Examen 2005/2006 Temps Réel

Le 19/5/2006, Tout document autorisé
Durée : 1 heure et demie

S'il vous plait, pour faciliter la correction, essayez de répondre aux questions dans l'ordre.
Il faut étayer vos réponses.

1. Si dans une machine on a 1 processeur par tâche à exécuter, les tâches sont-elles temps-réel ?
2. Quels sont les avantages et inconvénients de ne pas utiliser les interruptions matérielles ?
3. Peut-on faire un ordonnanceur temps réel n'utilisant pas les interruptions matérielles ?
4. La mémoire virtuelle est-elle nécessaire au fonctionnement des applications temps réel ?
5. Dans un système temps réel *dur*, quels sont les paramètres nécessaires à l'ordonnanceur pour lancer une tâche.
6. Comment calculer une capacité ?
7. Dans un système temps réel, que font les gestionnaires d'interruptions quand ils reçoivent une interruption matérielle ? (à part la sauvegarde/restauration du contexte).
8. Une liste chaînée est partagée par plusieurs processus. Proposez une algorithme pour la fonction d'insertion d'un élément dans une liste chaînée dans le cas d'une utilisation pour des processus temps réel.
9. Un processeur dispose d'un instruction atomique pour échanger le contenu de deux mots en mémoire. Donnez un exemple d'utilisation de cette instruction dans le cas du temps réel.
10. Un *hub* réseau peut-il être temps réel ? Et un *switch* ?
11. Un disque dur (hors système d'exploitation) est-il temps réel ?
12. Peut-on faire le test de la zone critique en réalité plutôt que théoriquement ?
13. En quoi un ordonnancement de processus pourrait-il être meilleur qu'un autre ?
14. Vous avez un programme à lancer périodiquement toutes les milli-secondes, avec une échéance égale à la période. Qu'utiliseriez vous pour le faire fonctionner (Linux, RTAI, ...) ?
15. Y-a-t-il une latence de démarrage lors de l'exécution d'un gestionnaire d'interruption avec RTAI ?
16. La table de *hashage* est-elle une bonne structure de donnée pour les tâches temps réel ?
17. Une personne mal intentionnée voudrait faire en sorte que votre tâche temps réel ne respecte pas son échéance. Que pourrait-elle faire à part éteindre la machine ? Sans accès physique à la machine ? Sans compte sur la machine ?
18. Où doit-on compter le temps d'exécution de l'ordonnanceur temps réel ?
19. Une tâche temps réel déborde de sa capacité. Que peuvent faire les différents ordonnanceurs que l'on a vu en cours ?
20. Comment calculer la plus petite échéance possible pour une configuration de machine donnée. On suppose que le travail de la tâche est négligeable.

Une correction.

1. Non, il peuvent dépasser leur échéance car :

   (a) Ils sont interrompus par les interruptions matérielles.

   (b) Ils font des appels systèmes.

   (c) Ils partagent l'accès mémoire, il faut donc penser à ce problème quand on calcule les capacités.

   (d) Ils peuvent avoir des sections critiques communes.

   Avoir un système temps réel améliore les 2 premiers points. Les deux derniers sont indépendant du type d'ordonnanceur utilisé (orienté temps réel ou non) car arrêter une tâche ne fera pas avancer plus vite les autres.

2. Avantages :

   (a) calcul de capacité facilité car pas d'interruption

   (b) pas de zones critique, donc pas de *deadlock*.

   Inconvénient :

   (a) Utilisation de temps CPU pour faire de la scrutation,

   (b) non prise en compte immédiate de changement d'état de périphérique.

3. Oui, en utilisant par exemple un ordonnanceur cyclique avec une scrutation de l'horloge pour déterminer le début des cycles mineurs. L'inconvénient est que si une tâche dépasse sa capacité, le système ne le sais qu'après et donc des tâche importantes peuvent ne pas commencer au bon moment.

4. On peut utiliser la mémoire virtuelle au lieu des adresses mémoire absolues **à condition que le contenu de la mémoire reste en RAM**. Si le *swap* est utilisé, les temps d'éxécution sont imprédictibles. La mémoire virtuelle ralentie les changements de contexte (reprogrammation du MMU) mais permet par exemple le déplacement de processus d'une barrette de RAM à une autre sans l'arrêter (pour la changer par exemple).

5. Tous les ordonnanceurs ont besoin de la période. EDF a besoin de l'échéance et LLF a besoin en plus de la capacité. **Mais pour tous les ordonnanceurs temps réel dur, MÊME HPF et EDF, il est nécessaire de connaître la capacité de la nouvelle tâche afin de n'accepter de la lancer que si le système assure le respect du temps réel dur**.

6. Il faut :

   - Déterminer le temps maximum d'exécution des instructions élémentaires et des appels système en considérant les caches vides.
   - Déterminer les données du processus maximisant la durée théorique totale.
   - Dans le cas de sections critiques, calculer le temps de suspension le plus long possible. Ceci dépend de l'ensemble des autres tâches.
   - Dans le cas d'informations venant de l'extérieur, évaluer ce que l'on peut.

7. On ne veux pas que les interruptions perturbent l'ordonnancement des tâches, leur durée doit être minimum. Donc le gestionnaire d'interruption :

   - réveille la tâche qui est en attente de l'interruption matérielle en utilisant un sémaphore par exemple.
   - Et l'ordonnanceur peut décider de refaire l'ordonnancement si la tâche réveillée est plus prioritaire que la tâche courante.

Le gestionnaire d'interruption se termine pour laisser la tâche active travailler.

Si la tâche réveillée est très prioritaire, il faut garantir un délai minimum entre 2 interruptions reçues pour garantir que le système est temps réel. Se délai minimum sera utilisé comme une période minimum.

8. On ne peut pas utiliser simplement l'algorithme classique d'insertion dans une liste car si 2 processus veulent ajouter une cellule au même endroit en même temps une seule des deux cellules sera ajouté. Il faut donc utiliser un *mutex* pour prendre le droit en écriture. Si la liste chaînée est longue et très souvent modifiée il faut stocker un *mutex* par cellule, sinon un *mutex* global sur la liste chaînée est suffisant.

   Il n'est pas nécessaire d'implémenter dans ce cas un système lecteur/écrivain car il ne peut y avoir de problème de parcours de liste dans le cas de l'insertion.

9. Le *test and set* peut être implémenté avec cette instruction. Mais le plus important est que de nombreux algorithmes n'ont plus besoin de sections critiques. Par exemple l'insertion dans une liste :

```
void inserre_apres(Cellule *endroit, Cellule *nouveau)
{
nouveau->suivant = nouveau ;
SWAP(endroit->suivant, nouveau->suivant) ;
}
```

10. Le travail du HUB est de répéter toutes les données reçues par tous les ports sur tous les autres. Il ne peut effectuer ce travail sans perdre de paquet quand il y a une collision. Il n'assure donc pas le service demandé, donc il n'est pas temps réel. Si le protocole réseau est *Token Ring* on peut considérer le HUB comme assurant un service temps réel.

    Dans le cas du *switch* il n'y a pas de collisions. Par contre il peut y avoir congestion quand trop de données arrivent en direction du même port. De plus le *switch* peut contenir des tampons, dans ce cas le temps de transit n'est pas constant. Pour que le transit soit temps réel il faut utiliser *Token Ring* (mais autant avoir des HUB plutôt que des *switchs* dans ce cas). Ou limiter le trafic des machines afin de garantir qu'il n'y aura pas de congestions sur les liens entre machines dialoguant en temps réel. On peut aussi utiliser la qualité de service que le *switch* comprennent afin de rendre prioritaire le trafic temps réel (VoIP par exemple).

11. Supposons tout d'abord un disque dur bête sans aucun tampon en entrée et sortie. Il est facile de lui associer une capacité pour les opérations de lecture et d'écriture : c'est le temps maximum pris pour effectuer l'opération. Ce temps ne se réduit pas au temps de déplacement de la tête d'un bout à l'autre du disque Il faut tenir compte du fait qu'il peut y avoir plusieurs essais nécessaires et que si un secteur est déclaré hors service il faut modifier la table des blocs invalides. On peut supposer que les secteurs invalides apparaîtront un par un sinon cela veut dire que le disque est en panne.

    Si le disque est intelligent on supposera qu'il fera mieux que le pire cas, en associant une échéance égale à la capacité calculée à chacune des requêtes. Donc on peut garder les capacités calculées initialement.

12. Cela n'a pas beaucoup de sens, il faudrait lancer simultanément les tâches, et les empêcher de terminer avant la fin de leur capacité. Dans ces conditions, autant faire le test théorique. Dans la pratique les ordonnanceurs s'assurent que les tâches ne dépassent pas leur capacité, en tout cas, ils devraient prévenir.

13. Plusieurs critères sont possibles :

    - Il permet de lancer plus de tâches tout en garantissant le temps réel.

- Il utilise moins de temps CPU pour lui même. (Redondant avec le point précédents)
- Minimise le nombre de changement de contexte, (Redondant avec le point précédents)
- Il est stable en cas de dépassement de capacité.

14. Avec RTAI, aucun problème cela marchera, mais on perd en fiabilité car il n'y a pas de protection mémoire entre tâches (vous avez souvent fait redémarrer les machines).

    Avec Linux (2.6) on n'a pas tous ces problèmes, mais :

    - la méthode passive peut souvent dépasser l'échéance (à cause de l'ordonnanceur) si le système est chargé.
    - la méthode active avec une *thread* temps réel répond au cahier des charges dans tous les cas par contre le système devient inutilisable pour faire autre chose.

15. Si l'interruption matérielle est associée à une *thread* alors elle ne démarrera que quand celle-ci aura le droit de démarrer. La latence sera importante car généralement si l'on fait cela c'est que l'on associe l'interruption à une tâche qui n'a pas la priorité la plus élevée.

    Si l'interruption matérielle n'est pas associée à une tâche (comme en TP) alors le gestionnaire d'interruption s'exécute immédiatement après la sauvegarde des registres. Cette approche complique le calcul des capacités car il faut tenir compte du temps CPU utilisé par le gestionnaire dans le pire cas (plein d'interruption) dans le calcul de la capacité de **toutes** les tâches.

16. Dans le cas du temps réel plusieurs problèmes se posent :

    - On ne peut pas tous bloquer pendant un agrandissement de la table de *hashage*.
    - Dans le pire cas les éléments peuvent se retrouver tous dans la même case. Le temps d'accès n'est plus constant mais en *log* du nombre d'éléments dans le pire cas si l'on utilise un arbre bien balancé pour stocker la "liste"

    Moralité, en temps réel dur la table de *hashage* n'est pas utilisable à du changement de taille mais aussi car elle n'est pas plus efficace qu'un arbre bien balancé.

17. Si le système est temps réel ont ne peut rien faire car normalement tout a été pris en compte. Évidemment le concepteur qui a calculé les capacités des tâches n'a pas pu tout imaginer.

    Avec un accès physique (pas de compte nécessaire) :

    - Brancher des périphériques conçus pour que la machine reçoivent des interruptions à vitesse maximale. Les tâches temps réel ne suspendent pas les interruptions afin que le système temps réel puissent les gérer avec la priorité qui leur est associée.
    - Faire monter la température au cas ou cela fasse ralentir le processeur (idée trouvée dans une copie d'étudiant)

    Avec un compte (on suppose non *root* sinon c'est trop facile) :

    - Lancer un processus vidant continuellement les caches du processeurs.
    - Utiliser à vitesse maximale des instructions générant des interruptions.

    Avec seulement un accès réseau :

    - Envoyer le maximum de petit paquets afin de générer le maximum d'interruptions.
    - Envoyer des paquets réveillant des tâches encore plus prioritaire.

Indépendamment de ces 3 cas, le processus temps réel manipule des données. Si vous avez accès aux données manipulées vous pouvez les modifier pour que le programme temps réel dépasse sa capacité. C'est évidement très dur à faire si l'on ne dispose pas des sources de celui-ci. Si par exemple il utilise une table de *hashage* sans avoir prévu de sécurités, alors vous pouvez vous amuser à envoyer des données pour générer le maximum de collisions dans la table.

18. Si l'ordonnanceur est bien fait, il tient compte de son temps d'exécution propre quand on lui demande lancer une nouvelle tâche. Le programmeur n'a donc pas à en tenir compte. De son coté, l'ordonnanceur à plusieurs choix possible :

    - Considérer que $x\%$ du temps CPU est utilisé pour l'ordonnancement, il multiplie donc les capacités des tâches par $\frac{1}{1-x}$
    - Se considérer comme une tâche comme les autres lancée périodiquement avec sa propre capacité.

19. Pour un ordonnanceur statique cyclique, il ne peut rien faire car il ne sais même pas quelle tâche a dépassé sa capacité.

    Si l'ordonnanceur connaît les capacités, il peut déduire grâce à un test comme le *Rate Monotonic* jusqu'à quel point le dépassement de capacité peut être autorisé tout en gardant un comportement temps réel. Si il n'y a pas de problème, on laisse le système en l'état est on préviens le programmeur.

    Si le système ne peut plus rester temps réel, on peut :

    - Tuer une tâche, de préférence la moins important, qui d'ailleurs n'est pas forcément la moins prioritaire.
    - Tuer la tâche qui a dépassé car on la suppose buggée ou bien ne fonctionnant pas dans des conditions normales.
    - Si cela a été prévu, ne faire tourner que la partie obligatoire de la tâche sans faire tourner la partie optionnelle.

    Ce choix est aussi difficile que le choix de la tâche à tuer quand le système d'exploitation n'a plus assez de mémoire virtuelle pour tous les processus.

20. Remarque : plus l'échéance est proche de la période plus la tâche est prioritaire.

    L'échéance minimale est théoriquement égale à la capacité, mais elle ne peut pas l'atteindre pour plusieurs raisons :

    - La résolution de l'horloge cadençant l'exécution de l'ordonnanceur peut être plus grande que la capacité. Dans ce cas, il faut ajouter ce temps à l'échéance.
    - Quand la tâche démarre, il y a un léger retard du au changement de contexte. Il est négligeable par rapport au temps précédent sauf si la tâche est réveillé *via* une interruption