TP MI03 - Réalisation d'un mini noyau temps réel (Parties 3 et 4).

Les tâches ont besoin d'échanger des données pour coopérer au sein d'une application. La façon la plus simple et la plus rapide de transmettre des données est de ne pas les transmettre, et donc d'utiliser des zones de données communes, par exemple des espaces de mémoire partagées entre les tâches, définis au moyen de variables globales ou alloués par des requêtes spécifiques du noyau.

Problèmes posés par le partage d'objets partagés

On suppose deux tâches A et B qui produisent des données devant être éditées sur une imprimante. L'emploi de l'imprimante par la tâche A exclut bien entendu son utilisation par la tâche B. Dans le cas contraire, les données imprimées seraient mélangées et en conséquence, incompréhensibles.

On qualifie d'**accès concurrents** des situations où **plusieurs tâches lisent et écrivent des données partagées et où le résultat dépend de l'ordonnancement des tâches**.

On remarque dans l'exemple que le problème peut être résolu de façon simple si l'accès à un objet par une tâche interdit son accès par d'autres tâches.

3ème partie : exclusion mutuelle

Pour éliminer les accès concurrents, il suffit de trouver un moyen d'interdire l'accès simultané à une donnée/ressource partagée. Les parties du programme qui conduisent à un conflit sont appelées **sections critique.** On appelle **l'exclusion mutuelle** le moyen de régler le problème d'accès concurrent.

L'ensemble des règles régissant l'accès à une section critique est donné par [Tanenbaum 87] :

1. Une section critique ne doit être occupée que par une seule tâche à la fois,
2. Aucune hypothèse ne doit être faite sur la vitesse relative et le nombre de tâches en exécution,
3. Aucune tâche suspendue en dehors d'une section critique ne doit en bloquer une autre,
4. Une tâche doit attendre un temps fini devant une section critique.

Il existe plusieurs techniques pour réaliser l'exclusion mutuelle :
- Masquage des interruptions, utilisé dans le noyau
- Variables verrou et attente active
    - Simples
    - Avec avertissement
- Instructions de type « Test and Set »

Question : expliquez les avantages et inconvénients des différentes techniques décrites ci-dessous pour réaliser l'exclusion mutuelle.

Dans tous les cas, ces techniques conduisent à l'utilisation d'une boucle d'attente active, qui présente l'inconvénient de consommer inutilement du temps processeur, et présente un risque d'interblocage dans le cas d'un noyau gérant des tâches de priorités multiples. Pour éviter cette attente active, on se propose d'ajouter un état supplémentaire aux tâches : **l'état suspendu (SUSP)**.

Une tâche peut passer de l'état EXEC à SUSP au moyen de la primitive *dort()*, et être réveillée (donc passer de l'état SUSP à EXEC) au moyen de la primitive *reveille()*.

On vous demande d'ajouter au mini-noyau temps réel la gestion de la suspension et du réveil d'une tâche. Vous réaliserez les deux primitives suivantes :

void dort(void) : endort (suspend) la tâche courante

void reveille(uint8_t tache) : réveille la tâche *tache*. Le signal de réveil n'est pas mémorisé si la tâche n'est pas suspendue.

Vous écrirez un petit programme de test mettant en œuvre ces deux primitives dans le cas du modèle de communication producteur/consommateur. Le programme devra comporter au moins deux tâches ; la première, le producteur, produira des entiers courts dans une file circulaire, la seconde, le consommateur, retirera ces entiers de la file et les affichera.

Pour la gestion de la file, on pourra s'appuyer sur les fichiers **fifo.c** et **fifo.h** disponibles sur Moodle (archive distribuables_tp2.tgz), qui représentent une implémentation minimaliste de file circulaire.

Question : implémentez les deux primitives de gestion de l'exclusion mutuelle dans le fichier **noyau.c** et un programme de test associé.

Question : expliquez en quoi votre programme de test permet de valider le bon fonctionnement du mécanisme d'exclusion mutuelle mit en place. Un screen de fonctionnement de votre noyau est attendu pour valider votre explication.

4ème partie : sémaphores

Dans la partie 3 du TP, on a vu que la résolution des conflits d'accès à des ressources partagées était un problème complexe et difficile à régler sans plus d'outils. Les sémaphores offrent une solution générale à ces problèmes.

Définition

Les sémaphores sont proposés pour la première fois en 1965 par Dijkstra. Il s'agit de compter le nombre de tâches endormies (ou de réveils en attente) à l'aide d'une variable appelée **sémaphore**. Un sémaphore *s* est composé de :

- un compteur *e(s)*, à valeur entière positive, négative ou nulle ;
- une file *f(s)*, permettant de stocker les tâches en attente sur le sémaphore.

Il est muni de deux primitives d'accès, *P(s)* et *V(s)*, qui permettent respectivement de prendre et de libérer une ressource associée au sémaphore. Toutes les opérations sur un sémaphore sont supposées indivisibles (ou *atomiques*).

P(s) réalise les actions suivantes :

- décrémenter *e(s)*
- si e(s) < 0 alors
    - bloquer la tâche ayant effectué la requête et mettre celle-ci dans la file *f(s*)

V(s) réalise les actions :

- incrémenter *e(s)*
- si e(s) <= 0 alors
    - sortir une tâche de la file *f(s)* et la relancer.

D'autre part, il est créé par une primitive permettant de fixer une valeur initiale du compteur **positive ou nulle.**

Principe de fonctionnement et propriétés

On peut imaginer un sémaphore comme une barrière, la valeur initiale du compteur représentant le nombre de tâches pouvant franchir la barrière en même temps (ou le nombre de tickets disponibles). Tant que des tickets sont disponibles, une tâche peut franchir la barrière (*P(s)*). Dès que tous les tickets sont épuisés, il faut attendre qu'une autre tâche restitue (*V(s)*) un ticket pour pouvoir passer la barrière.

On appelle ce type de sémaphore un **sémaphore à compte**. Il existe des cas particuliers, dépendant en général de la valeur à l'initialisation. Par exemple :

- sémaphores **binaires** ou **d'exclusion mutuelle (mutex)**, initialisés à 1, utilisés pour protéger des sections critiques ;
- sémaphores *privés*, en général initialisés à 0, pour lesquels un seul processus, dit **propriétaire du sémaphore**, peut utiliser *P(s)*. Ces sémaphores sont le plus souvent utilisés pour synchroniser des tâches entre elles et on peut faire l'économie de la file des tâches en attente, puisque seule la tâche propriétaire peut être en attente.

Implémentation

Dans l'archive disponible sur Moodle (distribuables_tp2.tgz), récupérez les fichiers **sem.h** et **sem.c**. Le fichier header contient l'ensemble des primitives pour gérer des sémaphores à compte. Les explications sont disponibles directement dans les fichiers.

Question : implémentez les fonctions de gestion des sémaphores pour une utilisation par le micro noyau temps-réel.

Note : ajoutez une fonction au noyau pour récupérer le numéro de la tâche courante plutôt que d'accéder directement à la variable *_tache_c*.

Question : améliorez votre programme de test d'exclusion mutuelle en remplaçant l'usage des primitives *dort()* et *reveil()* par des sémaphores. L'objectif est d'activer la tâche qui sert de consommateur seulement lorsqu'une donnée a été produite.

Question : améliorez encore votre programme en ajoutant plusieurs consommateurs et plusieurs consommateurs, afin de valider le bon fonctionnement de votre système de gestion des sémaphores. Détaillez vos choix quant au programme de test et n'oubliez pas de valider le bon fonctionnement par un screen de l'exécution de votre noyau.

Question bonus : les sémaphores sont un concept couramment enseignés via l'exemple du dîner des philosophes. Développez un programme de test qui utilise le micro noyau temps-réel et sa gestion des sémaphores pour proposer une solution à ce problème. N'oubliez pas de justifier votre solution ainsi que d'y associer un screen de bon fonctionnement.