2 - INTERRUPTION HORLOGE

1. CHOIX DU QUANTUM

On considère un système en temps partagé et on suppose, pour simplifier, qu'à chaque passage sur le processeur une tâche utilise la totalité de son quantum de temps. On suppose par contre qu'il y a un overhead de s millisecondes à chaque commutation.

1.1.

Quel est le pourcentage de temps perdu à cause des commutations ?

Le pourcentage de temps perdu à cause des commutations est s/q. On verra dans la question suivante qu'il s'agit en fait d'une évaluation globale et que le pourcentage de temps perdu augmente si une tâche n'utilise pas tout son quantum.

1.2.

On prend un quantum de 100 ms et un overhead de 1 ms. Quelle serait la durée totale (temps CPU consommé, en incluant les commutations) d'une tâche demandant 20 ms de calcul ? D'une tâche demandant 500 ms de calcul ?

On affecte le temps de commutation en fin de tâche à la tâche qui vient de se terminer. On a alors 1/20 = 5% de temps perdu pour la première tâche, contre 5/500 = 1% pour la deuxième.

1.3.

Quels sont les critères de choix du quantum?

En choisissant q trop petit, on multiplie les commutations et l'overhead devient non négligeable.

En prenant q grand, s/q devient faible, mais il ne donne pas forcément une bonne évaluation du pourcentage de temps perdu si un grand nombre de tâches n'utilisent pas la totalité du quantum. De plus, une tâche courte lancée en parallèle avec N autres tâches attend N.q avant de pouvoir s'exécuter, ce qui peut la pénaliser considérablement.

Tout est donc affaire de compromis, et fonction du nombre de processus simultanés prévu (degré de multiprogrammation). Plus ce nombre est faible, plus on peut augmenter q sans ralentir exagérément les tâches courtes.

1.4.

Comment peut-on expliquer, malgré l'évolution de la puissance des processeurs, que le quantum de la plupart des systèmes Unix reste fixé à 100 ms ?

Ces 100 ms correspondent en fait à la perception du parallélisme par l'être humain. Avec cet intervalle, deux tâches s'exécutant de manière pseudo-parallèle sont perçues par l'œil humain comme se déroulant en vrai parallélisme.

2. GESTION D'UNE INTERRUPTION HORLOGE

On considère le processeur JB007 qui dispose d'un registre temporisateur Rtempo, décrémenté automatiquement toutes les milli-secondes *en mode usager* et *en mode système*. Lorsque sa valeur atteint 0, Rtempo provoque une interruption (interruption horloge). Ce registre est initialisé automatiquement à une valeur RTMAX (celle où tous les bits sont à 1).

Rtempo est utilisé à la fois pour tenir à jour l'heure universelle dans un mot RHU, pour instaurer un tour de rôle avec quantum d'exécution et pour comptabiliser le temps CT utilisé par une tâche. Lorsque cette variable CT dépasse une valeur CTMAX, la tâche est détruite.

2.1.

Pourquoi décrémente-t-on Rtempo même en mode système, et pourquoi continue-t-on à le décrémenter lorsqu'il atteint 0 ? Que se passe-t-il lorsqu'on décrémente la valeur d'un registre alors que celui-ci est déjà à 0 ?

Dans cet exemple, Rtempo est utilisé entre autres pour la mise à jour de l'heure universelle. Il faut donc le décrémenter en permanence (on n'arrête pas le temps !), quelle que soit sa valeur et quel que soit le mode d'exécution.

La décrémentation d'un registre à 0 ramène tous les bits à 1. Après chaque passage à 0, le registre reprend donc automatiquement sa valeur maximum RTMAX. Cette valeur peut cependant être interprétée comme une valeur négative.

Le système gère une table TDT des descripteurs de tâches : TDT[i] contient l'ensemble des informations relatives à la tâche i. On considère pour l'instant que la valeur QX du quantum est telle que QX = RTMAX.

2.2.

Quelles sont les opérations concernant la gestion du temps que doit effectuer le système la première fois qu'il charge la tâche i pour l'exécuter ?

Il faut impérativement mettre à jour RHU à chaque modification de Rtempo puisque c'est la valeur de Rtempo qui permet de mesurer le temps écoulé. On s'est placé dans le cas où la seule valeur d'initialisation possible est RTMAX. Le temps écoulé depuis la dernière mise à jour est donc RTMAX – Rtempo. On initialise ensuite Rtempo pour mesurer le quantum de la tâche.

```
TDT[i].CT = 0;                /* gestion du temps tâche */
RHU = RHU + RTMAX - Rtempo;
Rtempo = RTMAX ;
RIT;
```

Les interruptions ont été masquées lors de l'interruption qui a provoqué le lancement de la tâche i. On suppose que RIT a pour effet de démasquer les interruptions.

Le système dispose d'une variable ITE qui contient l'indice de la tâche élue. On suppose qu'une tâche élue se voit toujours attribuer un quantum entier, même si elle avait précédemment perdu le processeur en ayant utilisé partiellement un quantum de temps (suite à une demande d'E/S par exemple).

2.3.

Pourquoi est-il important que la durée de traitement de l'interruption horloge ne soit pas trop longue ?

Deux raisons :

Parce qu'elle s'exécute à un fort niveau de priorité : les autres interruptions sont masquées pendant le traitement et pratiquement aucun événement ne peut donc être pris en compte.

Pour ne pas en rater une : si Rtempo repasse à 0 pendant le traitement de l'IT horloge, la mise à jour de RHU donnera une valeur fausse.

2.4.

Donnez l'algorithme de la routine de traitement de l'interruption horloge. Cette routine provoque une nouvelle élection.

Soit il faut être capable de gérer plusieurs valeurs d'initialisation pour le registre horloge. Ca n'est pas forcément trivial (stockage de la dernière valeur d'initialisation)

Soit on a, pour toutes les opérations temporelles une granularité égale à RHM

Description d'une commutation de T1 à T2.

```
sauvegarder le contexte de T1 (ITE)
TDT[ITE].CT += RTMAX - Rtempo;
si (TDT[ITE].CT ≥ TDT[ITE].CTMAX) {
      STOP ;
}
ITE = indice de la nouvelle tâche;
charger le contexte de T2 (ITE);
RHU := RHU + RTMAX - Rtempo;
Rtempo = RTMAX ;
RIT;
```

La procédure STOP force l'arrêt d'une tâche et libère l'ensemble des ressources qu'elle possède.

2.5.

Quels sont les avantages et les inconvénients de toujours initialiser le registre Rtempo à la même valeur ?

C'est avantageux d'avoir une seule valeur d'initialisation possible pour le registre, la mise à jour du temps est simple à traiter. Par contre, ça pose un problème si on veut lancer des tâches à une fréquence supérieure à 1 /RTMAX.

L'idée est donc d'augmenter la fréquence de l'interruption horloge sans diminuer le quantum : pour cela, il faut que la commutation n'ait lieu que toutes les n interruptions. C'est la technique utilisée sous Unix.

3. QUANTUM ET TICK

Sous Unix, un tick correspond au passage à 0 du registre temporisateur. Certaines tâches de l'interruption horloge sont effectuées à chaque tick, alors que d'autres ne sont gérées que tous les N ticks, N dépendant de la tâche à traiter. Par exemple, la commutation de tâches est traitée toutes les 100 ms alors que l'intervalle entre 2 ticks est de 10 ms.

Par contre, la mise à jour de l'heure universelle, du temps utilisé par la tâche ou le test des alarmes à déclencher (réveil de tâche, réémission de paquets, …) est effectué à chaque tick.

3.1.

Chaque fois qu'un processus perd le processeur, le système réajuste sa priorité en appliquant la formule :

$$p\_pri = p\_user + p\_cpu + p\_nice$$

où p_pri est la priorité de la tâche, p_user la priorité de base d'une tâche utilisateur, p_cpu le temps utilisé par la tâche, comptabilisé en nombre de ticks, et p_nice le paramètre d'une commande nice effectuée par l'utilisateur.

Réécrivez l'algorithme de la routine de traitement de l'interruption horloge en tenant compte de ces nouveaux mécanismes.

Comme tout est comptabilisé au tick près, on peut en faire autant pour le temps utilisé par la tâche. On incrémente à chaque tick le compteur p_cpu. On note TCK la constante égale à la durée du tick, qui est utilisée pour initialiser le registre temporisateur. A chaque tick, on incrémente le temps utilisé par la tâche de TCK (on peut s'amuser avec Rtempo, mais l'éventuel débordement qui n'est pas pris en compte ici le sera sur le prochain tick).

```
        sauvegarder le contexte de T1 (ITE)
        nbtick++;
        p_cpu++ ;
        RHU = RHU + TCK - Rtempo;
        TDT[ITE].CT += TCK;
        Tester s'il y a des alarmes à déclencher ;
        Si ((TDT[ITE].CT >= TDT[ITE].CTMAX) || (nbtick == 10))  {
              si (TDT[ITE].CT >= TDT[ITE].CTMAX) {
                    STOP ;
              } sinon {
                    si (nbtick == 10) {
                          TDT[ITE].p_pri += p_cpu;
                    }
              }
              p_cpu = 0;
              nbtick = 0 ;
              commut = 0 ;
              ITE = indice de la nouvelle tâche;
              charger le contexte de T2 (ITE);
        } sinon {
              restaurer le contexte de T1 ;
        }
}
```

3.2.

Lors du positionnement d'une alarme, l'instant de déclenchement peut être choisi à la micro-seconde près. Quelle est, en réalité, la précision du déclenchement ?

En réalité, le système n'examine les éventuelles tâches à lancer qu'à chaque tick. Par conséquent, la précision du déclenchement est celle du tick, soit en général 10ms.

3.3.

Comment peut-on mesurer le temps passé par la tâche en mode utilisateur et en mode système ?

La façon la plus simple de procéder (celle utilisée par Unix système V) est de faire des statistiques. Le système gère deux champs p_utime et p_stime. Si au moment du tick le processus s'exécute en mode utilisateur, p_utime est incrémenté, s'il est en mode système, c'est p_stime.

A priori, l'erreur d'estimation peut être assez importante sur une tâche courte. Par contre, une tâche longue qui passe 90% du temps en mode utilisateur et 10%en mode système a statistiquement une chance sur 10 que le tick se produise alors qu'elle est en mode système. On obtiendra alors une bonne approximation du temps réellement passé dans chacun des modes.

L'autre solution consisterait à stocker la valeur de Rtempo lorsqu'on passe d'un mode dans l'autre pour pouvoir mesurer le temps passé effectivement dans chacun des modes. C'est sans doute plus précis, mais beaucoup plus lourd.