TP 2

Systèmes temps-réel

Le but du TP est de se familiariser avec l'API POSIX permettant de créer des tâches temps-réel, en implémentant un certain nombre de situations courantes.

Rédigez un compte-rendu tout le long du TP, et pour chaque question d'implémentation, placez votre code dans un fichier `tp_##.c` différent correspondant au numéro de la question.

1 Introduction

La première fonction permettra de simuler un calcul d'un certain temps pour les experimentations.

Question 1. Écrivez une fonction `void do_work( unsigned int duration)` qui occupe le processeur pendant `duration` millisecondes. Effectuez les mesures avec la commande UNIX `time`, et quantifiez l'imprécision de la mesure.

Notes :

- Cette fonction est à calibrer selon la vitesse de votre machine. La commande UNIX `time` peut vous aider, mais attention au temps de lancement du programme!
- Utilisez les mêmes options d'optimisation pour tout le TP, car la vitesse d'execution de votre fonction en dépend beaucoup. Il est preferable de compiler sans optimisation.
- Si vous compilez avec des options d'optimisation, le compilateur risque de retirer une partie de votre code. L'emploi de
  `asm volatile ("nop")` permet de faire croire au compilateur que votre programme effectue un vrai travail.

2 Tâches sporadiques

Une tâche *sporadique* effectue un *travail* en réponse à un *évènement*. Ces évènements sont souvent des interruptions matérielles, mais ici nous allons simuler cet environnement par l'utilisation d'interruptions logicielles, implémentées sous UNIX à l'aide des signals.

Question 2. Implémentez un programme qui affiche un message à l'écran lorsque le signal SIGUSR1 lui est envoyé.

Les fonctions nécessaires pour l'implémentation de cette fonction sont `signal`, `pause`, et eventuellement `getpid`. Pour rappel, la commande `kill` permet d'envoyer un signal à un processus.

Question 3. Utilisez la fonction `do_work()` pour faire un travail plus conséquent dans le signal handler, et envoyez une "rafale" de signaux à la fonction. Que remarquez vous? Quelle est la condition sur les évènements envoyés qui permette d'éviter ce problème?

Question 4. Proposez et implémentez un mécanisme permettant d'améliorer le traitement de rafales d'interruptions. Est-ce que le problème est complètement résolu?

Question 5. En vous inspirant du mécanisme précédent, proposez et implémentez un nouveau mécanisme permettant de vérifier que l'interruption précédente a bien été traitée quand on recoit la suivante.

3 Tâches périodiques

3.1 Introduction

Si les tâches précédentes travaillaient sur arrivée d'un évènement (programmation event-triggered), les tâches suivantes déclenchent des travaux à certains moments dans le temps (programmation *time-triggered*).

Les tâches *périodiques* sont un certain type de tâches time-triggered, pour lesquelles un travail est effectué à intervalle régulier appellé *période*.

Question 6. Implementez une tâche périodique de période une seconde et qui execute un travail de 500ms de deux manière, l'une en utilisant `sleep()`, l'autre en utilisant `alarm()`. Quels sont les problèmes de cette méthode?

Ces limitations ont conduit la norme POSIX à s'enrichir de certaines extensions pour la gestion des timers.

Question 7. Implémentez une tâche périodique POSIX, à l'aide des fonctions `timer_create` et `timer_settime`.

Sous Linux, vous risquez d'avoir des problèmes d'édition de lien (fonction `timer_create` non définie). Utilisez le flag `-lrt` pour pouvoir compiler le programme.

3.2 Programmation en boucle

Soit un système nécessitant trois travaux périodiques T_2, T_3, T_4 telles que:

- T_2 doit executer une fonction `t2()` de durée 0.333s toutes les 2 secondes,
- T_3 doit executer une fonction `t3()` de durée 1s toutes les 3 secondes,
- T_4 doit executer une fonction `t4()` de durée 2s toutes les 4 secondes.

Question 8. Implémentez les trois tâches selon une approche de programmation en boucle, i.e. dans une seule grande tâche périodique.

Notes :

- Le découpage des traitements `t2()`, `t3()`, `t4()` en morceaux de durée arbitraire est autorisé, mais essayez de minimiser le nombre de découpages.
- En réalité, il est difficile de découper un programme en plusieurs morceaux d'une certaine durée; c'est ce qui fait la difficulté majeure d'application de la méthode.
- Une implémentation réelle de programmation en boucle ne comporte qu'une seule tâche périodique (même pas d'OS). Les temps d'exécution dans la tâche restent à peu près constant, ce qui facilite son emploi.
- Votre programme risque donc de ne pas fonctionner à cause de l'activité de l'OS; dans ce cas "augmentez" la vitesse du processeur en modifiant votre fonction `do_work()`.

Question 9. On remplace le processeur du système par un processeur 2 fois plus rapide (tous les traitements durent deux fois moins longtemps). Votre programme fonctionne-t-il toujours? Si non, proposez-en un qui marche indépendement de la vitesse d'execution. Trouvez experimentalement à partir de quelle vitesse de processeur vous pouvez executer le programme.

Question 10. L'exécution est elle déterministe (i.e. predictible et reproductible)?

3.3 Programmation preemptive

On reprend les trois travaux précédent, mais en les implémentant dans trois tâches périodiques différentes, de manière préemptive.

Question 11. Implémentez les travaux T1, T2, T3 sous forme de trois tâches périodiques.

Ordonnancement avec EDF L'algorithme EDF (earliest deadline first) ordonnance les tâches de la manière suivante:

- à tout moment, la tâche executée a l'échéance la plus proche.
- si une nouvelle tâche est prête dans le système et a une échéance plus proche que la tâche courante, alors la tâche courante est preemptée et la nouvelle tâche est executée à la place.

Question 12. En partant de l'hypothèse que les trois tâches démarrent en même temps de manière synchronisée, montrez que les trois tâches s'executent toujours correctement avec EDF

Il est difficile d'implémenter des ordonnanceurs en espace utilisateur sous Linux, donc en particulier EDF.

Ordonnancement à priorité fixe Les ordonnancements à priorité fixe se déroule de la manière suivante:

- à tout moment, la tâche executée a la priorité la plus elevée,
- si une nouvelle tâche est prête dans le système qui a une priorité plus elevée que la tâche courante, alors la tâche courante est preemptée et la nouvelle tâche est executée à la place.

Question 13. Peut-on trouver une répartition des priorités entre les tâches T_2, T_3, T_4 tel que le système soit ordonnancable? Et si le processeur est 1.3 fois plus puissant?

Question 14. Implementer le système sur trois tâches en leur affectant des priorités, et vérifier que ca marche (processeur 1.3 fois plus puissant)

Les fonctions permettant de changer la priorité d'un processus sous UNIX sont `sched_setparam` et `setpriority`.