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

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

Gestion du temps

Introduction

Dans cette séance, on va implanter la gestion du temps dans notre noyau. A la fin de la séance, on attend que le système affiche en haut à droite de l'écran une horloge indiquant depuis combien de temps le système a démarré, sous la forme `HH:MM:SS`.

Mise en place du traitement des interruptions

Principe général

Pour gérer le temps, on va devoir utiliser un mécanisme fondamental des systèmes : les interruptions. Le principe général de ce mécanisme est le suivant :

1. on est en train d'exécuter un programme quelconque ;
2. un événement prioritaire (e.g. on appuie sur une touche du clavier, un circuit horloge indique qu'un certain laps de temps est écoulé, ...) provoque un signal appelé interruption qui est transmis au processeur ;
3. le processeur arrête l'exécution du programme en cours, et commence à exécuter un programme spécial appelé traitant d'interruption dont le but est de gérer l'événement en question ;
4. une fois l'événement géré, le processeur reprend l'exécution du programme initial à l'endroit où il s'était arrêté.

Ce mécanisme en apparence simple entraîne plusieurs difficultés qu'on va devoir gérer comme détaillé ci-dessous.

Initialisation de la table des vecteurs d'interruption

Chaque source d'interruption porte un numéro unique (sur x86, il y en a 256) afin de permettre au processeur de l'identifier. Lorsque le processeur reçoit l'interruption N, il consulte la case numéro N d'une table appelée table des vecteurs d'interruption pour trouver l'adresse en mémoire du traitant à appeler. Cela impose évidemment que cette table soit elle-même a une adresse connue du processeur : dans notre cas, il s'agira de l'adresse 0x1000.
Chaque entrée de l'IDT occupe 2 mots consécutifs de 4 octets chacun et a le format suivant :

- le premier mot de l'entrée est composé de la constante sur 16 bits `KERNEL_CS` (bits 31 à 16) et des 16 bits de poids faibles de l'adresse du traitant (bits 15 à 0) ;
- le deuxième mot est composé des 16 bits de poids forts de l'adresse du traitant (bits 31 à 16) et de la constante 0x8E00 (bits 15 à 0).

La constante `KERNEL_CS` est définie dans `segment.h` et précise ce qu'on appelle un descripteur de segment. On ne rentrera pas dans les détails du fonctionnement de la mémoire sur x86 dans ce TP.
L'adresse (sur 32 bits) du traitant à activer est donc répartie sur les deux mots composant l'entrée dans la table, avec 16 bits dans chaque mot.
La constante 0x8E00 sert à préciser un certain nombre de choses dépassant le cadre de ce TP, dont notamment le fait que l'exécution du traitant se fait interruptions masquées : un traitant d'interruption ne peut donc pas être lui-même interrompu.
Attention : on rappelle qu'on travaille sur une machine (virtuelle) 32 bits (pas 64 bits comme la plupart des machines modernes), qui se trouve de plus être *little-endian*. Il est donc **très fortement recommandé**

d'écrire les deux mots par bloc de 32 bits (pas 64 bits d'un coup, ni 16 bits par 16 bits) pour être sûr que les différentes parties du contenu de la case seront bien stockées dans le bon ordre en mémoire.

Écriture d'un traitant d'interruption

Un traitant d'interruption est un programme très particulier qui ne s'écrit pas comme un programme classique car on doit prendre en compte de façon précise l'état du processeur aux moments où on entre et sort du traitant.

Une fois qu'il a trouvé l'adresse du traitant à appeler dans la table des vecteurs d'interruption, le processeur sauvegarde en mémoire deux informations importantes avant de passer la main au traitant : le contenu du registre des indicateurs `%eflags` et le compteur ordinal (qui pointe sur la prochaine instruction à exécuter dans le programme interrompu).

Le processeur ne sauvegarde notamment pas les registres généraux : c'est donc à la charge du traitant de sauvegarder ceux susceptibles d'être modifiés (et seulement ceux-ci). La façon la plus simple de le faire est d'utiliser les instructions `push` et `pop` qui copient et lisent respectivement des valeurs dans la pile d'exécution du traitant.

On fourni la partie assembleur du traitant d'interruption (à implanter dans le fichier `traitant.S`), dont on détaille le code ci-dessous :

```
# cette directive sert a rendre l'etiquette publique
    .globl traitant_IT_32
# debut du traitant
traitant_IT_32:
# sauvegarde des registres importants
    pushl %eax
    pushl %edx
    pushl %ecx
# appel a la fonction C realisant le traitant
    call tic_PIT
# restauration des registres importants
    popl %ecx
    popl %edx
    popl %eax
# fin du traitant
    iret
```

La fonction `tic_PIT` est à écrire en C : c'est elle qui va réaliser le travail concret du traitant, le code assembleur ci-dessus n'est qu'un enrobage nécessaire pour sauvegarder ce qui doit l'être.

Lorsqu'on commence à traiter une interruption, on doit le signaler à un composant matériel appelé contrôleur d'interruptions dont on parlera plus bas. Cette étape est nécessaire pour permettre à ce contrôleur de se remettre à écouter d'autres interruptions éventuelles : elle doit donc être réalisée le plus tôt possible dans le traitant (en pratique, tout au début de la partie C du traitant).

Pour cela, on va encore utiliser les opérations de communication avec les ports qu'on a vu à la séance précédente. Pour acquitter une des interruptions qu'on manipule dans ce TP, on doit envoyer la commande sur 8 bits 0x20 sur le port de commande 0x20. Comme on veut le faire en C au tout début de la partie du traitant d'interruption gérant l'affichage de l'horloge, on utilisera le bout de code suivant : `outb(0x20, 0x20)`.

A la fin de l'exécution du traitant, on doit utiliser une instruction particulière pour revenir au programme initial : `iret` (*Interrupt Return*) dont le fonctionnement se rapproche de l'instruction `ret` qu'on utilise classiquement à la fin d'une fonction, mais qui permet en plus de rétablir les indicateurs et le compteur ordinal originaux.

Utilisation d'une l'horloge matérielle comme source d'interruption

Gestion de l'IRQ0

L'interruption que l'on va utiliser dans cette séance est celle générée par un composant appelé horloge programmable dont le fonctionnement sera détaillé plus bas. En résumé, cette horloge va générer périodiquement des signaux pour signifier l'écoulement du temps.
L'horloge est connectée au contrôleur d'interruptions dont on a parlé plus haut. Lorsqu'elle émet un signal, celui-ci est transmis au contrôleur d'interruption via un canal appelé IRQ (*Interrupt ReQuest*). Dans le cas de l'horloge, il s'agit du canal IRQ0. Le contrôleur d'interruption transmet ce signal au processeur sous la forme d'une interruption : dans le cas de l'horloge, le contrôleur est programmé pour émettre l'interruption 32.
Il est possible de masquer ou démasquer chaque IRQ individuellement : si une IRQ est masquée, les signaux transmis seront ignorés par le contrôleur d'interruption. Dans ce TP, le masquage ou démasquage d'une IRQ se fera en deux temps :

1. il faut d'abord lire la valeur actuelle du masque sur le port de données 0x21 grâce à la fonction `inb` ;
2. l'octet récupéré est en fait un tableau de booléens tel que la valeur du bit N décrit l'état de l'IRQ N : 1 si l'IRQ est masquée, 0 si elle est autorisée : il faut donc forcer la valeur du bit N à la valeur souhaitée (sans toucher les valeurs des autres bits) et envoyer ce masque sur le port de données 0x21 grâce à la fonction `outb`.

Il est aussi possible de désactiver globalement toutes les sources d'interruption externes au processeur : c'est ce qu'on doit faire pour la majorité du code noyau, car celui-ci doit effectuer des opérations de façon atomique. Mais on doit aussi les activer aux endroits opportuns pour permettre l'arrivée et la prise en compte des interruptions utiles comme celle en provenance de l'horloge. On utilise pour cela les fonctions `void sti(void)` (active les interruptions externes) et `void cli(void)` (désactive toutes les interruptions externes). En résumé, la fonction `kernel_start` de votre noyau devrait au final ressembler à ceci :

```
void kernel_start(void)
{
// initialisations
    ...
// démasquage des interruptions externes
    sti();
// boucle d'attente
    while (1) hlt();
}
```

La fonction `hlt()` est définie dans `cpu.h` : elle a pour effet d'endormir le processeur (pour économiser de l'énergie). Le processeur sera réveillé par l'arrivée d'une interruption : il est donc essentiel que les interruptions soient démasquées avant d'appeler cette fonction.

Utilisation de l'horloge programmable

L'horloge programmable est un composant pouvant être paramétré de façon à générer des signaux à la fréquence voulue. Le circuit que l'on utilise dans ce TP émet un signal sur l'IRQ0 à une fréquence par défaut de 0x1234DD Hz (environ 1,19 MHz), ce qui est beaucoup trop rapide pour l'utilisation qu'on veut en faire.
Il est possible de régler la fréquence des signaux en utilisant les ports d'entrée-sorties associés à l'horloge programmable. Si on souhaite par exemple que l'horloge émette un signal toutes les 20 ms (50 Hz), on procédera de la façon suivante :

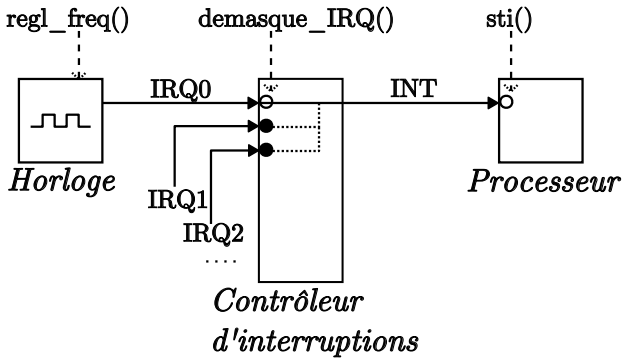
1. on envoie la commande sur 8 bits 0x34 sur le port de commande 0x43 grace à la fonction `outb` : cette commande indique à l'horloge que l'on va lui envoyer la valeur de réglage de la fréquence sous la forme de deux valeurs de 8 bits chacune qui seront émises sur le port de données ;

2. on envoie les 8 bits de poids faibles de la valeur de réglage de la fréquence sur le port de données 0x40 : cela peut se faire simplement par `outb((QUARTZ / CLOCKFREQ) & 0xFF, 0x40)` où `QUARTZ` vaut 0x1234DD et `CLOCKFREQ` vaut 50 ;
3. on envoie ensuite les 8 bits de poids forts de la valeur de réglage sur le même port 0x40.

On note que comme la valeur de réglage de la fréquence est limité à 16 bits, on ne pourra pas régler la fréquence d'émission du signal d'horloge à 1 Hz : il faudra donc que le traitant de l'interruption associé intègre un compteur pour savoir quand mettre à jour l'affichage du temps à l'écran.

Schéma global

On peut visualiser le chemin que prend le signal depuis l'horloge jusqu'au processeur grâce au schéma ci-dessous :



Conseils d'implantation

Un rappel de C

On va manipuler un pointeur de fonction dans cette séance. Supposons qu'on définit une fonction en C comme suit :

```
void fct(void)
{
   ...
}
```

On sait qu'on peut appeler cette fonction (c'est à dire executer son code) en écrivant simplement `fct();`. Mais l'idenfiant `fct` est aussi ce qu'on appelle un pointeur de fonction, c'est à dire une constante ayant comme valeur l'adresse de la fonction `fct`.
Dans le travail demandé ici, on va devoir écrire une fonction qui prend en paramètre un pointeur de fonction : `void init_traitant_IT(uint32_t num_IT, void (*traitant)(void))`. Ce traitant est la fonction assembleur `traitant_IT_32` définie dans `traitant.S`. On pourra donc initialiser la table des vecteurs d'interruption en appelant simplement `init_traitant_IT(32, traitant_IT_32);`.
Bien sûr, la fonction `traitant_IT_32` étant dans un fichier assembleur, il faudra rendre son prototype visible depuis le fichier C dans lequel on l'utilise.

Résumé du travail demandé

Le travail demandé peut être découpé de la façon suivante :

— écrire une fonction `ecrit_temps` qui prend en paramètre une chaine de caractères (ainsi que sa taille) et l'affiche en haut à droite de l'écran : c'est cette fonction qui sera appelée par le traitant d'interruption quand on devra mettre à jour l'affichage de l'heure ;

— écrire le traitant de l'interruption 32 qui affiche à l'écran le temps écoulé depuis le démarrage du système : ce traitant commence par une partie en assembleur pour sauvegarder les registres (qui est fournie dans le fichier `traitant.S`), mais l'acquittement de l'interruption et la partie gérant l'affichage doit être faite dans une fonction en C qu'on appelera `void tic_PIT(void)` (on attire au passage votre attention sur l'existence dans la mini-libc fournie d'une fonction `sprintf` qui vous sera vraisemblablement utile) ;
— initialiser l'entrée 32 dans la table des vecteurs d'interruptions, grace à la fonction mentionnée ci-dessus (`init_traitant_IT`) ;
— régler la fréquence de l'horloge programmable : la fréquence d'émission des signaux par l'horloge doit être une constante globale de votre système, afin de permettre facilement de la changer ;
— démasquer l'IRQ0 pour autoriser les signaux en provenance de l'horloge : on vous recommande d'écrire une fonction void `masque_IRQ(uint32_t num_IRQ, bool masque)` prenant en paramètre le numéro de l'IRQ (entre 0 et 7) à gérer ainsi qu'un boolean indiquant si on souhaite masquer ou démasquer l'IRQ en question ;
— démasquer les interruptions externes grâce à un appel à la fonction `sti()` comme expliqué dans le squelette de code donné plus haut.