Systèmes

Temps

Réel

Liste des TD :

Séance n° 1 : Etude de l'ordonnancement sous Linux (PC sous Debian)
Séance n° 2 : Génération de signaux temps réel via des GPIO sous Linux (carte Beaglebone)
Séance n° 3 : Gestion des événements sous Linux (étude des temps de réponse)

Document initialement rédigé par G. Jacquet et modifié par A. Aubert

Consignes pour l'utilisation de la carte BeagleBone.

Dans cette série de TP, vous allez utiliser une carte BeagleBone (sauf pour la première séance). Cette carte permet un accès facile aux ports d'entrées/sorties du micro-contrôleur am335x de Texas Instrument. On pourra y travailler soit en baremetal , soit avec un système d'exploitation mono-tâche, soit sous Linux (système d'exploitation multi-tâches).

Le développement d'applications pour cette carte se fera à partir d'un PC avec la chaîne de développement croisée Buildroot sous Linux Debian. Ce système d'exploitation sera mise en œuvre sur une machine virtuelle « VirtualBox » à partir de Windows : le nom de la machine est elec6, vous devez la charger à partir du partage enseign. Le répertoire d'installation par défaut sera d:\MV\elec6.

Deux utilisateurs sont définis :

- root (superutilisateur avec tous les droits sur la machine) mot de passe admintest
- tpuser (utilisateur classique avec des droits réduits) mot de passe tpuser

Remarque importante : la carte Beaglebone s'alimente en 5V directement via le port USB.

Autre remarque importante : cette carte « low cost » ne possède pas de protections sur ses entrées/sorties, il faudra donc être très prudent sur la connexion des fils extérieurs ainsi que sur la programmation des ports pour ne pas réaliser de court-circuit.

Séance n°2 : Génération de signaux temps réel via des GPIO sous Linux (carte Beaglebone)

Conseil de mise en oeuvre

Récupérer, via le partage, le répertoire TD2_gpio_linux et le placer sous votre répertoire de travail (/home/tpuser). Plusieurs sous-répertoires correspondant aux questions suivantes sont déjà créés.
Les Makefile présents dans les répertoires copient aussi le fichier exécutable dans le répertoire /share qui doit être partagé, celui-ci permettra d'exécuter les programmes à partir de la Beaglebone via un partage NFS. Il suffit alors de démarrer la beaglebone avec le système de fichier sur la SDCard et de monter le répertoire précédent sur le chemin /mnt (vous devrez bien entendu faire les opérations nécessaires que l'on ne vous rappelle pas ici !).
(si vous le désirez vous pouvez aussi réaliser le boot complet en NFS en modifiant le Makefile pour copier dans /targetfs)

Objectif de la séance

Les contraintes de temps (ou de temps réel) peuvent prendre deux aspects suivant l'objectif recherché :
- un traitement en flux qui doit produire sur une sortie un signal issu du traitement d'un ou de plusieurs signaux d'entrée. La contrainte principale est alors l'obtention d'une sortie après traitement sans perdre des informations d'entrée (c'est le respect d'un flux de données). Une contrainte annexe pouvant être le temps de décalage maximum admissible entre entrée et sortie.
- une contrainte de temps de réaction à un événement externe (alarme → action). L'événement pouvant être peu fréquent.

Lors de cette séance nous allons étudier la mise en œuvre sous Linux de contraintes de type « traitement en flux ». La séance 4 sera quant à elle consacrée à l'étude des réactions à un événement.
Afin de simuler un système de traitement en flux avec de fortes contraintes de temps, nous allons générer des signaux sur des ports GPIO de la carte Beaglebone. Nous pourrons visualiser ces signaux via un oscilloscope. Relativement à la séance précédente nous pourrons ainsi étudier plus précisément l'ordonnancement et les contraintes de temps sur un système multi-tâches. Plusieurs éléments seront à prendre en compte :
- la vitesse de génération de ces signaux (fréquence maximum atteignable)
- la continuité de la génération (avec la mise en évidence de temps inactif)

II.1 Lancement de 2 tâches concurrentes de génération de signaux

II.1.1 Génération d'un signal d'horloge (répertoire Q1_sys_gpio)

A l'aide du programme « sys_gpio1.c » fourni vous allez générer un signal d'horloge sur un port GPIO (le port gpio2_6 situé en P8_45). Ce programme utilise l'accès aux GPIO via le répertoire /sys/class/gpio prévu pour un accès manuel aux GPIO. Ce mécanisme est considéré comme « deprecated » à partir du noyau 4.8 au profit d'une librairie accessible par un chardriver sous /dev (libgpiod). La version de Linux utilisé étant plus ancienne nous garderons dans l'essentiel de la séance un accès via /sys/class/gpio assez lent. En fin de séance nous verrons une méthode plus rapide via un driver puis via une librairie particulière basée sur l'accès direct mémoire.

```
#include <fcntl.h>
#include <stdio.h>
#include "port_beagle.h"

void sys_write (char *sys_file,*val)
{
  int  file_desc ;

  file_desc = open (sys_file, O_WRONLY);
  write (file_desc,val,strlen(val)+1) ;
  close (file_desc) ;
}

main()
{
  /* port reservation and direction */
  sys_write ("/sys/class/gpio/export",GPIO2_6) ;
  sys_write ("/sys/class/gpio/gpio70/direction","out") ;

  /* infinite clock loop */

  while (1) {
      sys_write("/sys/class/gpio/gpio70/value","1") ;
//    usleep(1) ;
      sys_write("/sys/class/gpio/gpio70/value","0") ;
//    usleep (1) ;
      }

  return (0) ;
}
```

Le fichier port_beagle.h vous est fourni, il contient des constantes permettant de connaître les adresses des ports.

Via le Makefile, compiler ce programme et recopier l'exécutable dans le répertoire /share
Lancer ce programme et à l'aide d'un oscilloscope (ou d'un ersatz) visualiser cette horloge.
Donner la fréquence maximum obtenue (nous verrons à la fin du TD comment générer une horloge de manière plus efficace).

Le niveau haut et le niveau bas sont-ils approximativement de même durée ?
Evaluer la durée des instructions d'écriture sur un port en mesurant une demi-période.
L'horloge est-elle générée de manière continue, ou y a-t-il des moments d'interruptions ?

II.1.2 Génération de 2 signaux d'horloge concurrents

Dupliquer le fichier sys_gpio1.c sous le nom sys_gpio2.c. Modifier ce deuxième fichier pour l'adapter au port P8_43 (GPIO2_8) en utilisant les constantes du fichier inclus fourni. Modifier le Makefile pour générer les 2 programmes de la même manière.

Sous la beaglebone, lancer les deux en tâche de fond avec &.
Visualiser sur l'oscilloscope les 2 signaux en sortie des ports.

Que se passe-t-il ?
La fréquence des horloges a-t-elle changée ?
Donner la durée standard d'une tâche sous Linux avant commutation (TIC).
Peut-on voir le temps passé par le CPU pour faire le basculement de tâche ?

II.1.3 Génération des 2 signaux avec passage par le « scheduler ».

Modifier les 2 programmes précédents, en ajoutant dans la boucle deux temps d'attente de 1 micro-seconde par la fonction usleep(1) placée après la fonction close (ces commandes sont commentées dans la version initiale).
En théorie cette fonction ne devrait pas changer le comportement du système de génération car sa durée est faible relativement au temps d'exécution des fonctions dans la boucle mais ...
Qu'est ce qui a été modifié et pourquoi ?

Pour rappel (?), les fonctions de type sleep passe le processus dans un état S (interruptible Sleep) en attente de la fin du temps du timer. Le processus quitte donc la file d'attente des processus de type R (running) et le suivant prend la place.

La génération des signaux d'horloge est-elle complètement correcte ou bien y a-t-il encore des artefacts ?

Que se passe-t-il si on ne met des temps d'attente que dans un seul des 2 programmes ? Essayer.

II.1.4 Modification de la priorité des 2 processus générateur d'horloge.

En reprenant la première version des générateurs (sans usleep), lancer les 2 processus concurrents avec des priorités différentes via la commande nice. Pour ne pas avoir trop d'écart de temps cpu, prévoyez un écart de priorité faible.
(vous devez relancer un make dans le répertoire idoine pour que la recopie se fasse, à condition de changer la date des fichiers sources).
Comment sont générés les signaux d'horloge ?

Faire la même manipulation sur la deuxième version des programmes.

II.2 Modification de la classe d'ordonnancement des 2 processus générateur d'horloge

II.2.1 ordonnancement de type FIFO.

Copier complètement le contenu du répertoire précédent Q1_sys_gpio dans le répertoire de même niveau nommé Q2_sys_gpio_RT (cp -a Q1_sys_gpio/* Q2_sys_gpio_RT).
Modifier la première version des générateurs (sys_gpio) afin de lancer les 2 processus concurrents en leur donnant une priorité temps réel de type FIFO.

Les lignes suivantes en début de programme vont permettre de passer en temps réel type FIFO avec une priorité de 5.

```
# include <sched.h>

struct sched_param myparam ;

sched_getparam(0,&myparam);
myparam.sched_priority = 5 ;
sched_setscheduler(0,SCHED_FIFO,&myparam);
```

Le premier paramètre à 0 dans les fonctions sched_getparam et sched_setscheduler signifie que l'on modifie le processus courant (notre programme). Pour ne modifier que le paramètre de priorité (ici à 5) dans la structure sched_param on la lit puis modifie juste le paramètre intéressant. En fait elle ne contient que cette

valeur dans la version de Linux que l'on utilise. Dans un processus temps réel, on ne peut pas modifier la priorité par nice mais seulement via cette fonction.

Lancer les 2 processus avec une priorité identique.
Que se passe-t-il au niveau de l'interface utilisateur ?

Lancer les 2 processus avec une priorité différente (premier > second) puis l'inverse (second>premier)

Comment sont générés les signaux d'horloge ?

Faire la même manipulation sur la deuxième version des programmes (avec les usleep).

II.2.2 ordonnancement de type Round Robin.

Modifier la première version des fichiers afin de lancer les 2 processus concurrents en leur donnant une priorité temps réel de type Round Robin. (il suffit de modifier le paramètre SCHED_FIFO en SCHED_RR.

Lancer les 2 processus avec une priorité identique.
Lancer les 2 processus avec une priorité différente (premier > second) puis l'inverse (second>premier)

Comment sont générés les signaux d'horloge ?

Faire la même manipulation sur la deuxième version des programmes.

Certaines des solutions proposées pourraient paraître satisfaisantes, hormis le fait que le noyau reprend la main de temps en temps même sur les processus de type temps réel. Vérifier ce fait sur les chronogrammes.

II.2.3 Modification du temps réservé par le noyau pour les tâches non temps réel.

Afin de donner plus de temps aux processus temps réel, nous allons modifier 2 caractéristiques de l'ordonnanceur : la valeur du temps maximum avant que l'ordonnanceur reprenne la main sur une tâche temps réel définie dans /proc/sys/kernel/sched_rt_period_us = 1000000 par défaut (1 sec) et la valeur du temps réservé aux tâches temps réel : /proc/sys/kernel/sched_runtime_us = 950000.
Cela signifie que par défaut 50msec sont réservées toutes les secondes pour un processus temps partagé. On peut supprimer cette contrainte en écrivant -1 dans sched_rt_runtime_us. Le seul petit problème est que l'on perd toute interaction avec l'utilisateur, par exemple pour arrêter une tâche !

La fréquence de l'horloge que nous avons générée dans les exemples précédents est assez faible, nous allons voir plusieurs méthodes pour en augmenter la valeur et les problèmes que cela peut engendrer.

II.3 Accès aux GPIO par un LKM.

Ecrire un LKM générant une horloge sur le GPIO2_6 à partir du fichier *gpio_task_template.c* vu en TP driver.
Compiler le code ainsi écrit en modifiant un Makefile existant. Tester l'horloge générée. Quelle est la fréquence de cette horloge ? La comparer avec la fréquence de la question II.1.1
Modifier ce programme pour générer les 2 signaux sur GPIO2_6 et GPIO2_8 comme précédemment dans 2 tâches concurrentes. Conclusion