Mini-projet système : développement d'un noyau de système d'exploitation

Responsable : Christophe Rippert
Christophe.Rippert@Grenoble-INP.fr

Gestion des processus dans un système embarqué (ISSC / SLE)

Généralisation à N processus

Vous devez ensuite généraliser votre code pour N processus : pour les tests, on choisira N = 4.
On rajoute donc 2 nouveaux processus dans le système, `proc2` et `proc3` dont le code est similaire pour l'instant à celui de `proc1`.
La généralisation ne nécessite pas beaucoup de changements :

- la fonction ordonnance doit être adaptée pour implanter la politique du tourniquet, qui active les processus dans l'ordre de leur `pid` : 0, 1, 2, 3, 0, 1, 2, 3, ... ;
- on vous recommande de factoriser le code de création et d'initialisation des processus `proc1`, `proc2` et `proc3` avec une fonction `int32_t cree_processus(void (*code)(void), char *nom)` qui prend en paramètre le code de la fonction à exécuter (ainsi que le nom du processus) et renvoie le `pid` du processus créé, ou -1 en cas d'erreur (si on a essayé de créer plus de processus que le nombre maximum).

Ordonnancement préemptif

Dans la majorité des systèmes actuels, ce ne sont pas les processus qui se passent la main : les basculements d'un processus à l'autre sont provoqués par des événements venant de l'horloge système, et s'enchainent suffisamment rapidement pour donner à l'utilisateur l'impression que les processus s'exécutent en parallèle.
On va donc connecter l'ordonnanceur à l'interruption horloge, ce qui en pratique ne nécessite que très peu de modifications par rapport à ce que vous avez fait avant.
Les processus de tests seront maintenant les suivants :

```
void idle(void)
{
    for (;;) {
        printf("[%s] pid = %i\n", mon_nom(), mon_pid());
        sti();
        hlt();
        cli();
    }
}

void proc1(void) {
    for (;;) {
        printf("[%s] pid = %i\n", mon_nom(), mon_pid());
        sti();
        hlt();
        cli();
    }
}

... (idem proc2 et proc3)
```

Vous devez bien sûr remettre dans la fonction `kernel_start` toutes les initialisations nécessaires à l'interruption horloge que vous aviez géré pendant la séance 2 (note : ne mettez pas d'appel à `sti()` dans `kernel_start` : c'est la fonction `idle` qui activera les interruptions la première fois).
Vous devez penser à ajouter un appel à la fonction `ordonnance` à la fin de la fonction appelée par le traitant de l'interruption horloge, pour provoquer le changement de processus.
L'affichage obtenu doit être le même que pour la séance précédente : on doit voir les 2 processus prendre la main l'un après l'autre. Vous devez utiliser GDB pour voir s'afficher les traces de façon lisibles. On verra comment implanter un véritable mécanisme d'attente dans la partie suivante.

Endormissement des processus

On va maintenant implanter un mécanisme permettant d'endormir un processus pendant un certain nombre de secondes, de façon similaire à la fonction `sleep` de la bibliothèque C standard. Il s'agit d'une simple fonction `void dors(uint32_t nbr_secs)` qui prend en paramètre le nombre de secondes pendant lequel le processus doit dormir.
Une façon simple de mettre en oeuvre ce mécanisme consiste à rajouter un état `ENDORMI` et à garantir que la fonction d'ordonnancement n'activera pas les processus dans cet état tant que leur heure de réveil n'est pas dépassée.
Pour gérer le réveil, il faut stocker dans la structure décrivant chaque processus l'heure à laquelle il doit se réveiller. On mesurera le temps en nombre de secondes écoulées depuis le démarrage du système (une information déjà disponible depuis la séance 2 et qu'il suffit de rendre accessible à l'ordonnanceur). C'est la fonction d'ordonnancement qui devra réveiller tous les processus dont l'heure de réveil est dépassée.
Vous pourrez tester votre implantation avec par exemple les 4 processus ci-dessous, en supposant que `nbr_secondes` soit la fonction qui renvoie le nombre de secondes écoulées depuis le démarrage du système :

```
void idle()
{
    for (;;) {
        sti();
        hlt();
        cli();
    }
}

void proc1(void)
{
    for (;;) {
        printf("[temps = %u] processus %s pid = %i\n", nbr_secondes(),
          mon_nom(), mon_pid());
        dors(2);
    }
}

void proc2(void)
{
    for (;;) {
        printf("[temps = %u] processus %s pid = %i\n", nbr_secondes(),
          mon_nom(), mon_pid());
        dors(3);
    }
}

void proc3(void)
{
    for (;;) {
        printf("[temps = %u] processus %s pid = %i\n", nbr_secondes(),
```

```
          mon_nom(), mon_pid());
        dors(5);
    }
}
```

Le processus `idle` n'a lui bien sûr pas le droit de s'endormir, sinon on risquerait de se retrouver dans un système sans aucun processus activable !

Terminaison des processus

On va maintenant permettre aux processus de se terminer : vous pourrez alors enlever la boucle infinie autour du code des processus pour vérifier qu'ils se terminent bien. Le processus `idle` n'a bien sûr pas le droit de se terminer !

Terminaison explicite d'un processus

Pour commencer, il faut implanter une fonction `void fin_processus(void)` qui va réaliser le travail de terminaison d'un processus. Dans une première implantation, un processus voulant se terminer devra explicitement appeler cette fonction, comme par exemple :

```
void proc1(void)
{
    for (int32_t i = 0; i < 2; i++) {
        printf("[temps = %u] processus %s pid = %i\n", nbr_secondes(),
          mon_nom(), mon_pid());
        dors(2);
    }
    fin_processus();
}
```

La fonction de terminaison doit marquer le processus actif (puisque c'est forcément lui qui l'appelle) comme étant dans l'état `MORT` et ensuite passer la main à la fonction d'ordonnancement(). Il faut bien sûr aussi garantir que la fonction d'ordonnancement n'activera jamais un processus mort !
A ce stade, il peut être utile d'implanter une fonction `void affiche_etats(void)` qui affiche (par exemple en haut à gauche de l'écran) l'état de chaque processus du système (par un simple parcours de la table des processus).

Terminaison automatique d'un processus

Evidemment, dans un vrai système, on n'a pas besoin d'insérer un appel à `fin_processus` à la fin du code de chaque processus : la terminaison se fait automatiquement.
Une façon simple d'implanter cette terminaison automatique consiste à initialiser le sommet de pile de chaque processus avec l'adresse d'une fonction gérant la terminaison de celui-ci. On rappelle qu'on doit toujours copier l'adresse de début de la fonction dans la pile avant le premier changement de contexte (il suffit de décaler cette adresse pour qu'elle soit sous le sommet de pile).

Création dynamique de processus

Maintenant que les processus peuvent se terminer, il est intéressant de pouvoir en créer dynamiquement (sinon, on va rapidement se retrouver avec un système qui fait `idle` tout le temps).
En pratique, vous ne devez pas avoir grand chose à changer pour permettre à un processus d'appeler lui-même la fonction de création d'un processus (sous réserve qu'elle soit implantée proprement). Cette fonction devra ré-utiliser une case de la table des processus actuellement allouée à un processus `MORT`, en vérifiant bien toujours qu'on ne dépasse pas le nombre maximum de processus dans le système.
A vous de créer des tests significatifs pour vérifier que la terminaison et la re-création des processus se passent bien.

Pour aller plus loin

Si vous avez fini en avance, vous pouvez étendre votre système comme il vous plaira. A titre d'exercice, vous pouvez travailler sur la version ISI du projet (mais attention : le jour de l'examen de TP, les questions porteront sur la version ISSC et SLE).