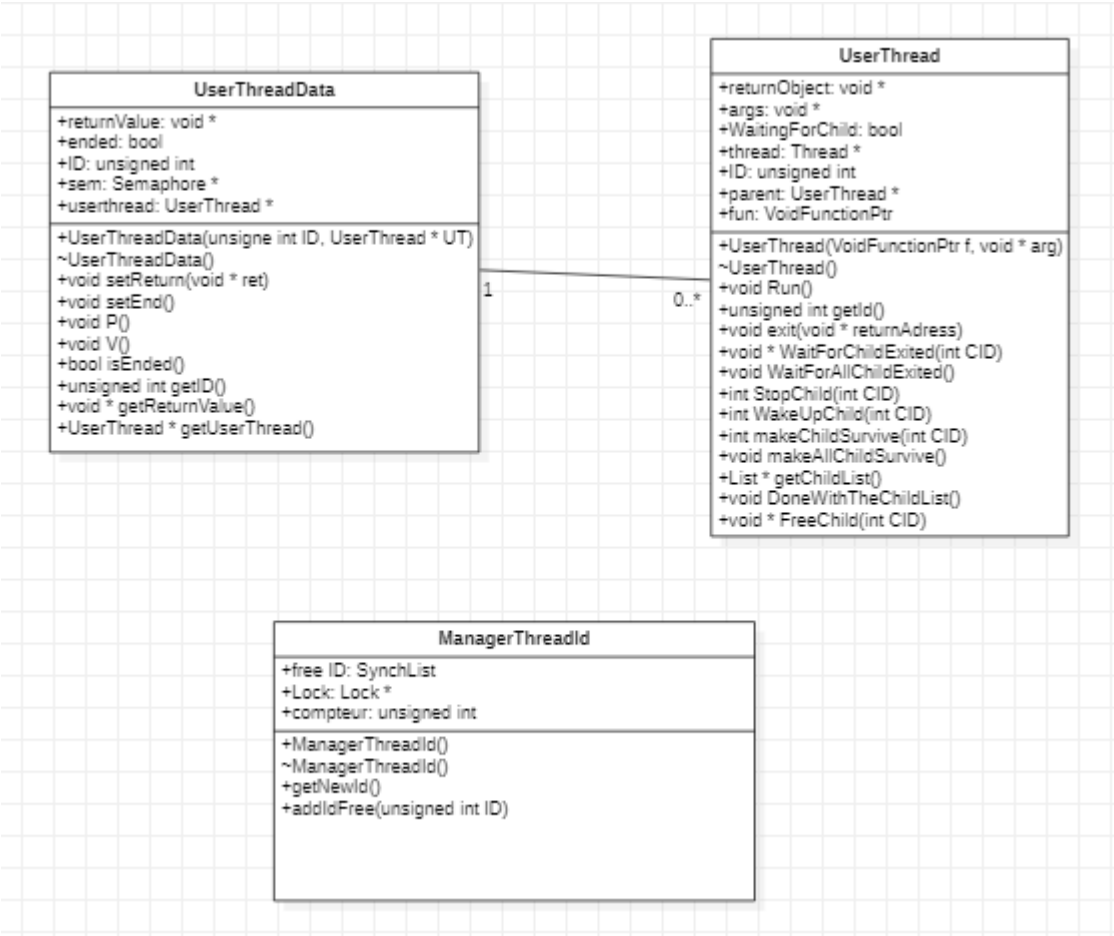
4 Implémentation

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 kerbel 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 pas alors utiliser les UserThread pour y appliquer les différentes opérations nécessaire.

L'assignation des identifiants d'UserThread se fait via le UserThreadManager. Le userThreadManager stock un variable compteur qui est incrémenter à chaque demande d'identifiant. C'est UserThread étant stocké sur des "unsigned int" on peux se voir limiter en nombre de thread parallèle par la limite de taille d'un unsigned int. Cependant, les identifiants sont recyclé. A chaque destruction de UserThread on réinjecter alors sont identifiant dans la liste du ThreadManager. Cela permet alors d'être sur qu'on programme à longue utilisation ne se retrouve jamais à cours d'identifiant. Néanmoins cela implique une plus grosse structure de stockage et une éventuelle perte temps par rapport à d'autre structure (exemple : bitmap)

5 Scolaire

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