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

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

Gestion des processus

Introduction

Le but des 4 séances restantes est de programmer un noyau de processus simpliste. On se limitera à un mode de fonctionnement volontairement très simplifié par rapport à un noyau de système réaliste. Les aspects fondamentaux des systèmes actuels, tels que les primitives de synchronisation, la communication inter-processus, ... seront vus en détail dans le cours d'Approfondissement proposé en option au 2e semestre.

Notions générales

Spécification des processus

On a vu en SEPC que les processus sont des unités d'exécution permettant d'implanter la notion de pseudo-parallélisme sur un système mono-processeur (remarque : en pratique, la grande majorité des processeurs actuels sont équipés de plusieurs coeurs d'exécution, mais on ne les exploitera pas dans ce TP). Ce pseudo-parallélisme est réalisé très simplement en accordant à chaque processus un temps d'exécution sur le processeur, puis une fois ce temps écoulé, en arrêtant le processus actif pour passer la main à un autre processus prêt à s'exécuter : l'entrelacement suffisamment rapide de l'exécution des différents processus donne à l'utilisateur l'illusion qu'ils s'exécutent réellement en parallèle.
Dans le cadre de ce TP, chaque processus sera défini par :

- son numéro, qui servira d'identifiant unique dans le système, et que l'on appelle typiquement `pid` (pour *Process IDentifier*) ;
- son nom, une chaine de caractères qui servira à produire des traces lisibles ;
- son état : un processus peut être élu (c'est à dire que c'est lui que le processeur est en train d'exécuter), activable (c'est à dire qu'il est prêt à être élu et qu'il attend que le processeur se libère), endormi pour une durée donnée, ... ;
- son contexte d'exécution sur le processeur : il s'agit en pratique d'une zone mémoire servant à sauvegarder le contenu des registres important du processeur lorsqu'on arrête le processus pour passer la main à un autre, et à restaurer ce contenu ensuite lorsque le processus reprend la main ;
- sa pile d'exécution : qui est l'espace mémoire dans lequel sont stockés notamment les variables locales, les paramètres passés aux fonctions, ...

Les processus que l'on va implanter partageront leur espace mémoire de données : il s'agit donc de processus légers (ou *threads*), par opposition à des processus lourds qui seraient totalement isolés dans des espaces d'adressage différents.

Rôle de l'ordonnanceur

Dans un noyau de système, l'ordonnanceur de processus est le composant qui gère l'entrelacement de l'exécution des différents processus. Il contient un certain nombre de variables d'état importantes, et notamment la table des processus : dans le noyau que l'on va implanter, il s'agira simplement d'un tableau de taille définie statiquement par une constante et contenant les structures de données représentant les processus activables. Il est aussi nécessaire de gérer un pointeur sur le processus élu (celui en cours d'exécution) pour accéder à sa structure interne si besoin est.
La partie principale de l'ordonnanceur est évidemment la fonction d'ordonnancement, c'est à dire la primitive qui implante la politique d'ordonnancement. Cette fonction est appelée à chaque fois qu'un

processus a épuisé le temps d'exécution qui lui était alloué, et elle se base sur la politique d'ordonnancement pour choisir quel processus va maintenant s'exécuter. Dans ce TP, on commencera par implanter la politique d'ordonnancement la plus simple, celle du tourniquet (*Round-Robin*) : si par exemple le système gère 4 processus de `pid` 0, 1, 2 et 3, alors l'ordonnanceur les fera s'exécuter dans l'ordre suivant : 0, 1, 2, 3, 0, 1, 2, 3, ...
Une fois que la fonction d'ordonnancement a choisi quel processus sera activé, et qu'elle a mis a jour ses variables internes en conséquence, elle doit provoquer l'arrêt de l'exécution du processus actif et démarrer le processus élu. Cette opération critique est réalisée par une primitive appelée traditionnellement changement de contexte (*context switch*).

Le changement de contexte d'exécution

Lorsqu'on veut arrêter l'exécution d'un processus pour donner la main à un autre, il faut sauvegarder le contexte d'exécution du processus actif, de façon à pouvoir reprendre son exécution exactement là où il en était lorsqu'il prendra à nouveau la main. En pratique, cela consiste à sauvegarder les valeurs des registres importants du processeur dans une zone mémoire réservée pour cela (dans la structure du processus telle que présentée plus haut).
Une fois le contexte du processus actif sauvegardé, on doit restaurer le contexte du processus élu (celui qu'on veut activer) : en effet, ce processus avait lui-même été arrêté auparavant dans un certain état et on veut reprendre son exécution dans cet état précis (le problème de la première exécution du processus sera expliqué plus bas).
Parmi tous les registres à restaurer, le pointeur de pile (`%esp` sur x86) est certainement le plus important : en effet, c'est ce registre qui contient l'adresse de la pile d'exécution du processus élu. Cette pile contient elle-même non-seulement les valeurs des variables locales, paramètres, etc. du processus, mais aussi et surtout, l'adresse de retour à la fonction qui était en cours d'exécution lorsque le processus a été arrêté. Une fois le pointeur de pile restauré à la valeur sauvegardé, il suffit donc de terminer la fonction de changement de contexte par l'instruction `ret` habituelle pour que le fil d'exécution redémarre dans le processus élu (remarque : ce point peut vous paraitre difficile à comprendre en lisant cette explication formelle, mais il s'éclaircira lorsque vous le mettrez en pratique sur un exemple concret).
Comme noté plus haut, la première exécution d'un processus pose un problème particulier. En effet, si le processus à activer n'a jamais été exécuté, son contexte n'a pas été sauvegardé (puisqu'il n'en a pas encore). Mais il faut tout de même s'assurer qu'après l'exécution de l'instruction `ret` du *context switch*, l'exécution démarrera bien au début de la fonction principale du processus. Une façon simple de garantir cela est de préparer la pile d'exécution du processus en stockant en sommet de pile l'adresse de la fonction principale du processus, et de placer l'adresse du sommet de pile dans la zone mémoire correspondant à la sauvegarde du registre `%esp`.
Note : on rappelle que le **sommet d'une pile** n'est pas le début de la zone mémoire allouée, mais **la fin de cette zone** (lorsqu'on empile des valeurs, on descend vers les adresses décroissantes).

Exemple avec deux processus

Pour commencer, on va travailler avec 2 processus qui vont se passer la main explicitement (dans les parties suivantes, le changement de processus sera déclenché par l'interruption de l'horloge, comme dans les systèmes classiques). Pour garantir qu'il n'y a pas d'interférence (et pour permettre le blocage du système), vous devez désactiver les interruptions. Le plus simple pour cela est d'enlever l'appel à la fonction `sti()` que vous aviez ajouté à la fin de votre fonction `kernel_start`.
Pour commencer, on va mettre au point le mécanisme de passage d'un processus à l'autre. Dans le système réalisé, il y aura donc seulement 2 processus :

- le processus `idle` (qui a par convention le `pid` 0) est le processus par défaut : en effet, dans un système, il doit toujours y avoir au moins un processus actif, car le système ne doit jamais s'arrêter ;
- le processus `proc1` (de `pid` 1) qui représentera un processus de calcul quelconque.

Passage d'un processus à l'autre

Dans une première étape, on va implanter simplement le passage du processus `idle` au processus `proc1`
Plus précisément, le code de ces processus sera :

```
void idle(void)
{
    printf("[idle] je tente de passer la main a proc1...\n");
    ctx_sw(..., ...);
}

void proc1(void)
{
    printf("[proc1] idle m'a donne la main\n");
    printf("[proc1] j'arrete le systeme\n");
    hlt();
}
```

Vous devrez donc voir apparaitre à l'écran la trace de `idle` suivi des deux traces de `proc1`, puis le système se bloquera grâce au `hlt()`.
Pour implanter ce petit test, vous aurez besoin de définir la structure des processus décrite plus haut. Précisément, le type structure de processus que vous devez définir comprendra :

- le `pid` du processus, sous la forme d'un entier (signé, car traditionnellement la fonction de création d'un processus renvoie -1 en cas d'erreur) ;
- le nom du processus : un chaine de caractères avec une taille maximum fixée ;
- l'état du processus : dans ce premier exemple, il n'y a que 2 états possibles : élu ou activable, vous pouvez toutefois déjà définir proprement les états comme une énumération de constantes qui sera facile à étendre par la suite ;
- la zone de sauvegarde des registres du processeur : il s'agira tout simplement d'un tableau d'entiers, puisqu'on manipule des registres 32 bits, et vous avez besoin de 5 cases car il n'y a que 5 registres importants à sauvegarder sur x86 (voir plus bas l'explication du changement de contexte) ;
- la pile d'exécution du processus, qui sera définie comme un tableau d'entiers d'une taille fixée par une constante, par exemple 512.

Ensuite, il faut définir la table des processus, c'est à dire tout simplement un tableau de structure de processus, dont la taille est là encore une constante prédéfinie dans le système. Dans cette première étape, cette constante sera fixée à 2.
Lors du démarrage du système, il faut initialiser les structures de processus de `idle` et `proc1` avec des valeurs pertinentes, c'est à dire :

- pour les deux processus, son `pid`, son nom et son état : par défaut, c'est le processus `idle` qui est élu le premier ;
- pour `proc1` la case de la zone de sauvegarde des registres correspondant à `%esp` doit pointer sur le sommet de pile, et la case en sommet de pile doit contenir l'adresse de la fonction `proc1` : c'est nécessaire comme expliqué plus haut pour gérer la première exécution de `proc1`.

Il n'est pas nécessaire d'initialiser la pile d'exécution du processus `idle` : en fait, ce processus n'utilisera pas la pile allouée dans sa structure de processus mais directement la pile du noyau (celle qui est utilisée par la fonction `kernel_start` notamment). De même, il n'est pas nécessaire d'initialiser la zone de sauvegarde de `%esp` pour `idle` puisque ce processus sera exécuté directement par la fonction `kernel_start`, qui devrait donc ressembler au squelette ci-dessous :

```
void kernel_start(void)
{
    // initialisation des structures de processus
    ...
    // demarrage du processus par defaut
    idle();
}
```

On vous fourni l'implantation du changement de contexte dans le fichier `ctx_sw.S`. Cette fonction est cruciale dans le mécanisme d'ordonnancement car c'est elle qui va effectuer le passage d'un processus à l'autre. On détaille son code ci-dessous :

```
    .globl ctx_sw
# Structure de la pile en entree :
```

```
#   %esp + 4 : adresse de l'ancien contexte
#   %esp + 8 : adresse du nouveau contexte
ctx_sw:
    # sauvegarde du contexte de l'ancien processus
    movl 4(%esp), %eax
    movl %ebx, (%eax)
    movl %esp, 4(%eax)
    movl %ebp, 8(%eax)
    movl %esi, 12(%eax)
    movl %edi, 16(%eax)
    # restauration du contexte du nouveau processus
    movl 8(%esp), %eax
    movl (%eax), %ebx
    movl 4(%eax), %esp
    movl 8(%eax), %ebp
    movl 12(%eax), %esi
    movl 16(%eax), %edi
    # on passe la main au nouveau processus
    ret
```

Le changement de contexte est une fonction particulière : on note déjà qu'il n'y a pas de mise en place d'un contexte d'exécution (ce qui serait contre-productif puisque la fonction doit justement manipuler les contextes de l'appelant et de celui à qui elle va passer la main).
Elle prend 2 paramètres de types pointeurs sur des entiers : il s'agit en fait des adresses des zones de sauvegarde des registres des contextes de l'ancien processus et du nouveau.
Le principe de la fonction est simple :

1. on commence par sauvegarder les registres dans la zone appropriée du contexte de l'ancien processus (celui qu'on veut arrêter) ;
2. on restaure ensuite les registres depuis la zone de sauvegarde du nouveau processus (celui qu'on veut re-démarrer) ;
3. on passe enfin la main au nouveau processus avec l'instruction `ret`.

On peut maintenant comprendre plus précisément le fonctionnement de l'initialisation de la pile de `proc1` que l'on a fait plus haut : lors de la restauration des registres de `proc1`, la valeur de `%esp` restaurée sera un pointeur sur le sommet de la pile de `proc1`, qui contient l'adresse de la fonction `proc1`. Lorsqu'on arrivera sur le `ret` à la fin du changement de contexte, c'est donc cette adresse qui sera dépilée et à laquelle le flot d'exécution sautera.
On remarque qu'on ne sauvegarde pas tous les registres du processus : en effet, il est inutile de sauvegarder les registres `%eax`, `%ecx` et `%edx` dans lequel le compilateur C n'a pas le droit de laisser des valeurs importantes lorsqu'il fait un appel de fonction. Plus tard, lorsqu'on branchement l'ordonnancement sur l'interruption horloge, on devrait formellement sauvegarder ces registres (puisqu'une interruption peut intervenir n'importe-quand), mais on le fait en fait déjà dans le traitant de l'interruption : inutile donc de le refaire dans le changement de contexte. On rappelle enfin que le registre des indicateurs `%eflags` est lui automatiquement sauvegardé par le processeur avant l'appel au traitant d'interruption.

Aller-retour entre les processus

Une fois que le test précédent fonctionne correctement, on va l'étendre pour faire revenir l'exécution à `idle` après être passé par `proc1` (et pour se convaincre que cela marche vraiment, on va faire l'aller-retour 3 fois...). Le code des processus sera donc maintenant :

```
void idle(void)
{
    for (int i = 0; i < 3; i++) {
        printf("[idle] je tente de passer la main a proc1...\n");
        ctx_sw(..., ...);
        printf("[idle] proc1 m'a redonne la main\n");
```

```
    }
    printf("[idle] je bloque le systeme\n");
    hlt();
}

void proc1(void)
{
    for (;;) {
        printf("[proc1] idle m'a donne la main\n");
        printf("[proc1] je tente de lui la redonner...\n");
        ctx_sw(..., ...);
    }
}
```

Ordonnancement selon l'algorithme du tourniquet

On va maintenant implanter un mécanisme d'ordonnancement simple connu sous le nom d'ordonnancement collaboratif : ce sont les processus eux-mêmes qui vont explicitement se passer la main en appelant la fonction d'ordonnancement. On continue à travailler avec les processus `idle` et `proc1` dont on donne le nouveau code ci-dessous :

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

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

On devrait voir s'afficher à l'écran alternativement la trace de chaque processus... mais tellement rapidement que c'est parfaitement illisible. Pour y voir quelque-chose, utiliser GDB pour faire du pas à pas et vérifier que les deux processus alternent bien correctement.
Vous devez compléter votre gestion de processus pour implanter ce test :

- la fonction `void ordonnance(void)` a pour rôle d'implanter la politique d'ordonnancement en choisissant le prochain processus à activer (comme il n'y en a que 2 pour l'instant, ça ne devrait pas poser de difficulté) et de provoquer le changement de processus en appelant la fonction `ctx_sw` avec les bons paramètres ;
- l'ordonnanceur a besoin de savoir quel est le processus en cours d'exécution : le plus simple pour cela est de conserver un pointeur vers la structure de processus sous la forme d'une variable globale `actif` (de type pointeur sur une structure de processus par exemple ou de façon équivalente, le `pid` du processus actif sous forme d'un entier signé) ;
- n'oubliez pas de changer l'état des processus dans la fonction `ordonnance` : le processus élu doit prendre l'état `ELU` et l'autre devenir `ACTIVABLE` ;
- les fonctions `int32_t mon_pid(void)` et `char *mon_nom(void)` renvoient simplement le `pid` et le nom du processus en cours d'exécution, elles ne posent pas de difficulté d'implantation.

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 les fichiers `tinyalloc.h` et `tinyalloc.c` : il vous suffit de l'inclure (avec `#include <tinyalloc.h>`) dans le fichier où vous en avez besoin pour pouvoir utiliser la fonction classique `malloc` 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 (les zones de sauvegarde des registres et les piles d'exécution resteront par contre des tableaux statiques : le gaspillage mémoire engengré par l'allocation d'une pile pour le processus `idle` qui utilise en fait la pile système n'est pas très important). 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 désactiver le processus actif (puisque c'est forcément lui qui l'appelle) et ensuite passer la main au prochain processus activable.
Mais elle ne doit pas libérer la structure du processus (on ne fait aucun appel à la fonction `free` dans tout ce projet) car on va permettre le recyclage des processus. Il faut pour cela ajouter une liste des processus zombies, c'est à dire des processus dont l'exécution est terminée, et bien sûr ajouter un nouvel état possible `ZOMBIE`. On comprendra l'intérêt de ce recyclage lorsqu'on ajoutera la création dynamique d'un processus.
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), c'est à dire de permettre à un processus d'appeler lui-même la fonction de création d'un processus.
Il va falloir modifier la fonction de création pour ajouter le recyclage des processus zombies : si la liste des zombies est vide, on ne change rien, et on ré-alloue une nouvelle structure de processus ; mais s'il y a au moins un processus zombie, on récupère sa structure sans en ré-allouer une nouvelle.
Ce mécanisme permet d'éviter des allocations mémoires (qui selon les algorithmes utilisés peuvent s'exécuter en un temps non-deterministe) à chaque création de processus. Bien sûr, il a comme inconvénient de ne jamais libérer la mémoire, mais un système moderne fournira alors un mécanisme de type ramasse-miettes qui pourra s'exécuter à des moments bien définis pour nettoyer la liste des zombies en désallouant toutes les structures allouées.

Pour aller plus loin

Si vous avez fini en avance, vous pouvez étendre votre système comme il vous plaira.
Une autre 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).