Programmation multi-threads

Objectif de la séance

Nous avons vu en cours que la gestion des threads se fait en utilisant des structures de données abstraites c'est qui n'exposent pas leur contenu, mais juste leurs méthodes. L'accès aux attributs de ces structures se fait par des fonctions d'accès. Toutefois avant d'utiliser ces structures, nous allons faire une gestion "naïve" et simplifiée des threads. Dans cette version:

- les structures sont ouvertes,
- il n'y a qu'une seule politique d'ordonnancement (FIFO),
- les threads ne peuvent être joints (on ne peut pas récupérer leur valeur de retour)

Définitions

thread

- Un thread peut être dans 5 états (vous ferez une énumération) :
  - CREATE : état de démarrage attribué au thread lorsqu'il n'a jamais été chargé sur un core.
  - READY : état dans lequel le thread est en attente du core.
  - RUN : état en cours d'exécution.
  - WAIT : état dans lequel le thread est en attente d'une ressource.
  - DEAD : état de sortie dans lequel le thread peut être détruit.

- La structure thread_t est définie par:
  - spinlock_t lock; // *protection contre les accès concurrents*
  - state_t state; // *CREATE, READY, ...*
  - unsigned cpuid; // *numéro du core affecté au thread*
  - list_t list; // *utilisé pour le chainage des threads dans les listes d'attente*
  - list_t rope; // *chaine tous les threads quelque soit leur état*
  - unsigned pws[CONTEXT_SIZE]; // *context*

- Les fonctions de manipulation des threads sont:
  - typedef void * thread_fct_t (void *); *Thread function type*
  - extern thread_t * thread_create (unsigned cpuid, thread_fct_t *start, void *arg);
  - extern unsigned thread_save (thread_t * thread);
  - extern void thread_restore (thread_t * thread);
  - extern void thread_exit (void * exit_value) attribute((noreturn));
  - extern void * thread_idle (void * arg);

  La fonction thread_create utilise malloc pour créer la structure thread et la pile du thread.

scheduler

- La structure schedroot_t:
  - spinlock_t lock; // *protection contre les accès concurrents*
  - thread_t * run; // *thread en cours d'exec*
  - thread_t * idle; // *thread idle*
  - list_t dead; // *liste des threads à détruire*
  - list_t ready; // *liste des threads en attente du core*

- Les fonctions de manipulation du scheduler sont:
    - extern void sched_init (void); // *initialise un scheduler*
    - extern void sched_create (thread_t *thread); // *ajoute un nouveau thread*
    - extern void sched_wakeup (thread_t *thread); // *remets un thread dans l'état READY*
    - extern void sched_yield (void); // *demande d'ordonnancement*
    - extern void sched_sleep (void); // *place le thread dans une liste d'attente*
    - extern void sched_exit (void); // *termine un thread (ici on ne récupère pas la valeur)*

- Le graphe ci-dessous illustre le comportement des fonction sur l'état des threads

mutex

- Afin que tous les états de threads soient utilisés, il est nécessaire d'ajouter une ressource qui pourra être partagée par plusieurs threads. La plus simple est le mutex qui est une variable à deux états: 0 pour libre, 1 pour occupé.
  La structure mutex_t est définie par:
    - spinlock_t lock; // *protection contre les accès concurrents*
    - int state; // *état du mutex*
    - list_t wait; // *liste d'attente des threads ayant demandé le mutex alors que celui-ci était occupé.*

- Les fonctions d'accès:
    - extern int mutex_init (mutex_t *mutex); // *initialise un mutex dont on passe le pointeur (pas de malloc)*
    - extern int mutex_lock (mutex_t *mutex); // *prend le verrou s'il est libre ne met en attente sinon*
    - extern int mutex_unlock (mutex_t *mutex); // *rend le verrou et réveille un des threads en attente s'il y en a un.*

Travail demandé

Vous allez créer plusieurs fichiers:

- scheduler.h + scheduler.c : les prototypes et fonctions du scheduler
- thread.h + thread.c : les prototypes et fonctions des threads + thread_idle()
- mutex.h + mutex.c : les prototypes et fonctions des mutex
- app.c : les threads de votre application

Je vous propose une application, mais vous pouvez en imaginer une autre plus intelligente. N threads identiques de type 1 qui se partagent un compteur commun protégé par un mutex. Un thread de type 2 va recevoir les compteurs de chaque instance des threads de type 1 et vérifier qu'il ne reçoit pas deux valeurs identiques (vous faites comme vous voulez pour faire cette vérification, mais vous pouvez profiter du fait que les valeurs envoyées par les threads de type 1 sont strictement croissantes...). Le compteur est une variable globale initialisée à 0. Chaque instance de thread de type 1 boucle sur la séquence suivante:

- prend le mutex
- prend la valeur du compteur
- rend le mutex
- attend un nombre aléatoire de cycles
- envoi de la valeur du compteur vers le thread de type 2

Pour faire l'envoi des valeurs de compteur, vous allez avoir besoin d'un moyen de communication N vers 1. Je vous laisse spécifier et implémenter ce moyen.