Master 1 - FST - Dpt. Info - 2016-2017
INF1105M Programmation Système Et Temps Réel
L. Gonnord & T. Excoffier
http://perso.univ-lyon1.fr/thierry.excoffier/COURS/temps_reel.html

TP système généraliste et temps de réponse

Objectifs

— Démontrer expérimentalement que Linux n'est pas un système temps-réel.
— Jouer avec les horloges et les signaux.
— Présenter des résultats expérimentaux sous forme graphique.

Crédit Cet énoncé reprend l'énoncé de T. Excoffier sur le même sujet (cf http://perso.univ-lyon1.fr/thierry.excoffier/COURS/COURS/TEMPS_REEL/HTML/tp_unix.html).

Durée 2 séances de 2h30. Rapport à rendre à la date indiquée sur TOMUSS. Binômes autorisés.

1 Problématique

Comme vu en cours, les systèmes généralistes ne sont pas temps-réel. Le but de ce TP est de montrer les variations de durée que l'on peut observer dans un cas particulier simple, *l'appel de fonction*. Dans ce TP, nous allons donc à intervalles réguliers, appeler une fonction `ma_fonction` (qui ne fait rien de particulier). Des horloges/timers seront utilisés afin de mesurer le **retard de déclenchement**, ie la différence entre la date d'appel de la fonction et la date d'exécution de la **première** instruction de cette fonction [1] :

— avec différentes façons de déclencher la mesure (attente active ou méthode passive) ;
— en stressant plus ou moins le programme (période plus ou moins courte) ;
— dans le cas d'une machine peu chargée ou très chargée, qui imprime ses résultats de différentes façons (`stdout`, fichier) ;
— dans le cas d'un processus à priorité (au sens unix) normale ou en changeant la priorité, ou encore dans le cas d'un *thread temps réel* (optionnel) ;
— (optionnel) machine monoprocesseur versus multiprocesseur.

2 Un peu d'aide

2.1 Le squelette du programme

Vous écrirez les fonctions/structures de données suivantes :

— Un `main` sera responsable de l'appel (direct ou indirect) à `ma_fonction` toutes les `dt` microsecondes ($dt = 100, 1000, \ldots$).

1. Attention, il est possible que ce retard soit une avance, pourquoi ?

— `ma_fonction` réalisera le travail demandé, suivant la méthode de déclenchement utilisée. La fonction sera responsable de la mesure du retard. Le i-ème appel de fonction fera l'objet d'une mesure t_i et le retard $r_i = t_i - (t_0 + i \times dt)$ sera stocké quelque part (t_0 est la date de la première mesure de temps de la fonction), on verra plus loin comment.

— Dans le cas de l'attente active (le cas le plus simple à traiter). Le `main` passera son temps à regarder l'heure et il déclenchera la fonction tous les `dt` millisecondes :

```
tant que simu_pas_finie
    regarder l'heure
    si heure > heureprec + dt alors
      ma_fonction_active(...)
fin tant que
```

— Dans le cas de la méthode passive on traitera le cas un peu plus compliqué de la synchronisation par timer. Le `main` configurera un *timer* qui envoie de manière périodique un *signal* :

```
mettre_en_route_timer
tant que simu_pas_finie
    pause() // La fonction pause attend le signal, et termine.
fin tantque
```

— Il faudra inventer des structures de données qui permettent de stocker les différentes valeurs, les configurations ; ainsi que des fonctions d'impression de ces structures de données. Il est recommandé de ne pas stocker toutes les différentes valeurs de retard/avance car il il en a beaucoup, mais un histogramme du nombre de retards par tranches de retard/avance. L'axe horizontal de l'histogramme est en log de 10. L'affichage pourrait alors être [2].

```
appels en avance                    exact                   appels en retard
  6     5     4     3     2     1     0     1     2     3     4     5     6
00000 00000 00000 00000 00000 00000 00033 00065 00000 00001 00000 00000 00001
```

2.2 Gestion du temps

Vous aurez besoin des fichiers entête suivants :

```
#include <stdio.h>    /* printf()                    */
#include <unistd.h>   /* pause()                     */
#include <time.h>     /* clock_gettime(), timer_settime() */
#include <signal.h>   /* signal(), SIGALRM           */
```

2.2.1 Horloge, calcul du temps

Pour calculer le temps, il est conseillé d'utiliser une horloge de type `CLOCK_MONOTONIC` (`clockid`) et la fonction `clock_gettime`. Les intervalles de temps utiliseront la structure suivante, définie dans `time.h` [3] :

```
struct timespec {
        time_t tv_sec;        /* seconds */
        long tv_nsec;         /* nanoseconds */
};
```

2. la colonne 1 compte le nombre de mesures entre 1 et 100 unités de temps, la colonne 2 entre 101 et 1000, ...
3. Ne pas oublier `-lrt` à l'édition de lien

Divers conseils :

— Une nanoseconde vaut 10^{-9} seconde.
— On pourra écrire des fonctions de soustraction de deux `timespec`.
— Le manuel est votre ami.

2.2.2 Timers, signaux, handlers

Pour la partie “réveil asynchrone” (méthode passive, et uniquement celle-ci), on utilisera un timer périodique :

```
int timer_create(clockid_t clockid, struct sigevent *sevp,
                         timer_t *timerid);

int timer_settime(timer_t timerid, int flags,
                        const struct itimerspec *new_value,
                        struct itimerspec * old_value);
```

Un peu d'explications supplémentaires :

— La structure de type `sigevent` sert à embarquer des informations avec le signal. Cela va nous servir à dire quelles actions faire lorsque le signal est reçu (voir plus bas).
— À la création du *timer*, il faut passer un clockid égal à `CLOCK_MONOTONIC`, et une adresse pour stocker l'id du timer (voir le man de `timer_create` pour un exemple). Cet id sera utilisé pour configurer le timer avec `settime`.
— Pour configurer la période du timer avec `settime`, vous pouvez vous reporter à l'exemple de la page de man de `timer_create`.
— Périodiquement, ce timer enverra le signal `SIGALRM`. Il faudra donc mettre le champ `SIGEV_SIGNAL` dans le champ `sigev_notify` de la structure `sigevent`, ainsi que `SIGALRM` dans le champ `sigev_signo` (man sigevent).
— La structure `sevp` de type `sigevent*` peut contenir plus d'informations, et en particulier une partie des statistiques que vous voulez calculer. Si vous ne voulez pas vous en préoccuper, vous pouvez utiliser des variables globales, mais ce n'est pas recommandé.

Pour récupérer le signal envoyé par le *timer*, il faudra utiliser une structure de type `sigaction`, que vous pouvez initialiser (dans le `main`, avant de déclencher le timer) de la façon suivante :

```
struct sigaction sa ;
memset(&sa, '\0', sizeof(sa)) ;
sa.sa_sigaction = ma_fonction_passive ;
sa.sa_flags = SA_SIGINFO ;
sigaction(SIGALRM, &sa, NULL) ;
```

La fonction `pause()` sera ici utile, on vous laisse découvrir pourquoi.

2.3 Threads temps-réel

Pour cette partie (facultative) on vous demande de regarder ce qui se passe lorsque c'est une `thread` temps réel qui lance périodiquement la fonction.

Rapportez-vous au cours. N'oubliez pas de compiler avec `-D_REENTRANT -lpthread`. Quelques remarques :

— Vous pouvez changer la priorité d'une thread avec la fonction `pthread_setschedparam`.
— Vous pouvez utiliser `pthread_t pthread_self()`.
— Faire attention aux droits de lancement (seul *root* peut passer en priorité temps-réel).

2.4 Automatisation des tests

Avant d'automatiser les tests, vous ferez en sorte que :

— la ligne de commande (de votre programme C) permette de choisir :
  — la période entre deux appels de fonctions (paramètre `dt`, en μs, par exemple 100, 1000, 10000, 100000) ;
  — la durée de l'expérience ;
  — des impressions plus ou moins verbeuses sur `stdout` ou des fichiers.

— vous sachiez (shell) choisir les priorités (`nice`), charger la machine (construire une énorme archive de fichiers système en parallèle semble une bonne idée [4]), lancer sur un coeur/un processeur (`taskset`), ...

Pour automatiser les tests, il est demandé d'écrire un script (Shell, Python, ...), qui permettra de lancer le programme dans différentes configurations (logiciel/système). Vous commencerez par faire des acquisitions courtes, afin de bien vérifier vos résultats avant de faire de grandes acquisitions de données [5]. Loguer les tests dans des fichiers permet éventuellement de ne pas relancer tous les tests en cas de plantage [6].

2.5 De jolis graphiques

Vous représenterez vos résultats sous la forme de graphiques résumant les résultats expérimentaux. Ces graphiques pourront être de différentes formes. Nous recommandons d'utiliser mathplotlib, où vous pourrez trouver une jolie galerie d'exemples : http://matplotlib.org/examples/index.html

Il est recommandé de bien réfléchir au nombre et au(x) type(s) de graphique(s) que vous allez montrer, notamment pour permettre les comparaisons décrites plus haut (variance, moyenne de performances dans différentes configurations). Il conviendra aussi de "grouper" les différentes mesures que vous faites pour que le(s) graphique(s) soient lisibles.

3 Rendu

Vous rendrez votre TP sous la forme d'une archive `tgz` sous TOMUSS [7]. Cette archive comprendra :

— Vos codes source documentés (sans produit de compilation).

— Les données acquises.

— Un rapport de 2 à 5 pages max en format PDF qui explique l'architecture de votre code ainsi que vos résultats expérimentaux. Vous expliquerez clairement quelles sont les parties que vous avez traitées, et les configurations testées (quelles machines, leurs caractéristiques). Rappel : un graphique non expliqué ne vaut rien.

4. ATTENTION : quand vous faites un `tar`, les accès disques en écriture n'ont pas lieu au moment où le `tar` est fait mais lorsque le système décide d'écrire sur le disque. Ceci peut avoir lieu plusieurs dizaines secondes après la fin de la commande `tar`. Il n'y a aucun moyen simple d'annuler ces écritures. À vous d'être astucieux pour l'exécution des tests.

5. Attention à bien dimensionner vos tests, faites des calculs, l'acquisition peut durer une nuit si vous prenez 100 secondes par test.

6. Le traitement Python du format csv est particulièrement efficace.

7. Il va sans dire que ce travail est personnel et propre à chaque binôme. Toute copie sera sanctionnée par un 0 non négociable.