TP2 – Système temps réels

FABRE Adrien & CHEKROUN Thomas

03/04/2015

Question 1

La fonction “do-work” va nous permettre de simuler une action d’une durée « duration » exprimé en milliseconde.

```
void *do_work(int duration)
{

    unsigned int i=0;
    while(i< (duration * 320000))
    {
        ++i;
        asm volatile ("nop");
    }
}
```

Pour implémenter cette fonction nous utilisons l'instruction *asm volatile ("nop");* qui permet de faire exécuter un réel calcul au processeur sans risque que le compilateur n'optimise la fonction et fausse nos mesures. Nous répétons donc cette instruction un nombre de fois proportionnel à la durée qui est passée en paramètre pour occuper notre processeur le temps voulu. Comme nous avons du trouver empiriquement la valeur de ce coefficient, la précision de notre fonction n'est bien sûr pas parfaite.

Après avoir trouvé le bon coefficient nous avons effectué un dernier test en passant à notre fonction le paramètre *6000* pour occuper notre processeur 6 secondes et voici le résultat de la commande *time ./o/tp2_1.c*:

```
adrien@samsung-adrien time ./o/tp2_1                ~/real_time_computing/TP2
my pid: 30646
./o/tp2_1  6,16s user 0,00s system 99% cpu 6,174 total
```

Nous avons donc une exécution à peu près comparable au temps que nous souhaitons (il est tout de même à noter que ce temps d'exécution peut varier de plus ou moins 1.5s au cours de nos tests, nous devrons donc garder à l'esprit au long de ce TP que nos mesures sont approximatives)

Question 2

La première étape pour faire réagir notre programme à un signal est d'implémenter la fonction qui sera exécutée à l'arrivée de ce signal. Ici la fonction affichera simplement un message prévenant de l'arrivée du signal:

```
/* Handles a signal by printing its number when the signal arrives */
void signalHandler(int signum){
    printf ("\nHey! You got a signal %d\n", signum);
}
```

Il nous faut ensuite préciser dans la méthode main que nous souhaitons lancer la méthode précédente à l'arrivée du signal. C'est ce que nous faisons avec la ligne 20 dans la capture suivante. L'appel à la méthode do_work() nous permet de faire s'exécuter le programme pendant un temps assez long pour que nous puissions tester l'envoi du signal:

```
16 int main (int argc, const char* argv[])
17 {
18     printf ("my pid: %d\n", getpid());
19     /* handler of SIGUSR1 signal */
20     signal (SIGUSR1, signalHandler);
21
22     do_work(60000);
23
24 }
```

Nous pouvons maintenant tester le bon fonctionnement de notre programme en lui envoyant un signal USR1 via la fonction shell kill. La capture suivante présente une exécution de notre programme durant laquelle nous lui avons envoyé 2 fois un signal:

```
adrien@samsung-adrien ./o/tp2_2                    ~/real_time_computing/TP2
my pid: 31733

Hey! You got a signal 10

Hey! You got a signal 10
adrien@samsung-adrien                              ~/real_time_computing/TP2
```

```
adrien@samsung-adrien kill -USR1 31733             ~/real_time_computing/TP2
adrien@samsung-adrien                              ~/real_time_computing/TP2
adrien@samsung-adrien kill -USR1 31733             ~/real_time_computing/TP2
adrien@samsung-adrien                              ~/real_time_computing/TP2
```

Question 3

On rajoute une exécution de la fonction do_work() dans le signal handler pour simuler un traitement long. En envoyant une rafale de signaux on remarque que tous les signaux envoyés ne sont pas interceptés comme le montre la capture d'écran suivante:

```
adrien@samsung-adrien ./o/tp2_3                       ~/real_time_computing/TP2
my pid: 443

Hey! You got a signal 10

Hey! You got a signal 10
adrien@samsung-adrien                                 ~/real_time_computing/TP2
```

```
adrien@samsung-adrien kill -USR1 443               [1]~/real_time_computing/TP2
adrien@samsung-adrien kill -USR1 443                  ~/real_time_computing/TP2
adrien@samsung-adrien kill -USR1 443                  ~/real_time_computing/TP2
adrien@samsung-adrien kill -USR1 443                  ~/real_time_computing/TP2
adrien@samsung-adrien kill -USR1 443                  ~/real_time_computing/TP2
adrien@samsung-adrien kill -USR1 443                  ~/real_time_computing/TP2
adrien@samsung-adrien kill -USR1 443                  ~/real_time_computing/TP2
adrien@samsung-adrien kill -USR1 443                  ~/real_time_computing/TP2
adrien@samsung-adrien kill -USR1 443                  ~/real_time_computing/TP2
adrien@samsung-adrien kill -USR1 443                  ~/real_time_computing/TP2
adrien@samsung-adrien                                 ~/real_time_computing/TP2
```

Sur 10 signaux envoyés au programme seulement 2 sont interceptés.

Cela vient du fait que l'OS utilise un registe à un bit à 0 ou à 1 pour indiquer qu'un signal a été reçu. Ainsi si le traitement du signal est long le bit n'a pas le temps d'être remis à 0 avant l'arrivée du signal suivant et certains signaux ne sont donc pas "vus".

La condition pour que les signaux soient tous interceptés serait qu'ils soient envoyés à un intervalle de temps plus grand que le temps d'exécution de la fonction *signalHandler()*.

Question 4

Pour améliorer le traitement des rafales nous allons mettre en place une variable globale qui servira à compter nos interruptions. Notre fonction de *signalHandler()* n'aura plus pour rôle d'exécuter un traitement lourd à l'arrivée d'un signal mais simplement d'incrémenter notre compteur global. Afin de garder une trace de nos interruptions nous créons également un tableau contenant leurs numéros.

Ensuite dans notre fonction main() nous dépilerons notre historique d'interruption et pour chacune nous effectuerons le traitement nécessaire:

```c
int CPT = 0;
int sigs[20];

/* function which pause the program with a millisecond precision */
void *do_work(int duration);

/* Handles a signal by printing its number when the signal arrives */
void signalHandler(int signum){
    sigs[CPT] = signum;
    ++CPT;
    printf ("Hey! You got a signal %d (CPT: %d)\n", signum, CPT);
}

int main (int argc, const char* argv[])
{
    printf ("my pid: %d\n", getpid());
    /* handler of SIGUSR1 signal */
    signal (SIGUSR1, signalHandler);

    do_work(60000);
    /* In this loop we handle the interruptions */
    while (CPT>0)
    {
        --CPT;
        printf("traitement du signal %d\n", sigs[CPT]);
        do_work(10000);
    }
}
```

Cette solution améliore le traitement des rafale mais elle ne le résout pas complètement: en effet si les signaux arrivent à un intervalle de temps inférieur à celui nécessaire à l'incrémentation du compteur certains seront toujours perdus.

Question 6

Implémentation d'une tâche périodique avec *sleep()*

La méthode *sleep()* à pour fonction d'endormir le processus pendant une période de temps qui lui est passée en paramètre.

Notre tâche à une période de 1s et un temps d'exécution de 500ms. Pour l'implémenter nous devons donc faire appel à notre méthode *do_work()* en lui passant en paramètre 500 et pour obtenir une période totale de 1s nous devons faire appel à la méthode *usleep()* en lui demandant une pause de 500ms. (Il est à noté que nous avions pendant un moment fait l'erreur de faire appel à la méthode *sleep()* en lui passant en paramètre 1 en plus de l'appel à *do_work()*).

L'implémentation est donc la suivante:

```c
int main (int argc, const char* argv[]) {
    int cpt = 0;

    while (cpt<20) {
        cpt++;
        printf("iteration %d\n", cpt);
        usleep(500);
        do_work(500);
    }
}
```

Implémentation d'une tâche périodique avec alarm*()*

La méthode *alarm()* à pour rôle d'envoyer un signal SIGALRM à un intervalle de temps régulier qui lui est passé en paramètre. Cette fonction s'utilise couplée avec la fonction pause() dont le rôle est de mettre le programme en pause en attendant l'arrivée d'un nouveau signal.

Nous allons donc dans chacune des itérations de notre boucle faire appel à la fonction *alarm()* en lui passant un paramètre de 1 qui représente le normbre de secondes avant lequel elle enverra son signal, on effectue ensuite la tâche de 500ms représentée par l'appel à *do_work()* puis on met le programme en pause. Après 500 nouvelles ms le signal programmé par *alarm()* est reçu par le programme qui se remet en marche après avoir effectué une tâche périodique de 1s:

```c
int main (int argc, const char* argv[]) {
    int cpt = 0;

    while (cpt<20) {
        cpt++;
        printf("iteration %d\n", cpt);
        alarm(1);
        do_work(500);
        pause();
    }
}
```

Il est a noter que par défaut la réception d'un signal SIGALRM met fin au processus nous avons donc du rajouter une méthode signalHandle() pour pouvoir continuer nos itérations. Cet ajout est présent dans nos codes sources mais n'est pas visible sur cette capture d'écran.

Le problème lié à l'utilisation de ces deux méthodes est le fait que le programme fait appel à des fonctions externes. Cela risque de causer un décalage dans le temps parce que les fonctions externes n'utilisent pas les même timers que le programme ce qui peut être très problématique dans un environnement temps-réel critique.

Question 7

Cette fois nous allons implémenter notre tâche périodique en utilisant un timer UNIX. Ce timer permet de définir un interval de temps avant lequel il enverra un signal. La différence avec la fonction alarm() de la question précédente est que l'on peut choisir quelle horloge le time utilise pour mesurer le temps. Nous avons donc créé une fonction start_timer permettant d'initialiser et de lancer un timer:

```
void start_timer(void)
{
    struct itimerspec value;

    /* les deux variables doivent avoir la meme valeur pour
     * permettre au timer de se réarmer à chaque fois
     */
    value.it_value.tv_sec = 1;
    value.it_value.tv_nsec = 0;

    value.it_interval.tv_sec = 1;
    value.it_interval.tv_nsec = 0;

    timer_create (CLOCK_REALTIME, NULL, &gTimerid);
    timer_settime (gTimerid, 0, &value, NULL);
}
```

La structure value contient deux champs qui représentent chacun un temps (avec une partie en secondes et une autre en millisecondes). Ces temps sont en fait le temps avant l'exécution du timer et le temps avant son réarmement. Dans notre cas nous les voulons en même temps pour que le timer fonctionne infiniment.

(Si l'on met:

- Value = 0 le timer ne se lancera jamais,
- Interval = 0 il ne se lancera qu'une seule fois.)

Le premier argument de la fonction *timer_create()* est la référence de l'horloge que le timer utilisera pour compter le temps et le troisième argument est un ID unique permettant d'indentifier le timer dans le programme. La fonction *timer_settime()* parle d'elle même.

Une fois initialisé et démarré de la sorte le timer enverra toutes les secondes un signal *SIGALRM*, que nous attraperons donc dans notre mains comme nous savons déjà le faire:

```c
void timer_callback(int sig) {
    printf("Nouveau signal envoye par le timer\n", sig);
    do_work(500);
}

int main(int ac, char **av)
{
    (void) signal(SIGALRM, timer_callback);
    start_timer();
    while(1);

}
```

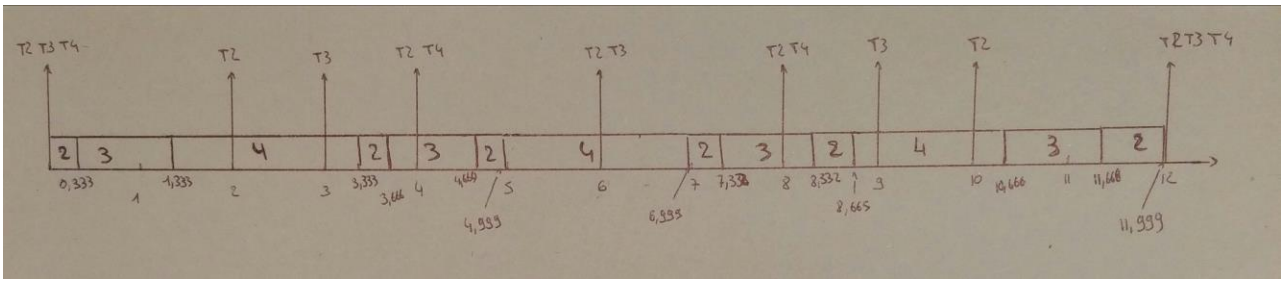
Notons qu'une fois le timer lancé, une boucle while vide permet au programme de rester actif pour récupérer les signaux du timer.

Question 8

Le sujet nous donne les trois taches telles que:

Tache	Temps d'exécution	Période
T2	1/3s	2s
T3	1s	3s
T4	2s	4s

Ce qui nous donne la politique d'ordonnancement suivante:



On remarquera que pour plus de simplicité, nous n'avons pas découpé les tâches en plusieurs parties afin de pouvoir implémenter chaque tâche avec une seule fonction. La capture suivante montre donc l'implémentation de la fonction main lançant les tâches dans l'ordre défini précédemment:

```c
int main(int ac, char **av) {

    while (1) {
        printf("BOUCLE!\n");
        T2();
        T3();
        T4();
        T2();
        T3();
        T2();
        T4();
        T2();
        T3();
        T2();
        T4();
        T3();
        T2();

    }

}
```

De plus les fonctions T2(), T3() et T4() représentent nos trois tâches:

```c
void *T2()
{
    int duration=333;
    unsigned int i=0;
    printf("T2\n");
    while(i< (duration * 320000))
    {
        ++i;
        asm volatile ("nop");
    }
}
```

```c
void *T3()
{
    int duration=1000;
    unsigned int i=0;
    printf("T3\n");
    while(i< (duration * 320000))
    {
        ++i;
        asm volatile ("nop");
    }
```

```
void *T4()

    int duration=2000;
    unsigned int i=0;
    printf("T4\n");
    while(i< (duration * 320000))
    {
        ++i;
        asm volatile ("nop");
    }
```

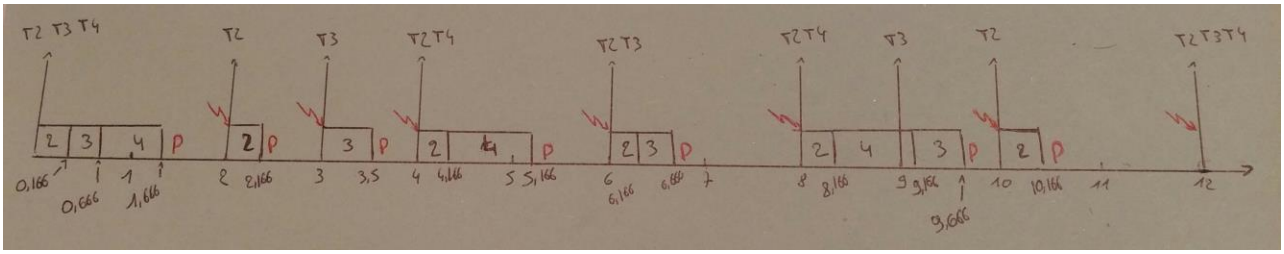
Avec la commande time() on observe un temps d'exécution d'environ 11 secondes ce qui est relativement proche du temps souhaité (la différence venant de l'imprécision du temps d'exécution des tâches définies de manière empirique.

Question 9

Cette fois nous doublons la vitesse du processeur en ce qui a pour effet de réduire de moitié le temps d'exécution de nos tâches nous nous trouvons avec la configuration suivante:

Tache	Temps d'exécution	Période
T2	1/6s	2s
T3	1/2s	3s
T4	1s	4s

Observons l'ordonnancement que nous avons produit avant de le commenter:



Ici les tâches sont toujours exécutable dans le temps qui nous est imparti mais le problème vient du fait que certaines tâches risquent de commencer avant leur échéance. Par exemple à 1.666s si l'on ne prévoit rien, la tâche 2 se lancera alors qu'elle ne doit pas commencer avec 2s.

Notre solution est la suivante: nous allons instaurer des pauses dans le programme pour permettre aux tâches de se lancer au bon moment. Deux solutions s'offrent à nous pour mettre en place ces temps d'attente: utiliser la fonction sleep() (ou usleep()) ou utiliser la fonction pause(). L'inconvénient de la fonction sleep() est qu'elle demande à ce qu'on lui passe en paramètre le temps que doit attendre le programme, on pourrait donc faire marcher le code pour une vitesse de calcul donnée mais en changeant cette vitesse l'ensemble du programme serait à re-coder.

Nous allons donc utiliser la méthode pause() qui permet de mettre en pause le programme pendant un temps indéfini: la reprise s'effectuera à la réception d'un signal. Sur notre schéma les "P" représentent les invocations des pauses et les éclairs rouges l'envoi des signaux pour réveiller le

programme. Nous pouvons observer que ces signaux sont à envoyer aux secondes 2, 3, 4, 6, 8, 10 et 12 de chaque itération.

Le code de la boucle principale est donc le suivant:

```
int main(int ac, char **av) {

    (void) signal(SIGALRM, timer_callback);
    start_timer();
    while (1) {
        printf("BOUCLE!\n");
        T2();
        T3();
        T4();
        pause();
        T2();
        pause();
        T3();
        pause();
        T2();
        T4();
        pause();
        T2();
        T3();
        pause();
        T2();
        T4();
        T3();
        pause();
        T2();
        pause();
    }

}
```

Il nous faut également un timer envoyant un signal toutes les 2 secondes, pour cela nous reprenons le timer de la question précédente en changeant les valeurs. De plus il nous faut également un timer qui attende 3secondes avant de lancer un signal puis devienne périodique sur 12 secondes. Nous n'avons cependant pas réussit l'implémentation de ce timer, nous pensons toutefois qu'il doit être possible de mettre en place une fonction appelée à la première réception d'un signal qui mette ensuite change les valeurs du timer pour qu'il devienne périodique sur 12 secondes.

Question 10

Tant que la vitesse du processeur permet d'exécuter toutes les taches dans le temps imparti le fonctionnement est déterministe puisque l'intérêt d'utiliser notre système de pause et de signal est que les taches seront toujours lancées à la bonne échéance quelle que soit l'avance que les taches précédentes aient pu prendre.