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 dynamique (ISI)

Passage à un modèle dynamique

Dans l'exemple précédent avec deux processus, on a travaillé avec un modèle très statique (structures contenant le contexte d'exécution allouées statiquement). Dans un système généraliste plus réaliste, on alloue les structures de contexte dynamiquement pour éviter de gaspiller de l'espace mémoire.
Pour pouvoir allouer dynamiquement de la mémoire, vous allez utiliser l'allocateur fourni avec les sources de départ du mini-projet et qui se trouve dans le fichier `malloc.c.h` : il vous suffit de l'inclure (avec `#include <malloc.c.h>`) dans le fichier où vous en avez besoin pour pouvoir utiliser les fonctions classiques `malloc`, `calloc`, `realloc` et `free` que vous connaissez. Attention : vous ne devez l'inclure que dans un seul fichier .c, celui qui contiendra le module de gestion des processus, et ne faire aucune allocation dynamique ailleurs dans votre noyau.
Commencez donc par modifier la table des processus pour qu'elle contienne non-plus directement les structures des processus, mais plutôt des pointeurs vers ces structures. Vous devez bien sûr allouer dynamiquement l'espace mémoire nécessaire aux structures lors de la création des processus, ainsi que les piles d'exécution. On ne gère pas pour l'instant la terminaison des processus, donc pas besoin de libérer l'espace mémoire.

Généralisation à N processus

Vous devez ensuite généraliser votre code pour N processus : pour les tests, on choisira N = 8.
On rajoute donc 6 nouveaux processus dans le système, `proc2`, `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, 4, 5, 6, 7, 0, 1, 2, 3, ... ;
- on vous recommande de factoriser le code de création et d'initialisation des processus `proc1`, `proc2`, ... 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).

Utilisation de listes de processus

Dans le système primitif actuel, la fonction d'ordonnancement parcourt la table des processus pour trouver le prochain processus à activer. Dans un système réaliste avec des milliers de processus activables, endormis, bloqués, ça ne serait pas très efficace.
On utilise plus couramment des listes chainées de processus, ordonnées selon la politique d'ordonnancement que l'on veut implanter. Ici, on veut gérer les processus de façon FIFO : il suffit donc de gérer la liste des processus activables comme une file, en activant (par exemple) systématiquement le processus en tête de liste et en insérant l'ancien processus élu en queue de la liste des activables.
Vous devez donc faire les modifications suivantes à votre système :

- rajouter un champ `suiv` dans votre structure de processus, défini comme un pointeur vers une structure de processus : ce champ va permettre de chainer les processus les uns aux autres ;

— définir une liste des activables : il est sûrement plus efficace de conserver deux pointeurs, un sur la tête et un autre sur la queue de la liste, pour garantir l'insertion en queue à coût constant.

On recommande aussi d'implanter directement une fonction d'extraction de la tête des activables, et une fonction d'insertion en queue, pour éviter d'avoir à dupliquer ce code, car il sera utilisé à plusieurs endroits du système par la suite. Ces fonctions doivent aussi changer l'état des processus manipulés.
Il est important de noter que dans tout le TP, on n'aura besoin que d'un seul champ `suiv` par processus, malgré le fait qu'on gérera à la fin 3 types de listes chainées : en effet, ces listes seront forcement disjointes, et un processus ne pourra se trouver que dans une seule liste à la fois.

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, 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 8 processus prendre la main l'un après l'autre. Vous devez utiliser GDB pour pouvoir voir les traces s'afficher de façon lisible.
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 gérer une liste des processus endormis : lorsqu'un processus appelle la procédure d'endormissement, on l'enlève de la file des activables et on l'ajoute dans cette liste des endormis, et vice-versa quand il se réveille. Il faut aussi rajouter un état pour les processus endormis (par exemple `ENDORMI`).

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.

On peut gérer librement la liste des endormis, mais il est plus efficace de l'ordonner selon l'heure de réveil des processus. Ainsi, les processus devant se réveiller en premier seront en tête de liste et il sera plus rapide et plus facile de les retirer de cette liste pour les insérer en queue de la file des activables. Là encore, on vous recommande d'implanter des fonctions de manipulation de la liste des endormis pour factoriser le code.

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 détruire le processus actif (puisque c'est forcément lui qui l'appelle). Elle doit donc libérer la structure du processus (en appelant la fonction `free`) et ensuite passer la main au prochain processus activable. Il ne faut pas oublier aussi de vider la case correspondant dans la table des processus.
Cependant, il y a un problème technique : si on libère la structure du processus avant d'appeler la fonction `ctx_sw`, celle-ci va sauvegarder les registres dans une zone mémoire non-allouée, ce qui est interdit (on pourrait aussi ne pas sauvegarder le contexte vu que le processus va mourir, mais cela impliquerait d'écrire une fonction de changement de contexte spécifique).
Une façon simple de contourner le problème consiste à gérer une liste des processus morts : lorsqu'un processus se termine, la fonction de terminaison l'insère dans la liste puis active le prochain processus, et c'est lors de l'appel suivant à la fonction d'ordonnancement que celle-ci détruira les contextes des processus morts. Un processus en train de mourir sera noté par un état particulier (par exemple, `MOURANT`).
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 : ce n'est pas très efficace, mais il s'agit d'une fonction de mise au point qui serait débranchée dans un système en production).

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).
Il est intéressant d'essayer d'utiliser tous les `pid` possibles dans le système, en évitant de ré-allouer toujours les mêmes, afin de construire des jeux de tests plus faciles à lire.

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.
Une extension simple consiste à gérer un système de priorité pour les processus : au lieu d'implanter un ordonnancement par tourniquet simple, vous implanterez une politique qui fait passer en premier les processus de forte priorité (évidemment, `idle` doit avoir la plus faible priorité pour n'être activé que lorsqu'il ne reste personne d'autre à traiter).
Vous pouvez implanter ce type d'ordonnancement par exemple en triant la liste des activables par ordre de priorité décroissante, ou (plus efficace) en gérant des files d'activables différentes en fonction de la priorité des processus (dans un système, l'intervalle des priorités est toujours fini et en général assez petit).