Croisière au cœur d'un OS*
Étape "7" : Espaces d'adressage, appels système et applications utilisateur

Résumé

Au cours des articles 6 et 6.5, nous avons étudié l'implémentation des *threads noyau*. Ce mois-ci, nous vous proposons de mettre en place différents *espaces d'adressage* et de faire fonctionner dans chacun d'entre eux des *threads utilisateur* qui pourront interagir avec le noyau au travers d'*appels système*.

Introduction

Au cours des articles 6 et 6.5, nous avons étudié le fonctionnement des *threads noyau*. Ceux-ci sont exécutés avec les *privilèges* du noyau : ils ont accès à toute la mémoire du noyau et ont le droit d'exécuter toutes les instructions du processeur, même les plus dangereuses. Potentiellement, ils peuvent donc endommager le noyau ou se gêner mutuellement. Ils sont donc réservés à l'exécution de quelques tâches de supervision internes au noyau, telles que la gestion de la *zone d'échange* (ou *swap*) par exemple.

Dans les systèmes d'exploitation modernes, on souhaite exécuter les différentes applications (*shell*, commandes, applications graphiques ou non) de manière à ce qu'elles soient cloisonnées les unes des autres. Ce cloisonnement permet d'éviter qu'une erreur de programmation dans l'une d'entre elles n'entraîne le plantage des autres. Pour les mêmes raisons, on souhaite que le noyau ne soit pas accessible directement par les applications. Sous Unix, ce cloisonnement est matérialisé par les *processus*.

En réalité, le cloisonnement est obtenu grâce aux *espaces d'adressage* : les applications peuvent fonctionner avec des adresses virtuelles identiques, mais ces adresses virtuelles ne se recouvrent pas en mémoire physique. C'est ce que nous présenterons en partie 1.1. De plus, le code de ces applications n'est pas exécuté avec les privilèges du noyau, mais avec des privilèges moins élevés, par des *threads utilisateur* : nous détaillerons cela en partie 1.2. Enfin, bien que les applications utilisateur soient séparées du noyau, celles-ci ont besoin de faire appel à ce dernier pour différents services (allocation de mémoire, accès aux périphériques, etc.). Il existe pour cela un mécanisme spécifique, l'*appel système*, que nous introduirons en partie 1.2.2.

La suite s'intéressera aux caractéristiques spécifiques de l'architecture `x86` concernant ces notions (section 2). En partie 3, nous présenterons l'implémentation bas niveau de ces mécanismes dans *SOS*. La partie 4 suivante traitera des aspects plus haut niveau, à savoir l'implémentation des threads utilisateur et des processus. Enfin, nous terminerons (partie 5) en montrant comment sont chargées et exécutées nos premières applications utilisateur au format ELF ! En annexe nous vous proposons de découvrir le débogage avec *Qemu* et *gdb*.

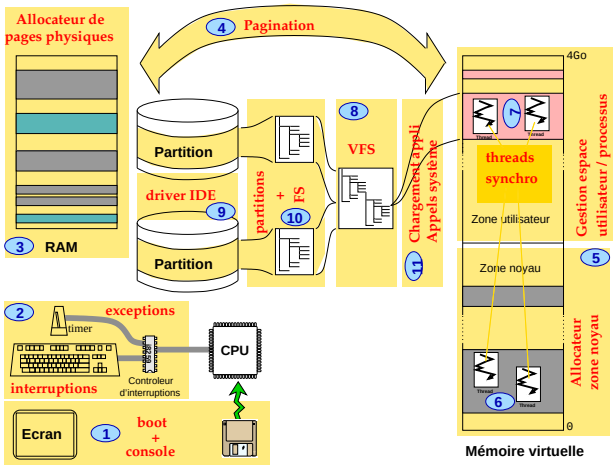

FIG. 1 – Programme des articles

1 Généralités

1.1 Espaces d'adressage

Dans l'article 4 concernant la mise en place de la pagination, nous avons évoqué la *multi-programmation*. Celle-ci consiste à exécuter plusieurs applications sur le même processeur. Nous avions rapidement présenté ses avantages en terme de cloisonnement et de partage de mémoire. Nous proposons maintenant d'étudier en détail le fonctionnement et la mise en place de cette *multi-programmation*.

1.1.1 Définition

L'unité de gestion de la mémoire (la *MMU*, voir l'article 4) effectue une traduction des adresses virtuelles vers les adresses physiques en fonction de tables de traduction fournies par le système d'exploitation. Dans SOS, un premier jeu de tables de traduction a été mis en place dans l'article 4 et permet au noyau de fonctionner au travers de la *pagination* prise en charge par la *MMU*.

On peut considérer qu'un jeu de tables de traduction d'adresses correspond à une fonction mathématique $f : V \rightarrow P$ où V est l'ensemble des adresses virtuelles et P l'ensemble des adresses physiques. Une fonction f donnée traduit certains éléments de V en certains éléments de P.

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 69 – Mars 2005 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

Rien n'empêche de définir un autre jeu de tables de traduction, c'est-à-dire une autre fonction g qui traduira certains éléments de V en certains éléments de P, d'une manière éventuellement différente de f. Ainsi, P vu au travers de f^{-1} donnera une certaine portion de V, qu'on appelle *espace d'adressage*. P vu au travers de g^{-1} donnera une autre portion de V, c'est-à-dire un autre *espace d'adressage*.

De la même manière qu'il a été possible (article 6) de passer d'un contexte d'exécution à un autre, il est possible de passer d'un espace d'adressage à un autre. Il suffit pour cela de configurer la MMU avec d'autres tables de traduction d'adresses. On réalise ainsi un *changement d'espace d'adressage*.

1.1.2 Utilisation : cloisonnement mémoire

Une solution pour que deux applications n'interfèrent pas est de les cloisonner en mémoire physique. C'est-à-dire qu'on fera en sorte qu'elles n'aient aucune adresse en mémoire physique en commun. Dans notre exemple précédent on s'arrangera donc pour que $f(V) \cap g(V) = \emptyset$.

Par exemple, dans la figure 2, le jeu de tables de traduction f définit un espace d'adressage A. Dans cet espace d'adressage A, trois pages virtuelles sont associées à trois pages physiques. Le jeu de tables de traduction g définit un autre espace d'adressage B comportant également trois pages virtuelles, mais qui sont associées à trois pages physiques différentes de celles de f. Ainsi tout code qui s'exécute dans A ne peut accéder qu'aux pages physiques jaunes, et tout code qui s'exécute dans B ne peut accéder qu'aux pages physiques vertes.

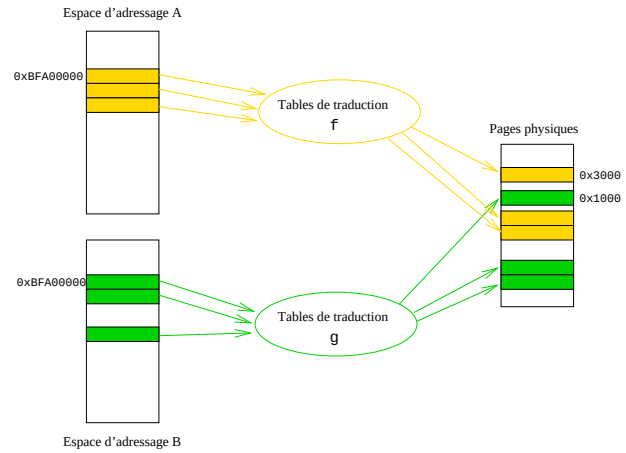

FIG. 2 – Deux espaces d'adressage A et B cloisonnés. Le code s'exécutant dans A n'a accès qu'aux pages jaunes, et le code s'exécutant dans B qu'aux pages vertes.

On s'aperçoit qu'en définissant un jeu de tables de traduction, et donc un *espace d'adressage* pour chaque application en exécution dans le système, on peut cloisonner celles-ci de manière à ce qu'elles n'interfèrent pas en mémoire physique. Il suffira d'accompagner les changements de contexte, du changement d'espace d'adressage correspondant. Une application fonctionnant dans l'espace d'adressage A ne pourra pas modifier la mémoire d'une application fonctionnant dans l'espace d'adressage B, même si ces deux applications manipulent les mêmes adresses virtuelles. Les *processus* Unix ont cette propriété.

1.1.3 Utilisation : Partage de mémoire

En plus de permettre le cloisonnement des applications, les *espaces d'adressage* permettent le partage de mémoire entre applications. En effet, il est possible de définir dans plusieurs *espaces d'adressage* des traductions *virtuel* $\rightarrow$ *physique* qui pointent vers la même page physique (auquel cas $f(V) \cap g(V) \neq \emptyset$). Ces espaces d'adressage peuvent alors accéder à la même zone de mémoire physique, qui est dite *partagée*, et ils peuvent donc s'échanger des informations. Par exemple dans la figure 3, la page physique violette est visible dans les deux espaces d'adressage A et B.

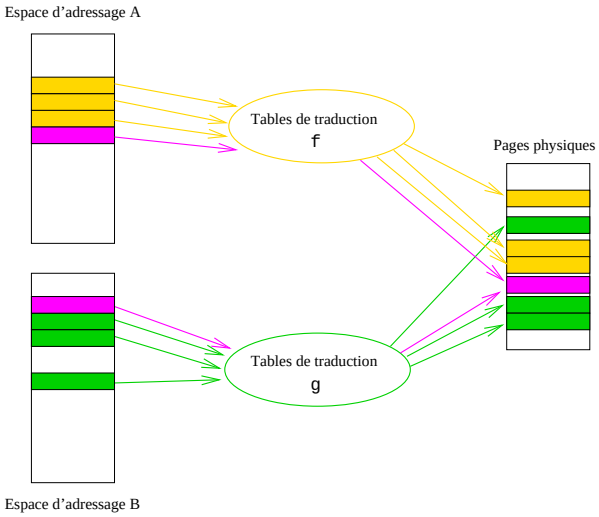

FIG. 3 – Partage de mémoire entre les espaces d'adressages A et B. A et B restent cloisonnés, sauf au niveau de la page violette dont le contenu est partagé et visible depuis les deux espaces d'adressage.

Cette possibilité de partage de mémoire est abondamment utilisée dans Unix : elle permet de partager le code des applications ou des bibliothèques partagées. On la retrouve également au cœur de primitives aussi essentielles que `exec(3)` ou `mmap(2)`. Ainsi, même si plusieurs instances d'une même application (par exemple `mozilla-firefox`) sont lancées dans différents processus, le code de cette application ne sera présent qu'une seule fois en mémoire physique. Les primitives de communication inter-processus de type *shared memory segment* (fonctions `shmget`, `shmat`, `shmctl`, etc.) utilisent également ce mécanisme.

1.1.4 Espace noyau et espace utilisateur

En général, chaque *espace d'adressage* est séparé en deux parties : une pour le noyau et une pour l'application utilisateur en exécution dans cet espace d'adressage. En effet, pour qu'une application puisse faire appel aux services du noyau au travers d'un *appel système* (voir la section 1.2.2), il est préférable que ce noyau soit directement accessible dans l'espace d'adressage courant. Il serait possible d'avoir un espace d'adressage spécifique pour le noyau, mais cela obligerait à effectuer un changement de contexte vers cet espace d'adressage à chaque *appel système*. De plus, le noyau peut avoir besoin d'accéder à certaines données de l'application réalisant un *appel système* : il est donc plus simple qu'ils se situent dans le même espace d'adressage.

La définition qu'on a faite de "noyau" d'OS (gestionnaire central de toutes les ressources) n'a alors de sens que si la

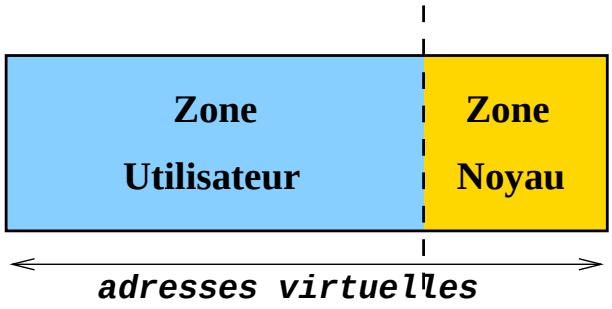

FIG. 4 – Découpage d'un espace d'adressage en deux zones : zone *utilisateur* et zone *noyau*.

partie noyau de tous les espaces d'adressage est identique. Sinon on n'aurait pas "un" noyau dans le système, mais plusieurs (certains OS fonctionnent sur ce principe).

Pour cela, plusieurs solutions existent. Par exemple, dans le noyau Linux, la partie noyau est une représentation à l'identique (*identity mapping*), moyennant une translation, de la mémoire physique (figure 5). On est alors assuré que, quoi que fasse le noyau avec sa mémoire, les parties noyau de toutes les tables de traduction seront toujours identiques et n'auront jamais à être modifiées. Bien entendu, cette représentation à l'identique est limitée par la taille de la partie noyau de l'espace d'adressage, à savoir 1 Go sous Linux. Avec les mécanismes du noyau Linux qui travaillent sur des adresses noyau purement virtuelles (`vmalloc`), le noyau est obligé de changer temporairement d'espace d'adressage.

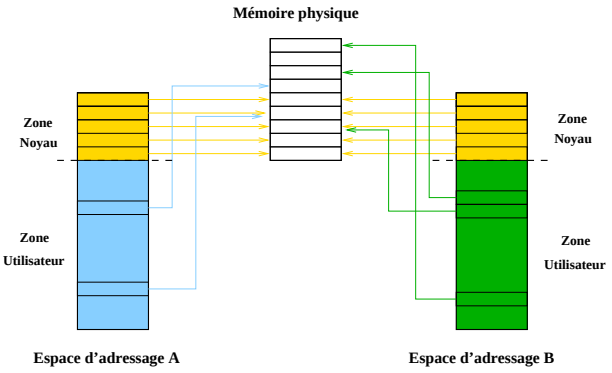

FIG. 5 – **Identity mapping** : La zone noyau des deux espaces d'adressage A et B est une représentation à l'identique (*identity mapping*) d'une partie de la mémoire physique. Les deux espaces d'adressage ont ainsi la même vision du noyau. Les pages virtuelles des zones utilisateur correspondent soit à des pages dans l'identity mapping, soit à des pages hors identity mapping.

Dans SOS, nous avons opté pour une solution différente : la partie noyau n'est pas une représentation à l'identique de la mémoire physique. Il n'empêche qu'on veut que tous les espaces d'adressage aient toujours la même vision de la partie noyau (figure 6). Il faut donc *synchroniser* les tables de traduction des différents espaces d'adressage. Cela signifie qu'à chaque fois qu'on modifiera la traduction d'une adresse virtuelle dans la partie noyau d'un espace d'adressage, on reportera cette modification dans tous les autres espaces d'adressage.

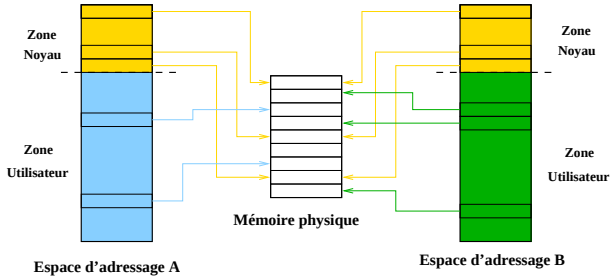

FIG. 6 – **Synchronisation** : La zone noyau des espaces d'adressage A et B représentent la même vision de la mémoire physique, mais ne sont pas un *identity mapping* de celle-ci. Les zones utilisateur, elles, diffèrent entre l'espace A et l'espace B

1.2 Threads utilisateur

Un *thread utilisateur* fonctionne de manière similaire à un *thread noyau* : c'est un contexte d'exécution élu de temps en temps par l'ordonnanceur. Les différences majeures entre un *thread noyau* et un *thread utilisateur* sont les suivantes :

- un thread utilisateur est associé à un espace d'adressage donné, défini à la création du thread. Ceci permet le cloisonnement mémoire vis à vis des threads des autres applications ;
- un thread utilisateur possède un "niveau de privilège" moins élevé que celui d'un thread noyau (voir la section suivante). Ceci interdit à une application de modifier l'état du noyau ou du système dans son ensemble.

1.2.1 Niveaux de privilège

Le processeur propose en général plusieurs niveaux de privilège, au minimum deux : un qui sera utilisé pour l'exécution du noyau (privilège dit "superviseur"), et l'autre qui sera utilisé pour l'exécution des applications utilisateur (privilège dit "utilisateur"). Ces notions de privilèges matériels superviseur/utilisateur n'ont rien à voir avec les droits "administratifs" qu'on trouve sous Unix : "super-utilisateur" (i.e. utilisateur `root`) / utilisateur normal.

Impact sur les accès mémoire. Pour **chaque** accès à une donnée en mémoire virtuelle, la MMU vérifie que le thread courant possède le niveau de privilège suffisant pour accéder à l'adresse souhaitée. Les niveaux de privilège requis pour accéder aux différentes adresses virtuelles sont consignés dans les tables de traduction d'adresses de l'espace d'adressage courant.

Quand un thread possède les privilèges du noyau, il peut modifier toutes les données situées à toutes les adresses virtuelles valides, i.e. pour lesquelles il existe une traduction vers une adresse en mémoire physique.

En revanche, un thread qui possède les privilèges utilisateur ne peut accéder qu'aux adresses virtuelles autorisées aux threads utilisateur (voir figure 7). Quand un thread ne dispose pas des privilèges suffisants pour accéder à telle adresse virtuelle, une exception est levée et c'est alors le noyau qui prend la main pour définir les mesures à prendre (tuer le thread fautif, modifier l'espace d'adressage, etc..).

Le mécanisme des niveaux de privilège permet donc de protéger le noyau des défaillances des applications utilisateur, même si la partie noyau réside dans le même espace

d'adressage que l'application.

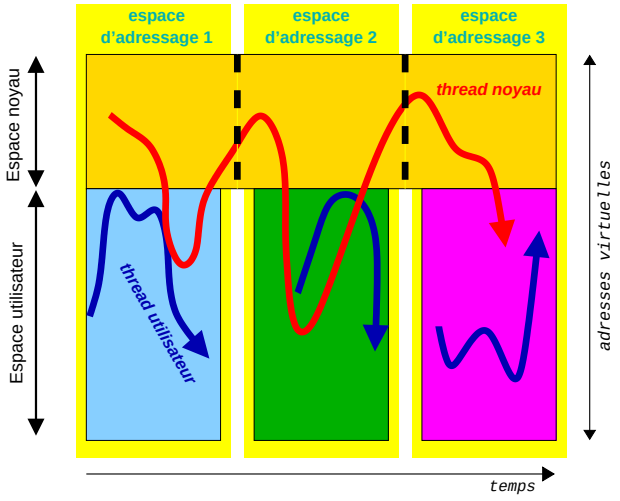

FIG. 7 – Chronogramme des accès mémoire par 4 threads lorsque 3 applications différentes sont successivement exécutées : 1 thread noyau qui peut accéder à toutes les adresses virtuelles de tous les espaces d'adressage, 3 threads utilisateur qui sont confinés à la partie utilisateur de leur espace d'adressage.

Impact sur l'exécution d'instructions. Le code exécuté avec les privilèges du noyau peut utiliser toutes les instructions disponibles sur le processeur. Par exemple, il pourra modifier le comportement du processeur vis à vis des interruptions, changer d'espace d'adressage, passer à un niveau de privilège inférieur sans contrôle, etc..

En revanche, le code exécuté avec des privilèges utilisateur se verra refuser l'exécution de certaines instructions. Par exemple, il sera interdit de changer d'espace d'adressage, de modifier le comportement du processeur vis à vis des interruptions, etc.. et de changer de niveau de privilège de façon incontrôlée ! Quand un thread tente d'exécuter une instruction pour laquelle il ne dispose pas des privilèges suffisants, une exception est levée et c'est alors le noyau qui prend la main pour définir les mesures à prendre (en général : tuer le thread fautif).

1.2.2 Appel système

Le rôle du système d'exploitation est de gérer les ressources matérielles et logiques du système et de les mettre à disposition des applications. Ces dernières doivent donc pouvoir faire appel au noyau pour utiliser les ressources.

Puisque le code des threads *utilisateur* s'exécute avec des privilèges limités, il n'est pas possible de faire directement appel au noyau à l'aide d'un simple appel de fonction. Il est nécessaire de "repasser en mode noyau", avec tous les privilèges de celui-ci, pour réaliser l'exécution du *service* demandé (allocation de mémoire, accès aux périphériques, etc.). Un mécanisme **contrôlé** et totalement défini par le système d'exploitation, permet cette augmentation de privilèges : l'*appel système*. Pour le traitement d'un appel système, on dira que le thread utilisateur passe "en mode noyau" momentanément avant de revenir en "mode utilisateur"

Sur la plupart des architectures, les *appels système* sont implémentés à l'aide d'interruptions logicielles. Celles-ci, contrairement aux exceptions et aux IRQs, sont déclenchées de manière explicite par le code en exécution à l'aide d'une instruction dédiée.

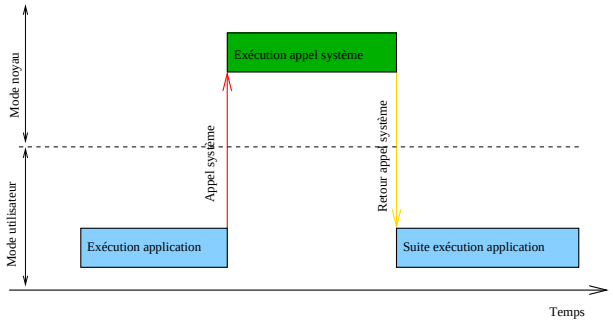

FIG. 8 – Passage en mode noyau lors d'un *appel système* effectué par un thread *utilisateur*, puis retour en mode utilisateur après exécution de l'*appel système*.

1.2.3 Contexte et piles

Chaque thread *noyau* possède une pile. Celle-ci est utilisée pour stocker les variables locales, les paramètres, les valeurs de retour et les adresses de retour des différentes fonctions et sous-fonctions exécutées par le thread (voir l'article 6).

Tout comme un thread noyau, un thread *utilisateur* a besoin d'une pile. Cette pile sera utilisée pour exécuter le code du thread lui-même, c'est-à-dire avec des privilèges *utilisateur*, limités. Elle sera donc située dans la zone *utilisateur* de l'espace d'adressage. On parlera de *pile utilisateur*.

Toutefois, un thread *utilisateur* peut faire appel aux services du noyau *via* des appels système. L'exécution du code de ces services s'effectue alors avec les privilèges noyau. Malheureusement la pile utilisateur n'est pas satisfaisante pour cela : d'une part elle n'est pas très fiable (elle n'est pas tenue d'être toujours présente en mémoire physique) et d'autre part elle risquerait d'être modifiée par les autres threads utilisateur s'exécutant dans le même espace d'adressage. L'appel système sera donc traité sur une pile différente, plus fiable (allouée de façon sûre) et dans la partie noyau de l'espace d'adressage : une *pile noyau*. Le plus souvent, chaque thread utilisateur possédera par conséquent une deuxième pile : *sa* propre pile *noyau*.

Cette pile noyau sera également utilisée en cas d'interruption (IRQ, exception) car la routine de traitement associée doit (en général) s'exécuter avec les privilèges noyau. Ainsi, en cas d'interruption d'un thread utilisateur, il y aura changement implicite de niveau de privilège.

1.3 Synthèse

Pour résumer, un thread noyau :
- s'exécute avec les privilèges du noyau, donc a le droit de tout faire, même d'agir sur toutes les applications ;
- n'est pas lié à un espace d'adressage particulier puisqu'il n'accède qu'à la partie noyau qui est commune à tous les espaces d'adressage.

Et un thread utilisateur :
- s'exécute avec les privilèges utilisateur. Grâce à la protection par niveaux de privilège, il n'a le droit d'agir que sur la partie utilisateur de son espace d'adressage et ne

peut pas exécuter des instructions qui auraient un impact sur tout le système ;
– est lié à un espace d'adressage particulier. C'est le noyau seul qui peut gérer les tables de traduction d'adresses des différents espaces d'adressage. Il est possible de réaliser le cloisonnement mémoire ou le partage contrôlé de mémoire entre applications ;
– peut faire appel à des services du noyau au travers d'un changement de privilège contrôlé : les appels système ;
– possède une pile utilisateur en zone utilisateur pour l'exécution du code lorsque le thread possède les privilèges utilisateur, et utilise une autre pile système en zone noyau quand il passe "en mode noyau" pour le traitement des appels système ou des interruptions. En général chaque thread utilisateur possède sa propre pile noyau pour cela.

Toutes ces règles établissent la *protection* du système. Chaque application s'exécute ainsi dans une "machine virtuelle" : tout ce qu'une application peut faire est contrôlé afin qu'elle ne puisse agir que sur elle seule. L'objectif est que **1)** les autres applications ne pourront pas être influencées par son comportement de façon incontrôlée (hors partage de mémoire physique et synchronisations), **2)** le noyau est isolé des applications et l'invocation des services du noyau par les applications (appels système) est totalement contrôlée.

2 Cas d'étude : l'architecture `x86`

Dans cette partie, nous revenons sur le support offert par l'architecture `x86` pour les 3 éléments qui nous intéressent dans cet article : la notion de privilège, de changement de privilège (pour les appels système) et de changement d'espace d'adressage.

2.1 Niveaux de privilège

L'architecture *x86* propose 4 niveaux de privilège (voir [1, Partie 4.5]) allant de 0 (le plus privilégié) à 3 (le moins privilégié). Dans la suite, quand nous dirons "privilège inférieur", cela signifiera donc "numériquement supérieur". À un instant donné, le niveau de privilège avec lequel le code s'exécute est déterminé par le segment de la GDT (voir l'article 2) pointé par le registre de segment `cs` ("Code Segment", voir l'article 2). Ce niveau de privilège s'appelle le "`CPL`", pour *Current Privilege Level*.

Le niveau de privilège 0 représente le niveau de privilège "superviseur", les autres niveaux représentent des niveaux de privilège "utilisateur". Assez souvent, on se contente d'utiliser seulement les niveaux de privilège 0 (pour le noyau) et 3 (pour les applications). C'est le cas dans Linux et dans SOS par exemple.

2.1.1 Protection des instructions

La plupart des instructions du processeur sont autorisées pour tous les niveaux de privilège. Mais certaines instructions sont réservées au code s'exécutant avec le niveau de privilège superviseur et sont interdites aux autres niveaux de privilège [2]. En cas de violation de cette règle, le plus souvent l'exception *General Protection Fault* est levée par le processeur.

Il en va ainsi par exemple des instructions `cli/sti` pour masquer/démasquer les interruptions matérielles, des instructions modifiant le registre `cr3` déterminant l'espace d'adressage courant, des instructions pour manipuler la *GDT*, l'*IDT* ou d'autres informations vitales au fonctionnement du système (voir [1, Partie 4.9]).

2.1.2 Protection mémoire

Pour ce qui est des accès mémoire, la règle est plus fine et la protection s'effectue à deux niveaux : segmentation et pagination.

Segmentation. Au niveau segmentation, la MMU vérifie pour chaque accès mémoire **1)** que le niveau de privilège souhaité pour accéder à tel segment (le `RPL`, *Requested Privilege Level*, défini dans les registres de segments) est compatible avec le `CPL` et **2)** que ce `RPL` est compatible avec le niveau de privilège déclaré dans la GDT ou la LDT (le `DPL`, *Descriptor Privilege Level*). La MMU vérifiera alors que (voir [1, section 4.6]) : $CPL \leq RPL \leq DPL$ (numériquement). En schématisant, on ne pourra accéder aux segments que si on possède un niveau de privilège numériquement inférieur ou égal à celui du segment. S'il y a violation de ces règles, la MMU lève l'exception processeur "*General Protection Fault*".

Dans SOS, la segmentation n'est quasiment pas utilisée car on adopte le modèle *flat* dans lequel les segments sont identiques et couvrent tout l'espace 0-4Go. Cependant on sera amené à définir au moins un segment de code supplémentaire avec un niveau de privilège 3. Ce segment permettra aux applications de s'exécuter avec un niveau de privilège (`CPL`) 3.

Dans SOS, ce n'est pas *via* la segmentation que la protection mémoire s'effectuera.

Pagination. Supposons maintenant que la pagination est activée. Pour chaque entrée dans le répertoire de pages (*PDE*) et chaque entrée dans les tables de pages (*PTE*), le bit *U/S* (voir l'article 4) indique si la table de pages / la page associée est accessible avec un niveau de privilège utilisateur. Lorsque le processeur exécute du code avec un niveau de privilège utilisateur (i.e. $CPL > 0$), il ne peut donc accéder qu'aux pages de la mémoire de niveau utilisateur, c'est-à-dire celles dont le bit *U/S* est à 1 à la fois dans l'entrée de répertoire de pages et dans l'entrée de table de pages correspondante. S'il y a violation de cette règle, la MMU lève l'exception processeur "*Page Fault*".

Autrement dit, on pourra réserver certaines pages pour le noyau (interdites aux applications) et d'autres qui seront accessibles par les applications. Dans SOS, le mécanisme de cloisonnement du noyau vis à vis des applications reposera sur ce principe.

2.2 Changement de privilège

En vertu de ce qu'on a dit précédemment, un thread avec un niveau de privilège supérieur peut utiliser les instructions et les données accessibles aux niveaux de privilège inférieurs. En revanche, pour qu'un thread de niveau de privilège inférieur puisse atteindre les données réservées aux niveaux de privilège supérieurs ou utiliser des instructions superviseur, il faut passer par des mécanismes étroitement contrôlés conjointement par le processeur et

l'OS. Ceux-ci s'apparentent à des appels de fonctions accompagnés d'une augmentation de privilège. Une telle fonction est appelée "appel système", elle constitue une passerelle depuis les applications vers le système d'exploitation. La liste de ces "fonctions" qui peuvent être ainsi appelées est définie par le noyau.

Comme à l'habitude dans l'architecture `x86`, il y a plusieurs mécanismes différents pour faire des choses semblables.

Call Gates. Les *Call Gates* [1, section 4.8.3] (les ingénieurs Intel ont beaucoup d'humour) sont des segments "système" particuliers. Si on fait un "saut" (i.e. presque un appel de fonction) vers de tels segments, ce saut est accompagné d'un changement de privilège. Il est aussi accompagné d'un changement de contexte complet avec sauvegarde et restauration de l'état processeur dans d'autres segments particuliers : les *TSS*, ou *Task State Segment*s [1, section 6.2.1] (voir la figure 9).

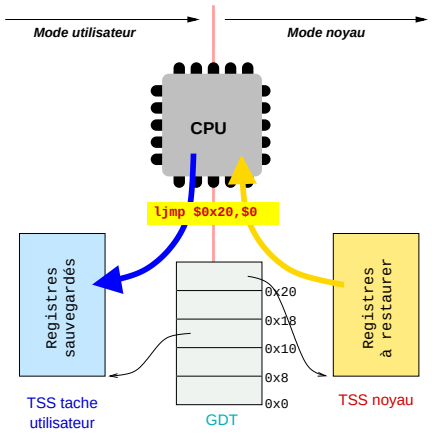

FIG. 9 – Changement de privilège à l'aide d'un *call gate*.

Un *TSS* n'est pas un segment comme les autres segments de la *GDT* : il n'est pas utile, dans le cadre de la segmentation, pour accéder à certaines zones de la mémoire physique. Il contient des informations sur le système. La documentation Intel différencie ces deux types de segments sous les dénominations *Code and data segments* (pour les segments tels que `SOS_SEG_KCODE` ou `SOS_SEG_UDATA`) et *System segments* (pour les segments de type *TSS* par exemple).

Comme les *TSS* doivent être définis dans la *GDT*, on est limité à 8191 *TSS* donc on ne peut stocker les états "que" de 8191 threads. De plus, la gestion des slots libres et occupés de la *GDT* doit être réalisée.

Cette solution est souvent qualifiée de "*hardware task-switching*" car c'est le processeur qui s'occupe de tout, y compris de la sauvegarde/restauration de l'état processeur. Ce mécanisme est très spécifique à l'architecture `x86`.

Interruptions. Une autre technique plus classique repose sur l'utilisation des interruptions. En effet, sur toutes les architectures de notre connaissance, une interruption (IRQ, exception, interruption logicielle) doit pouvoir être accompagnée d'un changement de privilège pour être prise en charge par le noyau.

L'architecture `x86` n'échappe pas à cette règle : toutes les interruptions référencées dans l'IDT (voir l'article 2) seront exécutées en mode superviseur. L'appel de ces routines se fera en général de manière matérielle (IRQs, exception). Il peut également se faire de manière logicielle (interruptions logicielles) par l'instruction `int` [2]. Dans ce cas on peut indiquer dans l'IDT quels niveaux de privilège ont le droit de faire l'interruption logicielle.

Comme avec les *Call Gates*, le changement de privilège utilise un *TSS*, mais cette fois-ci seule une infime partie de celui-ci est utilisée (le reste peut demeurer à 0). Elle sert à définir l'adresse de la pile noyau sur laquelle on doit passer pour faire le traitement de l'interruption (voir la figure 10). Aucun autre *TSS* n'est nécessaire pour stocker l'état du contexte interrompu. Il s'agira seulement de toujours tenir ce TSS à jour pour refléter la bonne adresse de pile noyau. Dans SOS, seul le noyau pourra s'en charger, étant donnée la valeur du *RPL* (voir la section 2.1.2) de ce *TSS*.

Cette fois-ci, la sauvegarde/restauration du contexte interrompu est à la charge du système d'exploitation, c'est pourquoi on la qualifie souvent de "*software task-switching*".

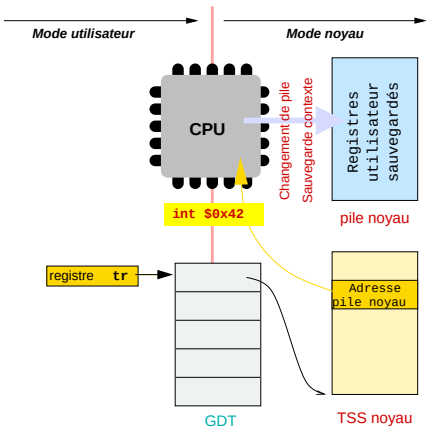

FIG. 10 – Changement de privilège à l'aide d'une interruption logicielle.

C'est cette dernière solution que nous avons adoptée dans SOS.

2.3 Changement d'espace d'adressage

Sur `x86`, l'espace d'adressage courant est complètement déterminé par la configuration de la segmentation (`GDT` et `LDT`) et par la configuration de la pagination si celle-ci est activée.

En changeant de `GDT` ou de `LDT`, on peut changer la configuration de l'espace d'adressage. Dans le cas de SOS, nous avons fixé la `GDT`, celle-ci ne change jamais.

Dans SOS (comme dans Linux), nous avons préféré nous appuyer sur la pagination pour manipuler les espaces d'adressage. Comme nous l'avons vu dans l'article 4, sur processeur x86 les tables de traduction pour la pagination fonctionnent avec deux niveaux d'indirection : un *répertoire de pages* et des *tables de pages*. L'adresse **physique** du *répertoire de pages* à utiliser est à tout moment présente dans le registre `cr3`. Puisqu'un répertoire de pages contient les références vers les tables de pages, il suffit, pour définir un espace d'adressage, de retenir l'adresse du répertoire de pages. Ainsi, pour passer d'un espace d'adressage à un autre, il suffira "simplement" de faire pointer `cr3` sur un autre répertoire de pages.

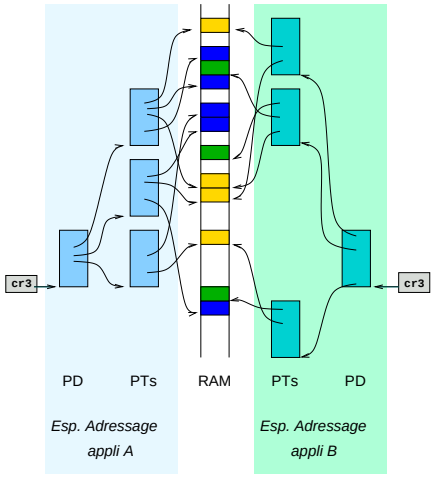

FIG. 11 – Changement d'espace d'adressage : il suffit de modifier l'adresse du PD contenue dans le registre `cr3`.

3 Implémentation dans *SOS* : aspects bas niveau

3.1 Introduction

Dans SOS, nous avons fait les choix suivants :

- Chaque thread utilisateur possède deux piles : une pile utilisateur quand il s'exécute en mode utilisateur et une pile noyau pour les appels système et pour le traitement des IRQs et des exceptions ;
- Chaque thread utilisateur est associé dès sa naissance à un unique espace d'adressage ;
- Les appels système sont réalisés par des interruptions logicielles ;
- Tous les espaces d'adressage sont parfaitement identiques en ce qui concerne la partie noyau. Un mécanisme de maintien de cette *synchronisation* est nécessaire pour cela.

Pour les threads utilisateur en mode noyau et pour les threads noyau, nous avons fait les choix suivants :

- Par défaut, les accès mémoire sont limités à la partie noyau de l'espace d'adressage courant ;
- Quand on passe d'un thread de ce type à un autre thread du même type (i.e. thread utilisateur en mode noyau ou thread noyau), on n'a donc pas besoin de changer d'espace d'adressage puisque les parties noyau sont toutes identiques ;
- Un thread de ce type peut cependant demander à accéder temporairement à la partie utilisateur d'un espace d'adressage précis au travers de la fonction `sos_thread_change_mm_context()` (voir la section 4.3.3).

Dans la suite de la présente section, nous détaillons comment SOS utilise les mécanismes bas niveau que nous avons évoqués dans la section 2 précédente. Nous verrons dans la section qui suivra comment SOS implémente les deux notions de plus haut niveau d'abstraction : les threads utilisateur et les processus. La figure 12 résume l'articulation de tous les sous-systèmes impliqués.

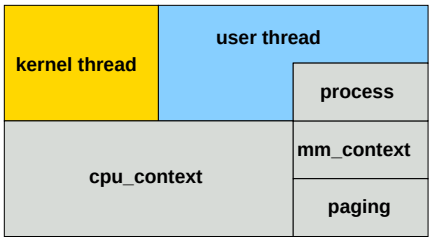

FIG. 12 – Les éléments que nous abordons ici et leur organisation les unes par rapport aux autres.

3.2 Niveaux de privilège

Conformément à ce que nous avons dit en section 2.1, sur `x86` la notion de niveau de privilège est étroitement liée à celle de segment. Jusqu'à l'article 7, la GDT, définie dans le fichier `hwcore/gdt.c`, comptait trois segments :

- Le segment *nul*, obligatoire ;
- Un segment pour le code, avec un niveau de privilège 0, sur lequel pointe le registre de segment `cs` ;
- Un segment pour les données, avec un niveau de privilège 0, sur lequel pointent les registres de segment `ds`, `es`, `fs` et `gs`.

Puisque tout le code du noyau est exécuté avec `cs` pointant sur un segment dont le niveau de privilège est 0, le noyau s'exécute avec les privilèges superviseur (qui l'eut cru ?).

L'exécution des *threads utilisateur* doit s'effectuer dans un niveau de privilège inférieur : nous avons choisi le niveau 3. Nous n'utiliserons pas les niveaux de privilège 1 et 2.

Pour permettre l'exécution de code au niveau de privilège 3, il faut donc définir au moins un segment de privilège 3 destiné à être le `cs` des threads utilisateur. En fait, deux nouveaux segments sont ajoutés dans la GDT. Ces segments ont été insérés dans le tableau `x86_segment_descriptor_gdt` du fichier `hwcore/gdt.c` :

```
[SOS_SEG_UCODE] = BUILD_GDTE(3, 1),
[SOS_SEG_UDATA] = BUILD_GDTE(3, 0),
```

Ces deux segments seront utilisés quand le processeur exécutera un *thread utilisateur*. Le premier est le segment de code, sur lequel `cs` pointera. C'est précisément parce que ce segment possède un niveau de privilège 3 que le processeur passera dans ce niveau de privilège, et appliquera les restrictions correspondantes (voir la section 2.1). Le deuxième segment sera celui pointé par les registres `ds`, `es`, `fs` et `gs` : il sera donc utilisé pour l'accès aux données. Ce dernier segment n'est pas vraiment obligatoire, il aurait pu être fusionné avec le segment de données utilisé par le noyau. Dans SOS en effet, les réelles restrictions d'accès aux données sont effectuées par la pagination.

3.3 Changement de privilège

Dans SOS, une application changera de privilège pour les appels système *via* une interruption logicielle (voir la section 2.2).

3.3.1 Premier aspect : le gestionnaire d'interruption

Sous Linux, l'interruption utilisée pour les appels système est l'interruption `0x80`. Sous *SOS*, nous avons choisi

SOS_SWINTR_SOS_SYSCALL, c'est-à-dire 0x42[1].

La fonction sos_swintr_subsystem_setup() du fichier hwcore/swintr.c se charge d'enregistrer le gestionnaire de bas niveau sos_syscall_wrapper() dans la table des interruptions. On notera que ce gestionnaire est enregistré dans l'*IDT* avec un lowest_priviledge de 3, ce qui permet aux threads utilisateur d'appeler cette interruption. Toutes les autres interruptions (exceptions et IRQs) ont un lowest_priviledge à 0, ce qui permet d'éviter qu'un thread utilisateur tente d'appeler une de ces interruptions en utilisant l'instruction int. Ce paramètre lowest_priviledge de sos_idt_set_handler() correspond au champ dpl d'un descripteur de l'*IDT* (voir [1, Partie 5.11]).

Le gestionnaire d'interruption de premier niveau, en assembleur, est présent dans le fichier hwcore/swintr_wrappers.S. Il est très similaire aux autres gestionnaires de premier niveau des IRQs et des exceptions (voir l'article 2), sauf qu'il appelle directement la fonction sos_do_syscall() vue plus loin, en section 4.4.1.

3.3.2 Deuxième aspect : localisation de la pile noyau

Lorsque le processeur exécute du code dans un mode non privilégié et qu'une interruption (IRQ, exception ou appel système) survient, il doit récupérer l'adresse de la pile noyau sur laquelle le gestionnaire d'interruption doit s'exécuter (voir la section 2.2). Sur x86, le processeur détermine cette adresse en récupérant les champs ss0 (segment de la pile noyau) et esp0 (adresse de la pile noyau) d'un segment particulier du système : le *TSS* (voir la section 2.2) indiqué par le registre tr (*Task Register*) du processeur.

Le système d'exploitation doit mettre à jour les champs ss0 et esp0 de ce TSS à chaque fois qu'on passe à un thread en mode utilisateur. L'objectif est que l'adresse de la pile noyau indiquée dans le TSS corresponde à l'adresse de la pile noyau du thread utilisateur en exécution. Sur uniprocesseur, comme il n'y a à chaque instant qu'un seul thread (utilisateur ou pas) qui peut s'exécuter, il suffit d'un seul *TSS* dans tout le système. Sur un système multi-processeur, il faudrait un *TSS* par processeur.

L'unique *TSS* de *SOS* est défini avec tous les autres segments de la *GDT*, dans hwcore/gdt.c :

```
static struct x86_segment_descriptor gdt[] = {
  ...
  [SOS_SEG_KERNEL_TSS] = { 0, } /* Initialized by
                                   register_kernel_tss */
}
```

La structure d'un TSS est spécifiée par Intel [1, section 6.2.1] : il s'agit d'une structure de 104 octets (voir sa définition dans hwcore/cpu_context.c), le segment peut éventuellement être plus gros si le programmeur le désire. L'unique *TSS* de SOS est défini dans le fichier hwcore/cpu_context.c et se nomme kernel_tss. La fonction sos_cpu_context_subsystem_setup() l'initialise puis l'enregistre dans l'entrée SOS_SEG_KERNEL_TSS de la *GDT* grâce à la fonction sos_gdt_register_kernel_tss() du fichier hwcore/gdt.c. Cette fonction s'occupe aussi de faire pointer le registre tr du processeur vers cette entrée SOS_SEG_KERNEL_TSS (instruction ltr).

[1]Comme il se doit pour tout informaticien qui se respecte ;-)

C'est la fonction sos_cpu_context_update_kernel_tss() qui est chargée de mettre à jour l'adresse de la pile noyau dans le *TSS* lorsqu'on effectue un changement de contexte vers un thread utilisateur :

```
void
sos_cpu_context_update_kernel_tss(struct sos_cpu_state *next_ctxt)
{
 [...]
     kernel_tss.esp0 = ((sos_vaddr_t)next_ctxt)
                       + sizeof(struct sos_cpu_ustate);
 [...]
}
```

Cette fonction est appelée :
- À chaque changement de contexte vers un thread en mode utilisateur (voir les fonctions de hwcore/cpu_context_switch.S).
- À la fin de chaque traitement d'interruption matérielle, en cas de retour vers un thread utilisateur interrompu (voir les fonctions de hwcore/irq_wrappers.S).
- À la fin de chaque traitement d'exception, si le thread interrompu était en mode utilisateur (voir les fonctions de hwcore/exception_wrappers.S).
- À la fin de chaque traitement d'appel système puisque le thread interrompu était forcément en mode utilisateur (voir hwcore/swintr_wrappers.S).

Ainsi, quand le thread utilisateur en mode utilisateur sera de nouveau interrompu, le processeur connaîtra la bonne adresse de la pile noyau à utiliser.

3.3.3 Troisième aspect : informations empilées par le processeur

Dans le cas où le traitement d'une interruption ne s'accompagne pas d'un changement de privilège, alors le processeur empile automatiquement les registres eflags, cs, eip ainsi qu'un éventuel *code d'erreur* sur la pile noyau. Il passe ensuite à l'exécution du gestionnaire de l'interruption (voir l'article 6 et la figure 13).

Dans le cas où le traitement d'une interruption s'accompagne d'un changement de niveau de privilège, alors le processeur empile des informations supplémentaires avant d'empiler eflags, cs et eip. Il empile en effet l'adresse de la pile utilisateur et le registre de segment de la pile utilisateur : il y a changement de pile vers la pile noyau, il faut donc sauvegarder l'état de la pile utilisateur [1, Figure 5-4].

À la fin du traitement d'une interruption, on utilise l'instruction iret. Dans le cas où on retourne dans le contexte utilisateur d'un thread utilisateur, il y a besoin d'un changement de niveau de privilège. Pour cela, iret s'attend à trouver l'adresse de la pile utilisateur et le sélecteur de segment associé, en plus des informations habituelles détaillées dans l'article 6 (eflags, cs et eip). Le processeur déterminera tout seul qu'il doit utiliser ces informations supplémentaires parce qu'il remarquera que le registre cs à restaurer correspond à un niveau de privilège inférieur.

3.4 Changement d'espace d'adressage

Dans *SOS*, un espace d'adressage est représenté par une structure mm_context définie dans le fichier hwcore/mm_context.c. On peut faire le parallèle entre cette structure et la structure sos_cpu_state. Au lieu de s'occuper des changements de contexte, le sous-système mm_context permet les changements d'espace d'adressage. Et au lieu de faire référence à l'état des registres du

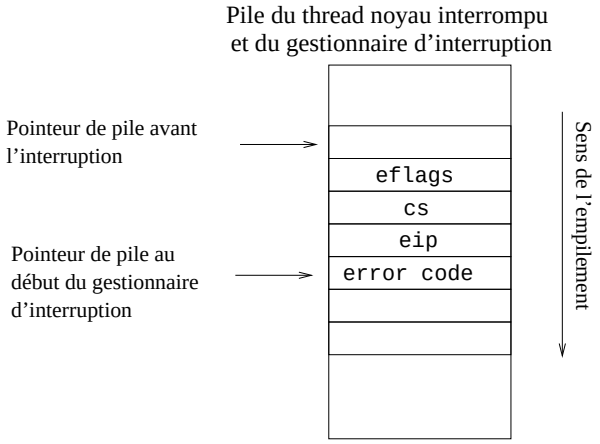

FIG. 13 – Utilisation de la pile lorsqu'il n'y a pas de changement de niveau de privilège. Les quatre informations `eflags`, `cs`, `eip` et éventuellement `error code` sont empilées par le processeur directement sur la pile du thread noyau qui était en cours d'exécution.

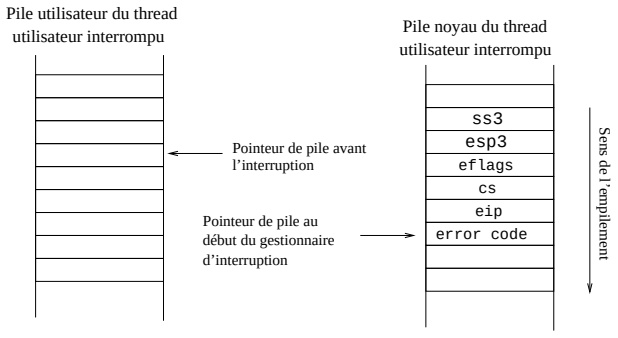

FIG. 14 – Utilisation de la pile lorsqu'il y a changement de niveau de privilège. Les informations sont empilées sur la pile noyau pointée par le TSS, et pas sur la pile utilisateur.

processeur, la structure `mm_context` fait référence à une configuration de la MMU, c'est-à-dire à un répertoire de pages.

Le sous-système `mm_context` est en quelque sorte une "sur-couche" au sous-système `paging`. Alors que `paging` s'occupe de l'espace d'adressage courant, `mm_context` s'occupe de créer de nouveaux espaces d'adressage et de changer d'espace d'adressage courant. Il a aussi pour rôle de maintenir les parties noyau de tous les espaces d'adressage identiques entre elles (voir les sections 1.1.4 et 3.4.3).

3.4.1 Description

La structure `mm_context` contient :
- L'adresse physique du répertoire de pages, `paddr_PD`. C'est cette adresse qui sera chargée dans le registre `cr3` pour passer dans l'espace d'adressage défini par ce répertoire de pages ;
- L'adresse virtuelle du répertoire de pages, `vaddr_PD`. Cette adresse virtuelle permet de synchroniser les parties noyau de tous les espaces d'adressage. Nous y revenons en section 3.4.3 ;
- Un compteur de références `ref_cnt` permettant de savoir si un processus et/ou le processeur utilise(nt) cet espace d'adressage ;
- Des pointeurs `next` et `prev` pour chaîner les `mm_context`.

Grâce aux pointeurs `next` et `prev`, tous les espaces d'adressage sont listés dans `list_mm_context` (fichier `hwcore/mm_context.c`). Cette liste permet au sous-système `mm_context` de connaître tous les espaces d'adressage pour lesquels il doit synchroniser la partie noyau.

3.4.2 Manipulation

Initialisation. La fonction `sos_mm_context_subsystem_setup` se charge d'initialiser le sous-système de gestion des `mm_context`. Tout d'abord, elle créé un *cache* d'objets pour allouer les structures `mm_context`. Ensuite, elle alloue une telle structure destinée à correspondre à l'espace d'adressage initial. En effet, avant d'activer la pagination, le noyau met en place un répertoire de pages et des tables de pages, mais il n'existe pas encore de structure `mm_context` associée. Pour la construire, la fonction récupère l'adresse physique du répertoire de pages courant en lisant le registre `cr3` du processeur (fonction `sos_paging_get_curent_PD_paddr()` de `hwcore/paging.c`). Elle le mappe ensuite en mémoire virtuelle manuellement en utilisant `sos_kmem_vmm_alloc()` puis `sos_paging_map()`. Cela permet à ce `mm_context` initial de respecter les conditions détaillées en section 3.4.3 dans la description du champ `vaddr_PD`.

Création. La fonction `sos_mm_context_create()` créé un nouvel espace d'adressage. Pour cela, elle commence par allouer un répertoire de pages : allocation en mémoire virtuelle d'abord par `sos_kmem_vmm_alloc()` (⇒ champ `vaddr_PD`) puis récupération de l'adresse physique correspondante par `sos_paging_get_paddr()` (⇒ champ `paddr_PD`). Elle initialise ensuite la partie noyau de cet espace d'adressage de manière à la synchroniser avec la partie noyau des autres espaces d'adressage (voir la section

3.4.3). Enfin, elle met en place le mirroring dans ce nouvel espace d'adressage (voir l'article 4) et ajoute la structure `mm_context` à la liste `list_mm_context`.

Changement d'espace d'adressage. La fonction `sos_mm_context_switch_to()` effectue un changement d'espace d'adressage de l'espace courant vers l'espace `mmctxt` passé en paramètre. Pour cela, il suffit de modifier le registre `cr3` du processeur pour qu'il pointe à l'adresse du répertoire de pages de l'espace d'adressage destination. Ceci est effectué par appel à la fonction `sos_paging_set_current_PD_paddr()` de `hwcore/paging.c`. Comme le processeur n'utilise plus l'espace d'adressage source, on peut décrémenter son compteur de références (fonction `sos_mm_context_unref()`). Et comme il utilise maintenant le nouvel espace d'adressage, on doit incrémenter le compteur de références de ce dernier :

```
sos_ret_t sos_mm_context_switch_to(struct sos_mm_context *mmctxt)
{
  if (mmctxt != current_mm_context)
  {
    struct sos_mm_context * prev_mm_context = current_mm_context;

    /* Actually change the address space */
    sos_paging_set_current_PD_paddr(mmctxt->paddr_PD);

    /* Exchange the mm_contexts */
    current_mm_context = mmctxt;

    /* Update the reference counts */
    mmctxt->ref_cnt ++;
    sos_mm_context_unref(prev_mm_context);
  }

  return SOS_OK;
}
```

Pour éviter les changements de contexte inutiles, on utilise la variable `current_mm_context` qui indique l'espace d'adressage actuellement sélectionné dans la MMU. Aucun changement d'espace d'adressage n'est effectué si l'espace d'adressage demandé correspond à `current_mm_context`.

Destruction. La fonction `sos_mm_context_unref()` décrémente le compteur de références d'un espace d'adressage. Quand ce compteur de références devient nul, l'espace d'adressage peut être retiré de la liste `list_mm_context` puis supprimé. Cette fonction est appelée par exemple quand on change d'espace d'adressage (`sos_mm_context_switch_to()`) ou quand un processus est supprimé (voir la partie 4).

3.4.3 Synchronisation de l'espace noyau

Comme nous l'avons vu dans la partie 1.1.4, toutes les applications doivent avoir une vue identique de la partie noyau de leur espace d'adressage. Le mécanisme qui permet cela est ce que nous appelons *synchronisation de l'espace noyau*. Cette synchronisation concerne la partie noyau elle-même, et non pas la partie de chaque espace d'adressage consacrée au *mirroring*. La partie consacrée au *mirroring* est le reflet du répertoire de pages et des tables de pages de l'espace d'adressage courant et reste donc propre à cet espace d'adressage.

On pourrait penser qu'il faut procéder à une synchronisation à chaque fois qu'on mappe/démappe une nouvelle page en partie noyau. En réalité, cela n'est pas nécessaire car les tables de pages (PT) qui sont utilisées pour mapper la partie noyau sont toutes identiques, donc elles peuvent être *partagées* par tous les répertoires de pages (voir figure 15). La synchronisation n'est alors nécessaire que lorsqu'on rajoute ou enlève une table de pages dans la partie noyau, c'est-à-dire lorsqu'on modifie une PDE concernant la partie noyau, ce qui est environ 1000 fois moins fréquent.

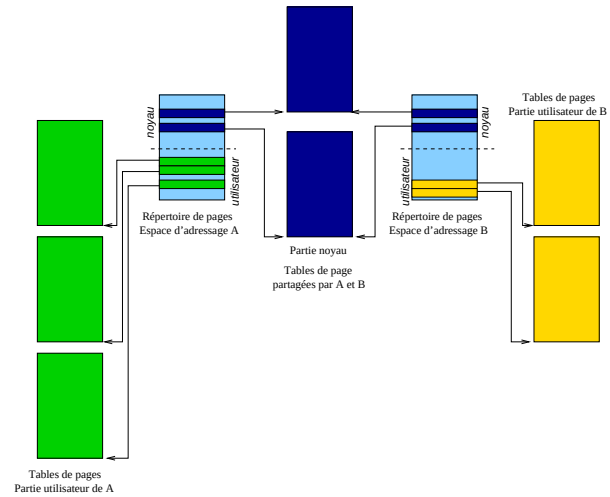

FIG. 15 – Les tables de pages couvrant la partie noyau des espaces d'adressage sont partagées : la synchronisation ne concerne donc que les répertoires de pages.

Les fonctions `sos_paging_map()` et `sos_paging_unmap()` ont donc été modifiées pour que lorsqu'une table de pages est ajoutée ou retirée dans la partie noyau de l'espace d'adressage courant, la fonction `sos_mm_context_synch_kernel_PDE()` soit appelée. Cette dernière parcourt la liste des espaces d'adressage et modifie l'entrée `index_in_pd` de tous les répertoires de pages par la valeur `pde` fournie.

C'est pour cela que tous les répertoires de pages (PD) sont mappés dans la mémoire virtuelle du noyau (champ `vaddr_PD` de la structure `mm_context`). Il faut en effet que `sos_mm_context_synch_kernel_PDE()` puisse accéder à ces répertoires de pages pour les modifier, i.e. qu'ils existent tous en mémoire virtuelle. Rappelons que le processeur ne sait en effet travailler qu'avec des adresses virtuelles.

4 Implémentation dans *SOS* : threads utilisateur et processus

Nous disposons maintenant des outils bas niveau pour gérer les changements de privilège et les changements d'espace d'adressage. Il ne reste plus qu'à les intégrer avec le reste du système, à savoir les threads, qui prennent alors la forme de *threads utilisateur*. Mais d'abord, intéressons-nous à l'entité de plus haut niveau représentant une application utilisateur : le processus.

4.1 *Process*

Dans SOS comme dans Unix, un processus, ou *process*, est le "container" des ressources utilisées par une application. Parmi ces ressources figurent les threads utilisateur formant l'application et l'espace d'adressage dans lequel ils s'exécutent. C'est la définition que nous adoptons pour le présent article. Dans les prochains articles,

parmi les ressources rassemblées dans le processus, figureront les descripteurs de fichiers ouverts et les régions de mémoire virtuelle utilisées dans la partie utilisateur de l'espace d'adressage.

4.1.1 Description

Pour l'instant, un *process* est donc composé d'un espace d'adressage, c'est-à-dire une référence vers une structure de type `mm_context`, et d'une liste de threads. Ces informations sont regroupées dans la structure `sos_process` définie dans `sos/process.c`. Un nom est également associé à chaque *process*, et un compteur `ref_cnt` permet de connaître le nombre de références au *process*.

4.1.2 Utilisation

Un *process* "vide", c'est-à-dire avec un espace d'adressage mais sans *thread utilisateur*, est créé par la fonction `sos_process_create_empty()`. Les fonctions `sos_process_ref()` et `sos_process_unref()` permettent respectivement d'incrémenter et de décrémenter le compteur de références d'un *process*. La dernière fonction a également pour effet de supprimer automatiquement le processus lorsque le compteur de références atteint 0, c'est-à-dire lorsque le *process* n'est plus utile.

Les fonctions `sos_process_register_thread()` et `sos_process_unregister_thread()` permettent respectivement d'ajouter et de supprimer un *thread* donné de la liste des threads d'un *process*. Elles ont également pour effet d'incrémenter et de décrémenter le compteur de références du *process* en utilisant les fonctions suscitées. Ainsi, le *process* ne sera pas supprimé tant qu'il contiendra au moins un *thread*.

4.2 Contexte d'un thread utilisateur

Comme pour les threads noyau, un thread utilisateur correspond d'abord à un *contexte d'exécution*.

4.2.1 Définition

Tout comme le contexte d'un thread *noyau*, le contexte d'un thread *utilisateur* sera sauvegardé sur la pile. Plus précisément, il sera sauvegardé sur la pile *noyau* du thread utilisateur : sa sauvegarde est en effet la conséquence d'une interruption (IRQ, exception, appel système). L'adresse de cette pile est indiquée par le *TSS* noyau. Le noyau s'occupe de la mettre à jour pour qu'elle corresponde à l'adresse de la pile noyau du thread utilisateur (voir la section 3.3.2).

Dans *SOS*, le contexte d'un thread utilisateur est décrit par la structure `sos_cpu_ustate` du fichier `hwcore/cpu_context.c`. Elle est similaire à la structure `sos_cpu_kstate` qui contient le contexte d'un thread *noyau* (voir l'article 6). Nous avons donc rassemblé la partie commune de ces deux structures dans une structure appelée `sos_cpu_state`.

La différence entre `sos_cpu_kstate` et `sos_cpu_ustate` correspond aux informations supplémentaires que le processeur empile en cas de traitement d'une interruption accompagné d'un changement de privilège (voir la section 3.3.3). En effet, le contexte d'un thread utilisateur ne sera perçu par le noyau... qu'après un changement de privilège, donc le processeur aura forcément empilé ces informations sur la pile noyau.

Dans SOS, ces informations supplémentaires correspondent aux champs `cpl3_esp` et `cpl3_ss` de la structure `sos_cpu_ustate` (voir la figure 14).

4.2.2 Initialisation d'un contexte de thread utilisateur

L'initialisation d'un thread utilisateur est similaire à l'initialisation d'un thread noyau. Il s'agit d'allouer une pile et de l'initialiser de manière compatible avec le mécanisme de sauvegarde et restauration de contexte.

Cette initialisation s'effectue *via* la fonction `sos_cpu_ustate_init()` du fichier `hwcore/cpu_context.c`. Elle prend en argument `user_start_PC`, l'adresse de début du code du thread utilisateur, `user_start_arg`, l'argument à passer au thread, `user_initial_SP` l'adresse à affecter initialement au pointeur de pile utilisateur, `kernel_stack_bottom`, l'adresse du bas de la pile noyau et `kernel_stack_size` la taille de la pile noyau. Elle retourne un contexte utilisateur de type `sos_cpu_state` via le pointeur `ctxt`.

La structure `sos_cpu_ustate` est allouée en haut de la pile noyau au début de la fonction puis initialisée à 0. Certains membres de cette structure sont ensuite initialisés :

- `eax` est initialisé avec la valeur de l'argument `user_start_arg` qui sera passé au thread lors de son lancement. Exceptionnellement, l'ABI IA32 System V[2] n'est donc pas respectée pour la fonction utilisateur initiale exécutée par le thread utilisateur ;
- `eip` est initialisé avec l'adresse de début du code du thread utilisateur, soit `user_start_PC` ;
- `cpl3_esp` est initialisé avec l'adresse du haut de la pile utilisateur, soit `user_initial_SP` ;
- `cs`, `ds`, `es` sont initialisés pour pointer sur le segment de données de niveau de privilège **3** de la *GDT* ;
- `cpl3_ss`, c'est-à-dire le registre de segment utilisé pour accéder à la pile lorsque le thread est en mode **utilisateur**, est également initialisé pour pointer vers le segment de données de niveau de privilège **3** dans la *GDT* ;
- `cpl0_ss`, c'est-à-dire le registre de segment utilisé pour accéder à la pile lorsque le thread est en mode **noyau**, est initialisé pour pointer vers le segment de données de niveau de privilège **0** dans la *GDT*.

En plus de cette fonction d'initialisation d'un contexte, le fichier `hwcore/cpu_context.c` propose une fonction permettant de savoir si un *thread* a été interrompu alors qu'il s'exécutait en mode utilisateur ou en mode noyau : `sos_cpu_context_is_in_user_mode()`. Elle consulte pour cela la valeur sauvegardée du registre de segment `cs`.

4.2.3 Changement de contexte

Jusqu'ici, les procédures de sauvegarde et de restauration du contexte d'un thread, implémentées dans le fichier `hwcore/cpu_context_switch.S`, permettaient d'effectuer ces opérations uniquement sur des threads *noyau*.

Comme nous l'avons vu dans la partie 4.2.1, le contexte d'un thread *utilisateur* est similaire à celui d'un thread *noyau*, sauf en ce qui concerne les informations empilées et dépilées automatiquement par le processeur.

[2] Application Binary Interface. Voir l'article 6.

Puisque celles-ci sont gérées automatiquement par l'instruction `iret` du processeur (voir la fin de la section 3.3.3), les routines `sos_cpu_context_exit_to()` et `sos_cpu_context_switch()` ne nécessitent pas de modifications à ce sujet.

En revanche, avant de retourner vers un thread *utilisateur*, il faut mettre à jour le *TSS* noyau avec l'adresse de la pile noyau du contexte destination (voir la section 3.3.2). C'est pourquoi un appel à la fonction `sos_cpu_context_update_kernel_tss()` a été ajouté.

Pour conclure, signalons qu'un changement de contexte ne s'occupe pas de changer d'espace d'adressage. Ce sera le rôle du sous-système `thread` que nous voyons ci-dessous.

4.3 Thread utilisateur

L'implémentation des fonctions de haut niveau de gestion des threads *utilisateur* est présente dans le même fichier que celle des threads *noyau*, c'est-à-dire dans `sos/thread.c`. Remarquez que nous avons renommé "`kthread`" de l'article 6.5 en "`thread`".

4.3.1 Description

Deux nouveaux champs importants ont été ajoutés à la structure `sos_thread`.

Le premier champ, `process`, est un pointeur vers le *process* auquel appartient le thread. Pour un thread utilisateur qui exécute du code en mode utilisateur, l'espace d'adressage courant doit toujours être celui défini par le champ `mm_context` de la structure `process` liée à ce thread. Pour différencier un *thread* noyau d'un thread utilisateur, il suffira donc de tester si ce champ `process` vaut `NULL` (⇔ thread noyau) ou pas (⇔ thread utilisateur).

Le second champ, `squatted_mm_context`, est un pointeur vers un *espace d'adressage*, c'est-à-dire une structure `mm_context`. Ce champ nécessite de plus amples explications, nous y reviendrons en section 4.3.3.

4.3.2 Création

La fonction qui permet de créer un nouveau thread *utilisateur* est `sos_create_user_thread()`. Elle prend en argument, entre autres, une référence vers un *process*. Après avoir alloué une structure `sos_thread`, cette fonction alloue une pile noyau pour le thread puis initialise le contexte du thread utilisateur en utilisant la fonction `sos_cpu_ustate_init()` (voir 4.2.2). Ensuite, elle enregistre le nouveau thread dans le *process* passé en argument, puis dans la liste globale des threads du système. Enfin, elle signale à l'ordonnanceur que ce nouveau thread est prêt pour l'exécution (appel à `sos_sched_set_ready()`).

4.3.3 Champ `squatted_mm_context`

Pour un thread noyau, ce champ est normalement nul. Cela est lié au fait qu'un thread noyau peut s'exécuter dans n'importe quel espace d'adressage. Il s'exécute en effet en général dans l'espace d'adressage courant, quel qu'il soit (voir la section 3.1). Il en va de même pour un thread utilisateur en mode noyau. Cela permet de limiter le nombre de changements d'espace d'adressage. Ces changements sont en effet très coûteux en temps parce qu'ils vident le cache de traduction d'adresses (TLB).

Toutefois, un thread noyau ou un thread utilisateur en mode noyau peut demander à utiliser un espace d'adressage précis. Ceci est par exemple le cas quand le noyau traite un appel système : il peut avoir besoin d'accéder à des données situées dans la partie utilisateur de l'espace d'adressage du thread. Dans ce cas, `squatted_mm_context` n'est pas nul et pointe sur l'espace d'adressage auquel on souhaite accéder. C'est la fonction `sos_thread_change_current_mm_context()` qui permet de modifier ce champ, en effectuant un changement immédiat d'espace d'adressage si besoin.

Ce champ `squatted_mm_context` sert à garantir qu'un thread noyau, ou un thread utilisateur en mode noyau, accédera toujours à l'espace d'adressage requis, même en cas de bloquage. En effet, en cas de bloquage il peut y avoir changement d'espace d'adressage (voir ci-dessous). Mais il faut assurer que lorsque le thread sera de nouveau élu, l'espace d'adressage auquel il a souhaité accéder sera de nouveau configuré dans la MMU. Sinon, le thread sera de nouveau élu mais il risquera d'accéder à des données d'un autre espace d'adressage que celui auquel il souhaitait accéder. Il faut donc avoir mémorisé l'identité de cet espace d'adressage à restaurer impérativement : c'est à cela que sert le champ `squatted_mm_context`.

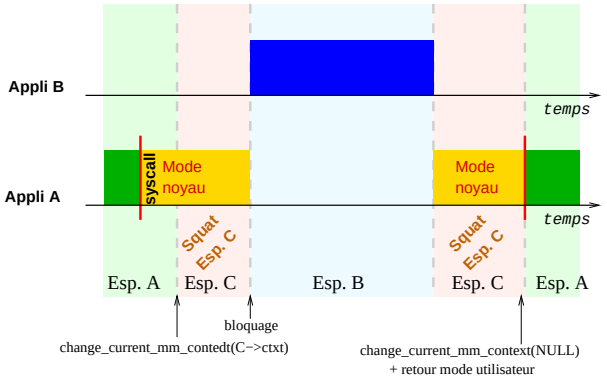

FIG. 16 – Un thread de l'application A fait un appel système. Cet appel système souhaite modifier l'espace d'adressage C mais il bloque et laisse la main à l'application B. Il faut qu'à son retour, il se retrouve dans l'espace de C pour continuer ses modifications.

4.3.4 Changement de thread et changement d'espace d'adressage

Comme pour l'article 6.5, la fonction centrale pour changer le thread courant est `_switch_to_next_thread()`, interne à `sos/thread.c`. Elle s'occupe de changer de contexte et de mettre à jour la variable interne `current_thread`.

Mais elle s'occupe également de changer d'espace d'adressage s'il y a :

- changement de contexte vers un thread utilisateur en mode utilisateur ;
- changement de contexte vers un thread noyau ou un thread utilisateur en mode noyau qui a requis l'accès à un espace d'adressage précis (i.e. champ `squatted_mm_context` non `NULL`).

Plus précisément, c'est la fonction interne `_prepare_mm_context()` qui s'en occupe. Tout ceci permet un changement d'espace d'adressage explicite.

4.3.5 Changement d'espace d'adressage implicite

Dans la section 3.3.2, nous avions signalé que la fin du traitement d'une interruption s'accompagnait de la mise à jour du TSS. Il en va de même pour la mise à jour de l'espace d'adressage courant.

En effet, la fin du traitement d'une interruption peut correspondre au retour vers un thread utilisateur en mode utilisateur, ou vers un thread noyau qui requiert l'accès à un espace d'adressage particulier (voir section 4.3.3). Dans ce cas, il faut que l'espace d'adressage au moment du retour soit le bon. En effet, pendant le traitement de l'exception, il peut y avoir eu bloquage (cas des exceptions et des appels système) avec changement d'espace d'adressage. Il faut donc éventuellement revenir vers l'espace d'adressage requis avant de restaurer le contexte du thread.

C'est pour cela que nous avons rajouté des appels à :

- `sos_thread_prepare_irq_switch_back()` à la fin du traitement des IRQs ;
- `sos_thread_prepare_exception_switch_back()` à la fin du traitement des exceptions ;
- `sos_thread_prepare_syscall_switch_back()` à la fin du traitement de l'appel système.

Ces fonctions ne font qu'appeler `_prepare_mm_context()` pour effectuer un changement d'espace d'adressage si nécessaire.

4.4 Appels système

4.4.1 Implémentation et protocole d'appel

Les appels système sont implémentés directement dans le gestionnaire de second niveau de l'interruption logicielle `0x42`, appelée par la routine assembleur décrite en section 3.3.1. Ce gestionnaire d'appels système est une simple fonction (fichier `sos/syscall.c`) :

```
sos_ret_t sos_do_syscall(int syscall_id,
                         const struct sos_cpu_state *user_ctxt);
```

Le paramètre `syscall_id` représente l'identifiant du service noyau demandé par le thread utilisateur. Pour l'instant, le noyau *SOS* propose deux services aux applications utilisateur (voir `sos/syscall.h` pour connaître leur identifiant) :

- Un service permettant de terminer le thread courant, similaire à l'appel système `exit()` sous Unix ;
- Un service permettant d'afficher un message sur la console *Bochs*.

Ces services récupèrent les arguments de l'appel système en utilisant les fonctions `sos_syscall_getXarg()`. Celles-ci sont implémentées dans le fichier `hwcore/cpu_context.c`. Dans SOS, nous avons choisi une convention d'appel système particulière. Cette convention n'est définie par aucune ABI, mais elle est classique et c'est elle que les fonctions `sos_syscall_getXarg()` exploitent :

- Le registre `eax` du contexte utilisateur interrompu contient l'identifiant du service noyau demandé. C'est lui qu'on retrouve en paramètre de `sos_do_syscall()` ;
- Les registres `ebx`, `ecx` et `edx` du contexte utilisateur interrompu contiennent les trois premiers arguments passés au service noyau demandé.

Lorsque l'appel système possède 1, 2 ou 3 arguments, alors ceux-ci sont directement passés dans les registres. La fonction `sos_syscall_get3args()` permet de récupérer ces valeurs dans le contexte du thread interrompu. Les fonctions idoines pour récupérer 1 ou 2 arguments utilisent également `sos_syscall_get3args()`. Pour les appels système avec 4, 5 ou 6 arguments, le dernier registre, `edx`, indique l'adresse d'un tableau contenant les autres arguments.

4.4.2 Fonctions utilitaires `sos_copy_from_user()` et `sos_copy_to_user()`

Dans le cas du traitement des appels système à plus de 3 arguments, nous venons de signaler qu'il est nécessaire de récupérer le contenu d'un tableau contenant les arguments 4, 5, ... Or ce tableau est dans la zone utilisateur puisqu'il est alloué sur la pile utilisateur du thread juste avant l'appel système (voir la section 5.1).

Pour cela, les fonctions `sos_syscall_getXargs()` avec $X > 3$ utilisent la fonction `sos_copy_from_user()`. Cette fonction demande explicitement l'accès à l'espace d'adressage du thread (voir la section 4.3.3) avant d'y accéder pour récupérer les données souhaitées. Cette fonction, ainsi que sa symétrique, `sos_copy_to_user()`, sont implémentées dans `sos/uaccess.c`.

Un autre exemple d'utilisation de ces fonctions est l'implémentation de l'appel système pour l'affichage d'une chaîne sur la console *Bochs*. Un des arguments est l'adresse de la chaîne de caractères à afficher. Celle-ci étant située dans la zone utilisateur, il faut utiliser `sos_copy_from_user()` pour y accéder :

```
case SOS_SYSCALL_ID_BOCHS_WRITE:
  {
    sos_uaddr_t user_str;
    unsigned int len;
    char * str;
    retval = sos_syscall_get2args(user_ctxt, & user_str, & len);
    str = (char*)sos_kmalloc(len + 1, 0);
    sos_copy_from_user((sos_vaddr_t) str, user_str, len);
    str[len] = '\0';
    sos_bochs_putstring(str);
    sos_kfree((sos_vaddr_t)str);
    vreak;
  }
```

5 Applications utilisateur

Nous voilà prêts à exécuter des applications utilisateur, mais lesquelles ?... Dans un système d'exploitation complet, les applications (*shell*, commandes, etc.) sont souvent chargées depuis des fichiers stockés sur le disque dur. Cela nécessite un pilote de périphérique pour le disque dur, un système de fichiers et éventuellement un *Virtual File System*, mécanismes qui seront étudiés dans de prochains articles de la série.

Puisque nous ne disposons pas pour le moment de ces éléments dans *SOS*, une solution de contournement a été mise en place. Elle consiste à embarquer directement dans le binaire du noyau le code et les données (i.e. binaires au format ELF) des applications utilisateur. Nous détaillons dans cette partie le fonctionnement de ce système.

5.1 Fonctionnement

Les applications utilisateur et quelques petites bibliothèques de fonctions utiles sont stockées dans le répertoire `userland/`.

En guise de petite démo, les fichiers `myprog1.c` à `myprog6.c` constituent le code des applications utilisateur. Ils prennent la forme d'applications Unix relativement standard, avec une fonction `main()`. Elles peuvent appeler quelques fonctions offertes par les petites bibliothèques.

Le fichier `libc.c` contient quelques fonctions minimales pour les applications utilisateur : de quoi terminer l'application (`exit()`) et de quoi écrire sur le port de débogage de Bochs (`bochs_write()`). Ce sont en l'occurrence des appels système.

Le fichier `crt.c` (pour "C Run-Time") contient le code de plusieurs fonctions importantes :

- La fonction `_start` qui est le point d'entrée de chaque application. En effet, `main()` n'est pas appelée directement, c'est la fonction `_start` qui l'appelle. En plus de cela, lorsque la fonction `main()` retourne, `_start` est chargée de demander au noyau de détruire l'application (appel système `exit()`) en lui transmettant la valeur de retour de la fonction `main()` ;
- Une fonction `_sos_syscall3()` qui permet de réaliser un appel système vers le noyau. Le paramètre `id` désigne le numéro de l'appel système, et les paramètres `arg1`, `arg2` et `arg3` constituent les arguments de l'appel système. Ces quatre valeurs sont stockées dans les registres `eax`, `ebx`, `ecx` et `edx`. La fonction réalise ensuite l'appel système proprement dit en demandant explicitement la levée d'une interruption via l'instruction `int $SOS_SWINTR_SOS_SYSCALL`. A cet endroit précis, c'est le noyau qui va exécuter le service requis. Une fois que celui-ci a terminé, l'application poursuivra son exécution à l'instruction suivante, c'est-à-dire `movl %%eax, %0` qui récupère la valeur de retour de l'appel système ;
- Des fonctions `_sos_syscall0()` à `_sos_syscall6()` qui utilisent `_sos_syscall3()` pour réaliser des appels système demandant de 0 à 6 arguments. Au delà de 3, les arguments supplémentaires sont stockés temporairement dans un tableau alloué sur la pile, dont l'adresse est passée en dernier argument de l'appel système ;
- Les fonctions `_sos_exit()` et `_sos_bochs_write()` qui réalisent les appels système permettant de demander ces services au noyau.

5.2 Compilation

Nous décrivons dans cette section le mécanisme de compilation des applications utilisateur et d'intégration au binaire final du noyau. Ce mécanisme est résumé dans la figure 17.

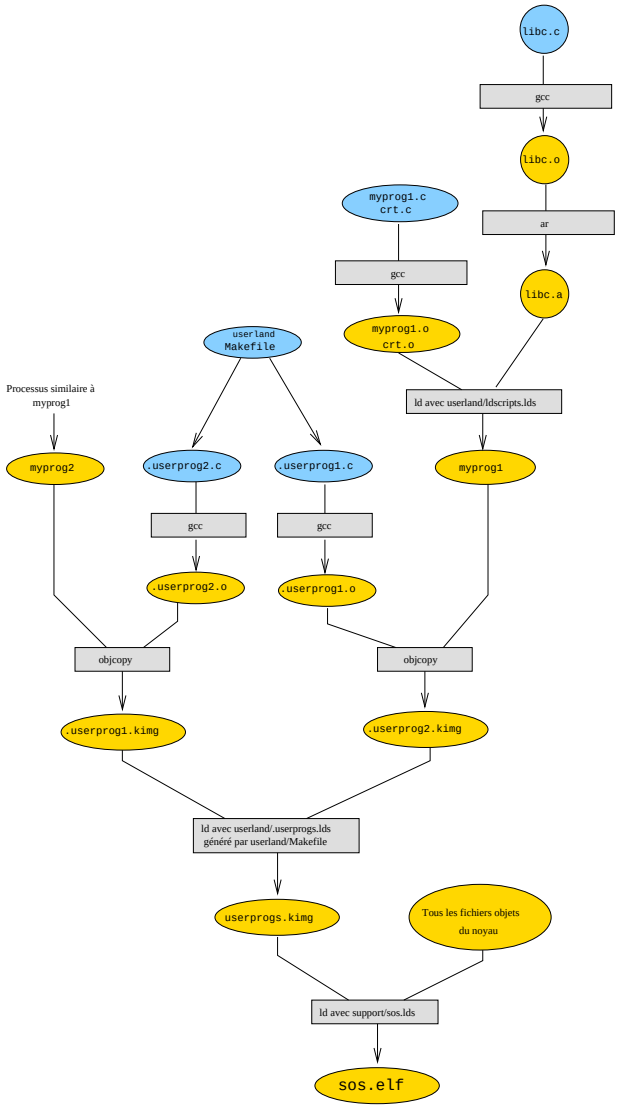

FIG. 17 – Processus de compilation des applications et d'intégration au binaire du noyau.

5.2.1 Compilation des applications proprement dite

Chaque application est compilée puis linkée statiquement avec la petite bibliothèque de fonction `libc.a` à l'aide du script de linkage `ldscript.lds`. Celui-ci définit l'emplacement des sections en mémoire virtuelle lorsque le programme sera chargé. En particulier, la directive `. = 0x80000000` indique que le programme sera chargé à l'adresse 0x80000000, soit 2 Go. Les applications sont linkées pour fonctionner à la même adresse, mais il s'agit d'adresses virtuelles. Comme chaque application fonctionnera dans son propre espace d'adressage, cela ne posera

aucun problème et c'est justement là un des intérêts de la mémoire virtuelle (on n'est pas obligé de recompiler les applications en fonction de celles qui sont déjà présentes dans le système).

5.2.2 Création du fichier `userprogs.kimg`

Le `Makefile` réalise ensuite une suite d'opérations pour construire un unique fichier `userprogs.kimg` contenant l'ensemble des applications, avec des marqueurs permettant de situer le début et la fin de chacune d'entre elles. Pour cela, il construit pour chaque application un petit fichier C nommé `.userprogX.c` où `X` est le numéro de l'application. Ce fichier contient simplement quelque chose comme :

```
extern char _begin_userprog1, _end_userprog1;
char *_userprog1_entry[] __attribute__((section(".userprogs_table")))
  = { "myprog1", &_begin_userprog1, &_end_userprog1 };
```

Les symboles `_begin_userprog1` et `_end_userprog1` seront définis par un script de linkage, et positionnés respectivement au début et à la fin du binaire enfoui dans le noyau (voir plus loin). L'adresse de ces symboles permet donc de délimiter dans l'image finale les limites d'une application. Ensuite, un élément `_userprog1_entry` est défini et positionné dans la section ELF `.userprogs_table`. Ce positionnement dans une section spécifique permettra à l'éditeur de liens de fusionner tous les éléments `_userprog1_entry`, `_userprog2_entry`, etc.. pour former un unique tableau contenant le nom de chaque application et leurs adresses respectives de début et de fin.

Le `Makefile` compile ce petit fichier C, puis à l'aide de la commande `objcopy`, intègre au fichier objet résultant le binaire de l'application précédemment compilée. Il utilise pour cela `objcopy` de la manière suivante :

```
objcopy --add-section .userprog1=myprog1 .userprog1.o .userprog1.kimg
```

Cette commande prend le contenu de `.userprog1.o` (le petit fichier C qui vient d'être compilé) et le copie dans `.userprog1.kimg` en ajoutant une section ELF nommée `.userprog1`. Cette section contient le contenu du fichier `myprog1`, c'est-à-dire le binaire au format ELF de notre application. Le fichier `.userprog1.kimg` contient donc à la fois le binaire de l'application et les informations concernant cette application.

En parallèle de cela, le `Makefile` créé un script de linkage `.userprogs.lds` qui est utilisé pour linker tous les fichiers `.userprogX.kimg` en un fichier final `userprogs.kimg`. Pour deux applications, le script de linkage est le suivant :

```
SECTIONS { .rodata . : {
  _begin_userprog1 = .; .userprog1.kimg(.userprog1); _end_userprog1 = .;
  .userprog1.kimg(.rodata); .userprog1.kimg(.data);
  _begin_userprog2 = .; .userprog2.kimg(.userprog2); _end_userprog2 = .;
  .userprog2.kimg(.rodata); .userprog2.kimg(.data);
  _userprogs_table = .; *(.userprogs_table) ; LONG(0);
} /DISCARD/ : { *(.text) *(.data) *(.bss) } }
```

Ce script de linkage définit une section `.rodata` qui contiendra les sections `.userprog1` et `.userprog2` des fichiers `.userprog1.kimg` et `.userprog2.kimg` et définit des symboles entourant ces sections. Ce sont les symboles `_begin_userprogX`/`_end_userprogX` que nous évoquions plus haut et qui sont référencés dans les fichiers `.userprogX.c`. Ils permettent de déterminer le début et la fin d'une application. Les sections `.data` et `.rodata` des fichiers `.userprogX.kimg` sont incluses car elles contiennent les chaînes de caractères représentant le nom des applications. De plus, la section `.userprogs_table` est également incluse, c'est elle qui contient la table des applications. Le symbole `_userprog_table` indique l'adresse du début de cette table et un 0 (`LONG(0)`) en marque la fin. Les sections `text`, `data` et `bss` ne sont pas inclues dans le fichier binaire résultant puisqu'elles ne contiennent rien d'intéressant.

À partir de ce script de linkage `.userprogs.lds` et des applications `.userprogX.kimg`, le fichier `userprogs.kimg` est créé par `ld`.

5.2.3 Intégration au noyau

Pour finir, le fichier `userprogs.kimg`, contenant les binaires de nos applications ainsi qu'une table les décrivant, est linké avec les autres fichiers objets du noyau. Cette opération est réalisée durant l'édition de liens finale qui utilise le script de linkage `support/sos.lds` et produit le fichier `sos.elf`.

La structure finale du fichier `sos.elf` est exposée dans la figure 18. Ce mécanisme de compilation, relativement complexe, permet d'enfouir les fichiers binaires ELF des applications dans le binaire du noyau. Il permet de se passer d'un système de fichiers pour stocker ces binaires. De plus, il montre la puissance des outils de compilation et d'édition de liens utilisant le format ELF.

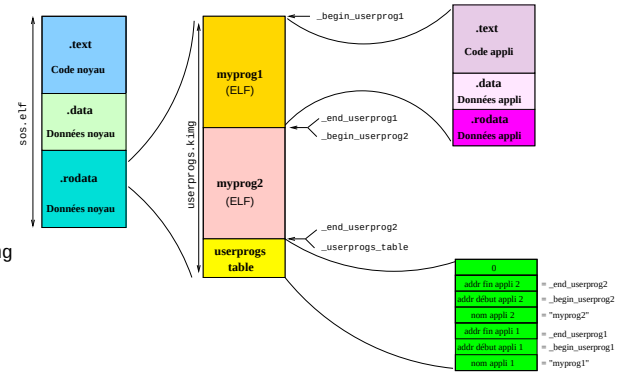

FIG. 18 – Structure finale du fichier `sos.elf` avec le code et les données du noyau. Au sein de ces dernières se trouvent les applications utilisateur.

5.3 Chargement

Le code noyau qui s'occupe du chargement des programmes utilisateur se trouve dans le fichier `sos/test-art7.c`. La fonction principale de ce code se contente de faire des appels à la fonction centrale de toute la démo : `spawn_program()`.

Cette fonction `spawn_program(nom_prog, nb_threads)` s'occupe de créer un nouvel espace d'adressage, de mapper le programme utilisateur de nom `nom_prog` dans cet espace, et d'y créer `nb_threads` threads utilisateur. Les `nb_threads` threads exécutent le même code : il s'agit de celui de la fonction `main()` du programme utilisateur `nom_prog`. Les différents programmes de test que nous proposons sont les suivants :

myprog1 : Affichage de 5 chaînes de caractères,

myprog2 : Affichage de 2 chaînes de caractères séparées par une grosse boucle consommant du temps processeur inutilement,

myprog3 : Écriture à une adresse interdite (i.e. dans le noyau) pour déclencher une exception “page fault”. Ceci provoque la terminaison du thread fautif,

myprog4 : Écriture à une adresse invalide (i.e. adresse non mappée en partie utilisateur) pour déclencher une exception “page fault”. Ceci provoque la terminaison du thread fautif,

myprog5 : Boucle infinie. Permet de montrer que, alors que le noyau est de type non-préemptible, les applications utilisateur sont ordonnancées en préemptif,

myprog6 : Appel d'une instruction assembleur réservée au mode superviseur (`cli`). Ceci provoque une exception “general protection fault” suivie de la suppression du thread fautif. Permet de montrer que l'application utilisateur s'exécute avec des privilèges restreints.

Tous ces tests affichent des messages sur le port `0xe9` de bochs/qemu (figure 19). Ils seront donc invisibles à l'œil nu sur machine réelle. Les seuls effets visibles seront que le compteur binaire de la tâche `idle` (en haut à gauche de l'écran) sera ralenti et que la charge processeur des applications utilisateur sera non nulle et fluctuante.

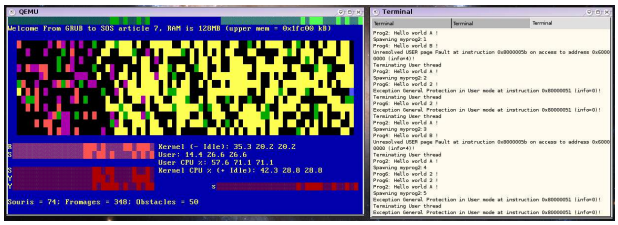

FIG. 19 – Aperçu de la petite démo (à droite : terminal pour observer les sorties sur le port `0xe9`)

Au passage, dans `sos/calcload.c`, nous donnons le code d'un système de calcul de la charge processeur. En fait, nous calculons 4 charges (charge noyau/utilisateur, pourcentage de processeur utilisé pour le noyau/les applications utilisateur). Et pour chaque charge nous donnons sa moyenne sur les 1, 5 et 15 dernières minutes (mise à jour toutes les secondes). Nous avons conservé la démo “souris” (article 6.5) et le test `test_thread()` parce qu'ils chargent le processeur en threads noyau.

La fonction `spawn_program()` fonctionne en quatre étapes, vues ci-dessous.

5.3.1 Récupération de l'adresse du programme utilisateur dans l'image du noyau

Le symbole `_userprog_table` marque le début d'une zone de données en partie noyau formant un tableau (voir la section 5.2.2). Chaque élément du tableau a la structure suivante :

```
struct userprog_entry
{
  const char *name;
  sos_vaddr_t bottom_vaddr;
  sos_vaddr_t top_vaddr;
};
```

Pour rechercher un programme de nom `"toto"`, le noyau parcourt ce tableau jusqu'à trouver une entrée dont le champ `name` vaut `"toto"`, ou `0` qui signifie qu'on a atteint la fin du tableau sans trouver le programme recherché. Ce `0` avait été ajouté par le script de linkage (voir la section 5.2.2).

Tout ceci est pris en charge par la fonction noyau `lookup_userprog(progname)` qui renvoie l'entrée `struct userprog_entry` correspondant au programme recherché, ou `NULL` si on ne l'a pas trouvé.

5.3.2 Création du nouvel espace d'adressage

La suite consiste à appeler `sos_process_create_empty()` pour créer un nouvel espace d'adressage (voir section 4.1.2), puis `sos_thread_change_current_mm_context()`. Cette deuxième fonction a pour rôle d'indiquer que le thread noyau courant a momentanément besoin de s'exécuter dans le nouvel espace d'adressage (voir la section 4.3.3). Cela va permettre de modifier ce nouvel espace d'adressage pour y copier le programme en partie utilisateur.

5.3.3 Copie du programme dans le nouvel espace d'adressage

C'est la fonction `load_elf_prog(prog_entry)` qui va s'occuper de copier le programme utilisateur au bon endroit en partie utilisateur.

Le paramètre `prog_entry` est le `struct userprog_entry` renvoyé par `lookup_userprog()`. Il indique le début et la fin de la zone du noyau qui contient le programme utilisateur. Cette zone de données correspond au contenu d'un fichier au format ELF (voir la section 5.2.2).

La fonction `load_elf_prog()` constitue un mini-chargeur ELF très basique. Elle utilise les structures de données définies par le format ELF32 [3] pour repérer les adresses virtuelles où les différentes parties du “fichier” doivent être chargées. Des pages physiques sont ensuite mappées en mémoire à ces adresses virtuelles et le contenu de ces parties du “fichier” est copié en mémoire virtuelle.

La fonction utilise également les structures ELF pour repérer l'adresse de début du programme, i.e. la fonction utilisateur `_start`.

5.3.4 Création des threads utilisateur

Pour terminer, c'est la fonction `spawn_program()` qui se charge elle-même de la création des threads utilisateur. Pour chaque thread, elle alloue une pile utilisateur de 1 page en partant de l'adresse la plus haute (i.e. `0xffffffff`) et en descendant de 12 pages à chaque thread (pour détecter les débordements de pile utilisateur). Puis elle appelle la fonction `sos_create_user_thread()` pour créer chaque thread. L'adresse de la première instruction utilisateur à exécuter est celle qui avait été retournée par `load_elf_prog(prog_entry)`.

Il ne reste plus qu'à relâcher la référence au nouveau processus (fonction `sos_process_unref()`) puis à indiquer qu'on ne souhaite plus s'exécuter obligatoirement dans le nouvel espace d'adressage (fonction `sos_thread_change_current_mm_context(NULL)`). Et voilà, la fonction `spawn_program()` a fait son devoir (voir la figure 20), les threads utilisateur peuvent maintenant commencer à vivre !

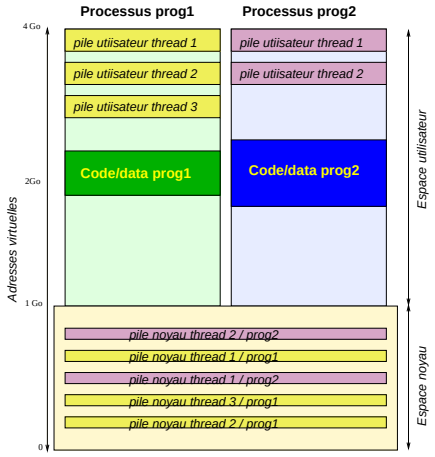

FIG. 20 – Configuration de la mémoire virtuelle de deux processus, après chargement des programmes (les adresses ne sont pas à l'échelle).

Conclusion

Cet article nous a permis de faire tourner nos premières applications utilisateur ! Il a été l'occasion d'introduire de nombreuses notions : protection, privilèges, changement de privilège, appels système, espace d'adressage, threads utilisateur, processus.

Vous êtes maintenant en mesure d'écrire vos propres appels système, d'enrichir la bibliothèque C utilisateur embryonnaire que nous fournissons, ou de compiler vos propres applications utilisateur (par exemple `gcc`...). Celles-ci sont pour l'instant enfouies directement dans le binaire du noyau par une subtile combinaison de scripts et de Makefile. Mais, à terme, nous disposerons d'un système de fichiers qui nous permettra de charger nos applications d'une façon plus conventionnelle.

L'article du mois prochain sera un complément de celui-ci. Il s'intéressera à la gestion de la partie utilisateur de chaque espace d'adressage. Nous avons décidé de le baptiser "article 7 et demi". L'article qui suivra entamera une série d'articles consacrés au système de fichiers. Dans un premier temps, les deux domaines n'auront rien à voir, mais ils se rejoindront.

Voilà, il est temps de vous laisser vous approprier cet article et son code. Nous faisons une pause d'un mois. Nous vous donnons donc rendez-vous au début du joli mois de Mai !

Déboguer avec Qemu

Introduction

Dans le premier article de la série, nous avons présenté l'émulateur de machine PC *Qemu* [4]. En plus de permettre de réaliser des tests plus rapidement, *Qemu* permet également de déboguer le système qui s'exécute en son sein. Pour cela, il s'interface avec le débogueur *gdb* [5]. Nous vous proposons de découvrir cette possibilité qui pourra vous être utile pour déboguer vos expérimentations autour de SOS ou d'autres systèmes d'exploitation.

Démarrer le débogage

Pour s'interfacer avec *qemu*, *gdb* utilise une connexion au travers d'une socket réseau sur le port 1234.

Il faut tout d'abord lancer `qemu` :

```
qemu -S -s -fda fd.img -boot a
```

L'option `-s` demande à *Qemu* d'ouvrir le port 1234 pour le débogage avec `gdb`. L'option `-S` précise que *Qemu* ne doit pas lancer la simulation, mais attendre qu'on la lance au travers du débogueur.

Une fois *Qemu* lancé, celui-ci ne démarre pas la simulation, il attend le débogueur. Nous lançons donc *gdb*, puis dans le *shell* du débogueur, nous lui indiquons de se connecter à *Qemu* :

```
target remote localhost:1234
```

On peut maintenant lancer la simulation en tapant la commande `c` ("continue"). A tout instant, on peut stopper la simulation en pressant `Ctrl+C`.

Points d'arrêt

Il est possible de positionner des points d'arrêt dans le code, c'est-à-dire des adresses auxquelles l'exécution va s'interrompre pour laisser la main au débogueur. Les points d'arrêt sont positionnés avec la commande `break` dans *gdb*. Par exemple, `break *0x2092dc` positionne un point d'arrêt à l'adresse `0x2092dc`.

Les relations entre adresses en mémoire et noms de fonctions dans SOS sont listées dans le fichier `sos.map` généré à chaque compilation. Toutefois, *gdb* nous propose une autre solution : lire directement la table des symboles du fichier `ELF sos.elf`. Pour cela, il faut utiliser la commande `symbol-file sos.elf` dans le *shell* de `gdb`, ou lancer `gdb` en lui passant `sos.elf` en paramètre.

Grâce à cela, il est maintenant possible d'écrire `break sos_main`, ce qui positionne un point d'arrêt au début de la fonction `sos_main()`. Il est évidemment possible de positionner plusieurs points d'arrêt, de lister les points d'arrêts (`info break`) ou de supprimer un point d'arrêt (`delete break num_point_arret`).

La session suivante positionne deux points d'arrêt, les liste, supprime l'un d'entre eux, puis lance la simulation (commande `c`). Celle-ci est automatiquement interrompue lorsque le processeur rencontre le point d'arrêt positionné sur la fonction `CreateMouse()`. Si on souhaite de nouveau reprendre la simulation, il suffit d'utiliser la commande "`c`".

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000fff0 in ?? ()
(gdb) symbol-file sos.elf
Reading symbols from sos.elf...(no debugging symbols found)...done.
(gdb) break sos_main
Breakpoint 1 at 0x2092dc
(gdb) break CreateMouse
Breakpoint 2 at 0x20a82d
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x002092dc <sos_main>
2   breakpoint     keep y   0x0020a82d <CreateMouse>
(gdb) delete break 1
(gdb) info break
Num Type           Disp Enb Address    What
2   breakpoint     keep y   0x0020a82d <CreateMouse>
(gdb) c
Continuing.

Breakpoint 2, 0x0020a82d in CreateMouse ()
(gdb)
```

Observer la mémoire et les registres

`gdb` permet d'observer le contenu de la mémoire et des registres du processeur. Par exemple :

```
(gdb) x /8xw sos_main
0x2092dc <sos_main>:    0x83e58955 0x458b58ec 0xe445890c 0xff9c5be8
0x2092ec <sos_main+16>: 0x9a52e8ff 0x04c7ffff 0x00001024 0x9a66e800
```

Étant donné que les données à cette adresse sont des instructions processeur, on peut demander à `gdb` de nous les traduire en assembleur :

```
(gdb) x /4iw sos_main
0x2092dc <sos_main>:    push   %ebp
0x2092dd <sos_main+1>:  mov    %esp,%ebp
0x2092df <sos_main+3>:  sub    $0x58,%esp
0x2092e2 <sos_main+6>:  mov    0xc(%ebp),%eax
```

L'état des registres peut être consulté via la commande `info registers`, et il est également possible de les modifier :

```
(gdb) info registers eip
eip            0x2092e0 0x2092e0
(gdb) set $eip+=4
(gdb) info registers eip
eip            0x2092e4 0x2092e4
```

Tous les registres ne sont pas accessibles avec `gdb`, tels les registres de débogage ou les registres système (`cr3`, `tr`, ...). Pour ceux-là, on utilisera la commande "info registers" du "monitor" de qemu (Ctrl-Alt-2).

Variables et code source

Pour l'instant, nous ne pouvons positionner des points d'arrêt qu'au début d'une fonction, et nous ne pouvons pas simplement afficher la valeur des variables locales ou des paramètres d'une fonction.

En effet, lorsque nous chargeons le fichier `sos.elf` via `symbol-file`, *gdb* nous précise "*no debugging symbols found*". Pour inclure les symboles de débogage, il faut recompiler SOS avec l'option `-g` de *gcc* : il suffit d'ajouter cette option à la variable `CFLAGS` du `Makefile` principal.

Grâce à cela, il est possible d'exécuter le code ligne par ligne, avec le code source sous les yeux et en ayant la possibilité de consulter les variables locales et les paramètres. La session suivante montre l'étendue des possibilités :

```
(gdb) c
Continuing.

Breakpoint 1, sos_main (magic=0, addr=0) at main.c:1740
1740    {
(gdb) print addr
$1 = 0
(gdb) n
1756      mbi = (multiboot_info_t *) addr;
(gdb) n
1759      sos_bochs_setup();
(gdb) print mbi
$2 = (multiboot_info_t *) 0x2c6a0
(gdb) print *mbi
$3 = {flags = 2023, mem_lower = 639, mem_upper = 130048, boot_device = 16777215, cmdline = 8192,
  mods_count = 0, mods_addr = 0, u = {aout_sym = {tabsize = 16, strsize = 40, addr = 2781184,
      reserved = 13}, elf_sec = {num = 16, size = 40, addr = 2781184, shndx = 13}}, mmap_length = 48,
  mmap_addr = 340580, drives_length = 0, drives_addr = 340628}
(gdb) print mbi->flags
$4 = 2023
(gdb) set mbi->flags=12
(gdb) print mbi->flags
$5 = 12
```

Cet exemple montre qu'on peut afficher la valeur d'un paramètre de la fonction courante (`print addr`), que le code source C est affiché au fur et à mesure de son exécution ligne par ligne (commande `n`), que les variables locales sont également accessibles (`print mbi`), que *gdb* a connaissance des types de données (`print *mbi`) et qu'on peut modifier aisément n'importe quel champ dans n'importe quelle structure (`set mbi->flags=12`).[3]

Conclusion

gdb associé à *Qemu* permettent de faciliter grandement le débogage d'un système d'exploitation. Nous terminerons cette petite annexe en précisant que lors de son exécution, *gdb* cherche un fichier `.gdbinit` dans le répertoire courant et qu'il exécute toutes les commandes de ce fichier. Ceci est pratique pour exécuter systématiquement les commandes `target remote localhost :1234`, `symbol-file sos.elf` et `break mon_symbole` par exemple.

The end.

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 Gabriel

Références

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

[2] Intel Corp. Intel architecture developer's manual, vol 2, 1997.

[3] TIS Committee. TIS ELF. `www.x86.org/ftp/manuals/tools/elf.pdf`, 1995.

[4] Fabrice Bellard. *The qemu CPU emulator*. GPL, `http://fabrice.bellard.free.fr/qemu/`, 2004.

[5] Projet GNU. *GDB : The GNU Project Debugger*. GPL, `http://www.gnu.org/software/gdb/gdb.html`.

[3] A noter que la rédaction de cette annexe a permis la découverte d'un bug dans *Qemu*. L'écriture d'une zone de la mémoire avec une commande comme `set mbi->flags=12` ne fonctionnait pas correctement. Un patch a été envoyé le 15 janvier 2005 à l'équipe de développement de *Qemu* et intégré le 17 janvier dans le CVS. Il faut donc utiliser une version postérieure à cette date pour pouvoir modifier la mémoire.