# Croisière au cœur d'un OS*
# Étape 2 : segmentation et interruptions


## Résumé

Grâce au premier article, nous sommes en mesure de charger notre noyau en mémoire. Dans cet article, notre croisière se poursuit en configurant le processeur pour qu'il interagisse proprement avec les périphériques (interruptions) et la mémoire (segmentation). C'est parti !

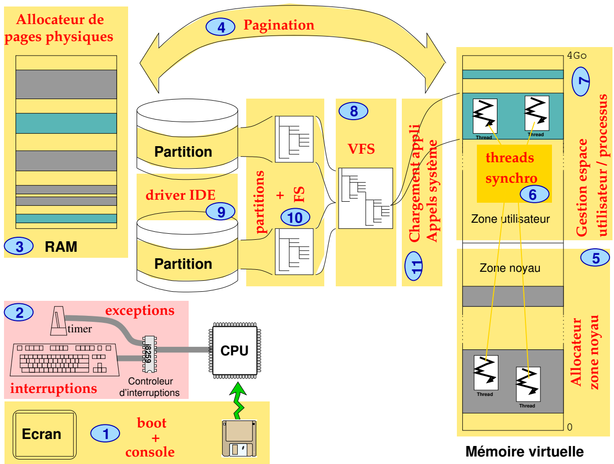




FIG. 1 – Programme des articles

## Introduction

Dans l'article précédent, nous avons décrit comment un noyau d'OS était chargé en mémoire principale, soit au moyen du chargeur de boot Grub, soit au moyen d'un chargeur de boot que nous proposions. Nous avions en particulier présenté quelques détails relatifs au mode dit *réel* historique dans lequel le processeur fonctionne lors de la phase de boot, et peu détaillé comment configurer le processeur pour un fonctionnement correct en mode dit *protégé*, dans lequel fonctionne SOS.

Dans cet article, nous allons justement aborder ce point, au travers de la configuration des éléments primordiaux de ce mode : la configuration de la segmentation pour adresser la totalité de l'espace mémoire physique possible 0-4Go sans encombre, la configuration du vecteur d'interruptions du processeur en mode protégé (`IDT`), et la configuration des contrôleurs d'interruptions pour un fonctionnement correct en mode protégé.

La figure 1 situe cet article dans la série. Elle n'insiste pas sur l'aspect segmentation que nous présentons pourtant ici, car SOS utilise un modèle de segmentation qui ne sera plus modifié une fois la configuration initiale effectuée. Comme dans le précédent article, nous commençons par décrire très brièvement les caractéristiques générales de l'architecture PC relativement à ces notions, puis nous détaillons le fonctionnement de SOS.

## 1 Quelques éléments d'architecture du PC

### 1.1 Caractéristiques générales du mode protégé

Dans l'article précédent, nous avions précisé que le processeur était passé du mode réel au mode protégé avant l'exécution de SOS, soit par le chargeur de boot *Grub*, soit par notre secteur de boot. Nous y revenons ici, car nous préférons reconfigurer proprement le processeur dans ce mode par nous-mêmes, afin de ne pas risquer d'écraser les structures de configuration mises en places par Grub ou notre secteur de boot.

La particularité la plus visible de ce mode protégé est que les registres, données et adresses sont sur 32 bits, ce qui permet d'accéder aux adresses physiques jusqu'à $2^{32}$ octets, soit 4 Go.

En fait, ce qui justifie l'adjectif "protégé" de ce mode correspond à deux notions :

1. la gestion flexible et personnalisable de la mémoire, qui permet d'isoler les programmes les uns des autres en mémoire physique (*cloisonnement*),
2. la définition de *niveaux de privilèges* qui permet de réserver certaines opérations de *supervision* de l'ensemble du système, donc très dangereuses, à une portion limitée du système (le plus souvent : le noyau).


*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 63 – Juillet/Août 2004 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

### 1.1.1 Adressage mémoire en mode protégé

**Différents types d'adresses.** En mode protégé, les adresses utilisées par les programmes et le système d'-exploitation subissent deux conversions au niveau du processeur avant de devenir des adresses physiques (voir la figure 2).

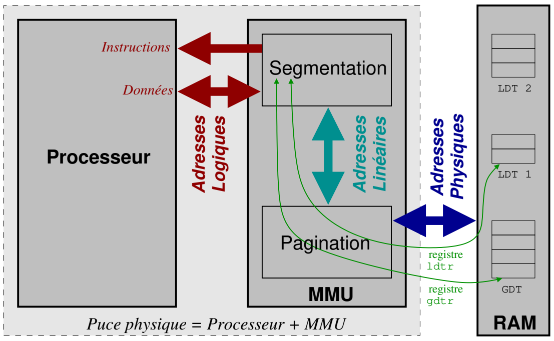



FIG. 2 – Traduction d'adresses dans les processeurs `x86`

On distingue ainsi trois types d'adresses (terminologie Intel) :

1. les adresses **logiques** utilisées par le cœur du processeur, et qui constituent les adresses mentionnées dans les programmes et le système d'exploitation,
2. les adresses **linéaires**,
3. les adresses **physiques**, *ie* celles relatives au bus mémoire.

La conversion des adresses logiques vers les adresses linéaires s'effectue grâce au mécanisme de *segmentation*, qui est obligatoire en mode protégé. Ce mécanisme de segmentation diffère de celui utilisé en mode réel. Nous y reviendrons plus loin.

La conversion des adresses linéaires en adresses physiques s'effectue grâce au mécanisme de *pagination*, qui est optionnel. S'il n'est pas activé, alors les adresses linéaires sont directement équivalentes aux adresses physiques. En ce qui nous concerne, la pagination est justement désactivée jusqu'à l'article 4.

**La MMU, et les tables de traduction d'adresses.** C'est une puce particulière, l'unité de gestion de la mémoire ou *MMU*, qui se charge de la *traduction d'adresses* du domaine logique au domaine physique, grâce à des *tables de traduction d'adresses* : `GDT` et `LDT` pour la segmentation, *répertoire et tables des pages* pour la pagination. Sur la plupart des processeurs modernes, dont les `80x86` font partie, il y a vraiment une distinction entre cœur de processeur et MMU, même si tous deux sont situés sur la même puce de silicium.

Il faut bien garder à l'esprit que, en mode protégé, le cœur du processeur ne perçoit *jamais* les adresses physiques qu'il manipule : il travaille dans un univers où toutes les adresses sont purement fictives, et délègue *intégralement* à la MMU les phases de traduction de ces adresses logiques en adresses physiques.

Segmentation comme pagination permettent de s'abstraire des adresses physiques manipulées par les programmes qui s'exécutent. Grâce à cela, il est possible d'avoir deux programmes différents qui sont chargés à la même adresse logique, mais qui sont en fait situés à des adresses physiques différentes en mémoire physique : il suffit pour cela de définir deux tables de traduction d'adresses, et de dire à la MMU de passer d'une table à l'autre quand on dit au processeur de passer d'une tâche à l'autre. On pourra constater que la gestion des processus Linux possède cette propriété. Faire l'équivalent sans avoir ni de notion de segmentation, ni de pagination serait possible mais beaucoup plus lourd.

### 1.1.2 Niveaux de privilèges

Le principe précédent permet le chargement très simple de plusieurs applications sans trop se préoccuper des adresses physiques. Mais, avec plusieurs applications dans le système, il y a risque qu'une application écrase les données/instructions des autres en mémoire (bug ou attaque). D'où la nécessité de *cloisonner* les applications, en désactivant ou en cachant les tables de traduction d'adresses des autres applications pendant que l'une d'elles est en train de s'exécuter.

Mais il faut tout de même trouver le moyen de gérer toutes les tables de traduction des applications du système, pour par exemple mettre en place les tables de traduction lors de la création d'une nouvelle application. C'est pourquoi on distingue deux *niveaux de privilèges* : le niveau *superviseur* qui a le droit de manipuler toutes les tables de toutes les applications du système, et le niveau *utilisateur* dans lequel chaque application n'a accès qu'à ses propres tables de traduction d'adresses. Passer d'un niveau de privilège à l'autre peut porter plusieurs noms : exceptions, appels système, ou encore *trappes*. Tandis qu'un accès invalide à un élément avec un niveau de priorité illégal conduit à une *exception de violation de protection*. À noter que la notion de niveaux superviseur/utilisateur n'a rien à voir avec celle de super-utilisateur (root) / utilisateur normal : nous parlons ici d'une protection au niveau matériel, alors que la notion de *droits* root a à voir avec une organisation logique au niveau applicatif.

Les processeurs Intel supportent en réalité 4 niveaux de privilèges, de 0 à 3 [1, section 4.5]. Le niveau *superviseur* correspond au niveau 0 [1, section 4.9], les autres niveaux sont de type *utilisateur*. Dans certains OS, les trois niveaux utilisateur sont mis à profit pour établir une hiérarchie entre *services* proches du noyau (niveaux 1 et 2), et *applications* indignes de confiance (niveau 3) : les services (systèmes de fichiers, piles réseaux, ...) ont d'autant plus de droits pour appeler des fonctions restreintes du noyau, que leur niveau de privilège est proche de 0. Dans SOS comme dans Linux, nous n'utilisons que les niveaux 0 (noyau) et 3 (applications utilisateur). La plupart des architectures autres que `i80x86` ne supportent par ailleurs que 2 niveaux de privilèges.

À part dans les derniers articles, nous travaillerons avec les droits superviseur, *ie* au niveau de privilège 0. Pour le moment, nous détaillons donc très peu les subtilités liées aux mécanismes de protections fondés sur ces niveaux de privilèges.

## 1.2 Segmentation en mode protégé

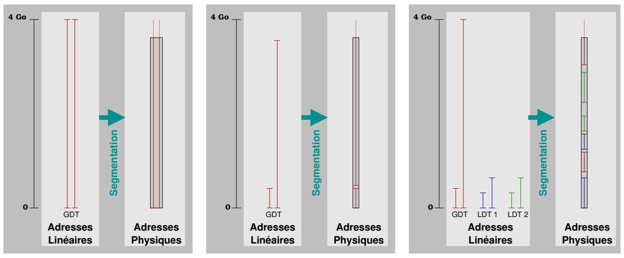
### 1.2.1 Principe

En mode réel comme en mode protégé, chaque donnée dans la mémoire est identifiée par une adresse de la forme `segment:offset`. En mode réel, l'adresse finale est calculée par le processeur de la manière suivante : `adresse = segment * 16 + offset` et les segments font tous 64 Ko. Dans ce mode, il n'y a pas à proprement parler de *table* de traduction d'adresses : il y a plutôt une arithmétique figée.

Le mécanisme de segmentation en mode protégé permet de définir des segments personnalisés en terme de taille, de positionnement ou de droits d'accès, *via* de véritables tables de traduction d'adresses. C'est le système d'exploitation qui doit fournir la description des segments à la MMU, dans des tables appelées *GDT* pour *Global Descriptor Table* ou *LDT* pour *Local Descriptor Table* [1, section 2.1.1]. Le système doit obligatoirement posséder une *GDT*, et peut optionnellement créer une ou plusieurs *LDT*.

Comme son nom l'indique, la *GDT* a pour vocation à être *globale* à tout le système, et à changer le moins possible durant toute la vie du système. Elle contiendra en principe les zones d'adresses constantes d'une tâche à l'autre. Elle pourra par exemple contenir les segments de code et de données pouvant être partagés par toutes ou partie des tâches, telles que le code/données de l'OS, le code/données des programmes et bibliothèques chargées en mémoire. Et les *LDT* ont pour vocation à être propres à chaque tâche. Elles pourront par exemple contenir les segments de code ou de données propres à chaque tâche du système.

En découpant ainsi la segmentation en deux niveaux global/local, de nombreuses combinaisons s'offrent au concepteur de l'OS pour organiser la gestion de la segmentation. La documentation Intel [1, section 3.2] en décrit trois : *i)* le *Basic Flat Model* dans lequel un unique segment couvre la totalité de l'espace mémoire physique possible (*ie* 4 Go), *ii)* le *Protected Flat Model* dans lequel le segment de code occupe une zone mémoire distincte des zones de données, et *iii)* le *Multisegment Model* dans lequel de nombreux segments sont définis : typiquement un segment par programme ou bibliothèque chargé en mémoire (en général dans la *GDT*), et un segment pour stocker les données locales de chaque tâche (en général dans la *LDT*).




(a) Flat (b) Protected Flat (c) Multisegment

FIG. 3 – Modes de segmentation standards, 2 segments par table(s) utilisée(s), pagination désactivée

La plupart du temps, le modèle *Basic Flat Model* est choisi : le mécanisme de pagination permet une gestion beaucoup plus fine de la mémoire, et la segmentation n'est donc pas particulièrement utile. C'est le modèle que choisit d'utiliser SOS, et c'est celui des versions de Linux récentes : nous justifierons ce choix dans l'article 4. Les premières versions de Linux adoptaient le modèle des autres OS commerciaux, qui se rapprochait du modèle *multisegment*.

### 1.2.2 Registres liés à la segmentation

Dans l'écriture d'une adresse logique sous la forme `segment:offset`, `segment` et `offset` peuvent être des entiers ou des registres. Nous décrivons ici le cas "registres", qui donne une idée plus précise du fonctionnement global.

Les *registres d'offset* sont n'importe quel registre général (`eax, ebx, ecx, edx`), ou le pointeur d'instruction courante (`eip`), ou les registres de pile (`esp, ebp`). Tous ces registres sont sur 32 bits, ce qui autorise les segments à faire jusqu'à 4Go.

Les *registres de segment* sont les registres `cs, ds, es, fs, gs` et `ss`, tous d'une largeur de 16 bits, et les valeurs qu'ils contiennent sont appelées des *sélecteurs de segment*. Le registre `cs` est réservé pour désigner le segment contenant le code en exécution, et `ss` est réservé pour désigner le segment contenant la pile. Les autres registres peuvent être utilisés pour manipuler des données. [1, section 3.6.2].

Ces *sélecteurs de segment* permettent d'identifier le segment à utiliser pour les accès mémoire, parmi les $8191+8191$ possibles (*GDT* + *LDT*). Ils contiennent trois informations [1, section 3.4.1] :

- Sur les 2 bits de poids faible, le *Requested Privilege Level*, qui indique avec quel niveau de privilège on souhaite accéder au segment.
- Sur le bit suivant, une indication sur la localisation du descripteur de segment : dans la *GDT* si le bit est à 0, ou dans la *LDT* courante si le bit est à 1.
- Les 13 bits restants constituent l'index du descripteur de segment dans la table (*GDT* ou *LDT*).

Ainsi, pour accéder à une donnée en mémoire, la MMU va procéder en plusieurs étapes (voir la figure

4) :

1. Récupération dans le sélecteur de segment du bit indiquant quel table doit être utilisée (GDT ou LDT) et de l'index du segment dans la table.
2. Récupération dans la table indiquée, du descripteur du segment concerné (voir plus bas).
3. L'adresse de base du segment sera finalement ajoutée à l'offset pour former l'adresse (linéaire) finale.

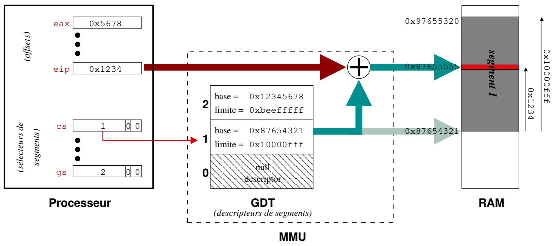



FIG. 4 – Segmentation sur `x86` (pagination désactivée)

### 1.2.3 Tables liées à la segmentation

Les tables *GDT* et *LDT* contiennent des *descripteurs de segments*, sur 64 bits. Les segments peuvent être de type *system* ou de type *code or data*. Pour le moment, nous n'utiliserons que ce dernier type qui correspond aux segments contenant des données ou des instructions en mémoire.

Un descripteur de segment contient l'adresse de base du segment, la taille du segment, le type du segment, et le niveau de privilège requis pour accéder au segment (voir [1, section 3.4.3] pour une description complète).

La MMU dispose de registres `gdtr` et `ldtr` (voir la figure 2) pour désigner les adresses de la *GDT* et de la *LDT* courantes. Chacun de ces registres est sur 48 bits : 32 bits pour l'adresse (linéaire) de base de la table, 16 bits pour la taille. Les instructions processeur spécifiques `lgdt`, `sgdt`, `lldt` et `sldt` permettent de charger ou de stocker les registres `gdtr` et `ldtr`.

## 1.3 Gestion des interruptions en mode protégé

### 1.3.1 Principe

Les interruptions sont des évènements qui interrompent le flot d'exécution du processeur (voir article 1). À chaque interruption, une routine de traitement appelée *gestionnaire d'interruption*, ou *interrupt handler*, ou encore *interrupt service routine* (*ISR*), peut être associée. Les interruptions peuvent être classées en trois catégories :

**Les exceptions.** Ce sont des interruptions générées en interne par le processeur suite à une erreur. Par exemple, la division par zéro déclenchera une exception, de même qu'une violation de protection mémoire (problème de niveaux de privilèges, instructions réservées au mode superviseur). Sur les processeurs x86, les exceptions sont au nombre de 32 [1, section 5.3]. Ces interruptions sont *synchrones*, c'est-à-dire que leur déclenchement est directement lié aux instructions exécutées, bien qu'elles ne soient pas déclenchées *explicitement* par le flot d'instructions.

**Les IRQ (pour *Interrupt ReQuest*), ou *interruptions matérielles*.** Ce sont des interruptions provenant du matériel externe au processeur. Ces évènements peuvent être par exemple l'appui d'une touche, la réception d'un paquet sur la carte réseau, ou la terminaison d'un transfert sur le disque. Là encore, ces interruptions ne sont pas déclenchées explicitement par le flot d'instructions. Mais elles sont cette fois *asynchrones* car l'instruction qui est interrompue n'est pour rien dans l'apparition de l'interruption.

**Les interruptions logicielles.** Ce sont des interruptions déclenchées explicitement et de manière synchrone par le logiciel, *via* des instructions spécifiques : `int`, `int3`, `into` et `bound` [1, section 5.1.2.2]. Ainsi, comme nous l'avions vu dans l'article 1, en mode réel, le BIOS propose au programmeur des routines d'interruptions logicielles permettant de lire ou écrire sur les disques, d'afficher à l'écran, etc... en utilisant des interruptions logicielles. En mode protégé, les interruptions logicielles sont également utiles. C'est en utilisant ces interruptions que les programmes utilisateur font appel au noyau : on parle d'*appel système*.

Les routines de traitement des interruptions sont implantées par le développeur du système d'exploitation. Suite à une division par zéro par exemple ou à une erreur de protection mémoire, le système d'exploitation peut décider de détruire la tâche ayant provoqué l'erreur. Suite à un appui sur une touche de clavier, le système d'exploitation peut effectuer un traitement visant à prendre en compte l'appui sur cette touche (enregistrement dans un buffer, affichage à l'écran, etc...).

Sur architecture x86, on peut disposer d'au maximum 256 interruptions. En mode protégé, les 32 premières interruptions sont réservées aux 32 exceptions du processeur, les autres interruptions étant utilisables pour les *IRQ* ou les interruptions logicielles.

## 1.4 Les *IRQ*

Comme nous l'avons déjà signalé, les *IRQ* sont des interruptions en provenance du matériel, et sont concentrées vers le processeur par des *contrôleurs d'interruptions*, aussi appelés *PIC* pour *Programmable Interrupt Controllers*. Sur les ordinateurs PC, il y a deux contrôleurs d'interruptions branchés en cascade, appelés *8259A* : le *maître* relié au processeur, et l'*esclave* relié au maître. Sur les processeurs récents, ces contrôleurs sont simulés par le processeur lui-même pour conserver la compatibilité ascendante ; Intel préconise cependant d'utiliser dorénavant l'*APIC* (pour *Advanced PIC*) du processeur à la place. C'est d'ailleurs parce qu'ils utilisent ces APIC que le `/proc/interrupts` de certains noyaux Linux renferment des IRQ dont le numéro est largement supérieur à 15. Dans SOS, nous continuerons d'utiliser les `8259A`.

Chacun de ces contrôleurs peut gérer jusqu'à 8 lignes d'interruption, c'est-à-dire être connecté à 8 matériels différents. Sur le contrôleur maître, la ligne d'interruption 2 est dédiée à la liaison avec le contrôleur esclave.

| Maître | Esclave | Utilisation |
|---|---|---|
| IRQ0 | | Timer |
| IRQ1 | | Clavier |
| IRQ2 | | Connexion avec l'esclave |
| | IRQ9 | Réservé |
| | IRQ10 | Réservé |
| | IRQ11 | Réservé |
| | IRQ12 | Réservé |
| | IRQ13 | Coprocesseur arithmétique |
| | IRQ14 | Contrôleur disque |
| | IRQ15 | Réservé |
| IRQ3 | | Port série 2 |
| IRQ4 | | Port série 1 |
| IRQ5 | | Port parallèle 2 |
| IRQ6 | | Contrôleur disquette |
| IRQ7 | | Port parallèle 1 |

TAB. 1 – Utilisation standard des IRQ

Au démarrage, le système d'exploitation va devoir configurer ces contrôleurs d'interruptions pour déterminer à quelle interruption du processeur correspondra quelle *IRQ*. En effet, lorsque l'*IRQ0* est levée, en mode protégé ce ne peut pas être la routine de traitement de l'interruption 0 du processeur qui est exécutée, puisque les 32 premières interruptions sont réservées pour les exceptions.

Dans SOS, nous programmerons les contrôleurs d'interruption de manière à faire correspondre l'*IRQ* $x$ à l'interruption $x + 32$ sur le processeur.

Par ailleurs, dans le tableau précédent, les *IRQ* sont classées de la plus prioritaire à la moins prioritaire. Ainsi, lors du traitement d'une *IRQ*, si une *IRQ* plus prioritaire survient, alors le traitement en cours sera suspendu, de manière à traiter l'*IRQ* prioritaire. On peut donc imbriquer les traitements d'interruptions.

### 1.4.1 Registres et données liés à la gestion des interruptions

Le registre du processeur dit *eflags* (voir [1, section 3.6.3]) comporte un bit `IF` qui permet d'activer ou de désactiver les interruptions matérielles. La position de ce bit peut être directement contrôlée grâce aux instructions `sti`, *reSTore Interrupts* pour activer toutes les interruptions et `cli`, *CLear Interrupts* pour les désactiver.

Un contrôle plus fin à la granularité de chaque interruption, ou par niveaux de priorité des interruptions, peut être obtenu en envoyant des commandes aux contrôleurs d'interruptions (voir la section 3.4.2).

### 1.4.2 Table liée à la gestion des interruptions

Les routines de traitement des interruptions sont consignées dans l'*IDT*, pour *Interrupt Descriptor Table* : lors de l'arrivée de l'interruption $n$, le processeur va aller lire l'entrée $n$ de l'*IDT* pour trouver l'adresse de la routine de traitement. Les entrées de l'*IDT* sont de trois types :

– Le type **trap gate** est le type le plus simple : on donne l'adresse de la routine, le segment de code à utiliser, ainsi que le privilège requis pour exécuter l'interruption si elle est de type logicielle. Lorsque l'interruption va être traitée, le processeur va simplement exécuter la routine stockée à l'adresse donnée dans l'entrée de l'*IDT*.

– Le type **interrupt gate** est strictement similaire au **trap gate**. La seule différence est qu'avant d'appeler la routine, le processeur va désactiver les interruptions.

– Le type **task gate** est un peu différent. Il va restituer les registres et les tables de traduction d'adresses sauvegardés dans un segment spécial appelé *TSS* pour *Task State Segment* : il y a *changement de contexte* vers la tâche sauvegardée dans le *TSS*. Nous ne détaillerons pas pour l'instant ce point, mais nous aurons l'occasion d'y revenir ultérieurement.

L'*IDT* est créée par le développeur de l'OS, son adresse et sa taille doivent être fournies au processeur dans le registre de 48 bits `idtr` : 32 bits pour l'adresse (linéaire) de base de la table, 16 bits pour la taille [1, section 5.8], bien qu'Intel précise que la table ne devrait pas renfermer plus de 256 entrées (soit 2048 octets). Les instructions `lidt` et `sidt` permettent respectivement de charger ou de stocker le registre `idtr`.

### 1.4.3 Fonctionnement d'une routine de traitement d'interruption

Avant d'exécuter une routine de traitement d'interruption de type *trap* ou *interrupt gate* (les deux types qui nous intéressent pour le moment), le processeur va empiler diverses informations sur la pile, comme lors d'un appel de fonction. Les informations empilées sont le registre `eflags`, le segment de code `cs`, le pointeur d'instruction `eip` et un éventuel code d'erreur. Ce code d'erreur n'est utilisé que pour certaines exceptions, pas pour les *IRQ* ni les interruptions logicielles (voir [1, section 5.3]).

Ces informations vont permettre de revenir à l'instruction qui a été interrompue (interruptions matérielles, logicielles, et exceptions de type *fault* selon la terminologie Intel [1, section 5.4]), ou à l'instruction qui suit (exceptions de type *trap*), une fois que la routine d'interruption sera terminée.

Le traitement d'une interruption se réalise donc généralement en 3 temps :

1. Sauvegarde de tous les registres, à savoir les registres généraux et les registres de segment.
2. Traitement effectif de l'interruption.
3. Restauration de tous les registres sauvegardés sauf `eflags`.
4. Retour à l'instruction interrompue ou à celle qui suit : instruction `iret`, qui est similaire à l'instruction de retour de fonction `ret`, sauf que `iret` dépile également `eflags`.

En général, les opérations 1 et 3 sont réalisées par une petite routine en langage assembleur, et la routine 2 est implémentée en langage C. Elles garantissent que, quoi que puisse faire la routine d'interruption, celle-ci ne va pas interférer avec le programme interrompu : tous les registres sont restitués à l'identique avant de revenir au programme interrompu.

À noter que lorsque le *lancement* (*ie* avant l'étape 1) d'une routine de traitement d'une exception provoque une faute (violation de protection, adresse invalide, débordement du segment de pile), une exception spéciale est levée : l'exception *faute double*. Et si le *lancement* de la routine associée à la faute double provoque à son tour une faute, le processeur est réinitialisé : il y a *faute triple* et reboot immédiat de la machine. Si cependant une exception survient *pendant* l'exécution d'une routine d'exception, les traitements sont simplement imbriqués.

#### 1.4.4 *Top-halves* et *bottom-halves*

Le plus souvent, on préférera désactiver les interruptions matérielles pendant qu'on est en train de traiter une interruption matérielle, en particulier parce que le traitement d'une interruption commence souvent par dialoguer avec le matériel pour récupérer des informations.

Si les routines de traitement d'interruption sont totalement non-interruptibles, alors le système perd en réactivité pendant le traitement d'une interruption matérielle, puisqu'aucune autre interruption matérielle ne peut être servie pendant ce temps. Dans ce cas, il est très chaudement recommandé que les routines de traitement des interruptions matérielles soient le plus rapide possible.

Ce n'est pas toujours possible ni évident. C'est pourquoi la plupart des OS découpent les traitements d'interruptions matérielles en 2 parties. La première partie (*top half*) est non-interruptible et contient le minimum pour prendre en compte l'information signalée par l'interruption, en interrogeant le matériel au besoin. Il s'agit par exemple de réagir à une interruption *clavier*, en ajoutant dans un buffer le code clavier de la touche appuyée. La deuxième partie (*bottom half*) s'occupe de traitements plus gourmands en processeur, qui ne nécessitent plus l'accès au matériel ayant généré l'interruption, et qui sont totalement ou partiellement interruptibles : on peut servir d'autres interruptions pendant ces traitements. Il s'agit par exemple de traduire les codes clavier relevés dans le buffer précédent, et de les traduire en code ASCII en tenant compte des séquences de touches (Ctrl, Alt, ...), du type de clavier (azerty, qwerty, ...), etc...

Dans le cas des interruptions logicielles ou des exceptions, ce découpage est également possible, pour permettre la prise en charge des interruptions matérielles pendant le traitement de l'interruption.

Sous Linux par exemple, les bottom halves existent sous plusieurs formes "historiques" : *BH* (pour... *Bottom Half*) et *taskqueues* dans les vieux noyaux, *tasklets* et *softirqs* pour les noyaux 2.4, et *workqueues* avec les noyaux 2.6. Pour plus de simplicité, dans SOS nos routines d'interruption seront uniquement des *top-halves* de type *intyerrupt gate*, c'est-à-dire en un seul morceau non interruptible par défaut.

# 2 Mise en place de la segmentation dans SOS

Que ce soit *Grub* ou notre secteur de boot, tous deux placent le processeur en mode protégé, et donc s'occupent déjà de mettre en place la segmentation en configurant leur *GDT*. Mais, à la fois pour éviter de l'écraser et pour avoir la maîtrise de notre propre *GDT*, nous redéfinissons la nôtre.

## 2.1 Configuration générale

Dans le cadre de SOS, nous avons fait le choix d'utiliser un modèle de segmentation *flat* dans la *GDT*, avec deux segments, un pour le code, un pour les données : en effet, certains attributs des descripteurs de segments ne sont pas équivalents pour les segments de code et les segments de données dans la norme Intel [1, section 3.4.3.1]. Nos deux segments couvrent tous deux 4 Go, soit la totalité de l'espace physique adressable. Ces deux segments seront utilisés pour exécuter le code du noyau et pour accéder aux données du noyau, leur niveau de privilège sera donc 0 ; des segments avec un niveau de privilège 3 seront définis beaucoup plus tard. Nous ne définissons aucune *LDT*.

## 2.2 Mise en place

### 2.2.1 Configuration de la *GDT*

Au total, nous aurons non pas deux entrées dans la *GDT*, mais trois. En effet, le premier descripteur doit être obligatoirement le *null descriptor*, qui sert à détecter les erreurs bêtes de manipulation erronée des sélecteurs de segments. Nos segments de code et de données seront situés après.

`hwcore/gdt.c` définit la structure d'un descripteur de segment `hwcore/gdt.c:struct x86_segment_descriptor` sous la forme d'un champ de bits, d'après la spécification Intel [1, section

3.4.3] (voir la section 1.2.3). On pourra remarquer la directive `__attribute__ ((packed))` dont le rôle est de forcer `gcc` à respecter scrupuleusement l'organisation en mémoire imposée par le code source, en l'empêchant de rajouter des espaces entre les champs, chose qu'il s'autorise à faire par défaut afin d'optimiser les accès aux membres des structures.

Ensuite, la *GDT* est initialisée, sous forme d'un tableau de descripteurs initialisés par la macro `hwcore/gdt.c:BUILD_GDTE(rpl, is_code)` qui s'occupe de remplir les champs de la structure précédente pour former des segments commençant à l'adresse linéaire 0, et de taille 4Go. Les deux paramètres de la macro permettent de préciser le niveau de privilège nécessaire pour accéder au segment, et si le segment est de type code ou données. La *GDT* est initialisée par le compilateur même :

```
/** The actual GDT */
static struct x86_segment_descriptor gdt[] = {
  [SOS_SEG_NULL]  = (struct x86_segment_descriptor){ 0, },
  [SOS_SEG_KCODE] = BUILD_GDTE(0, 1),
  [SOS_SEG_KDATA] = BUILD_GDTE(0, 0),
};
```

Les macros `SOS_SEG_NULL`, `SOS_SEG_KCODE`, `SOS_SEG_KDATA` sont définies dans `hwcore/segment.h`, désignent les index des descripteurs de segments dans la *GDT*, et sont respectivement égales à 0, 1 et 2.

Enfin la fonction `hwcore/gdt.c:sos_gdt_setup()` se charge de la configuration de la MMU pour qu'elle prenne en compte cette *GDT*. Il s'agit simplement de configurer le registre `gdtr` à partir de la variable `gdtr`. La structure de cette variable, `hwcore/gdt.c:struct x86_gdt_register`, est à nouveau la traduction littérale des spécifications d'Intel [1, figure 3-11]. L'initialisation de la variable `gdtr` suit également les spécifications d'Intel :

```
struct x86_gdt_register gdtr;

/* Address of the GDT */
gdtr.base_addr = (sos_ui32_t) gdt;

/* The limit is the maximum offset in bytes
   from the base address of the GDT */
gdtr.limit     = sizeof(gdt) - 1;
```

La suite de la fonction `hwcore/gdt.c:sos_gdt_setup()` est une routine assembleur qui :

1. Appelle l'instruction `lgdt` pour charger le registre `gdtr` à partir de cette variable `gdtr`.
2. Charge le registre de code `cs` et les registres de données `ss, ds, es, fs, gs` avec les sélecteurs associés. La valeur des sélecteurs est établie par la macro `hwcore/segment.h:SOS_BUILD_SEGMENT_REG_VALUE()` selon la spécification Intel (voir la section 1.2.2). Le chargement du registre de code se fait de manière détournée, par un saut dit *long*, dans lequel on précise le sélecteur de segment, en plus de l'*offset* habituellement suffisant, et qui correspond à celui de l'instruction qui suit le saut.

# 3 Mise en place de la gestion des interruptions dans SOS

## 3.1 Configuration générale

La gestion des interruptions se fait à deux niveaux : la gestion des interruptions matérielles et des exceptions par le processeur *via* l'*IDT* d'un côté (fichiers `hwcore/idt.c`, `hwcore/irq.c`, `hwcore/exception.c`), et la gestion des contrôleurs d'interruptions matérielles de l'autre (`hwcore/i8259.c`). La figure 5 résume les éléments qui constituent cette gestion, et qui sont détaillés dans la suite.

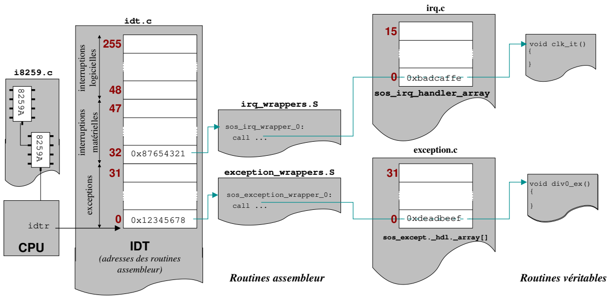



FIG. 5 – Gestion des interruptions dans SOS (256 entrées de type *interrupt gate* dans l'IDT)

## 3.2 Mise en place du vecteur d'interruptions

Dans `hwcore/idt.c`, la gestion du vecteur d'interruptions est en tous points conforme à la spécification Intel (voir la section 1.4.2). La fonction d'initialisation `hwcore/idt.c:sos_idt_setup()` se contente d'initialiser le registre `idtr` du processeur de sorte qu'il pointe vers le tableau de descripteurs `hwcore/idt.c:idt[]` :

```
/* Address of the IDT */
idtr.base_addr  = (sos_ui32_t) idt;

/* The limit is the maximum offset in bytes
   from the base address of the IDT */
idtr.limit      = sizeof(idt) - 1;

/* Commit the IDT into the CPU */
asm volatile ("lidt %0\n"::"m"(idtr):"memory");
```

De façon analogue à ce qui se passe pour la *GDT*, les 256 (*ie* la valeur de `hwcore/idt.h:SOS_IDTE_NUM`) entrées de l'*IDT* sont définies sous la forme du champ de bits `hwcore/idt.c:struct x86_idt_entry`. Elles sont remplies une à une juste avant de faire les opérations précédentes, par une boucle qui indique que ce sont toutes des *interrupt gates*, qui sont désactivées par défaut.

Le véritable cœur de la gestion de l'*IDT* est constitué de la fonction `hwcore/idt.c:sos_idt_set_handler(index, handler_addr, lowest_priviledge)`. Cette fonction a pour rôle d'associer l'adresse `handler_address` d'une routine, à une entrée particulière de l'*IDT* configurée en tant qu'*interrupt gate* ; lorsque cette adresse vaut `NULL`, l'entrée de l'*IDT* associée est désactivée.

Comme on peut s'en rendre compte, la gestion de cette *IDT* se limite effectivement à la seule gestion des entrées dans cette table. Toute la partie "traitement" des interruptions et exceptions est reportée dans les deux sous-systèmes suivants : interruptions matérielles (IRQ) et exceptions. Nous laissons le traitement des interruptions logicielles de côté pour le moment.

## 3.3 Gestion des exceptions

La norme Intel impose que les 32 premières entrées de l'*IDT* correspondent aux routines de traitement des exceptions. Le fichier `hwcore/exception.c` prend en charge la gestion de ces 32 exceptions, dont la liste est donnée dans `exception.h` : la fonction principale étant `hwcore/exception.c:sos_exception_set_routine()` dont le rôle est d'associer une routine de traitement à une exception indiquée en paramètre.

Pour cela, les fonctions qui y sont définies utilisent bien évidemment celles de `hwcore/idt.c` pour positionner l'adresse des routines. Cependant, les paramètres fournis à la routine située à cette adresse ne sont pas tous identiques suivant le numéro de l'exception : pour certaines exceptions, le processeur fournit un code d'erreur, pour d'autres pas. Pour cette raison, la gestion des exceptions fonctionne en 2 étapes afin d'appeler les *véritables* routines avec un protocole uniforme (voir la figure 5) :

- Les routines mentionnées dans l'*IDT* correspondent à des routines assembleur, et ne sont pas modifiées durant toute la vie du système.
- Ces routines assembleur appellent suivant un protocole standard, la *véritable* routine associée, dont elles vont chercher l'adresse dans le tableau `hwcore/exception.c:sos_exception_handler_array[]`.

Un cas particulier est à mentionner pour l'exception *faute double* qui peut mener à une faute triple (voir la section 1.4.3) et donc à un reboot immédiat de la machine si le processeur détecte une erreur pendant qu'elle est en train d'appeler la routine associée. Pour cette exception, nous définissons une routine de traitement particulière en assembleur, pas modifiable, et qui fait le maximum pour qu'aucune faute triple ne puisse pas avoir lieu. Cette routine est une boucle infinie (instructions `1: hlt; jmp 1b`), ce qui est toujours mieux qu'un reboot immédiat. Bref, si votre machine se bloque dans SOS, il est probable qu'une faute double ait été détectée.

### 3.3.1 Routines assembleur

Elles sont toutes définies dans `hwcore/exception_wrappers.S`, qui utilise les extensions de macros de l'assembleur GNU `gas` : les macros `.irp` et `.endr` permettent de récrire plusieurs fois le même morceau de code en faisant itérer une variable (ici `id`) sur un ensemble de valeurs (les numéros d'exception). Elles sont ainsi découpées en 2 paquets de routines : celles qui sont destinées aux exceptions avec code d'erreur, et celles sans code d'erreur (voir la section 3.3).

Ces routines assembleur commencent, comme indiqué dans la section 1.4.3, par sauvegarder les registres de la tâche interrompue, afin de ne pas altérer son fonctionnement. Cette sauvegarde se fait en les empilant sur la pile de la tâche interrompue. Puis ces routines configurent la pile suivant un modèle standard, de sorte qu'elles puissent appeler la *véritable* routine de traitement (écrite en général en C) qui devra posséder un prototype standard quel que soit le numéro d'exception auquel elle est rattachée. La figure 6 donne cette configuration de la pile standard qui est établie par ces routines assembleur.

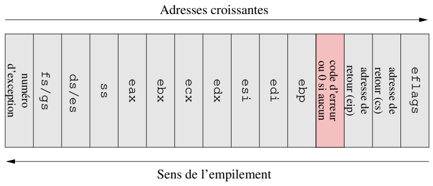



FIG. 6 – État de la pile standard, juste avant l'appel de la routine *véritable*

L'étape d'appel de la *véritable* routine d'exception repose sur le tableau d'adresses des routines `hwcore/exception.c:sos_exception_handler_array[]`, défini dans le fichier, `hwcore/exception.c`. Cette façon de procéder permet d'utiliser les adresses de ces fonctions le plus simplement possible, puisqu'on on en a besoin dans la fonction, en C, `hwcore/exception.c:sos_exception_set_routine()`, pour remplir les entrées de l'*IDT*. Pour l'exception $id$, il s'agit d'appeler la $id^{eme}$ routine de ce tableau en lui fournissant en paramètre le numéro de l'exception :

```
pushl $\id /* Arg #1 of the handler = index of the exception */
leal  sos_exception_handler_array,%edi
call  *\id*4(%edi)
```

Au retour des *véritables* routines, les registres de la tâche interrompue sont restaurés et la pile est restituée dans l'état attendu par le processeur. Enfin la routine assembleur retourne à la tâche interrompue par l'intermédiaire de l'instruction `iret` (voir la section 1.4.3).

### 3.3.2 Routines *véritables*

Tout l'intérêt des routines assembleur est de cacher les phases de sauvegarde/restauration de l'état de la tâche interrompue, et d'uniformiser le protocole d'appel à la routine *véritable*. À ce niveau, le prototype de la routine *véritable* est ainsi défini par le type suivant, qui ne dépend plus du type d'exception (avec/sans code d'erreur) :

```
typedef void (*sos_exception_handler_t)(int exception_number);
```

C'est-à-dire une fonction qui reçoit le numéro de l'exception fourni par la routine assembleur, et qui ne renvoie rien.

## 3.4 Gestion des interruptions matérielles

Pour les interruptions matérielles, le mécanisme est quasiment identique à celui des exceptions, sauf que le code est déporté dans `irq.c` et `irq_wrappers.S`.

La principale différence réside dans le début des routines assembleur, juste avant d'appeler les routines *véritables* :

```
/* Send EOI to PIC. See Intel 8259 datasheet
   available on Kos website */
movb  $0x20, %al
outb  %al, $0x20
```

Ceci correspond à un message d'acquittement (*EOI* pour *End Of Interrupt*) envoyé au contrôleur d'interruption, qui signifie : "*c'est bon, j'ai bien noté que tu m'as signalé une interruption matérielle, et je la prends en charge*". De la sorte, le contrôleur d'interruptions peut signaler d'autres interruptions pendant l'exécution de la *véritable* routine de traitement de l'interruption, ce qui augmente la réactivité du système, mais pose quelques risques de débordement de pile. Puisque la façon de signaler l'*EOI* n'est pas la même suivant le contrôleur qui est responsable de l'interruption signalée (maître/esclave), on regroupe le code des routines assembleurs de `hwcore/irq_wrappers.S` en deux paquets.

Pour terminer, le prototype de la *véritable* routine de traitement de l'interruption matérielle est analogue à celui des exceptions :

```
typedef void (*sos_irq_handler_t)(int irq_level);
```

Les interruptions sont désactivées par défaut pendant l'exécution de cette routine, puisque l'entrée de l'*IDT* correspondante est de type *interrupt gate* (voir la section 1.4.4). Mais il est possible de les réactiver dans la routine *véritable* (voir la section 3.4.2), auquel cas il pourra y avoir imbrication des routines puisque l'*EOI* aura déjà été envoyé au contrôleur d'interruptions.

### 3.4.1 Gestion des contrôleurs d'interruptions

Lorsque la machine démarre, le Bios configure les deux contrôleurs d'interruptions (Intel 8259A) pour que les IRQ 0..7,8..15 soient perçues comme les interruptions 0x8..0xf,0x70..0x77 par le processeur. Or, en mode protégé, la norme Intel précise que les interruptions 0..31 côté processeur sont réservées pour les exceptions. Il faut donc reconfigurer les contrôleurs pour leur indiquer que leurs IRQ 0..15 doivent maintenant correspondre aux interruptions 32..47 côté processeur. Et il faut maintenir dans cette reconfiguration la topologie standard de type PC des deux contrôleurs : le deuxième (esclave) est branché au premier (maître) par l'intermédiaire de la ligne d'interruption 2 du maître.

Pour cela, SOS communique avec les deux contrôleurs *via* les ports d'entrée/sortie `0x20/0x21` pour le maître, et `0xa0/0xa1` pour l'esclave. Pour chaque contrôleur, l'étape de configuration procède de quatre étapes successives dans `hwcore/i8259.c:sos_i8259_setup()`. Chaque étape, `ICW1` à `ICW4` (pour *Initialization Control Word*), consiste en l'écriture d'un octet sur un des ports de chaque contrôleur. Nous ne donnons pas les détails ici, nous renvoyons aux commentaires de `hwcore/i8259.c`, et à la spécification du composant [2, page 10 et suivantes].

### 3.4.2 Masquage des interruptions

Une fois l'initialisation des contrôleurs faite, le reste des fonctions se limite à positionner le masque des interruptions matérielles autorisées sur chaque contrôleur. C'est ce que font les fonctions `hwcore/i8259.c:sos_i8259_disable_irq_line(int numirq)` et `hwcore/i8259.c:sos_i8259_enable_irq_line(int numirq)`, en dialoguant avec les ports d'entrée/sortie des contrôleurs d'interruptions.

Mais dans la pratique, pour masquer les interruptions, on sera le plus souvent amenés à les désactiver au niveau du processeur directement, et en bloc, par les instructions `cli` et `sti` (voir la section 1.4.1). Pour appeler ces instruction, nous définissons les macros `hwcore/interrupt.h:sos_disable_IRQs(flags)` et `hwcore/interrupt.h:sos_restore_IRQs(flags)` :

– La première sauvegarde l'état courant du bit `IF` du registre `eflags` du processeur dans la variable `flags` passée en paramètre, avant d'appeler `cli` qui positionne `IF` à 0 si ce n'était pas déjà le cas.
– La deuxième restaure le bit `IF` du registre `eflags` tel qu'il avait été sauvegardé précédemment dans la variable `flags`.

En général, `flags` est une variable allouée dans la pile, *ie* une variable locale à la fonction englobante. Cela permet les imbrications de blocs `disable/restore` du type : "`sos_disable_IRQs(); A; sos_disable_IRQs(); B; sos_restore_IRQs(); C; sos_restore_IRQs()`", en garantissant que le bloc `C` s'exécutera avec les interruptions désactivées comme on l'attend.

# 4 Testons !

La petite démonstration associée à cet article est des plus simples. Nous allons illustrer le fonctionnement des interruptions seulement, considérant que la segmentation est établie une fois pour toute et peu digne d'intérêt dans SOS. Pour cela, nous allons faire générer une interruption matérielle dite d'*horloge* (ou *Timer*) par une puce autrefois périphérique au processeur (Intel 82C54), et aujourd'hui simulée par le *chipset*. Et nous allons aussi faire générer une exception par notre programme de test même (division par zéro).

La figure 7 présente une photo d'écran peu excitante de la démonstration : à gauche en rouge le nombre d'exceptions observées en représentation binaire, à droite en vert le nombre d'interruptions d'horloge. Bien sûr, tout cela *paraît* bouger simultanément et à vue d'œil.

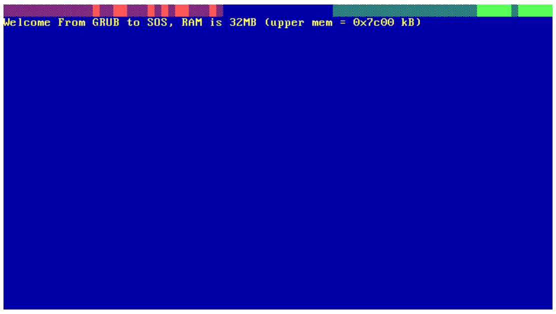



FIG. 7 – Copie d'écran après chargement du noyau

## 4.1 Mise en place de l'interruption d'horloge

Le but est de demander à un périphérique extérieur de générer une interruption, et d'y associer la routine de notre choix. Nous aurions pu choisir le clavier, mais pour des raisons de simplicité dans la mise en place, notre choix s'est porté sur une horloge matérielle, de configuration simple.

Sur un PC, il y a moult bases de temps disponibles. Sur les PC d'origine (XT), il y a le compteur matériel `i82C54` (maximum 1MHz environ) ; Sur la génération suivante (AT), il y a l'horloge dite "temps-réel" (*RTC* pour *Real-Time Clock*) "RTC CMOS" (un `MC146818`) à plus faible fréquence (quelques kHz) et surtout sauvegardée par pile pour maintenir la date même quand la machine est éteinte ; Sur les pentium, il y a les compteurs associés au signal d'horloge du processeur (certains *MSR*, pour *Machine State Registers*) mis à jour à la fréquence de plusieurs GHz de nos jours. Pour des raisons de simplicité et parce qu'elle est la plus standard sur les PC, nous avons choisi la première base de temps : le compteur `i82C54`.

Cette puce est cadencée à 1193182 MHz, et comporte 3 compteurs qui permettent chacun de générer un signal de fréquence comprise entre $\frac{1193182}{65536} = 18.2$ Hz et $\frac{1193182}{1}$ Hz. En fait chacun des compteurs peut être programmé dans 6 modes de fonctionnement différents [3] : nous choisirons le mode "2" pour la génération périodique d'une impulsion. Le compteur 0 est le seul qui est connecté au processeur, par l'intermédiaire de l'IRQ 0 ; le compteur 1 est destiné au rafraîchissement de la RAM (périodique de période $15\mu s$), et le compteur 2 est connecté au haut-parleur interne du PC. Il suffit donc de programmer la valeur du compteur 0, qui est décrémenté toutes les $\frac{1}{1193182}s$, pour que l'interruption matérielle 0, levée à chaque fois que ce compteur vaut 0, le soit à la fréquence souhaitée. Cela est effectué très simplement en suivant les spécifications du `i82C54` [3] dans la fonction `hwcore/i8254.c:sos_i8254_set_frequency(unsigned int freq)`, par dialogue avec la puce sur 4 ports d'entrée/sortie.

Une fois l'horloge programmée pour une génération de l'*IRQ 0* à 100 Hz environ, il suffit de connecter notre routine de traitement associée (dans `sos/main.c`) :

```
/* Configure the timer so as to raise the IRQ0 at a 100Hz rate */
sos_i8254_set_frequency(100);

/* Binding some HW interrupts and exceptions to software routines */
sos_irq_set_routine(SOS_IRQ_TIMER,
                    clk_it);
```

Quant à la routine associée à cette interruption, `sos/main.c:clk_it()`, son rôle est d'incrémenter la variable globale `clock_count` (définie en tant que variable locale en `static`), et d'afficher sa valeur au format binaire à l'aide de la fonction `sos/main.c:display_bits()` :

```
static void clk_it(int intid)
{
  static sos_ui32_t clock_count = 0;
  display_bits(0, 48,
        SOS_X86_VIDEO_FG_LTGREEN | SOS_X86_VIDEO_BG_BLUE,
        clock_count);
  clock_count++;
}
```

## 4.2 Mise en place de la routine d'exception `division par 0`

Le principe est d'associer la routine `sos/main.c:divide_ex()` à l'exception division par 0. Ceci se fait de façon analogue à ce qui précède, mais avec la fonction `hwcore/exception.h:sos_irq_set_routine()`. Et la routine `divide_ex()` fait l'équivalent de la routine `clk_it()` précédente, mais en rouge à l'écran.

## 4.3 Programme principal

Le reste du programme principal est très simple. Avant de faire les opérations précédentes, il s'agit d'appeler les fonctions de mise en place de la *GDT*, de l'*IDT*, et de l'initialisation des sous-systèmes d'IRQ et d'exceptions :

```
sos_gdt_setup();
sos_idt_setup();
sos_exceptions_setup();
sos_irq_setup();
```

Et après avoir associé les interruptions matérielles/les exceptions, aux routines `clk_it()` et `divide_ex()`, il s'agit *i)* de réactiver les interruptions qui étaient désactivées par défaut par *Grub* ou notre secteur de boot (instruction `asm volatile ("sti");`) pour que l'interruption d'horloge puisse être prise en charge par notre routine.

Et *ii)*, il s'agit de générer l'exception *division par zéro*. Pour cela, le code paraît relativement compliqué, mais sa complication a pour objectif de leurrer `gcc` afin qu'il n'optimise pas trop, et afin qu'il y ait effectivement division par zéro à un moment ou à un autre. L'effet désiré est alors obtenu : l'exception est levée. D'après la spécification Intel, cette exception appartient à la catégorie *fault* (voir la section 1.4.3), ce qui signifie que lorsque notre routine d'exception se termine, l'adresse à laquelle le processeur retourne est l'exacte adresse de l'instruction qui a fait la division par zéro, ce qui signifie qu'il va y avoir à nouveau une division par zéro immédiatement après la fin de la routine, et donc rééxécution de la routine à nouveau... tout ceci sans fin.

On observera d'ailleurs que le compteur du nombre d'exceptions observées évolue beaucoup plus vite que le compteur du nombre d'interruptions d'horloge. En effet, la première évolue en proportion de la fréquence du processeur, en général beaucoup plus rapide que la fréquence programmée dans le `82C54`.

## SOS en short

Nous avons maintenant un noyau relativement autonome, capable d'être chargé en mémoire, d'accéder à toute la mémoire physique sans surprise, et de réagir aux événements extérieurs. Certains OS s'arrêtent d'ailleurs là dans les fonctionnalités élémentaires qu'ils supportent, en particulier les petits OS embarqués. Nous choisissons d'aller légèrement plus loin pour SOS, avec la gestion de l'allocation dynamique de mémoire physique (prochain article), et la pagination (article d'après). D'ici là, sortez tongues, palmes, et tuba, car le moment est venu d'une petite escale estivale bien méritée !...

Thomas Petazzoni et David Decotigny
`Thomas.Petazzoni@enix.org` et `d2@enix.org`
Site de SOS : `http://sos.enix.org`
Projet KOS : `http://kos.enix.org`

*Á Joseph*

## Références

[1] Intel Corp. Intel architecture developer's manual, vol 3, 1997.

[2] Intel Corp. Intel 8259A datasheet. `http://kos.enix.org/docs.php`, 1988.

[3] Intel Corp. Intel 82C54 datasheet. `http://kos.enix.org/docs.php`, 1994.