Examen 2009/2010 Temps Réel

Le 3/5/2010, Tout document autorisé
Durée : 1 heure et demie

1. Expliquez ce qu'est une échéance. Donnez des exemples de valeurs pour des échéances dans des applications concrètes. Prenez des exemples avec des valeurs d'échéance très différentes les unes des autres en ordre de grandeur.

2. Pourquoi dans les critères permettant de savoir si un ensemble de tâches supporte le temps réel on considère toujours des tâches périodiques ? Si une tâche n'est pas périodique, que pourrait-on faire pour quand même faire le test ?

3. Les systèmes d'économies d'énergies arrêtent certains matériels sur la carte mère ou dans le processeur. Est-ce gênant pour le temps réel ? Détaillez précisément votre réponse.

4. Pour économiser du temps CPU, vous ajoutez des caches logiciels dans vos applications temps réel. Quelles en sont toutes les conséquences ?

5. L'ordonnanceur s'aperçoit qu'il a largement le temps de terminer avant son échéance la tâche en train de s'exécuter. Peut-il réduire la fréquence du processeur pour économiser de l'électricité ? Détaillez largement votre réponse.

6. Dans les langages interprétés qui supportent le *multi-thread* (par exemple Python) toutes les données sont partagées et doivent donc être accédées et modifiées dans des sections critiques. Donc une seule *thread* peut s'exécuter à la fois. Mais alors, quel est l'intérêt de faire du *multi-thread* ?

Soit l'API suivante pour des fonctions devant fonctionner en *multi-thread* :

- **void set_item(Table *table, int index, void *value)**
- **void *get_item(Table *table, int index)**
- **void *pop_last_item(Table *table)**
- **void push_last_item(Table *table, void *value)**
- **int get_length(Table *table)**

1. Ces fonctions doivent-elles être réentrantes ?

2. Quelles remarques/commentaires/critiques/explications pouvez-vous donner sur les prototypes de ces fonctions ?

3. Que contient la structure `Table` après son initialisation ?

4. Pour chacune de ces fonctions, proposez la meilleure implémentation possible. Quels sont les avantages et inconvénients de vos implémentations ?

5. Dans une application on veut faire une FIFO. Plusieurs *threads* font des *push*, et une seule *thread* fait les lectures. Proposez une implémentation efficace, de préférence sans ajouter de section critique.

6. Proposez une implémentation sans verrous des 5 fonctions initiales et indiquez les inconvénients de cette implémentation.

Une correction

1. *Expliquez ce qu'est une échéance. Donnez des exemples de valeurs pour des échéances dans des applications concrètes. Prenez des exemples avec des valeurs d'échéance très différentes les unes des autres en ordre de grandeur.*

   Si une tâche périodique à une échéance e et une période p alors à chaque période elle doit terminer son exécution avant $np + e$

Échéance	Tâche
10^{-4} s	Contrôle de la pièce en lévitation
10^{-4} s	Génération de son avec le buzzer
10^{-2} s	Envoit d'un buffer de donnée à une carte son
10^{-1} s	Mise à jour d'une information sur un écran
10^{0} s	Vérification d'un mot de passe via le réseau
10^{0} s	Affichage d'une page web
10^{1} s	Écriture des caches sur un support physique
10^{2} s	Démarrage d'un ordinateur (non périodique)
10^{4} s	Sauvegardes journalière
10^{5} s	Calcul de prévision météo à une semaine

2. *Pourquoi dans les critères permettant de savoir si un ensemble de tâches supporte le temps réel on considère toujours des tâches périodiques ? Si une tâche n'est pas périodique, que pourrait-on faire pour quand même faire le test ?*

   Avec une tâche apériodique on ne peut prévoir la quantité de ressources CPU utilisée. Pour faire le test vérifiant que le système supporte le temps réel, il est nécessaire d'associer une période minimum à la tâche. C'est en fait la durée minimal entre 2 déclenchements.

   Attention, le test n'est pas fiable dans ce cas, car la tâche apériodique pourrait vouloir démarrer son traitement **après** le début de sa période *fictive*. Et les tests d'ordonnancement (et ordonnanceurs) le supportent mal. Deux solutions : la tâche apériodique regarde en début de chaque période si elle a du travail à faire (retard de démarrage maximum égal à la période) ou bien on la met dans un serveur de tâches sporadiques qui règle le problème de l'exécution "dos à dos" mais n'assure pas l'éxécution temps réel.

   Conclusion : pour assurer le respect de l'échance d'une tâche sporadique il faut donner comme période et échéance la moitié de l'échéance de la tâche sporadique et l'ordonnancer comme une tâche périodique classique.

   Évidemment, le système sera considérablement sous utilisé car la tâche ne sera pas lancée aussi souvent.

3. *Les systèmes d'économies d'énergies arrêtent certains matériels sur la carte mère ou dans le processeur. Est-ce gênant pour le temps réel ? Détaillez précisément votre réponse.*

   La plupart du temps, le matériel ne peut pas passer en mode économie d'énergie car les tâches temps réel étant périodiques le matériel est constamment utilisé.

   Si néanmoins, le matériel se met en mode économie tout seul, alors il suffit de considérer ceci dans le calcul de la capacité d'une tâche en considérant qu'elle démarre en mode économie et donc qu'il faut un certain temps pour redémarrer le matériel. Dans ce cas de figure, on ne pourra pas faire tourner autant de tâches qu'avant sur la machine tout en certifiant leur comportement temps réel.

   Finalement, c'est généralement l'ordonnanceur qui décide de passer le matériel en mode économie. Dans ce cas, il est facile pour lui de rallumer le matériel un temps suffisant avant que la tâche en ai besoin. Dans ce cas, le système est toujours temps réel, on fait tourner autant de tâches qu'avant et on fait des économies d'énergies.

4. *Pour économiser du temps CPU, vous ajoutez des caches logiciels dans vos applications temps réel. Quelles en sont toutes les conséquences ?*

   Il faut recalculer toutes les capacités pour prendre en compte la gestion du cache. En effet, dans le pire cas la donnée n'est pas dans le cache ou bien elle est obsolette. Les capacités sont donc augmentées et il est possible que l'on ne puisse pas faire fonctionner autant de tâche en garantissant le temps réel.

   L'utilisation de cache fait en moyenne gagner du temps. Un de ses effets négatifs est que l'on utilise plus de mémoire et que l'on déborde du cache du processeur. Mais cela n'est pas gênant car les capacités supposent que les tâches n'utilisent pas le cache du processeur.

   Finalement, en ajoutant les caches, il est possible que des systèmes d'économies d'énergie se mettent en route car le processeur est moins chargé. Avec beaucoup de malchance, cela peut donc ralentir l'exécution de vos programmes !

5. *L'ordonnanceur s'aperçoit qu'il a largement le temps de terminer avant son échéance la tâche en train de s'exécuter. Peut-il réduire la fréquence du processeur pour économiser de l'électricité ? Détaillez largement votre réponse.*

   Si l'ordonnanceur s'aperçoit qu'il a largement le temps de terminer avant son échéance c'est qu'il connaît obligatoirement les capacités des tâches. C'est par exemple une ordonnanceur LLF qui normalement peut connaître sa marge de manoeuvre **en tenant compte de TOUTES les tâches**.

   Il peut donc baisser la fréquence du processeur à la fréquence adéquate tout en respectant la contrainte de temps réel.

   Mais, en fait, il est beaucoup plus simple (et énergétiquement rentable) d'attendre que le processeur soit inutilisé et de le suspendre complètement jusqu'à son prochain réveil (en tenant compte de son temps de réveil).

6. *Dans les langages interprétés qui supportent le* multi-thread *(par exemple Python) toutes les données sont partagées et doivent donc être accédées et modifiées dans des sections critiques. Donc une seule* thread *peut s'exécuter à la fois. Mais alors, quel est l'intérêt de faire du* multi-thread *?*

   Dans la question, il est dit : *une seule* thread *peut s'exécuter à la fois* ceci implique qu'il n'y a qu'un seul verrou pour protéger toutes les sections critiques.

   Tous ceux qui ont répondu que les *threads* pouvaient travailler sur leurs données personnelles ont oublié que dans ce cas il devait y avoir plusieurs verrou (un par donnée) et qu'en plus de prendre beaucoup de place mémoire cela rend les *deadlock* inextricables.

   Revenons à la question : quel est donc l'intérêt de faire du *multi-thread* ? Et bien tout simplement quand une *thread* fait un appel système long elle peut libérer le verrous pour qu'un autre puisse continuer. Ceci permet donc d'avoir des programmes simples car ils n'ont pas besoin de faire des entrées/sorties asynchrones.

Soit l'API suivante pour des fonctions devant fonctionner en *multi-thread* :

- **void set_item(Table *table, int index, void *value)**
- **void *get_item(Table *table, int index)**
- **void *pop_last_item(Table *table)**
- **void push_last_item(Table *table, void *value)**
- **int get_length(Table *table)**

Remarques générales : quand plusieurs *threads* lisent et modifient le tableau en même temps, les résultats dépendent de la manière dont les *threads* sont ordonnancées, c'est donc "aléatoire". Par contre, le programme ne doit pas planter, et le push/pop doit garantir de ne pas perdre de valeur ou faire des doublons.

1. *Ces fonctions doivent-elles être réentrantes ?*

   Une fonction réentrante est une fonction que l'on a le droit d'appeler simultanément à partir de plusieurs *thread*. Les fonctions de cette API manipulent toutes une structure de donnée partagée, elles doivent donc toutes (ou presque) travailler dans une section critique. Si les fonctions précédentes ne sont pas réentrantes cela signifie que l'on doit rentrer dans la section critique en verrouillant avant de les appeler (ce qui est lourd au niveau programmation et entraine des bug mais peut faire gagner du temps dans certains cas). Si les fonctions sont réentrantes cela va simplifier la programmation.

2. *Quelles remarques/commentaires/critiques/explications pouvez-vous donner sur les prototypes de ces fonctions ?*

   - Pour ne pas bloquer toutes les tables à la fois, chaque table doit contenir un verrou puisque celui-ci n'est pas passé en paramètre.
   - Ces fonctions permettent de gérer une pile.
   - `get_item` et `pop_last_item` ne peuvent retourner d'erreur si l'élément n'existe pas. On doit donc considérer que le pointeur NULL indique que la case est vide.
   - On ne sait rien du comportement de `set_item` quand on indique un index inexistant.
   - Il manque l'attribut `volatile` sur la table.
   - Il n'y a pas de fonction d'initialisation de la table.

3. *Que contient la structure* `Table` *après son initialisation ?*

   Comme vous êtes en examen, il est préférable de prendre la structure de donnée la plus simple et qui est à accès direct, donc le tableau. La structure `Table` contiendra :

   - `pthread_mutex_t mutex` Un mutex déverrouillé.
   - `int size` Le nombre d'éléments présents dans le tableau.
   - `void **items` Les données.

   Pour optimiser en temps CPU, on peut ajouter un champ indiquant la taille effectivement allouée dans le tableau. Ceci permet de ne pas allouer les éléments un par un.

   Il est aussi intéressant d'indiquer lorsque l'on crée la table une fonction appelée lorsqu'un élément est enlevé de la table. Ceci permet de détourner le bug de l'API de `set_item` qui ne retourne pas la valeur effacée.

4. *Pour chacune de ces fonctions, proposez la meilleure implémentation possible. Quels sont les avantages et inconvénients de vos implémentations ?*

   Version sans optimisation des allocations. Ces implémentations sont lentes car elles verrouillent à chaque fois.

   On pourrait éviter plein de verrouillages si l'on stockait la taille maximum de la table car on n'aurait pas besoin de verrouiller pour savoir si un indice est dans la table car on stockerait NULL dans les cases détruites et on ne réduirait pas la taille allouée.

```
void set_item(Table *table, int index, void *value)
{
lock(table->mutex) ;
if ( index < table->size )
    table->items[index] = value ;
else
    ERROR ;
unlock(table->mutex) ;
}

void *get_item(Table *table, int index)
{
void *v = NULL ;
lock(table->mutex) ;
if ( index < table->size )
    v = table->items[index] ;
unlock(table->mutex) ;
return v ;
}

void *pop_last_item(Table *table)
{
void *v = NULL ;
lock(table->mutex) ;
if ( table->size > 0 )
    {
    v = table->items[table->size] ;
    table->size-- ; // La taille sera réduite au prochain 'push'
    }
unlock(table->mutex) ;
return v ;
}

void push_last_item(Table *table, void *value)
{
lock(table->mutex) ; // Ce verrou sera tenu longtemps...
table->items = realloc(table->items,
                       table->size * sizeof(*table->items)) ;
table->item[table->size] = value ;
table->size++ ;
unlock(table->mutex) ;
return v ;
}

int get_length(Table *table)
{
return table->size ; // Pas de verrou nécessaire.
}
```

5. *Dans une application on veut faire une FIFO. Plusieurs* threads *font des* push, *et une seule* thread *fait les lectures. Proposez une implémentation efficace, de préférence sans ajouter de section critique.*

L'implémentation suivante ne fait pas de décalage de tableau. Elle ne verrouille que quand la FIFO est vide afin de vider complètement le tableau. Pour se passer de verrous il faudrait gérer la table différemment, une réponse est dans la question suivante.

```
typedef struct { Table *table ; int index_read ; } Fifo ;

void fifo_push(Fifo *f)
{
   pop_last_item(f->table) ;
}

void *fifo_pop(Fifo *f)
{
   void *v = push_get_item(f->table, f->index_read) ;

   if ( v )
       f->index_read++ ; // Pas de verrou : un seul lecteur
   else
      {
         lock(f->table->mutex) ;
         if ( get_length(f->table) == f->index_read )
            {
              f->table->size = 0 ;
              f->index_read = 0 ;
            }
         unlock(f->table->mutex) ;
      }
   return v ;
}
```

6. *Proposez une implémentation sans verrous des 5 fonctions initiales et indiquez les inconvénients de cette implémentation.*

   Plein de solutions possibles.

   La plus propre est la liste chainée, mais elle nécessite une instruction atomique pour faire l'échange et puis il faut gérer la mémoire. C'est un peu long à faire en examen...

   On peut aussi faire en sorte que chaque *thread* ait ses structures de données personnelles qui n'ont pas besoin d'être protégée. C'est long d'accès et compliqué à programmer.

   La plus simple : on réserve un tableau de taille fixe et on ne stocke pas la longueur. On met NULL pour les éléments détruits.

   On ne stocke pas la table mais la suite des instructions permettant de la recréer. Un processus, recompacte cette suite de temps en temps. Cette technique est utilisée par les systèmes de fichiers conçu comme des fichiers de log (NILFS par exemple).