API IENAC-Threads

***Mots-clés** : multitâche, coopératif, préemptif, thread, verrou, pile d'exécution*

On souhaite développer sous UNIX une API (Application Program Interface) offrant la création et la gestion de threads.

Etape 1

Le modèle multitâche retenu est le multitâche **coopératif**, c'est à dire que chaque thread choisira à quel moment il souhaite libérer le processeur au profit d'un autre thread. L'ordonnancement des threads se fera à l'intérieur d'un même processus.

Une priorité est attribuée à chaque thread au moment de sa création. Au moment d'élire un thread, l'ordonnanceur choisira le thread en tête de la file d'attente la plus prioritaire (gestion en FIFO pour chaque niveau de priorité). Les valeurs de priorité vont de 1 à 5, 1 étant la plus prioritaire.

A- Services offerts par l'API

t_create() **Création d'un nouveau thread.**

Doit permettre de passer des paramètres au thread et précise sa priorité.

t_exit() **Terminaison d'un thread.**

Appelée par le thread qui se termine. Permet de récupérer la valeur de retour du thread.

t_yield() **Suspension d'un thread**

Appelée par le thread qui cède ainsi la main à l'ordonnanceur.

t_wait() **Attente de la terminaison d'un thread**

Permet de récupérer la valeur de retour du thread.

t_lock_create() **Création d'un verrou (mutex)**

Offre une possibilité de synchronisation entre threads. On donnera un nom à chaque verrou créé.

t_lock() **Attente sur un verrou.**

Cède la main à l'ordonnanceur.

t_unlock() **Libère un verrou**

Débloque une tâche en attente sur un verrou.

t_lock_destroy() **Destruction d'un verrou**

B- Travail demandé

- Etude technique

  Vous présenterez l'algorithme des fonctions ainsi que les structures de données utilisées en expliquant vos choix.

- Dossier de tests

  Vous donnerez la liste des tests nécessaires à la validation de votre code

- Codage

  Vous réaliserez ces fonctions ainsi les programmes de test permettant de valider leur fonctionnement.

  Un fichier d'entête contiendra le prototype des fonctions ainsi que la définition des variables globales nécessaires à la mise en œuvre de la gestion multithread.

  Le code devra être lisible facilement (nom des fonctions et variables explicites, commentaires)

Etape 2

Le modèle multitâche coopératif, ne permet pas une gestion équitable du processeur (cf. cours de Systèmes d'Exploitation). Un ordonnanceur **préemptif** doit pouvoir décider quand une tâche doit céder la main à une autre.

On va choisir dans cette étape de rendre notre gestion multithread préemptive. L'ordonnancement se fera toujours selon les priorités, mais un thread qui n'invoque pas la fonction yield() au bout d'un quantum de temps défini devra céder la main au profit d'une autre. Ce quantum de temps sera une valeur définie dans notre API.

Travail demandé

Vous réaliserez l'étude technique de la solution (choix techniques et algorithmes).

Mécanismes utilisés

On veut mettre en œuvre une gestion de threads dans l'espace utilisateur. On n'utilisera dons pas les mécanismes existants (bibliothèque pthread), mais il est vivement conseillé de s'inspirer de leur fonctionnement !

Les fonctions sigsetjmp() et siglongjmp() permettent respectivement de sauvegarder le contexte d'exécution d'une fonction et de reprendre son exécution. On les utilisera pour effectuer les commutations de contexte.

Manipulation de la pile d'exécution

```
int sigsetjmp(sigjmp_buf env, int value) ;
```

Sauvegarde l'environnement d'exécution (état de la pile : sommet de pile, registres courants, …) dans un buffer, et si *value* est non nul, sauvegarde le masque de signaux courants. La valeur de retour de cette fonction est 0 au premier appel (sauvegarde), et sinon (appel suivant) prend la valeur du paramètre *value2* de la fonction siglongjmp().

```
int siglongjmp(sigjmp_buf env, int value2) ;
```

Permet de se débrancher du contexte d'exécution courant (saut non local) et de continuer l'exécution à l'endroit préalablement sauvegardé par sigsetjmp() dans la variable *env*.

Attention !

- Le buffer *env* doit avoir été initialisé par l'appel à sigsetjmp(), les valeurs de pile placées au-dessus de l'environnement repris sont perdues.
- L'environnement doit encore exister dans la pile au moment de l'appel, sinon le résultat du saut sera indéterminé.