Croisière au cœur d'un OS*
Étape 6 : Multitâche et changement de contexte

Résumé

Au cours de cet article et du suivant, nous vous proposons la mise en place du multitâche, qui consiste à donner l'impression que le système exécute plusieurs programmes simultanément. Ce premier volet présente les mécanismes bas niveau pour passer d'un programme à un autre.

Introduction

L'aventure SOS repart en ce début d'année 2005 en attaquant un autre aspect important d'un système d'exploitation : la gestion de la ressource *processeur*.

Un processeur est une machine complexe mais qui n'est capable d'exécuter qu'une seule instruction à un instant donné[1]. Or tout système d'exploitation moderne doit permettre d'exécuter plusieurs programmes "simultanément". En réalité les différents programmes ne sont pas exécutés simultanément, mais l'un après l'autre, par tranches de temps. Mais ces tranches de temps sont suffisamment courtes pour donner l'illusion à l'utilisateur d'une exécution en parallèle.

Le processeur peut donc être considéré comme une ressource matérielle à partager entre les différents programmes, au même titre que la mémoire. Cette ressource nécessite une gestion logicielle de la part du système d'exploitation.

Cet article s'intéresse à une première partie de la gestion du processeur : le passage d'un programme à un autre, mécanisme plus connu sous le nom de *changement de contexte*. La technique choisie pour réaliser cela dans SOS sera décrite dans la section 3. Elle repose exclusivement sur la manipulation de données dans la pile, c'est pourquoi nous expliquerons d'abord le fonctionnement de la pile dans l'architecture `x86` (section 2). Enfin, comme à l'habitude, nous présenterons les petites démonstrations qui sont au nombre de 3 ce mois-ci (section 4). En annexe, on trouvera un bonus consacré au *backtracing* dans l'architecture `x86`.

Avant de poursuivre, signalons que nous parlons ici de "programmes" au sens très large. Plus tard ces programmes prendront la forme de threads ou d'applications utilisateur. Précisons également que le patch qui accompagne le code apporte plus de modifications que celles qui sont réellement liées au présent article : nous avons en effet corrigé quelques bugs et petits détails cosmétiques portant sur le code des articles précédents.

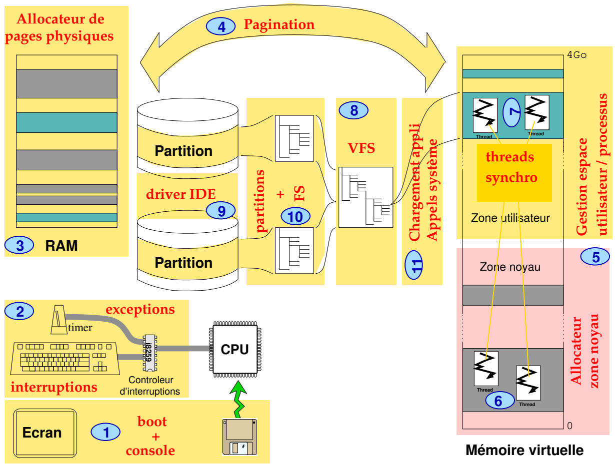


FIG. 1 – Programme des articles

1 Contexte d'exécution

Avant d'expliquer comment effectuer un changement de contexte, il convient de définir ce que nous entendons par *contexte d'exécution*, ou *contexte processeur*. Pour cela, il faut revenir rapidement sur les principes de base de l'exécution de programmes sur un processeur.

Un programme est constitué d'une suite d'instructions que le processeur exécute en séquence. Les instructions agissent sur la mémoire, sur des registres et sur les bus d'entrée/sortie pour l'accès aux périphériques. Si on veut passer d'un programme à l'autre, il s'agit de faire une photographie de l'état du programme, de la ranger, puis de sortir la photographie d'un autre programme et de la mettre en place dans le processeur. Cette opération est appelée *changement de contexte* et c'est cet état du programme qu'on appelle le *contexte d'exécution*.

La "photographie", ou sauvegarde du contexte, correspond à la sauvegarde des registres du processeur. Pour être exhaustif, il faudrait aussi sauvegarder le contenu de la mémoire et l'état des périphériques concernés par le programme, mais cela serait trop coûteux et, pire, inutile.

Notre implantation du changement de contexte repose sur la pile. La pile est un élément que nous avons plusieurs fois évoqué dans les articles précédents. C'est

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 68 – Janvier 2005 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

[1]On ignore ici les notions de *pipeline* ainsi que les processeurs type SMT (Simultaneous Multi-Threading, architecture Alpha) ou *Hyper-Threading* (Intel).

donc l'occasion de la présenter plus en détails maintenant. Nous reviendrons sur le changement de contexte dans la section qui suivra.

2 La pile

2.1 Généralités

Du point de vue du programmeur, un programme est constitué de fonctions appelant d'autres fonctions qui appellent d'autres fonctions, et ainsi de suite. Bref, il y a "imbrication" des appels de fonctions.

Pour chaque fonction, le programmeur peut définir des "variables locales" qui ne seront utilisables que lorsque le processeur sera entré dans la fonction et qui seront perdues dès qu'il en sortira. Dans l'idéal on voudrait stocker toutes ces variables dans les registres du processeur. Dans cet idéal, il faudrait donc un grand nombre de registres, puisque ni le nombre de variables locales à chaque fonction, ni leur taille (pour des tableaux), ni le nombre d'appels de fonctions imbriqués ne sont limités. Malheureusement, dans la pratique le nombre de registres reste réduit. L'idée est donc d'utiliser temporairement une zone de la mémoire (une *frame*) pour y stocker les données propres à chaque appel de fonction, puis de libérer cette zone lorsque la fonction "retourne" à sa fonction appelante.

Sur certaines architectures (Sparc et Alpha en particulier), toutes ces zones sont situées en partie sur le processeur et en partie en mémoire (technique de "renommage de registres"). Mais dans le cas le plus simple, elles sont intégralement en mémoire principale : dans ce qui suit, nous détaillons uniquement ce cas-là puisque c'est comme cela que fonctionne l'architecture `x86`. Ces zones de mémoire forment la "pile", ou *stack* en anglais. C'est le compilateur qui génère automatiquement toutes les instructions pour la gérer. La pile doit être de taille limitée mais suffisamment grande pour contenir une profondeur raisonnable d'imbrication d'appels de fonction. Si on la dimensionne mal, les frames risquent d'en sortir (on parle d'un "débordement de pile"), ce qui peut se traduire par des écrasements de données.

D'une manière générale, la pile fonctionne comme une pile d'assiettes, c'est-à-dire que le dernier élément empilé sera le premier élément dépilé. On parle ainsi de file *LIFO*, pour *Last In First Out*. Ce type de fonctionnement correspond tout à fait à ce qu'on cherche pour stocker les variables locales aux fonctions : à chaque instant, les variables de la fonction en cours d'exécution sont accessibles en tête de pile et celles des fonctions appelantes se situent juste derrière. Lorsque la fonction appelée "retournera", on "dépilera" la portion de la pile qui lui était affectée, et alors on retrouvera naturellement les variables locales de la fonction appelante en tête de la pile.

Chaque appel de fonction donne ainsi lieu à l'utilisation d'une portion contiguë de la pile : la "*frame*" en anglais, ou "cadre de pile" en français. Au final, au gré des appels de fonctions, les cadres de pile s'empilent dans la pile par empilements/dépilements successifs de valeurs (voir la figure 2). Remarquons que ceci entraîne

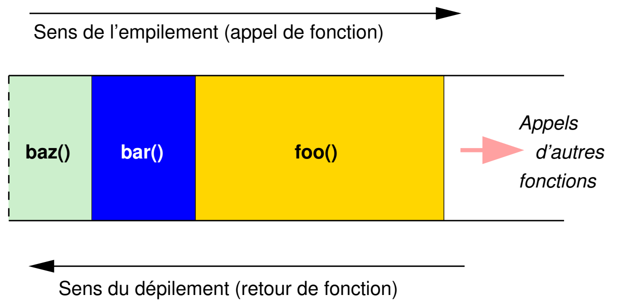


FIG. 2 – Emplacement des *frames* dans la pile les unes par rapport aux autres. Exemple avec une fonction `baz()` qui appelle `bar()` qui appelle `foo()`.

qu'un algorithme récursif écrit sous la forme de fonctions récursives (i.e. des fonctions qui s'appellent elles-mêmes, directement ou indirectement) sera grand consommateur de pile.

Sur `x86`, chaque "frame" rassemble les variables locales de la fonction associée et des valeurs temporaires utilisées par le compilateur pour le cœur de la fonction ou pour appeler d'autres fonctions.

2.2 Manipulation de la pile sur `x86`

Sur `x86`, la pile est le plus souvent manipulée par des instructions spécifiques : `push` (pour empiler une valeur ou la valeur contenue dans un registre) et `pop` (pour dépiler en transférant la valeur dépilée dans un registre ou vers la mémoire). Elle peut aussi être manipulée directement par les instructions habituelles d'accès à la mémoire (`mov`, `lea`, ...).

Le registre processeur `esp` contient à tout instant l'adresse courante dans la pile : c'est le "*pointeur de pile*" ou *Stack Pointer* (SP) en anglais. Il s'agit d'un registre **fondamental**, au même titre que le registre `eip`, le "compteur de programme", qui renferme l'adresse de l'instruction à exécuter.

Le registre `ebp` sert à mémoriser l'adresse de la frame courante : c'est le "*pointeur de frame*" ou *Frame Pointer* (FP) en anglais (sur `x86`, on dirait "Base Pointer" pour expliquer les lettres "BP"). Ce registre n'est pas fondamental, il permet essentiellement de faciliter le debogage. On peut désactiver son utilisation à l'aide de l'option `-fomit-frame-pointer` de `gcc`.

Sur `x86`, la pile est *descendante* [1, Chapitre 6.2], c'est-à-dire qu'un empilement décrémente le pointeur de pile (`esp`) et qu'un dépilement incrémente ce pointeur de pile, comme l'indique la figure 3. Dans la suite de l'article, quand on parlera de "bas" de la pile, on fera référence aux adresses basses, c'est-à-dire aux derniers empilements effectués. Et inversement quand on parlera de "haut" de la pile.

2.3 Étude d'un petit programme C

Afin d'étudier l'utilisation de la pile, nous vous proposons d'observer un programme C simple dont nous allons étudier l'assembleur. Ce qui suit est valable pour

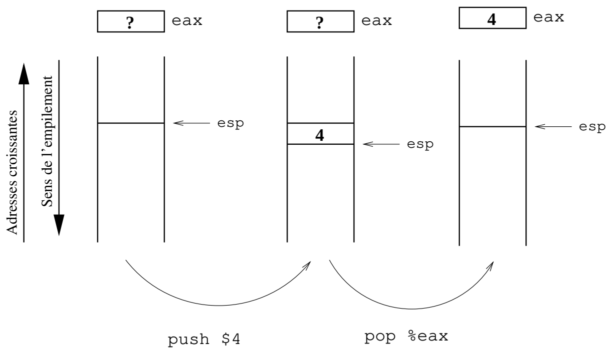


FIG. 3 – Empilement et dépilement

tout type de code, qu'il corresponde à du code noyau ou à du code d'application utilisateur.

Pour étudier l'assembleur, soit on demande à `gcc` de nous fournir le source assembleur après compilation (option `-S`), soit on compile le source en langage machine normalement (option `-c`) puis on utilise un désassembleur (par exemple `objdump -S machin.o`).

Le code C de notre programme :

```
int foo(int a, int b)
{
  int c;
  int t[12];
  c = a + b;
  return c;
}

int bar(int d)
{
  int e;
  e = d + 2;
  e = foo(e, 12);
  return e;
}
```

Pour l'exemple, ce code C a été compilé en utilisant `gcc 2.7.2.3` avec l'option `-O0` pour désactiver toutes les optimisations. Cette configuration a été choisie car c'est celle qui offrait le code le plus clair, certes pas le plus optimisé. D'autres versions de `gcc` ou d'autres niveaux d'optimisations donneront un fonctionnement identique (heureusement !) mais un code assembleur différent.

Le code assembleur généré par `gcc` est le suivant :

```
foo:
 [ 0]      pushl %ebp
 [ 1]      movl %esp,%ebp
 [ 3]      subl $52,%esp
 [ 6]      movl 8(%ebp),%edx
 [ 9]      addl 12(%ebp),%edx
 [ c]      movl %edx,-4(%ebp)
 [ f]      movl -4(%ebp),%eax
 [12]      jmp .L1
.align 4
.L1:
 [14]      leave
 [15]      ret

bar:
 [18]      pushl %ebp
 [19]      movl %esp,%ebp
 [1b]      subl $4,%esp
 [1e]      movl 8(%ebp),%edx
 [21]      addl $2,%edx
 [24]      movl %edx,-4(%ebp)
 [27]      pushl $12
 [29]      movl -4(%ebp),%eax
 [2c]      pushl %eax
 [2d]      call foo
 [32]      addl $8,%esp
 [35]      movl %eax,%eax
 [37]      movl %eax,-4(%ebp)
 [3a]      movl -4(%ebp),%eax
 [3d]      jmp .L2
.align 4
.L2:
 [40]      leave
 [41]      ret
```

Pour faciliter les explications, nous avons rétabli à la main les offsets des instructions (dans la marge, en hexadécimal) à l'aide d'un désassemblage par `objdump`.

Afin de mieux appréhender la suite, le shéma 4 suivant est utilisé. Ce schéma représente la configuration de la pile établie par `gcc` pour notre petit exemple et sera expliqué au fur et à mesure dans ce qui suit.

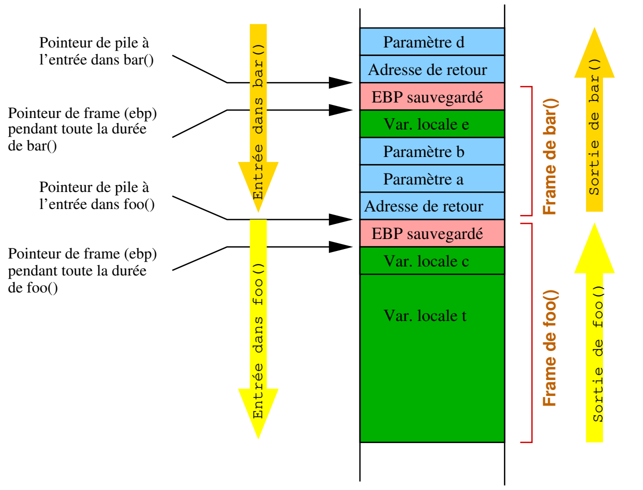


FIG. 4 – Utilisation de la pile par les fonctions `foo()` et `bar()`

2.3.1 Prologue de fonction

Repérage de la "frame" (optionnel). À chaque début de fonction (offsets 0-1 pour `foo` et 18-19 pour `bar`) on aperçoit les deux instructions `pushl %ebp` et `movl %esp, %ebp`. L'instruction `pushl %ebp` empile la valeur du registre `ebp` sur la pile, puis `movl %esp, %ebp` transfère la valeur du registre `esp` (le pointeur de pile) dans le registre `ebp` (le pointeur de *frame*).

Ainsi, à tout instant, le registre `%ebp` contient l'adresse du début de la frame pour la fonction en cours, et à cette adresse figure l'adresse du début de la frame de la fonction appelante. Cette caractéristique permet de pouvoir retracer quelle suite d'appels de fonction a mené jusqu'à la fonction en cours (nous en reparlerons en annexe A).

Le code ainsi généré est facultatif : il suffit d'appeler `gcc` avec l'option `-fomit-frame-pointer` pour qu'il disparaisse, auquel cas le registre `ebp` n'est plus utilisé par `gcc` : le compilateur se débrouillera en utilisant directement le pointeur de pile `esp`.

Allocation des variables locales dans la pile (systématique). Ensuite, `gcc` alloue les variables locales. Pour cela, il leur réserve de la place dans la pile par une simple soustraction sur le registre `esp`. Par exemple, dans `foo` les variables locales occupent

48 octets (pour le tableau) + 4 octets (pour l'entier), soit 52 octets, d'où la soustraction `subl $52,%esp` (offset 3). `bar` fait évidemment de même, mais pour 4 octets seulement (offset 1b).

Accès aux variables locales dans le reste de la fonction. Par la suite, les variables locales seront repérées relativement au registre `ebp` qui marque le début de la frame. Par exemple dans `bar`, la variable `e` est réservée juste après l'endroit où `ebp` avait été empilé, soit 4 octets après le début de la frame : ceci se traduit en assembleur ATT par "`-4(%ebp)`" dans les expressions comme `movl %eax,-4(%ebp)` (offset 37 : la variable `e` prend la valeur du résultat de la fonction `foo`). Dans le cas où on aurait compilé avec l'option `-fomit-frame-pointer`, `gcc` aurait repéré les variables locales relativement au registre `esp`.

Dans la suite, nous allons nous limiter au code de la fonction `bar`. La fonction `foo` suit évidemment le même principe. Sa présence nous permet surtout de présenter comment les frames correspondant à 2 appels de fonctions s'organisent dans la pile (cf. figure 4).

2.3.2 Cœur de la fonction

Repérage des arguments. Nous venons de présenter comment sont repérées les variables locales à une fonction. Pour repérer les arguments passés à la fonction par la fonction appelante, le mécanisme est similaire.

Sur `x86`, les arguments des fonctions sont empilés sur la pile[2] avant d'effectuer l'appel de fonction proprement dit (instruction `call`). Ceci veut dire que la fonction repère les arguments qui lui sont passés en regardant *avant* le début de la frame.

Autant `gcc` est libre de faire comme il le désire pour gérer les variables locales, autant il est obligé de respecter une norme pour le repérage des arguments passés aux fonctions. En effet, `gcc` doit être capable de compiler des programmes ou des bibliothèques de fonctions qui doivent pouvoir utiliser / être utilisés par du code généré par d'autres (versions de) compilateurs.

Dans notre cas (compilateur de Linux), les règles sont définies par l'ABI IA32 ("Application Binary Interface" pour l'"Intel Architecture 32bits") associée au standard "System V" [2]. Entre autres choses, cette ABI spécifie que les arguments passés à une fonction seront empilés dans l'ordre suivant : empilement du dernier argument d'abord, de l'avant-dernier ensuite, etc. jusqu'au premier argument. Le premier argument est ainsi l'argument le plus bas dans la pile.

Dans notre exemple, dans `foo`, l'argument `a` se traduit en assembleur par `8(%ebp)` et l'argument `b` par `12(%ebp)`. De même, dans la fonction `bar`, l'argument `d` s'écrit `8(%ebp)` (voir la traduction de "`e = d + 2`" en assembleur, à l'offset 21).

[2] En fait, certains arguments peuvent être passés dans les registres (mot clef `register` dans les prototypes de fonctions).

2.3.3 Appel de fonction

Examinons maintenant ce qui se passe lorsque `bar` appelle la fonction `foo`.

Mise en place des arguments. Pour réaliser l'appel à la fonction `foo`, il faut tout d'abord préparer les arguments passés à l'appel de `foo`, c'est-à-dire les empiler conformément à l'ABI IA32 System V : le dernier d'abord (offsets 27 à 2c).

Saut vers la fonction appelée (Offset 2d). Il s'agit de réaliser l'appel à la fonction proprement dit, *via* l'instruction `call` (voir [3, CALL]). Cette instruction commence par empiler l'adresse de l'instruction qui suit le `call` de manière à ce que la fonction appelée (ici `foo`) sache à quelle instruction `bar` devra être reprise. Ensuite, l'instruction `call` saute à l'adresse passée en paramètre (ici l'adresse du symbole `foo`). La séquence des 2 instructions qui suivent sont équivalentes au `call` :

```
   ...
   pushl $.L42
   jmp foo
.L42:
   ...
```

Nous ne détaillons pas la fonction appelée (`foo`). Considérons qu'elle s'exécute et qu'elle utilise l'adresse de retour empilée par le `call` pour que `bar` puisse poursuivre juste après le `call`.

Remise en place de la pile au retour de la fonction appelée. Une fois l'exécution de la fonction `foo` terminée, la fonction `bar` poursuit son exécution à l'offset `32`.

L'espace alloué sur la pile pour les paramètres de l'appel de fonction (8 octets) est libéré. Pour cela, il suffit d'incrémenter `esp` de 8 octets : `addl $8, %esp`. On remarquera que c'est la fonction appelante (`bar`) qui s'occupe de cette désallocation, c'est-à-dire celle qui a consommé la pile en empilant les arguments pour `foo`.

Ainsi, même si on se trompe dans le nombre d'arguments passés à `foo`, la fonction `bar` remettra la pile dans l'état où elle était avant l'appel, quoi qu'il arrive, même si `foo` attendait plus/moins d'arguments. La seule conséquence sera que `foo` aura une partie de ses arguments incorrects si elle en attend plus que ce que `bar` lui transmet.

Récupération du résultat de la fonction appelée. L'ABI IA32 System V précise que le résultat retourné par une fonction doit être contenu dans le registre `eax`.

Les instructions (offsets 35 et 37) qui suivent cet appel de fonction se chargent de stocker le résultat de la fonction appelée dans la variable locale (ici : `e`, i.e. `-4(%ebp)`).

2.3.4 Épilogue (i.e. mot-clef "`return`" en C)

Mise en place de la valeur à retourner. Conformément à l'ABI IA32 System V, pour retourner la valeur `e`, il faut la stocker dans `eax` (offset 3a). Ensuite (offset 3d), `gcc` demande à l'assembleur de sauter quelques octets plus loin (offset 3d) de sorte que l'instruction qui suit soit alignée sur 4 octets (directive `.align 4`). Ceci est une "optimisation" (inutile) de `gcc`.

Restauration de la pile et de la "frame" (i.e. désallocation des variables locales). A l'offset `40`, l'instruction `leave` (voir [3, LEAVE]) remet la pile dans l'état où elle était à l'entrée de la fonction `bar` et repositionne le pointeur de frame `ebp` pour retrouver la frame de la fonction appelante. Pour cela, elle réutilise ce qui avait été fait par le prologue (voir la section 2.3.1) et effectue les opérations exactement symétriques à celles du prologue, ce qui pourrait aussi s'écrire :

```
movl    %ebp,%esp
popl    %ebp
```

Retour dans la fonction appelante. Pour finir, à l'offset `41` l'instruction `ret` retourne à la fonction appelante en restaurant dans `eip` la valeur préalablement empilée par l'instruction `call` ayant appelé la fonction `bar`. Cette instruction serait équivalente à l'instruction suivante, qui est interdite sur `x86` : `popl %eip`.

2.4 Synthèse

Les figures 5 et 6 résument respectivement l'utilisation de la pile au fil des différentes étapes de l'appel et du retour d'une fonction.

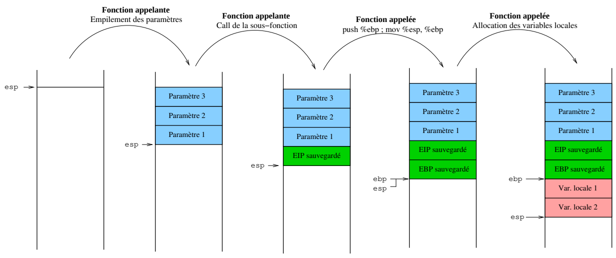


FIG. 5 – Utilisation de la pile lors de l'appel d'une fonction

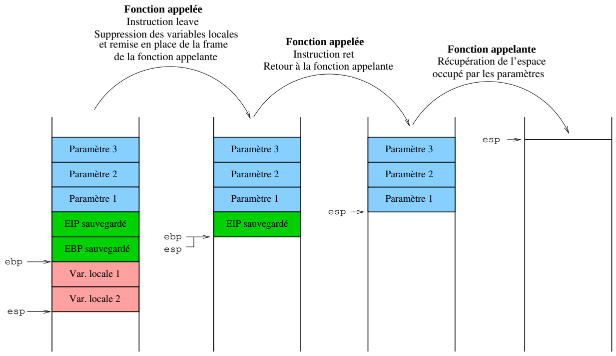


FIG. 6 – Utilisation de la pile lors du retour d'une fonction

On retiendra que l'exécution d'un appel de fonction se fait en utilisant une portion de la pile nommée *frame*. Cette notion de frame est purement logique, essentiellement interne au compilateur et au debogueur. Cependant, sur `x86`, le compilateur génère par défaut les instructions pour qu'elle soit accessible par le contenu du registre `ebp`, sauf si l'option `-fomit-frame-pointer` de `gcc` est utilisée.

Les variables locales d'une fonction appelée sont situées en dessous de l'adresse de la *frame* correspondante, tandis que les paramètres passés à la fonction sont situés au dessus de l'adresse de la *frame*.

3 Changement de contexte

Comme nous l'avions dit dans la section 1, pour réaliser du multitâche, c'est-à-dire l'exécution de tâches les unes à la suite des autres, il faut pouvoir passer d'une tâche à une autre. Pour cela, il faut sauvegarder l'état processeur du programme à interrompre quelque part en mémoire et restituer sur le processeur l'état d'un autre programme précédemment interrompu ou nouvellement créé. Comme nous l'avons dit, par "état du programme" ou "contexte d'exécution", nous entendons : les registres du processeur.

Certes, parmi ces registres figurent `esp` et `ebp` que nous venons de présenter. Mais cela n'explique pas pourquoi nous avons voulu présenter la pile et le changement de contexte dans le même article. La raison est simple : dans SOS, les contextes processeur des programmes interrompus sont sauvegardés... dans leur pile. Cela évite d'allouer des structures de données spéciales supplémentaires. En effet, un programme interrompu n'a pas besoin de sa pile donc on peut utiliser cette pile pour y stocker le contenu des registres. Dans ces conditions, un contexte d'exécution dans SOS correspond à la portion de la pile qui renferme la valeur des registres sauvegardés au moment du changement de contexte.

Dans un premier temps, nous considérons que les changements de contexte dans SOS sont volontaires. C'est-à-dire qu'il faut explicitement appeler la fonction de changement de contexte en précisant le contexte destination. Dans SOS, cette fonction s'appelle `sos_cpu_kstate_switch()`.

Toutes les fonctions que nous allons étudier ce mois-ci se situent dans `hwcore/cpu_context.h` pour les déclarations et dans `hwcore/cpu_context.c` et `hwcore/cpu_context_switch.S` pour les définitions. L'interface de programmation que nous définissons est assez proche de l'interface `makecontext()/swapcontext()` de SUSv2 ("Single UNIX Specification, Version 2") disponible avec les bibliothèques C GNU récentes. Il s'agit d'une version allégée des "coroutines" qu'on pourrait définir comme une sorte d'ancêtre des threads "modernes" (nous y reviendrons très rapidement à la fin de la section 4.3). Dans notre implantation, nous avons intégré un mécanisme de *stack poisoning* pour détecter l'usage de variables locales non initialisées ainsi qu'un mécanisme basique de détection des débordements de pile. Nous ne les détaillerons pas ici.

3.1 Implantation du changement de contexte dans SOS

Le passage d'un programme `from_context` à un programme `to_context` se déroule donc en trois étapes distinctes :

1. Sauvegarde sur la pile de `from_context` de la valeur des registres du processeur ;
2. Changement de pile pour « passer » sur la pile de `to_context` ;
3. Restauration depuis la pile de `to_context` de la valeur des registres du processeur.

C'est exactement ce que fait la fonction `sos_cpu_kstate_switch(from_context, to_context)` définie en assembleur dans `hwcore/cpu_context_switch.S`.

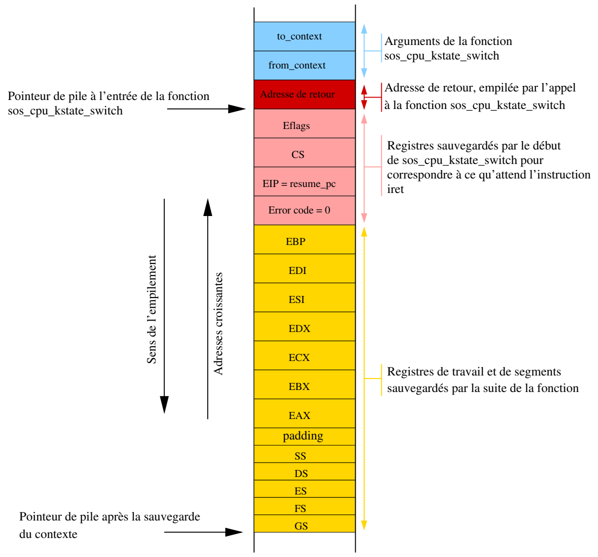


FIG. 7 – État de la pile à l'issue des empilements pour la sauvegarde du contexte

3.1.1 Sauvegarde du contexte d'exécution

Première étape : sauvegardes des registres. Il s'agit d'empiler la valeur de tous les registres, sauf le pointeur de pile `esp` (cela n'aurait en effet aucun intérêt car on aurait besoin du pointeur de pile pour retrouver le pointeur de pile sauvegardé...). Le code assembleur correspondant est :

```
sos_cpu_kstate_switch:
  pushf              // (eflags)
  pushl %cs          // (cs)
  pushl $resume_pc   // (eip)
  pushl $0           // (error code)
  pushl %ebp
  pushl %edi
  ...
  pushw %fs
  pushw %gs
```

On remarque que deux valeurs sont empilées artificiellement :

– "`$resume_pc`" : c'est l'adresse de la première instruction qu'exécutera le processeur une fois que ce contexte sera restauré sur le processeur. En effet, dans les conventions qu'on s'est fixées et que nous voyons plus loin, cette valeur est placée à l'endroit précis du contexte d'exécution qui sera transféré vers le compteur de programme (registre `eip`). Dans SOS, le symbole `resume_pc` est situé en toute fin de la fonction (nous y reviendrons plus loin).

– "`$0`" : ceci sert à traiter de façon uniforme tous les contextes d'exécution sauvegardés dans le système, que ceux-ci correspondent à des changements de contexte volontaires comme ici, ou à des programmes interrompus par des interruptions matérielles, logicielles, ou des exceptions. Nous y reviendrons en section 3.4.2.

La figure 7 représente le contenu de la pile utilisée par la fonction `sos_cpu_kstate_switch(from_context, to_context)` à l'issue de ces empilements.

Imaginons que nous dépilions maintenant ces valeurs vers les registres dans l'ordre inverse de leur empilement. On retrouverait alors exactement les valeurs que les registres avaient lors de l'appel de la fonction. Sauf pour le compteur de programme `eip` car, à la suite des dépilements, il aura été affecté à l'adresse du symbole `resume_pc` (voir ci-dessus), c'est-à-dire pour que l'instruction à exécuter après dépilement soit "`ret`" :

```
resume_pc:
  ret
```

Autrement dit, après les dépilements dans l'ordre inverse, nous nous retrouverions à la fin de la fonction, comme si elle n'avait rien fait. C'est exactement ce que nous souhaitons ! L'intérêt, c'est qu'entre la fin des empilements et les dépilements, nous pouvons librement exécuter un autre programme qui modifie tous les registres si besoin, ni vus ni connus par le programme appelant la fonction `sos_cpu_kstate_switch()`.

Deuxième étape : sauvegarde du pointeur de pile. Il ne manque rien ? Si, bien sûr : il faut stocker quelque part la valeur du pointeur de pile (`esp`) du contexte source si nous voulons pouvoir le restaurer ultérieurement, c'est-à-dire faire les dépilements à partir du bon endroit. C'est le rôle de l'argument `from_context` de `sos_cpu_kstate_switch()` : indiquer l'adresse où ce pointeur de pile doit être stocké.

Tout ce que nous avons expliqué précédemment sur le repérage des arguments passés aux fonctions s'applique ici. L'endroit où `esp` est à stocker correspond au premier argument (`from_context`) passé à la fonction, soit 4 octets avant le début de la frame.

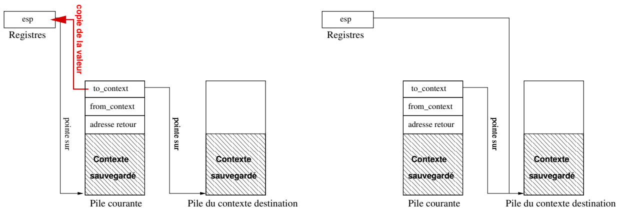
Pour situer le début de la frame, nous avons choisi de ne pas utiliser le registre `ebp`. Par conséquent, pour repérer les arguments, nous emploierons le pointeur de pile `esp`. Or celui-ci a été décalé par rapport au début de la frame suite aux empilements que nous venons d'effectuer. Après ces empilements (56 octets), le premier argument se situe donc à 56+4=60 octets au dessus du pointeur de pile. Une fois ceci connu, la sauvegarde de esp à l'adresse indiquée par cet argument s'écrit en deux instructions :

```
movl  60(%esp), %ebx
movl  %esp, (%ebx)
```

3.1.2 Changement de pile

Nous allons préparer le pointeur de pile du contexte destination sur le processeur pour qu'il ne reste plus que les dépilements à effectuer. Pour cela, il suffit de récupérer la valeur de `esp` sauvegardée pour le contexte à restaurer (figure 8). Cette valeur est transmise en tant que deuxième argument (`to_context`) à la fonction, c'est-à-dire 8 octets au dessus de la frame, soit à `esp+64` :

```
movl 64(%esp), %esp
```



(a) Pendant l'affectation de `esp`

(b) Juste après l'affectation de `esp`

FIG. 8 – Changement de pile.

Et voilà, nous sommes prêts à dépiler les registres du programme `to_context` !

3.1.3 Transfert du nouveau contexte sur le processeur

Il suffit de dépiler tous les registres dans l'ordre inverse de leur empilement :

```
popw %gs
popw %fs
...
popl %ebp
addl $4, %esp /* Ignore "error code" */

/* This restores the eflags, the cs
   and the eip registers */
iret /* equivalent to: popfl ; ret */
```

Comme on peut le voir dans ce code, tous les registres sont restaurés avec de simples dépilements sauf les trois registres `eflags`, `cs` et `eip` qui sont restitués en une seule instruction : `iret`. Nous avions vu cette instruction dans le cadre du traitement des interruptions (article 2). Nous détournons un peu son usage ici, mais son effet est identique.

Le rôle de cette instruction sera entre autres de sauter à l'adresse de l'instruction sauvegardée dans le contexte. Si le contexte avait été sauvegardé par `sos_cpu_kstate_switch()`, alors ie processeur saute au `ret` situé au niveau du symbole `resume_pc` (voir la section 3.1.1). Et si le contexte avait été sauvegardé par une interruption, alors le processeur saute à l'instruction où il avait été interrompu.

Voilà, nous avons sauvegardé le contexte d'un programme et transféré celui d'un autre sur le processeur.

3.1.4 Prototype de la fonction de changement de contexte

Nous venons de voir que la fonction `sos_cpu_kstate_switch()` est implantée en assembleur. Puisqu'elle doit être appelée depuis du code C, il s'agit de lui associer un prototype. Nous avons dit que cette fonction prend deux arguments :

1. L'adresse où la valeur pointeur de pile `esp` du contexte `from_context` doit être sauvegardée ;
2. La valeur du pointeur de pile `esp` du contexte "destination" `to_context`.

Autrement dit, le premier argument est un pointeur vers une adresse et le deuxième est une adresse. On aurait donc pu définir le prototype C suivant pour la routine assembleur : `sos_cpu_kstate_switch(sos_vaddr_t *from_context, sos_vaddr_t to_context)`. En fait, comme nous le voyons dans ce qui suit, ce n'est pas tant l'adresse de `esp` qui nous intéresse, que les valeurs des registres qui sont empilées dans la pile à cette adresse. C'est pourquoi nous écrivons `struct sos_cpu_kstate *` à la place de `sos_vaddr_t` (dans les deux cas, il s'agit bien d'une adresse). Le prototype de la fonction C s'écrit donc :

```
void sos_cpu_kstate_switch(struct sos_cpu_kstate **from_ctxt,
                           struct sos_cpu_kstate *to_ctxt);
```

Nous revenons sur ce `struct sos_cpu_state` un tout petit peu plus loin.

3.2 Changement de contexte sans sauvegarde

Dans SOS on définit la fonction `sos_cpu_kstate_exit_to()` qui est très proche de la fonction `sos_cpu_kstate_switch()` précédente. La différence est qu'elle n'effectue pas la sauvegarde du contexte du programme en cours d'exécution. Elle se contente de transférer le contexte d'un autre programme sur le processeur.

Cette fonction est destinée à être utilisée lorsque le programme qui l'appelle se termine : il faut en effet passer au contexte d'un autre programme sans se soucier de pouvoir revenir au contexte du programme courant désormais terminé. C'est pourquoi on fournit en plus à `sos_cpu_kstate_exit_to()` l'adresse d'une "fonction de rappel" (*callback* en anglais) qu'elle doit appeler. Ce callback (`reclaiming_func`) sera responsable de la libération des données allouées pour le contexte terminé (typiquement la pile) juste avant de passer au contexte destination.

Le prototype du callback est le suivant :

```
void sos_cpu_kstate_function_arg1_t(sos_ui32_t arg1);
```

L'argument qu'il prend est exactement celui qu'on fournit à `sos_cpu_kstate_exit_to()` (argument `reclaiming_arg`).

Nous verrons un exemple d'utilisation de `sos_cpu_kstate_exit_to()` dans la première démo (section 4.1).

3.3 Structure d'un contexte

3.3.1 Définition

Après l'empilement des registres pour la sauvegarde d'un contexte, la valeur du pointeur de pile `esp` est stockée quelque-part (voir la section 3.1.1). Elle est donc utilisable par le reste du système pour, par exemple, observer dans quel état était le programme interrompu. Sa valeur correspond à l'adresse du dernier empilement effectué pour sauvegarder le contexte. À cette adresse figure le contenu du registre `gs`. 16 bits plus haut (en adresse) figure la valeur du registre `fs`, etc. (voir la section 3.1.1, figure 7). Autrement dit on peut représenter ces empilements par la structure C suivante :

```
struct sos_cpu_kstate {
  /* (Lower addresses) */

  /* These are SOS convention */
  sos_ui16_t  gs;
  sos_ui16_t  fs;
  sos_ui16_t  es;
  sos_ui16_t  ds;
  sos_ui16_t  ss;
  sos_ui16_t  alignment_padding; /* unused */
  sos_ui32_t  eax;
  sos_ui32_t  ebx;
  sos_ui32_t  ecx;
  sos_ui32_t  edx;
  sos_ui32_t  esi;
  sos_ui32_t  edi;
  sos_ui32_t  ebp;

  sos_ui32_t  error_code;
  sos_vaddr_t eip;
  sos_ui32_t  cs;
  sos_ui32_t  eflags;

  /* (Higher addresses) */
} __attribute__((packed));
```

Le fait de représenter cette adresse de `esp` autrement que sous la forme d'une simple adresse permet d'accéder simplement (en C) à la valeur de tous les registres d'un contexte sauvegardé.

Mis à part pour les 4 derniers champs de la structure, l'ordre défini est une simple convention fixée par SOS. La seule contrainte est que cet ordre doit rester cohérent avec l'ordre des empilements/dépilements effectués dans les routines assembleur concernées. En ce qui concerne les trois derniers champs, ceux-ci doivent obligatoirement être à cet emplacement dans cet ordre pour pouvoir être exploités par l'instruction `iret` (voir la section 3.1.3). Nous reviendrons sur le quatrième champ "`error_code`" dans la section 3.4.1.

3.3.2 Utilisation

La structure `struct sos_cpu_kstate` est opaque à l'utilisateur : elle est définie et utilisée dans `hwcore/cpu_context.c` seulement.

Pour tous les autres fichiers source, seule l'adresse de la structure a un sens. Son contenu est en effet totalement spécifique à l'architecture `x86`. Dans un soucis d'écriture de code générique pour les autres fichiers, nous avons souhaité que ceux-ci n'aient pas à connaître les détails de cette structure.

Nous définissons cependant des fonctions d'accès aux champs de la structure `struct sos_cpu_kstate` qui ont un sens pour toutes les architectures et qui risquent d'être utiles à d'autres fichiers source :

sos_cpu_kstate_get_PC(ctxt) : récupère le compteur de programme (i.e. registre `eip` sur `x86`) du contexte sauvegardé `ctxt`,

sos_cpu_kstate_get_SP(ctxt) : récupère le pointeur de pile (i.e. registre `esp` sur `x86`) du contexte sauvegardé `ctxt`. Avec l'implantation que nous venons de détailler, cette valeur est exactement l'adresse de `ctxt`.

3.4 Caractéristiques des contextes d'exécution dans SOS

Il y a trois raisons de sauvegarder les registres d'un programme :

1. Pour effectuer un changement de contexte volontaire. C'est ce que nous avons détaillé avec la fonction `sos_cpu_kstate_switch()`.
2. Lorsqu'une interruption matérielle (IRQ) est traitée (voir l'article 2 et le fichier `hwcore/irq_wrappers.S`), à l'origine pour que les handlers d'interruption n'aient aucune influence sur les registres du programme interrompu.
3. Lorsqu'une exception processeur est traitée, pour les même raisons.
4. Lorsqu'une interruption logicielle est traitée, pour les même raisons.

3.4.1 Une structure identique pour tous les cas

Pour ces quatre cas, dans SOS les registres sont sauvegardés dans la pile. On s'est arrangé pour que la configuration de la pile soit strictement identique pour les quatre cas. Autrement dit, la structure `struct sos_cpu_kstate` représente correctement cette configuration de la pile dans les quatre cas.

De la sorte, depuis la fonction `sos_cpu_kstate_switch()`, on peut restaurer le contexte d'un programme sauvegardé par une exception processeur ou une interruption matérielle. De même, depuis un handler d'exception, on peut restaurer le contexte d'un programme interrompu par une exception ou qui a été sauvegardé par appel à `sos_cpu_kstate_switch()`. Toutes les autres combinaisons sont possibles.

Ceci explique aussi pourquoi nous avons rajouté le champ `error_code` dans la structure `struct sos_cpu_kstate` : parce que certaines exceptions empilent un code d'erreur dans la pile [4, Section 5.13]. Pour les autres interruptions et pour la fonction `sos_cpu_kstate_switch()`, la notion de code d'erreur n'a aucun sens, c'est pourquoi nous empilons artificiellement `0` à l'emplacement de ce champ.

3.4.2 Des handlers d'IRQ et d'exceptions qui peuvent accéder au contexte du programme interrompu

Nous avons modifié les caractéristiques des gestionnaires d'exception et d'IRQ (vus dans l'article 2) pour qu'ils puissent accéder à l'état des contextes interrompus. Désormais, les gestionnaires de premier niveau (en assembleur) passent un deuxième argument aux gestionnaires d'exception (écrits en C) : il s'agit d'un pointeur vers le contexte sauvegardé. Ceci leur permettra de réagir à l'interruption en fonction de l'état du contexte interrompu.

Pour cela, les prototypes d'un gestionnaire d'exception et d'IRQ ont été changés dans `hwcore/exception.h` et `hwcore/irq.h` :

```
typedef void (*sos_exception_handler_t)
   (int exception_number,
    const struct sos_cpu_kstate *cpu_kstate);

typedef void (*sos_irq_handler_t)
   (int irq_level,
    const struct sos_cpu_kstate *cpu_kstate);
```

3.5 Initialisation d'un nouveau contexte

Il s'agit de créer *ex-nihilo* un contexte en tous points identique à un contexte interrompu par un changement de contexte. Un nouveau contexte est donc lié à 2 choses :

1. Une pile. C'est cette pile qui contient la structure `struct sos_cpu_context` que nous voulons initialiser ici,
2. L'adresse de la première instruction a exécuter par le nouveau contexte une fois qu'il aura été transféré sur le processeur. En pratique, cette adresse correspond à celle d'une fonction.

Dans SOS, c'est la fonction `hwcore/cpu_context.c:sos_cpu_kstate_init()` qui est chargée de la préparation de la pile du nouveau contexte et d'initialiser la structure `struct sos_cpu_context` (située dans cette pile) convenablement.

3.5.1 Description générale

La fonction `sos_cpu_kstate_init()` prend de nombreux arguments pour permettre l'initialisation du contexte :

ctxt : L'adresse à laquelle sera retournée un pointeur sur le contexte, une fois initialisé. C'est cette information qui sera utilisée pour désigner le contexte lors des changements de contexte ;

start_func : L'adresse de la fonction correspondant au code à exécuter lorsque le contexte sera transféré sur le processeur ;

start_arg : L'argument à passer à la fonction `start_func`;

stack_bottom : L'adresse du bas de la pile à utiliser pour ce contexte ;

stack_size : La taille de la pile à utiliser pour ce contexte ;

exit_func : L'adresse de la fonction à exécuter lorsque la fonction `start_func` retournera. Cette fonction est censée ne jamais retourner ;

exit_arg : L'argument passé à la fonction `exit_func` lorsqu'elle sera exécutée.

3.5.2 Mise en place du nouveau contexte

La fonction `sos_cpu_kstate_init()` positionne le nouveau contexte en haut de la nouvelle pile désignée à la fonction par les arguments `stack_bottom` et `stack_size`. Tous les registres du nouveau contexte sont mis à zéro, sauf :

– les descripteurs de segments `cs`, `ds`... (voir l'article 2) qui sont positionnés à `SOS_SEG_KCODE` pour le segment de code et à `SOS_SEG_KDATA` pour les autres segments (ci-dessous une version volontairement raccourcie et fausse du code) :
```
(*ctxt)->cs  = SOS_SEG_KCODE;  /* Code */
(*ctxt)->ds  = SOS_SEG_KDATA); /* Data */
(*ctxt)->es  = SOS_SEG_KDATA;  /* Data */
(*ctxt)->ss  = SOS_SEG_KDATA;  /* Stack */
```
– le compteur ordinal (i.e. registre `eip` du contexte quand il sera transféré sur le processeur) qu'on affecte à l'adresse de la première instruction d'une fonction particulière : `core_routine()` :
```
(*ctxt)->eip = (sos_ui32_t)core_routine;
```
– le registre `eflags` qui est initialisé de telle sorte que le bit "Interrupt Flag" ou IF (voir l'article 2 et [1, section 3.4.4]) soit positionné. Cela signifie que le nouveau contexte autorisera les interruptions matérielles à l'interrompre :
```
(*ctxt)->eflags = (1 << 9); /* set IF bit */
```

3.5.3 Présentation de la fonction `core_routine()`

Cette fonction constitue les traitements que tous les nouveaux contextes exécuteront. Elle prend 2 callbacks en argument, ainsi que l'argument à passer à chacun de ces callbacks.

Le premier callback (`start_func`) correspond à la fonction principale du nouveau programme (analogue à la fonction `main` sous Unix) : il s'agit du même callback que celui passé en argument de `sos_cpu_kstate_init()`. Le deuxième callback (`exit_func`) correspond aux traitements à effectuer une fois que le callback précédent aura terminé (même remarque). Le prototype et le code de cette fonction sont alors tout simplement :

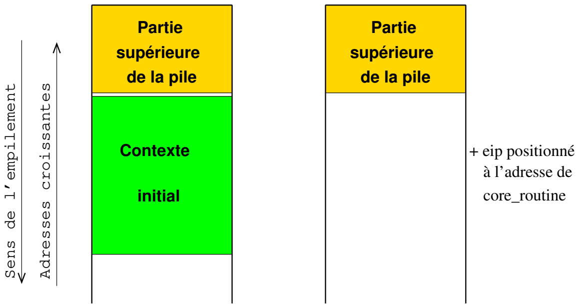
```
static void
core_routine (sos_cpu_kstate_function_arg1_t *start_func,
              sos_ui32_t start_arg,
              sos_cpu_kstate_function_arg1_t *exit_func,
              sos_ui32_t exit_arg)
{
  start_func(start_arg);
  exit_func(exit_arg);

  /* exit_func should NOT return ! */
  SOS_ASSERT_FATAL(! "ERROR");
}
```

Il est entendu que cette fonction `core_routine()` ne doit pas terminer, c'est-à-dire retourner vers aucune fonction appelante. Elle est en effet la première fonction du programme, donc aucune autre fonction dans le programme ne l'a appelée.

En pratique, soit le callback `start_func` appellera `sos_cpu_kstate_exit_to()` (voir la section 3.2) pour signaler la fin du programme, soit il retournera. Dans ce dernier cas, c'est au callback `exit_func` de signaler la fin du programme par un appel à `sos_cpu_kstate_exit_to()`. Dans ces conditions, `core_routine()` ne retournera pas, comme convenu.

3.5.4 Préparation de l'appel à `core_routine()`

Il reste à expliquer comment la pile du nouveau contexte est configurée pour que la fonction `core_routine()` reçoive ses 4 arguments au lancement du nouveau contexte. Pour cela, il suffit de se souvenir que, juste après le transfert du nouveau contexte sur le processeur, tout le contenu du nouveau contexte que nous venons d'initialiser aura été dépilé. Autrement dit, on passe de la configuration de la figure 9(a) à celle de la figure 9(b).



(a) Juste après l'initialisation par `sos_cpu_kstate_init()`

(b) Juste après que le nouveau contexte aura été transféré sur le processeur

FIG. 9 – Étapes précédent le début de l'exécution du nouveau contexte.

Les principales subtilités de la fonction `sos_cpu_kstate_init()` vont être de remplir convenablement la "partie supérieure de la pile" des figures précédentes. En effet, conformément à ce que nous avons décrit dans la section 2.3.3, la fonction `core_routine()` attend la configuration de la pile représentée sur la figure 10.

3.5.5 Est-ce la seule technique de changement de contexte ?

Dans l'article 7, nous verrons une autre technique de changement de contexte qui reposera sur un mécanisme intégralement pris en charge par le processeur. Elle complétera celle présentée ici en permettant de changer de niveau de privilège.

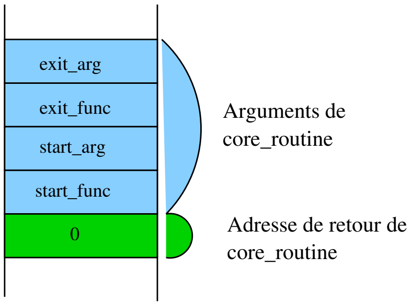


FIG. 10 – Contenu de la partie supérieure de la pile pour que `core_routine()` soit correctement appelée.

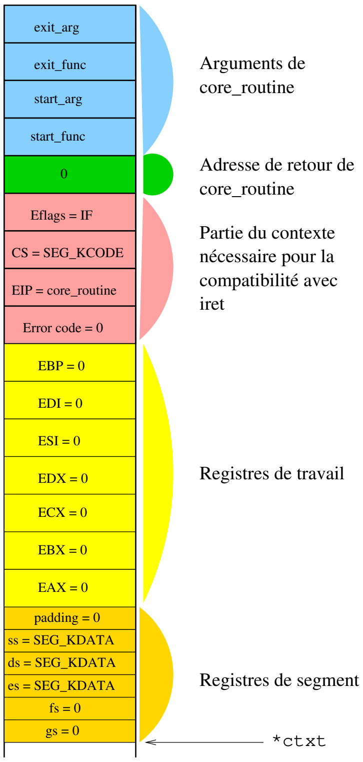


FIG. 11 – Allure de la pile suite à l'exécution de `sos_cpu_kstate_init`

Sinon il existe quantités d'autres techniques de changement de contexte dont le principe est identique à celle-ci. Certaines ne sauvegardent pas les contextes sur la pile, mais ailleurs en mémoire, dans une structure allouée spécialement pour cela (cas de Linux). D'autres ne reposent pas sur une fonction analogue à `sos_cpu_kstate_switch()` : ils remarquent que les gestionnaires d'interruption ont un fonctionnement qui correspond à ce qui est nécessaire (sauvegarde de registres, gestion de l'interruption, restauration d'un[autre] contexte). Ils passent donc par une interruption logicielle pour réaliser le changement de contexte.

4 Testons !

Ce mois-ci, nous proposons pas moins de 3 démos, dont 1 assez grosse que nous n'expliquerons pas dans le détail (la troisième). L'ensemble de ces démo est disponible dans `sos/main.c`.

4.1 "Hello world" à 2 programmes

Cette démo simple permet de montrer comment on se sert des contextes d'exécution.

Nous allons créer 2 "programmes" qui, ensemble, vont afficher la chaîne "`Hello world\n`" sur le port `0xe9` de bochs. Chaque programme affiche 1 caractère puis "passe" le processeur à l'autre programme qui va afficher le caractère suivant puis va "repasser" le processeur au premier, etc. Ce ping-pong entre les 2 programmes s'arrête quand un des programmes rencontre le "retour chariot" (`'\n'`), auquel cas il y a changement de contexte vers le contexte "primordial", i.e. celui de `sos_main()`. La figure 12 représente le chronogramme de cette petite démo.

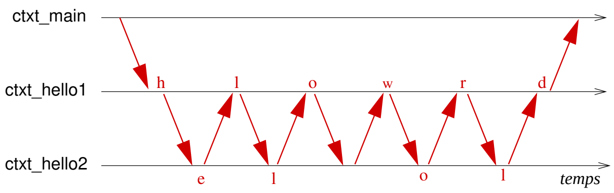


FIG. 12 – Chronologie des changements de contexte.

4.1.1 Le code des 2 programmes

Le code du premier programme correspond à :

```
static void hello1 (char *str)
{
  for ( ; *str != '\n' ; str++)
    {
      sos_bochs_printf("hello1: %c\n", *str);
      sos_cpu_kstate_switch(& ctxt_hello1, ctxt_hello2);
    }
}
```

Le principe est simple : on lui passe en argument la chaîne qu'il doit imprimer (i.e. `"Hlowrd"` pour le premier programme). Après chaque caractère qu'il imprime, il effectue un changement de contexte vers le deuxième programme. Le code du deuxième programme est comparable, si ce n'est que la chaîne qu'on lui passe en argument est `"el ol\n"` et qu'il effectue le changement de contexte dans le sens inverse (i.e. vers le contexte du programme 1).

4.1.2 Mise en place des deux programmes

La mise en place des contextes des programmes par `sos_main()` nécessite d'allouer une pile pour chaque programme. Puis il faut initialiser chacun des deux contextes en précisant à la fois quelle fonction ils doivent exécuter (voir ci-dessus), l'argument qu'on passe à la fonction (i.e. le morceau de chaîne que chaque programme doit afficher) et également quelle fonction doit être appelée lorsque le programme se termine (`exit_hello12()`). Par exemple, pour la mise en place du premier programme, on doit écrire :

```
struct sos_cpu_kstate *ctxt_hello1;
struct sos_cpu_kstate *ctxt_hello2;
struct sos_cpu_kstate *ctxt_main;
sos_vaddr_t hello1_stack, hello2_stack;
...

#define DEMO_STACK_SIZE 1024
  /* Allocate the stack */
  hello1_stack = sos_kmalloc(DEMO_STACK_SIZE, 0);

  /* Initialize the coroutines' contexts */
  sos_cpu_kstate_init(&ctxt_hello1,
                      (sos_cpu_kstate_function_arg1_t*) hello1,
      (sos_ui32_t) "Hlowrd",
                      (sos_vaddr_t) hello1_stack, DEMO_STACK_SIZE,
                      (sos_cpu_kstate_function_arg1_t*) exit_hello12,
      (sos_ui32_t) hello1_stack);
```

4.1.3 Lancement du ping-pong

La suite consiste à faire un changement de contexte vers le premier programme pour amorcer le ping-pong :

```
  /* Go to first coroutine */
  sos_bochs_printf("Printing Hello World\\n...\n");
  sos_cpu_kstate_switch(& ctxt_main, ctxt_hello1);

  /* The first coroutine to reach the '\n' switched back to us */
  sos_bochs_printf("Back in main !\n");
```

On remarquera qu'on a pu "utiliser" le contexte primordial `ctxt_main` (i.e. celui de la fonction `sos_main()`) sans l'avoir initialisé comme les deux autres. En effet, nous sommes en train d'exécuter du code quand nous appelons `sos_cpu_kstate_switch()`, ce qui définit le contexte d'exécution primordial : pas besoin de l'initialiser puisqu'il existe déjà.

Sous bochs/qemu, le ping-pong devrait afficher :

```
Printing Hello World\n...
hello1: H
hello2: e
hello1: l
hello2: l
hello1: o
hello2:
hello1: w
hello2: o
hello1: r
hello2: l
hello1: d
Back in main !
```

4.1.4 Terminaison des programmes

Pour finir, revenons sur la fonction `exit_hello12()`. Cette fonction est passée en argument de `sos_cpu_kstate_init()` pour être appelée lorsque les programmes se terminent (fin de `hello1()` par exemple) avec en paramètre l'adresse de la pile du programme qui s'est terminé :

```
static void exit_hello12(sos_vaddr_t stack_vaddr)
{
  sos_cpu_kstate_exit_to(ctxt_main,
 (sos_cpu_kstate_function_arg1_t*) reclaim_stack,
 stack_vaddr);
}
```

Cette fonction se contente de changer de contexte sans sauvegarder celui qui se termine (voir la section 3.2). Le contexte destination est celui de la fonction `sos_main()`, ce qui marque donc la fin du ping-pong pour l'affichage de "Hello world". La fonction demande aussi d'appeler la fonction `reclaim_stack()` pour désallouer la pile passée en paramètre :

```
static void reclaim_stack(sos_vaddr_t stack_vaddr)
{
  sos_kfree(stack_vaddr);
}
```

Et voilà ! Ainsi, le premier programme qui se terminera, ou même qui appellera "manuellement" `sos_cpu_kstate_exit_to()`, verra sa pile désallouée automatiquement. La pile de l'autre programme ne sera pas désallouée, ceci est laissé à titre d'exercice.

4.2 Allocation de pages à la demande

Notre deuxième démo présente la mise en place d'un mini-système de *demand paging*. Le *demand paging* consiste à allouer à la volée la mémoire physique nécessaire lorsque des *défauts de page* surviennent lors de l'accès à une zone de mémoire. C'est une technique habituelle utilisée dans le systèmes d'exploitation classiques car elle permet de n'allouer que la mémoire physique réellement utilisée. Dans notre exemple, on alloue 916 Mo d'espace de mémoire virtuelle alors que seuls 2 Mo seront réellement utilisés (voir la figure 13).

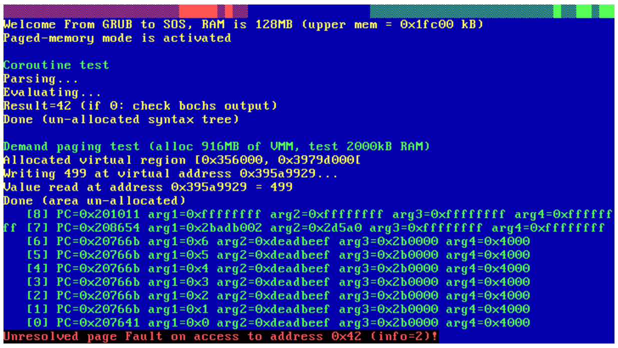


FIG. 13 – Aperçu de la petite démo

4.2.1 Gestionnaire d'exception

D'abord nous devons mettre en place un gestionnaire d'exception pour l'exception *défaut de page*. Celui-ci se chargera d'allouer une page physique et de la mapper correctement en mémoire virtuelle avant de relancer l'exécution de l'instruction ayant généré l'exception.

Ce gestionnaire est la fonction `pgflt_ex` du fichier `sos/main.c`. En premier lieu, elle récupère l'adresse de l'accès mémoire qui a provoqué l'exception en utilisant la fonction `sos_cpu_kstate_get_EX_faulting_vaddr()`.

Ensuite, le gestionnaire de l'exception *défaut de page* utilise la fonction `sos_kmem_vmm_is_valid_vaddr()` pour déterminer si l'adresse virtuelle fautive est située ou non dans une région mémoire valide. Si ce n'est pas le cas, alors l'état du contexte interrompu, passé en paramètre du gestionnaire de l'exception (voir la section 3.4.2), est affiché (il s'agit d'un backtrace, voir l'annexe A) et le système est arrêté. On dit que la "faute de page" est "non résolue",

Sinon, la fonction alloue une page physique en utilisant `sos_physmem_ref_physpage_new()`, puis la mappe à l'adresse fautive en utilisant `sos_paging_map()`. Chaque page physique allouée par le gestionnaire incrémente un compteur affiché en rouge en haut à gauche de l'écran.

À la sortie de ce gestionnaire, l'exécution du code interrompu reprend là où il s'était arrêté. L'instruction fautive ne déclenche plus le défaut de page puisque l'accès mémoire est maintenant valide.

4.2.2 Utilisation

La fonction `test_demand_paging()` permet de tester ce mécanisme en réalisant plusieurs accès en écriture au sein d'une zone de mémoire virtuelle en *demand paging*. Cette zone de mémoire de 916 Mo environ est allouée en utilisant `sos_kmem_vmm_alloc()`. Le deuxième paramètre à 0 indique que l'on ne souhaite pas que de la mémoire physique soit immédiatement associée à la région virtuelle dès sa création, ceci afin de tester le *demand paging* et aussi parce que peu de machines disposent réellement de plus de 916 Mo de RAM.

La figure 14 résume le déroulement des opérations lors d'un *défaut de page*.

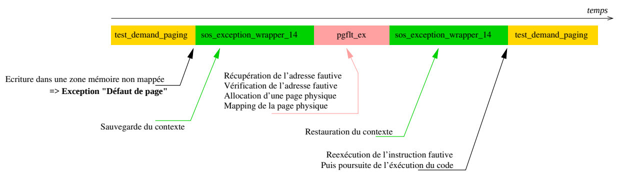


FIG. 14 – Déroulement de l'exécution de `test_demand_paging` lors d'un *défaut de page*

Pour finir, on choisit d'écrire à l'adresse virtuelle `0x42` depuis une fonction récursive (`test_backtrace()`). Ceci a pour conséquence de provoquer une exception "faute de page". Mais l'adresse virtuelle `0x42` n'est couverte par aucune région de mémoire virtuelle. C'est pourquoi le gestionnaire d'exception arrête le système en affichant la chaîne d'appels de fonctions (voir l'annexe A) qui a conduit à la faute (voir le bas de la figure 13).

4.3 Pipeline pour l'évaluation d'expressions arithmétiques

Le but de cette démonstration est d'analyser et d'évaluer des expressions arithmétiques entières du type "1+toto*(42+y)" en utilisant une méthode un peu originale faisant intervenir 3 programmes. Étant donné qu'on calcule avec des entiers, il y aura des erreurs d'approximation (par exemple : 1/2 vaudra 0).

On utilise :

- un programme (`reader`) qui lit caractère par caractère le contenu de la chaîne,
- un programme (`lexer`) qui consomme chacun de ces caractères pour extraire les composantes syntaxiques de l'expression (chiffre 1, opérateur '+', nom de variable "toto", parenthèse ouvrante, etc.),
- un programme (`parser`) qui consomme chacune de ces composantes syntaxiques pour construire l'arbre syntaxique habituel (la grammaire est fournie dans le source).

Pour faire l'analogie avec le "pipe" des shells Unix, on peut représenter l'articulation entre ces programmes par "`reader | lexer | parser`". Sauf qu'ici c'est chaque programme qui décide précisément à quel autre programme il "passe" le processeur et non le système d'exploitation sous-jacent. Nous fournissons un quatrième programme, `eval`, chargé de parcourir l'arbre syntaxique fourni par le `parser` pour effectuer l'évaluation de l'expression initiale.

Notons qu'on aurait pu tout faire avec des appels de fonctions normaux. Mais, en utilisant des contextes différents du contexte primordial (celui de `sos_main()`), il nous a été possible d'utiliser des piles beaucoup plus grandes que celle de 16 ko disponible par défaut pour `sos_main()`. Ceci est intéressant essentiellement parce que les programmes `parser` et `eval` sont récursifs, donc gros consommateurs de pile.

Nous ne détaillons pas davantage cette démo ici.

Nous dirons juste que nous parlons beaucoup de "coroutines" dans le code. Une coroutine peut être vue comme une généralisation des appels de fonction. En effet, la relation "appel de fonction" est par définition asymétrique : une fonction appelante choisit quelle fonction elle peut appeler, mais une fonction appelée ne choisit pas vers quelle fonction elle peut retourner. À l'inverse, une coroutine appelée par une autre peut décider de redonner le processeur à qui elle veut, y compris à une coroutine qui n'a rien à voir avec celle qui l'a appelée. Une coroutine est en quelque sorte un ancêtre des "threads", mais avec un "ordonnancement" manuel (nous reviendrons sur ces termes dans le prochain article). Les contextes d'exécution tels que nous les avons présentés sont une version un peu allégée de ces coroutines.

Conclusion

Cet article nous a permis de détailler la manipulation de la pile dans l'architecture `x86` en partant d'un exemple concret de code généré par `gcc`.

Nous avons ensuite présenté une notion qui n'a *a priori* rien à voir avec la pile : le changement de contexte qui pose les bases pour faire du multitâche. La technique de changement de contexte retenue dans SOS repose exclusivement sur la pile.

Le mois prochain, le deuxième volet de cette "6ème étape", que nous avons décidé d'appeler "Article 6 et demi", présentera la gestion de tous les contextes présents dans le système. Nous quitterons les considérations bas niveau pour nous placer à un niveau d'abstraction plus élevé : celui des threads (noyau) et de l'ordonnancement.

Voilà, nous espérons que vous n'avez pas trop pris la pile en pleine face. Il est maintenant temps pour vous de changer de contexte et d'aller reprendre une activité normale. Par exemple une activité à base de galette, de reines, de rois et de cidre(s).

Dernières nouvelles : *via* la liste de diffusion de SOS, Xavier Grave nous a annoncé que son projet "Xada", une version de SOS entièrement récrite en Ada, s'appelle désormais "Toy Lovelace". Á l'heure où ces lignes sont écrites, Xavier est en train de s'inspirer de l'article 5 de SOS. Et il semble s'orienter petit à petit vers un portage sur PowerPC ! Pour plus de détails, n'oubliez pas d'aller faire un tour sur le site de SOS `http://sos.enix.org`. D'ailleurs, allez-y régulièrement car nous y mettons les *erratas* (page "Bugs") des articles et les patches associés.

A *Backtrace*

A.1 Présentation

Les amateurs de `gdb` connaissent très certainement sa commande "`bt`" (ou *backtrace* en toutes lettres). Elle permet de connaître la chaîne d'appels de fonctions ayant (par exemple) mené à un bug et de voir la valeur les arguments qui étaient passés aux fonctions :

```
# gdb ~/a core
GNU gdb 6.1-debian
...
#0  g (i=11) at /home/d2/a.c:7
7       int g(int i) { return *((int*) 0x42); }
(gdb) bt
#0  g (i=11) at /home/d2/a.c:7
#1  0x080483fa in f (i=8, j=9, k=10) at /home/d2/a.c:6
#2  0x080483e6 in e (i=6, j=7) at /home/d2/a.c:5
#3  0x080483c2 in d (i=1) at /home/d2/a.c:4
#4  0x080483a6 in c (i=3, j=4, k=5) at /home/d2/a.c:3
#5  0x08048392 in b (i=1, j=2) at /home/d2/a.c:2
#6  0x0804836e in a (i=0) at /home/d2/a.c:1
#7  0x08048422 in main () at /home/d2/a.c:11
```

Dans cet exemple, on détermine très rapidement que le bug s'est produit dans la fonction `g`, elle-même appelée depuis la fonction `f` avec l'argument `11`, elle-même appelée depuis la fonction `e` avec les arguments `8, 9, 10`, etc. jusqu'à la fonction `main`.

Nous proposons ici de démystifier la magie qui se cache derrière cette commande.

A.2 Rappels sur la configuration de pile établie par gcc

Nous allons rappeler une partie de ce qui a déjà été dit dans la section 2. gcc génère le code nécessaire pour que le pointeur de frame (registre ebp) contienne à tout moment l'adresse du début de la frame de la fonction courante. Et à cette adresse, gcc place une sauvegarde du pointeur de frame de la fonction appelante. Rappelons que ceci n'est plus valable si on utilise l'option -fomit-frame-pointer de gcc.

La figure 15 suivante résume les éléments présents dans la pile qui peuvent être utiles pour le *backtrace*.

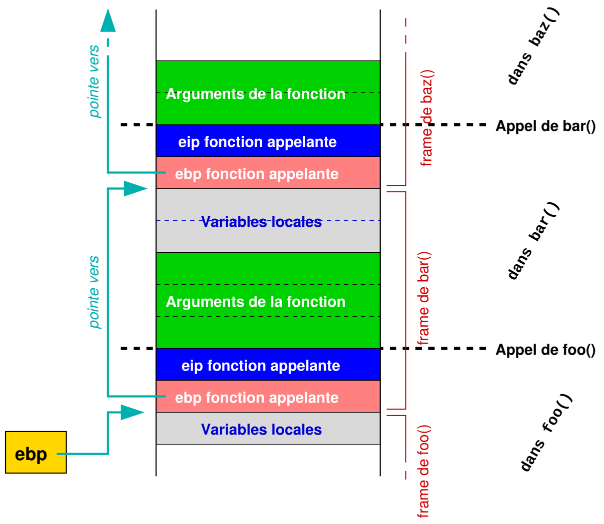


FIG. 15 – Éléments remarquables dans la pile pour l'implantation du backtrace. Exemple avec la fonction baz qui appelle bar qui appelle foo.

De cette figure, nous pouvons établir les règles suivantes :

1. Quand on connaît ebp, pour accéder aux arguments passés à la fonction associée, il suffit de regarder les données à partir de l'adresse valeur(ebp) + 8 octets.
2. Quand on connaît ebp, pour savoir dans quelle fonction on a été appelé, il suffit d'aller regarder l'entier situé à l'adresse valeur(ebp) + 4 octets.
3. Quand on connaît ebp associé à une fonction, pour connaître la valeur de ebp de la fonction appelante, il suffit de regarder l'entier situé à l'adresse valeur(ebp).

La troisième règle définit le principe de récurrence qui permet de récupérer l'adresse de tous les ebp de toutes les fonctions dans la chaîne d'appels.

A.3 Principe de l'implantation dans SOS

C'est la fonction sos_backtrace() qui permet de remonter la chaîne d'appels de fonction, soit à partir de la fonction courante, soit à partir d'un contexte processeur fourni en argument. Elle reprend le principe de récurrence précédent et appelle un callback à chaque étape. Le callback est fourni par l'utilisateur et peut par exemple afficher chaque appel de fonction et ses arguments.

Nous ne donnons pas son code ici, juste l'idée générale de son algorithme :

ALGO 1 Principe du fonctionnement du backtrace.

frame_addr := ebp initial
PC_appelé := eip
TANT QUE frame_addr contenu à l'intérieur de la pile **FAIRE**
  adresse_dans_la_fonction := PC_appelé
  adresse_arguments := frame_addr + 8
  Appel de callback(adresse_dans_la_fonction, adresse_arguments)

  / Remonte à l'appel de fonction précédent */*
  PC_appelé := valeur_à_l_adresse(frame_addr + 4)
  frame_addr := valeur_à_l_adresse(frame_addr)
FIN TANT QUE

Le callback est libre de faire ce qu'il veut mais il doit avoir le prototype suivant (l'algorithme précédent ne le reflète pas) :

```
void sos_backtrace_callback_t(sos_vaddr_t PC,
                              sos_vaddr_t params,
                              sos_ui32_t depth,
                              void *custom_arg);
```

Autrement dit, pour chaque appel de fonction dans la chaîne d'appels, on lui passe l'adresse de l'instruction où la fonction s'apprêtera à être reprise, l'adresse des paramètres, l'indice de profondeur dans la chaîne d'appels (0 : fonction courante, 1 : fonction appelante, etc.) et une adresse fournie par l'utilisateur au moment de l'appel à sos_backtrace().

A.4 Exemple d'utilisation

Pour illustrer l'utilisation de cette fonction sos_backtrace(), on pourra se reporter à la fonction sos/main.c:dump_backtrace(). Le callback utilisé est une fonction qui affiche le backtrace à la fois sur le port 0xe9 de bochs et sur la console. Celui-ci affiche également les 4 premiers arguments de chaque fonction appelée, même si cette dernière possède plus ou moins de 4 arguments en réalité (sur x86 il n'y a aucune moyen de savoir combien d'arguments ont été réellement passés).

Cette fonction dump_backtrace() est en particulier appelée dans le gestionnaire d'exception "faute de page" (voir la section 4.2.2) pour afficher la chaîne d'appels ayant conduit à un défaut de page non résolu. On obtiendra par exemple la chaîne d'appels suivante :

```
[0] PC=0x207641 arg1=0x0 arg2=0xdeadbeef arg3=0x2b0000
[1] PC=0x20766b arg1=0x1 arg2=0xdeadbeef arg3=0x2b0000
[2] PC=0x20766b arg1=0x2 arg2=0xdeadbeef arg3=0x2b0000
[3] PC=0x20766b arg1=0x3 arg2=0xdeadbeef arg3=0x2b0000
[4] PC=0x20766b arg1=0x4 arg2=0xdeadbeef arg3=0x2b0000
[5] PC=0x20766b arg1=0x5 arg2=0xdeadbeef arg3=0x2b0000
[6] PC=0x20766b arg1=0x6 arg2=0xdeadbeef arg3=0x2b0000
[7] PC=0x208654 arg1=0x2badb002 arg2=0x2d5a0 arg3=0xffffffff
[8] PC=0x201011 arg1=0ffffffff arg2=0xffffffff arg3=0xffffffff
Unresolved page Fault on access to address 0x42 (info=2)!
```

On s'aidera ensuite de la sortie de objdump -S sos.elf pour savoir dans quelle fonction se situent les adresses 0x207641, 0x20766b, etc.

Note : si on demande à `gcc` d'optimiser le code généré (options `-OX`), il transformera la fonction récursive `test_backtrace()` qui nous sert de test pour afficher ce qui précède. Cette fonction est originellement récursive et il en fait une fonction itérative. Ceci a pour conséquence que la chaîne d'appel sera beaucoup moins profonde.

The end.

David Decotigny et Thomas Petazzoni
`d2@enix.org` et `Thomas.Petazzoni@enix.org`
Merci à Nessie pour sa relecture, ses remarques constructives et ses propositions.
Site de SOS : `http://sos.enix.org`
Projet KOS : `http://kos.enix.org`

Á William Henry

Références

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

[2] System V Application Binary Interface - Intel386 Architecture Processor Supplement, Fourth Edition. `http://www.caldera.com/developers/devspecs/abi386-4.pdf`.

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

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