

Christophe Surun
Denis Kyriacou

M1 – IN2

Compte-Rendu TP2 Systèmes temps réel

-

API POSIX

Question 1

voir fichier tp2_1.c

Pour cette question, au moment d'exécuter le programme que nous avons compilé, nous avons rajouté dans la console la commande « time ».

```
g++ tp2_1.c -lrt -o tp
```

```
time ./tp
```

Voici le résultat de l'exécution :

Fin du travail

```
real    0m0.995s
```

```
user    0m0.992s
```

```
sys     0m0.000s
```

Les résultats sont assez aléatoires (dépendent du processeur, et bien sûr de la machine sur laquelle on travaille).

#define PROCESS 325000, obtenu par approximations successives, nous donne un temps réel d'exécution proche de 1 secondes, c'est cette valeur que nous conserverons pour la suite de ce TP.

Question 2

voir fichier tp2_2.c

Ici, nous avons tout d'abord créé la fonction « écoute ». Ensuite, nous avons associés cette fonction à un signal grâce à cette instruction :

```
signal(SIGUSR1, ecoute);
```

Ensuite, nous effectuons le kill d'un processus obtenu précédemment, et nous envoyons au même moment le signal SIGUSR1.

```
killpg(lepid,SIGUSR1);
```

Ainsi, la fonction ecoute est bien appelée au moment où le programme émet le signal associé.

Question 3

voir fichier tp2_3.c

Nous créons comme demandé une rafale de signaux, comme ci-dessous :

```
for(i=0; i<1000; i++)  
  
    {  
  
        lepid = fork();  
  
        kill(lepid,SIGUSR1);  
  
    }
```

Ainsi, de nombreux signaux de type SIGUSR1 sont créés.

La fonction « écoute » toujours associée à ce signal affiche dans la console un message qui témoigne de la réception de ce signal. Cependant, pour 1000 signaux envoyés, nous observons uniquement moins d'une dizaine qui sont reçus !

Le problème dans cette situation, c'est que le traitement d'un signal est effectué en parallèle de la tâche principale (la fonction main); ainsi, il ne peut y avoir aucun contrôle sur le bon traitement de ces 1000 signaux.

Une solution serait donc de traiter ces signaux dans la tâche principale, ce qui est implémenté dans la question suivante.

Question 4

voir fichier tp2_4.c

Dans ce fichier, nous avons donc amélioré le traitement de cette rafale. Pour résoudre le problème, nous avons créé deux variables globales qui permettent de compter les signaux reçus, et de compter ceux qui ont réellement été traités.

Le problème est en partie réglé, cependant il subsiste toujours des erreurs. En effet, si un signal est reçu lors d'un traitement, ce nouveau signal n'est pas traité. C'est un problème que nous allons une nouvelle fois traiter dans la question suivante.

Question 5

voir fichier tp2_5.c

Nous avons amélioré la gestion des deux variables globales. En effet, il y a des situations problématiques dans ces deux cas :

- `stack == stack2`
- `stack < stack2`

Ces deux situations sont synonymes qu'une interruption a été reçue alors qu'on traitait la précédente.

En considérant ces deux cas particuliers, on évite ainsi ce problème.

Question 6

voir fichier `tp2_6.c`

Nous avons créé pour cette question deux fonctions, l'une faisant appel à la méthode `sleep`, et l'autre la méthode `alarm`.

En laissant un choix à l'utilisateur du programme via un menu, on peut vérifier le temps réel d'exécution de ces méthodes (paramètre de 500 ms entré à titre indicatif).

- `Sleep` nous donne des résultats convenables
- `Alarm` en revanche nous donne des résultats complètement faussés.

Le problème vient probablement du fait qu'il est mauvais de mélanger dans un même programme ces deux instructions. En effet, `alarm` s'exécute pendant le temps demandé, ou bien jusqu'à ce qu'il rencontre un autre signal `alarm`. Mais puisque `Sleep` est lui-même créé à partir de `Alarm` (extrait d'une documentation sur cette méthode « *It is usually a mistake to intermix [alarm](#) and [sleep](#) calls, because [sleep](#) may be internally implemented on your system with [alarm](#).* ») nous pouvons affirmer que l'utilisation dans le même programme de ces deux méthodes est la source du problème que nous rencontrons ici.

Question 7

voir fichier `tp2_7.c`

// Pour Denis

Question 8

voir fichier `tp2_8.c`

On considère trois travaux périodiques :

On pose $TH = 0.333$ secondes.

- T2 d'une durée de TH , toutes les deux secondes
- T3 d'une durée de $3TH$, toutes les trois secondes
- T4 d'une durée de $6TH$, toutes les quatre secondes

Ci-après, voici comment nous avons découpé les tâches sur une durée totale de 12 secondes.

t		1		2		3		4		5		6		7		8		9		10		11		12				
2	x				x				x				x				x				x							
3		x	x	x				x			x	x			x	x			x		x	x	x					
4					x	x		x	x				x	x	x			x						x	x	x	x	x

Nous avons remarqué qu'il était nécessaire qu'une durée minimale de 12 secondes s'écoule pour que les trois tâches se synchronisent (en effet 12 est le plus petit nombre divisible par 2, 3 et 4).

Dans le fichier source ci-joint, nous avons appelé consécutivement des fonctions associées à ces travaux, pendant des durées correspondant au nombre de TH indiqué sur le tableau précédent. Le résultat visible à la fin de l'exécution nous montre bien que les tâches se sont bien exécutées dans un intervalle qui leur est attribué, et pendant une durée correcte.

[Question 9](#)

voir fichier tp2_9.c

Un processeur deux fois plus rapide se traduit par ce nouveau define :

```
#define PROCESS 162500
```

Le programme fonctionne toujours, mais évidemment les « secondes » n'en sont plus puisque le processeur a été modifié.

[Question 10](#)

L'exécution est, pour le programme que nous avons réalisé, déterministe. En effet, toutes les fonctions sont appelées dans la même tâche, et sont donc effectuées toutes les une après les autres.

La présence d'une boucle while permet de vérifier cette propriété, ainsi que la reproductibilité de cet ensemble d'instructions.

Ordonancement avec EDF (Earliest Deadline First)

Question 12

On applique EDF en reprenant les mêmes tâches que celles utilisées pour la question 8

Les cases rouges indiquent les deadlines. Les O indiquent que la tâche est terminée

t			1			2			3			4			5			6			7			8			9			10			11			12	
2	O											O				O						O				O										O	
3		x	x	O									x	x	O									x	x	O								x	x	O	
4						x	x	x	x	x	O							x	x	x	x	x	O						x	x	x	x	x	O			

- T2 d'une durée de TH, toutes les deux secondes
- T3 d'une durée de 3TH, toutes les trois secondes
- T4 d'une durée de 6TH, toutes les quatre secondes

Comme l'exige cette méthode d'ordonnement, nous avons effectué en priorité les tâches avec la deadline la plus proche. En cas de deadlines synchrones, nous avons décidé de poursuivre la tâche en cours jusqu'à fin de l'exécution.

De façon assez surprenante, nous n'avons pas eu besoin de découper les tâches en morceaux pour cette situation.

Ordonancement avec priorités fixes

Question 13

On attribue des priorités à chacune des tâches :

- T2 priorité 2
- T3 priorité 3 (plus importante)
- T4 priorité 1 (la plus faible)

Les priorités sont fixes et non modifiables pendant toute la durée de l'exécution.

Voici le résultat que nous obtenons, en supposant que les trois tâches démarrent de façons synchrone.

t			1		2		3		4		5		6		7		8		9		10		11		12
2				O																					
3	x	x	O																						
4					x	x																			

On remarque que dès la 4ème seconde, la tâche T4 n'a pas pu d'exécuter complètement puisque la priorité de T3 est supérieure à celle de T4.

En donnant 3,2,1 comme ordre de priorités, on observe le même problème. Si on donne à T4 la plus grande priorité, T2 n'aura pas le temps de d'exécuter avant sa deadline.

Ainsi, il n'est pas possible d'avoir un système ordonnancable qui utilise des priorités fixes.

Considérons maintenant un processeur 1,3 fois plus puissant.

- T2 doit durer 0,256 secondes
- T3 doit durer 0,769 secondes
- T4 doit durer 1,538 secondes

Par soucis de représentation des résultats, on arrondit ces durées comme ci-dessous :

QT = 0,25 s

- T2 doit durer 0,25 secondes (1 QT)
- T3 doit durer 0,75 secondes (3 QT)
- T4 doit durer 1,5 secondes (6 QT)

Voici donc une nouvelle table qui nous permet de visualiser l'exécution de ces tâches avec ces nouvelles contraintes.

t				1			2				3				4				5				6				7				8
2				O											/																
3	x	x	O												/	x	x	O	/												
4					x	x	x	x																							

Les priorités suivantes ont été choisies :

- T2 priorité 2
- T3 priorité 3 (plus importante)
- T4 priorité 1 (la plus faible)

Les « / » symbolisent les moment ou l'ordonnanceur n'a aucune tâche à effectuer.

Il est inutile de faire une trop grande représentation, puisqu'on remarque rapidement que avec une telle vitesse toutes les tâches ont largement le temps de s'exécuter.