

TP Système temps réel n°2

Sommaire

Question 1	2
Question 2	2
Question 3	3
Question 4	3
Question 5	4
Question 6	4
Question 7	5
Question 8	6
Question 9	9
Question 10	10
Question 12	11
Question 13	12

Question 1

Nous avons créé une fonction qui permet d'attendre un temps indiqué en millisecondes. Et cette attente simule un vrai travail de la part du processeur.

```
#define TIME_CST 330000

void do_work(unsigned int duration) /* duration en millisecondes */
{
    int i;

    for(i=0;i<TIME_CST *duration;i++) asm volatile ("nop");
}
```

Par la suite, à chaque fois que nous voudrions attendre un temps donné nous utiliserons cette fonction.

Question 2

Nous avons ensuite créé un programme permettant de détecter l'arrivée d'un signal et de faire afficher un message à chacun des signaux entrants.

```
void affiche()
{
    printf("Hello world!\n");
}

int main()
{
    int child;

    signal(SIGUSR1,affiche);

    child = fork ();

    kill(child,SIGUSR1);

    pause();
}
```

```
    return 0;  
}
```

Question 3

On intègre la fonction `do_work` (cf. q1) dans notre fonction signal handler de manière à rendre son travail plus conséquent et alors nous essayons d'envoyer une rafale de signal et à partir de ce moment nous observons que nous ne récupérons pas tous les signaux émis. Ils ne sont pas tous captés. Le souci est lié au fait que lorsque que le programme est entrain de traiter un signal il ne peut capter un autre signal.

La raison pour laquelle tous les signaux ne sont pas traités c'est que lors d'un traitement les multiples interruptions reçus ne sont pas comptabilisées. On n'en compte qu'une seule en réalité. De ce fait toutes les interruptions arrivant durant un traitement ne seront comptabilisées comme une seule et unique, d'où les pertes de signaux.

Question 4

Pour réduire la perte il nous faut chercher à rendre le plus disponible le handler pour l'écoute d'entrée de signaux tout en autorisant les traitements. En séparant le traitement de l'écoute cela nous permet de rendre bien plus disponible notre processus pour l'entrée d'un signal.



Sur le schéma précédent le gestionnaire d'interruption est nommé : gestion d'écoute / gestion

Ce découpage semble alors parfait et nous permettre d'intercepter toutes les interruptions. Mais ce n'est pourtant toujours pas le cas car le gestionnaire d'interruption a toujours un temps d'exécution qui est non nulle et supérieur à zéro. De ce fait si l'on observe deux interruptions sur un temps où le gestionnaire d'interruption a déjà intercepté un signal alors il ne pourra pas le traiter.

Cela marche de la même manière en cas de multiples signaux au cours d'une gestion nous ne pourrions considérer qu'une interruption.



Question 5

Ces méthodes sont utiles pour protéger les systèmes du trop de signaux pouvant être émis de la part d'un système déficient. Ainsi les systèmes mono tâche peuvent être protégés d'un surplus de messages, interruptions. La finalité est la possibilité d'ignorer même les systèmes déficients qui émettraient trop d'informations.

Question 6

Nous devons ici implémenter une tâche périodique de période une seconde et ce en deux travaux. L'un des deux étant réalisé par notre `do_work()` et l'autre par la fonction `sleep()`.

Le souci qui apparaît avec cette méthode est le fait qu'une interruption pourrait se manifester avant notre fonction `sleep()` et de ce fait nous perdrons le temps réel à attendre au sein de notre `sleep()` pour avoir une période de une seconde.

```
void signal_handler()
```

```
{
```

```
    printf("Alarme\n");
```

```
    do_work(500);
```

```
    sleep(0.5);
```

```
}
```

```
...  
gettime();  
sleep(500);  
...
```

 interruption de temps X

Question 7

Ce problème de gestion de temps a vu naître une solution, tâche périodique POSIX qui à l'aide de deux fonctions `timercreate()` et `timer_settime` permette de solver le problème précédent et ceux sans erreur possible.

```
#define PERIOD_SEC 1

#define PERIOD_NSEC 10

void handler()

{

    printf("Top !\n");

}

int main()

{

    timer_t timer_id;

    struct itimerspec info_timer;

    signal(SIGALRM, handler);

    timer_create(CLOCK_REALTIME, NULL, &timer_id);

    info_timer.it_value.tv_sec = PERIOD_SEC;

    info_timer.it_value.tv_nsec = PERIOD_NSEC;

    info_timer.it_interval.tv_sec = PERIOD_SEC;

    info_timer.it_interval.tv_nsec = PERIOD_NSEC;

    timer_settime(timer_id, 0, &info_timer, NULL);

    pause();

    return 0;

}
```

Ici notre code permet d'envoyer de manière périodique des signaux et de les traiter de la sorte. C'est en réalité notre fonction qui est lié au temps réel et non pas à la vitesse de traitement de notre CPU ou d'autres critères hardware qui va générer les signaux. De ce fait ce programme est

transportable d'un ordinateur à un autre. La fonction timer étant réglée pour s'adapter en fonction de la vitesse du CPU. La période sera toujours la même.

On a réussi à soustraire la dépendance hardware/temps d'exécution pour obtenir une réalisation en temps réel.

Question 8

La programmation en boucle est une programmation de diverses tâches devant se répéter au cours du temps. Ces dernières tâches sont alors ordonnancées dans ce qu'on appelle un contrôle de flot statique. La boucle pourra alors se répéter un nombre de fois défini ou indéfini.

Nous avons 3 tâches à réaliser au cours du temps avec chacune leur contrainte et leur temps d'exécution. Le but étant de les arranger de manière à trouver une période complète du système pour ainsi lancer dans une boucle cette période.

T2 : 0,333 secondes temps de réalisation

Toutes les 2 secondes T2 doit être réalisé

T3 : 1 seconde temps de réalisation

Toutes les 3 secondes T3 doit être réalisé

T4 : 2 secondes temps de réalisation

Toutes les 4 secondes T4 doit être réalisé

Avec ses tâches et leurs contraintes une multitude de possibilités est envisageable mais seulement quelques une seront optimales. Les raisons pour lesquels certaines l'une peut être dite optimale face à d'autres est lié à divers critères : période la plus petite possible et séquence au sein de la période qui se répète au cours du temps. Ce dernier point nous permet donc non plus de coder en dure l'ordre de réalisation des tâches mais il permet de découper la période en différentes séquences qui se répètent. L'avantage que l'on tire de ce dernier point sera discuté dans la question suivante.

Les diverses ordonnancement réfléchis et développer à cette question sont les suivants :

Ordonnancement	T2	T3	T4A	T2	T4B	T3	T2	T4B
Temps d'exécution	0,333	1	0,667	0,333	1,333	1	0,33	1,333
Temps limite inférieure	0	0	0	2	0	3	4	4
Temps écoulé	0,333	1,333	2	2,333	3,666	4,666	4,996	6,329
Temps limite supérieure	2	3	4	4	4	6	6	8

Ordonnancement	T2	T4A	T3	T2	T4B	T3	T2	T4A
Temps d'exécution	0,333	0,667	1	0,333	1,333	1	0,333	0,667
Temps limite inférieure	6	4	6	8	8	9	10	8
Temps écoulé	6,662	7,329	8,329	8,662	9,995	10,995	11,328	11,995
Temps limite supérieure	8	8	9	12	12	12	12	12

Ordonnancement	T2	T3	T4	T2	T4	T3	T2	T3	T4
Temps d'exécution	0,33	1	0,7	0,33	1,3	0,4	0,333	0,6	1,1
Temps limite inférieure	0	0	0	2	0	3	4	3	4
Temps écoulé	0,33	1,33	2,03	2,37	3,67	4,066	4,399	4,999	6,099
Temps limite supérieure	2	3	4	4	4	6	6	6	8

Ordonnancement	T2	T4	T3	T2	T3	T4	T2	T4	T3
Temps d'exécution	0,33	0,9	0,8	0,33	0,2	1,5	0,333	0,5	1
Temps limite inférieure	6	4	6	8	6	8	10	8	9
Temps écoulé	6,43	7,33	8,13	8,47	8,67	10,17	10,5	11	12
Temps limite supérieure	8	8	9	10	9	12	12	12	12

Nous observons ici deux découpages légèrement différents qui réalisent pourtant le même travail et ce pour un période total de même ampleur, soit 12 secondes

Question 9

Pour un ordinateur qui réalisera les tâches deux fois plus rapidement nous observerons un problème de temps dû au respect des contraintes de temps imposées, voir l'exemple numéro 1 avec division du temps par deux :

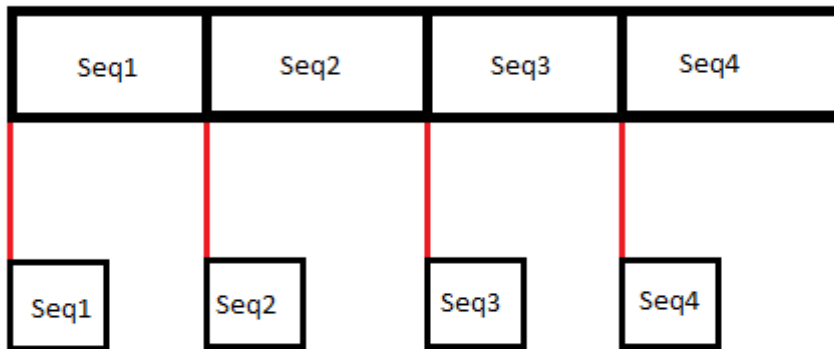
Ordonnancement	T2	T3	T4A	T2	T4B	T3	T2	T4B
temps d'exécution/2	0,1665	0,5	0,334	0,167	0,667	0,5	0,165	0,667
Temps limite inférieure	0	0	0	2	0	3	4	4
temps total /2	0,1665	0,6665	1	1,167	1,833	2,333	2,498	3,165
Temps limite supérieure	2	3	4	4	4	6	6	8

Ordonnancement	T2	T4A	T3	T2	T4B	T3	T2	T4A
temps d'exécution/2	0,1665	0,3335	0,5	0,167	0,667	0,5	0,167	0,334
Temps limite inférieure	6	4	6	8	8	9	10	8
temps total /2	3,331	3,6645	4,165	4,331	4,998	5,498	5,664	5,998
Temps limite supérieure	8	8	9	12	12	12	12	12

Nous observons bien que les contraintes de temps ne sont pas respectées ici lorsque nous changeons d'ordinateur pour un ordinateur 2 fois plus rapide.

La solution, ici, est celle qui a été introduite dans la question précédente. Il suffit de découper notre période en diverses séquences pouvant se réaliser à un temps donné. Cela aura pour effet de comprimé le temps de réalisation des séquences (qui se devront de respecter les contraintes de temps elles aussi). C'est juste que le processeur aura du temps libre pour d'autres actions entre deux séquences.

Pour représenter ce qui est expliqué ci-dessus voici un schéma explicatif :



Les lignes rouges représentent les contraintes de temps pour les séquences. Les séquences étant les cadres noirs numérotés de 1 à 4. L'ensemble des séquences représente une période complète, une boucle.

Le cas tout en haut correspondrait à la configuration de base et celle du dessous à un ordinateur deux fois plus rapide de traitement (environ). Pour le second ordinateur en forçant le démarrage de séquence à un temps donné on observe bien un respect des contraintes de temps ainsi que du temps libre pour le traitement d'autres tâches de la part du processeur.

Question 10

Un modèle d'exécution séquentiel est déterministe, cela fait partie de ces avantages (cf. cours).

Question 12

Ordonnancement	T2	T3	T4	T2	T3	T2	T4	T2	T3	T2	T4	T3	T2
Temps d'exécution	0,33	1	2	0,33	1	0,33	2	0,33	1	0,33	2	1	0,333
Temps limite inférieure	0	0	0	2	3	4	4	6	6	8	8	9	10
Temps écoulé	0,33	1,33	3,33	3,67	4,67	5	7	7,33	8,33	8,67	10,67	11,67	12
Temps limite supérieure	2	3	4	4	6	6	8	8	9	10	12	12	12

Voici l'ordonnancement dans le cas d'un EDF. On observe donc bien que les tâches respectent toujours les contraintes de temps qui leurs sont attribuées.

Question 13

Ordonnancement	Temps d'exécution	Temps limite inférieure	Temps écoulé	Temps limite supérieure
T2	0,333	0	0,333	2
T3	1	0	1,333	3
T4	0,667	0	2	4
T2	0,333	2	2,333	4
T4	0,667	0	3	4
T3	1	3	4	6

ERREUR ! T4 !

Priorité : T2, T3, T4

Ordonnancement	Temps d'exécution	Temps limite inférieure	Temps écoulé	Temps limite supérieure
T4	2	0	2	4

ERREUR ! T2 !

Priorité : T4...

Ordonnancement	Temps d'exécution	Temps limite inférieure	Temps écoulé	Temps limite supérieure
T3	1	0	1	3
T2	0,333	0	1,333	2
T4	0,667	0	2	4
T2	0,333	2	2,333	4
T3	1	3	3,333	6
T4	1,333	0	4,666	4
ERREUR ! T4 !				

Priorité : T3, T2, T4

Ordonnancement	Temps d'exécution	Temps limite inférieure	Temps écoulé	Temps limite supérieure
T3	1	0	1	3
T4	2	0	3	2
ERREUR ! T2 !				

Priorité : T3, T4, T2

Donc nous observons qu'aucune solution n'est possible pour ce qui est de fixer un ordre de priorité des tâches. Dans tous les cas nous avons abouti à une erreur.

Ordonnancement	Temps d'exécution	Temps limite inférieure	Temps écoulé	Temps limite supérieure
T2	0,22311	0	0,22311	2
T3	0,67	0	0,89311	3
T4	1,10699	0	2,0001	4
T2	0,22311	2	2,22321	4
T4	0,23301	0	2,45622	4

Avec un ordinateur qui va 1,3 fois plus vite on observe qu'il est plus que réalisable de fixer un ordre des priorités sur les tâches étant donné que l'ensemble des tâches est réalisé en moins de 3 secondes alors que ces dernières devaient se finir en 4 secondes maximums dans l'exemple ci-dessus.