RENPAR/CFSE/SYMPA
Paris, cité des sciences et de l'industrie, 24 – 27 avril 2001

Utilisation de la segmentation mémoire du processeur à partir d'un langage de haut niveau

Benoît Sonntag

INRIA-CCH-Université Henri Poincaré,
LORIA, équipe Mirò, Campus scientifique, BP 239, 54506 Vandoeuvre-lès-Nancy - France
bsonntag@loria.fr

Résumé
Cet article est le fruit d'une étude sur la mise en place d'un nouveau système d'exploitation entièrement en objets. En effet, à partir de l'analyse de nos besoins en matière de communication pour l'élaboration de nos mécanismes objets, différents problèmes sont apparus. Ici, nous en dénonçons un qui concerne le manque de flexibilité de la gestion mémoire par l'absence d'utilisation de la segmentation. Actuellement, certains processeurs offrent des mécanismes élaborés de gestion de la mémoire par segmentation. Mais malheureusement, ils sont inutilisables dans le contexte d'un programme C ansi par exemple. A première vue, la mise en place d'un système d'exploitation utilisant ce mécanisme nécessite de réécrire complètement le compilateur pour que ce dernier prenne en compte les indexations d'adressage mémoire. Cet article apporte une solution simple et efficace pour permettre la prise en charge de la segmentation processeur d'un programme C en mémoire. Cette solution ne nécessite aucune modification massive du compilateur.

Mots-clés : Segmentation, mémoire virtuelle, compilateur, système d'exploitation, communication inter-processus

1. Introduction

Certains processeurs actuels offrent des mécanismes élaborés de segmentation comparables à ceux de *Multics* [3]. La raison essentielle de la quasi inexistence de ces mécanismes dans de nombreux systèmes d'exploitation est causée par les compilateurs. Pourtant ce mécanisme permet d'apporter une meilleure gestion de la mémoire et de ses protections. De plus, nous connaissons depuis longtemps (cf. Multics, 1972) les possibilités qu'elle offre en matière de communication inter-processus, et la facilité qu'elle apporte pour l'implantation de la mémoire partagée. Sans perte de généralité, notre étude est faite sur l'implantation de la segmentation d'un processeur hautement répandu : la famille des processeurs Intel 80386 et ses versions supérieures. Nous profitons des qualités de la segmentation pour la mise en place d'un nouveau système d'exploitation, *Isaac*, notre sujet de recherche. Le projet *Isaac* a pour but d'étudier et d'intégrer des concepts objets au cœur du système d'exploitation lui-même. Néanmoins, la solution que nous apportons pour l'utilisation de la segmentation dans un programme C reste valide pour un autre système. Cette solution nous paraît intéressante et originale dans sa simplicité d'implantation.

Nous commençons dans la section 1.1 à rappeler certains éléments de la gestion mémoire d'Unix. Ensuite, la section 1.2 présente notre projet de système d'exploitation ainsi que notre utilisation de la segmentation du processeur. La section 1.3 détaille le problème de l'indexation mémoire dans les langages de haut niveau. La section 2 présente notre méthode suivie, en section 3, de mesures d'efficacité. Des travaux connexes sont présentés en section 4, avant de conclure en section 5.

1.1. Rappel sur la gestion mémoire de Linux sur une architecture segmentée

Dans un système UNIX, un processus occupe une grande partie de l'espace virtuellement adressable ce qui ne facilite pas la communication et la mise en place de mémoire partagée entre plusieurs processus [5] (voir figure 1).

En effet, l'implantation actuelle en mémoire d'un programme C dans une architecture segmentée n'utilise pas la segmentation du processeur, où plus exactement, utilise le même segment pour la pile et les données. Il est clair que cette utilisation du processeur va à l'encontre des objectifs et de l'attente du constructeur. Cette disposition ne permet pas de détecter de manière efficace le débordement de la pile dans le tas. Dans *Linux*, la solution actuelle consiste à utiliser un segment virtuellement et anormalement grand pour empêcher le recouvrement dans les limites de taille qu'impose le système à un processus. Chaque processus se voit forcé d'utiliser les mêmes emplacements virtuels de mémoire, sollicitant ainsi l'utilisation abusive de table de pagination. En effet, chaque processus possède sa propre table de pagination, et donc son propre espace linéaire virtuel de 4Go. La taille d'une table de pagination peut atteindre jusqu'à 4Mo de mémoire vive.

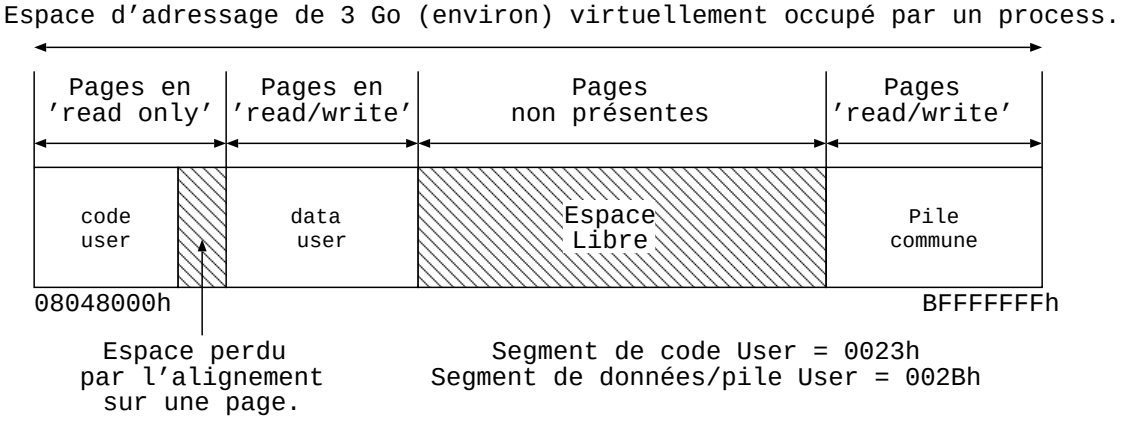

Figure 1: Vision de l'espace adressable d'un processus dans *Linux/Intel*.

Par ailleurs, nous pouvons aussi remarquer que la protection en écriture du code dans *Linux* sur une architecture segmentée de type *Intel 386* n'est pas faite au niveau du segment de code comme le suggère *Intel* [4] : le segment de code est un alias sur le segment de données/pile. De ce fait, rien n'empêche au niveau de la segmentation d'utiliser le segment de données pour écrire dans le code (voir figure 1). Néanmoins, une protection en écriture existe au niveau des pages mémoires contenant le code. Cette dernière remarque implique l'alignement du code sur la taille d'une page. En moyenne, la moitié de la dernière page allouée est libre et perdue (fragmentation interne). S'il y a n segments en mémoire et si la taille des pages est de p octets, la fragmentation fait perdre $np/2$ octets. En prenant en considération que le segment de code n'est pas extensible, nous pourrions imaginer un système d'allocation qui prend en compte cela et concatène à l'octet près l'ensemble des segments non extensibles. La protection inter-processus serait maintenue par la segmentation et la fragmentation interne disparaîtrait.

1.2. L'utilisation de la segmentation du processeur dans le système d'exploitation Isaac

Notre problème de segmentation nous a été posé dans le cadre de la mise en place d'un nouveau système d'exploitation *Isaac* [6] basé sur la notion de partage d'objets. Dans cette partie, nous allons présenter la problématique dans le cadre de notre projet.

Comme dans le système Mach [8], nous mettons l'accent sur la nécessité de partage de zone mémoire entre différents processus légers. Par exemple, dans le problème classique de producteur/consommateur, on peut envisager que le producteur et le consommateur soient des processus différents qui se partagent un tampon de données. Notre approche est d'étendre ce partage en une coopération plus fine entre différents processus de petite taille. Nous matérialisons cette coopération au sein de notre système par l'utilisation d'objets dynamiques, homogènes et rapides, qui fonctionnent directement sur le matériel. Aucune autre couche système ne doit être requise pour la mise en place de ces objets et aucune autre entité que l'objet ne doit être présente dans notre système. Cela fait toute l'originalité et la puissance du projet.

Notre étude soulève de nombreux problèmes liés essentiellement aux deux contradictions suivantes : flexibilité versus performances et communication versus protection système. De plus, une certaine compatibilité avec l'existant n'est pas à négliger et doit être prise en compte.
Parmi les concepts objets existants, nous avons choisi d'implanter ceux présents dans les langages à base de prototype. Notre système se constitue de petits exécutables représentant chacun un prototype d'objet. Chaque objet pourra communiquer avec d'autres par héritage ou par envoi de message à la manière des langages à objet comme Self [10]. Néanmoins, il nous semble utile de préciser que la granularité de nos objets est moins fine que dans ce type de langages possédant des objets de type *entier* ou *boolean*. En effet, dans notre système d'exploitation, nous représentons une entité physique du matériel par un seul objet (exemple : un disque dur, un écran, un clavier). Dans le cadre de cet article, nous considérons que nos objets appartiennent tous au même processus.
Il nous paraît évident, que les choix au niveau du gestionnaire mémoire sont primordiaux pour assurer une communication rapide entre nos objets. La communication s'effectuera essentiellement par des envois de messages à un objet client ou parent. Pour simplifier ce mécanisme, nous avons besoin d'un espace mémoire commun à l'ensemble des objets. Cet espace mémoire fonctionne naturellement comme une pile. Nous savons que la segmentation permet la séparation du code, des données et de la pile dans des espaces d'adressage logiquement indépendants. Mais aussi, elle facilite le partage et la protection de ces espaces [7]. Notre idée est d'utiliser au mieux ce mécanisme pour permettre la mise en place d'une pile commune d'exécution entre plusieurs objets possédant des données qui leur sont propres (voir figure 2). Comme dans *Multics* [3] nous avons choisi de profiter des avantages de la pagination et de la modularité de la segmentation.

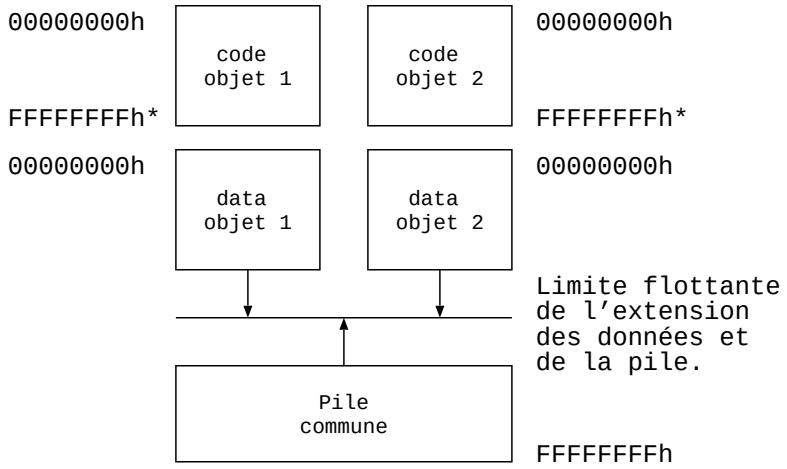

Figure 2: Vision segmenté de 2 objets *Isaac* : les deux objets partagent la même pile d'exécution.

De plus, si l'on désire que notre système ne reste pas un objet de laboratoire, il est nécessaire qu'il possède tous les atouts pour permettre une compatibilité avec l'existant. Recréer entièrement un compilateur adapté à nos besoins n'est pas souhaitable pour deux raisons évidentes : le temps de développement sur une architecture donnée devient trop imposant, et la réutilisation des programmes C existants devient difficile.
Donc pour des questions de portabilité, de compatibilité et de réutilisabilité, nous générons nos objets avec un compilateur C.
Cela nous permet d'établir une base solide pour une compatibilité avec les produits GNU. Nos objets sont compilés et peuvent être vus comme un programme C indépendant. Par exemple, nous considérons le pro-

gramme *emacs* comme un objet communiquant avec l'objet *libc* assurant la compatibilité << *Unixienne* >> avec l'ensemble de notre système.
Nous produisons un programme C pour chaque objet *Isaac* et nous utilisons une pile commune à tous les objets appartenant au même processus. Mais les compilateurs C ne permettent pas de séparer l'espace pile de l'espace données de manière transparente, nous proposons une solution qui le permet sans modifier le compilateur. De manière générale, cela permet une gestion de la mémoire et des protections plus fines qu'actuellement.
De plus, une autre conséquence réside dans la réduction de l'espace virtuellement occupé par un programme. Ce gain permet d'avoir un unique espace d'adressage pour plusieurs objets/programmes (voir figure 3).

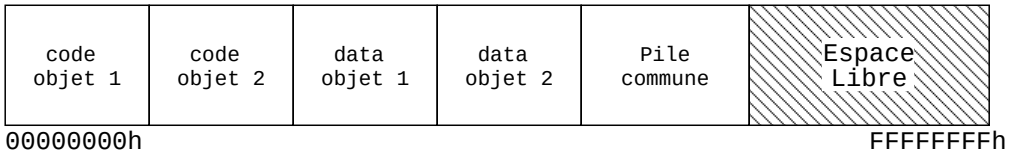

Figure 3: Vision de l'espace linéaire virtuel d'*Isaac*.

1.3. Le problème de la segmentation à partir d'un programme C

Dans cette partie, nous allons expliquer pourquoi le langage C n'est pas ou peu adapté à l'utilisation de la segmentation. Notre démonstration reste valable pour la plupart des langages et compilateurs actuels.
Que ce soit sur un processeur 32 bits ou 64 bits, l'adresse d'un élément (variable ou fonction) dans un programme *C* n'est matérialisée que par un seul index. Pour adresser un élément en utilisant la segmentation, nous avons besoin d'un couple d'index : le premier indique le numéro du segment à utiliser (registre de segment), le second représente l'offset pointant sur l'élément en question.
En règle générale, le principal problème réside dans la séparation en deux segments distincts des données et de la pile d'un programme en mémoire. En effet, dans l'exemple ci-dessous, nous pouvons constater que le système est incapable de déduire le segment contenant la donnée pointée par la variable `Ptr`. Elle peut pointer aussi bien sur le segment de pile, que sur les données ou encore le code.

```
int global ;
void main()
{ int local ;
  int *Ptr ;

  switch (getch()) {
    case 'G': Ptr = &global; break;
    case 'L': Ptr = &local; break;
    case 'C': Ptr = main; break;
  } ;

} ;
```

Il est facilement imaginable de connaître par une analyse de flots le segment utilisé pour la plupart des pointeurs. Néanmoins, cette solution n'est pas totale et oblige une modification massive du compilateur. Par exemple pour une architecture Intel, il faudrait modifier un compilateur pour considérer un pointeur, non pas sur 32 bits mais sur 48 bits prenant ainsi en compte le numéro du segment sur 16 bits. De plus, sans une analyse de flots pour connaître dans la plupart des cas le segment à considérer, il devient

obligatoire de charger un registre de segments pour chaque accès à un pointeur. Il paraît évident que les performances globales d'exécution seraient profondément atteintes.

2. Notre approche

Nous tenons tout d'abord à signaler que notre solution est applicable sans modification massive du compilateur et qu'elle admet une perte de performance acceptable. En effet, nous ne modifions pas les instructions d'utilisation de pointeurs que génère le compilateur. Notre technique utilise une bonne part de considération logique et repose sur une astuce durant l'exécution du programme.
Lors d'une instruction de pointage, le processeur doit connaître le segment en question. Mais, cette information est perdue par le compilateur car le pointeur ne possède qu'un offset.
Un ensemble de pré-indexation des registres de segment est par défaut correcte. Ce sont ces considérations qui vont nous permettre de résoudre la majeure partie des cas.
• Le code utilise toujours le segment de code par défaut. C'est à dire que toute instruction indirecte de branchement ou d'appel utilise de manière implicite le registre du segment de code. Donc, comme les pointeurs de fonction ou de label sont toujours utilisés dans le cadre de l'exécution d'un bloc d'instructions, il n'y a pas de réel problème de segmentation. Une autre utilisation de ce type de pointeur n'a aucun sens dans un programme correctement écrit.
• L'accès aux variables locales est toujours par défaut sur le segment de pile. Il n'y a donc pas de problème d'accès aux variables locales, même si le segment de pile est différent du segment de données. Par exemple sur une architecture *Intel*, l'accès aux variables locales utilise le registre d'index/d'offset *ebp* qui, de manière implicite, prend le registre de pile *ss* comme segment.
• Les variables globales ou allouées sont par défaut sur le segment de données. Par exemple sur une architecture *Intel*, toutes les instructions n'utilisant pas les registres d'index *ebp* (base de pile) et *esp* (sommet de pile) prennent par défaut le registre *ds* (Segment de data) comme segment en vigueur.
Il ne reste que les pointeurs de données qui peuvent être sur le segment de données et parfois sur la pile. Par défaut le processeur prend toujours le registre de données pour pointer. C'est dans cette unique ambiguïté que notre système entre en jeu.
Pour garantir une bonne efficacité et lever l'ambiguïté dans tous les cas, nous avons décidé d'agir non pas sur le code généré, mais durant l'exécution du programme. Par défaut, le processeur prendra un des deux segments, ce n'est qu'en cas d'échec que nous agissons. La difficulté réside à détecter une erreur d'adressage de segment lors de l'utilisation d'un pointeur et de rediriger l'adressage vers le bon segment. Celle-ci repose sur trois questions essentielles :
1) Comment détecter l'erreur d'adressage due à la segmentation ?
2) Comment et sur quel segment faut-il rediriger cette instruction d'adressage ?
3) Comment distinguer cette erreur par rapport à une erreur réelle d'adressage ?

2.1. Comment détecter l'erreur d'adressage due à la segmentation ?

Pour apporter une réponse à la première question, nous considérons les deux remarques suivantes : tout d'abord, nous savons qu'un segment de type pile commence au sommet de pile jusqu'à l'adresse la plus élevée (4Giga sur processeurs 32 bits). Par contre, un segment de données commence à l'offset zéro et s'étend jusqu'au sommet du tas. Nous pouvons donc établir les intervalles suivants :

$$I_{data} = [0..x] \text{ et } I_{stack} = [y..z] \text{ avec } x < y \text{ et } z : \textit{ Limite d'offset adressable.}$$

Ces deux intervalles sont à l'image des segments déclarés et maintenus par le système d'exploitation. Ils nous garantissent qu'un adressage avec une erreur de segmentation provoque une violation au niveau du système d'exploitation. Cette violation se traduit par une exception système de débordement qui nous permet de résoudre le problème lors de l'exécution de l'instruction fautive. Nous allons donc agir dans l'exception de débordement de segment pour rediriger le registre de segment vers le bon, et ainsi permettre de poursuivre l'exécution du programme au niveau de l'instruction fautive.

2.2. Sur quel segment faut-il rediriger l'instruction d'adressage ?

Cela nous amène à répondre à cette seconde question de manière triviale, le choix du segment se restreint à deux possibilités : le segment de pile ou celui des données. Une analyse grossière de l'instruction fautive permet de déterminer le registre de segment en question et de connaître sa valeur. Si ce registre pointe sur le segment de données alors on le redirige sur le segment de pile ou inversement. Notre technique de bascule permet de résoudre le problème de segmentation uniquement si cela est nécessaire.

2.3. Comment distinguer cette erreur par rapport à une erreur réelle d'adressage ?

La distinction d'une erreur de segmentation par rapport à une erreur d'adressage est quelque chose de beaucoup plus délicat. Cette vérification doit être rapide, car elle n'amène rien d'autre qu'une protection. En effet, avec la mise en place actuelle de notre bascule dans l'exception de débordement, une erreur d'adressage provoquerait une boucle sans fin de déclenchement d'exception. Pour résoudre ce problème, nous avons pensé à deux implantations possibles.

La première façon consiste simplement à vérifier si l'élément pointé appartient à l'un des deux intervalles. Cette première technique n'a pas été retenue dans notre implantation pour une raison d'efficacité. En effet, s'il est rapide de détecter, par une analyse grossière de l'instruction fautive, le registre de segment en question, il n'en est pas de même pour connaître la valeur du pointeur sur certains jeux d'instructions (*Intel* par exemple).

La deuxième technique repose sur l'utilisation d'une autre exception système : l'exception << trace >> . En effet, en mode Trace, le processeur déclenche l'exception trace à la fin de chaque instruction. Lors d'un déclenchement d'exception de débordement, deux cas sont à envisager : La donnée pointée est sur l'autre segment (pile ou donnée) ou le programme est effectivement en échec ! Pour permettre de lever cette dernière ambiguïté, nous passons le processeur en mode *trace* , nous redirigeons le registre en question sur le supposé bon segment et nous ré-exécutons l'instruction. Ainsi, en cas d'admission de cette instruction par le processeur, l'exception trace est déclenchée. En cas d'échec, le processeur redéclenche l'exception de débordement. A ce niveau, nous savons si le pointeur est correct ou si il s'agit d'une erreur irrécupérable. Dans le cas d'un pointeur correct, on repasse le processeur en mode normal dans l'exception trace pour poursuivre l'exécution du programme.

Le corps de l'exception de débordement :

```
Si (FlagTrace) Alors
    Erreur fatale !
Sinon
    Si (Reg.Segment == Segment.data) Alors
        Reg.Segment <- Segment.Stack.
    Sinon
        Reg.Segment <- Segment.data
    fsi
    FlagTrace <- True.
End.
```

Le corps de l'exception trace :

```
FlagTrace <- False.
```

Cette solution est correcte, mais pas très efficace. En effet, le déclenchement d'exception est coûteux, et notre algorithme oblige deux déclenchements d'exceptions pour chaque problème de segmentation.

C'est pour cela, que nous avons imaginé de modifier légèrement cet algorithme pour permettre de réduire le coût à un seul déclenchement d'exception dans la majeure partie des cas. Si on ne passe pas le processeur en mode << trace >> , nous pouvons constater que l'exception de débordement se produira de manière répétée sur la même instruction. L'idée est de conserver dans l'exception de débordement la position de l'instruction fautive. Ce n'est quand cas d'égalité entre la position précédente et la position courante que nous déclenchons le mode << trace >> .

Le corps de l'exception de débordement avec optimisation :

```
Si (FlagTrace) Alors
    Erreur fatale !
Sinon
    Si (Reg.Segment == Segment.data) Alors
        Reg.Segment <- Segment.Stack.
    Sinon
        Reg.Segment <- Segment.data
    fsi
    Si (PC_old == PC_fault) alors
        FlagTrace <- True.
    fsi
    PC_old <- PC_fault.
End.
```

Remarque : Le corps de l'exception << trace >> ne subit pas de modification.
Avec cette modification et pour un programme correct, le déclenchement de deux exceptions pour résoudre un problème de segmentation est quasiment inexistant.
Nous pouvons constater que la solution que nous apportons peut être utilisée pour résoudre d'autres problèmes que le nôtre liés à la communication ou la gestion de la mémoire virtuelle dans les systèmes d'exploitation existants ou futurs.

2.4. Implantation de la méthode

Nous avons implanté cette méthode sur notre système d'exploitation qui utilise le compilateur libre GCC sur une architecture Intel 386 [4]. Les modifications apportées au compilateur sont les suivantes :
• Détection et sortie du segment de code vers le segment de données des tables d'index de label générées lors de certains << switch >> importants (pour séparer le segment de code des données).
• Détection et sortie du segment de code vers le segment de données des variables constantes. (Pour séparer le segment de code des données).
• Réécriture du fichier décrivant le mapping en mémoire du code et des données pour le << linker >>, forçant ainsi le démarrage des offset à zéro pour les données.
Nous pouvons remarquer qu'aucune modification n'est interne au compilateur proprement dit, et que tant que GCC ne change pas sa syntaxe d'assembleur et le fichier de mapping pour le << linker >>, nos modifications sont portables et suivent l'évolution des versions du compilateur.

3. Mesure d'efficacité

Pour permettre de quantifier au mieux la perte de performance de notre méthode, nous avons effectué deux mesures avec deux variantes. La première mesure donne une idée de l'efficacité de notre méthode durant l'exécution d'un programme. La seconde mesure sert à quantifier la perte de performance dans le pire des cas.
Pour chaque mesure, la première variante, consiste à retirer l'utilisation du mode << trace >> pour comparer la rapidité lors de l'exécution dans le cas d'une comparaison des intervalles avec le pointeur fautif. La seconde variante représente l'implantation avec la vérification par déclenchement du mode << trace >>.
Les mesures ont été effectuées sur un *Intel pentium II* à 400MHz.

3.1. Exécution globale d'un programme

Cette mesure a pour but de donner une idée du nombre de déclenchements du mécanisme lors du déroulement d'un programme. Comme notre système n'est actuellement qu'un prototype, le seul code de taille acceptable que nous possédons pour ce type de mesure est le système lui-même. Il possède environ 100 000 lignes de C, notre mesure est faite sur l'ensemble du chargement du système, du boot jusqu'au chargement de l'interface graphique. Nous avons aussi effectué le chargement du système avec l'ancienne disposition des segments (donc sans déclenchement d'exception), mais la différence de performance en terme de temps d'exécution est indiscernable.

	Nombre de déclenchement d'exceptions	Temps d'exécution
Sans segmentation	0	82,27 secondes
Sans *trace*	9015	82,92 secondes
Avec *trace*	9015	82,92 secondes

Nous pouvons constater que le nombre de déclenchements de notre mécanisme est relativement faible. Il affecte les performances globales de moins d'un pour cent sur le temps d'exécution. Cela nous paraît acceptable en vue des possibilités que nous offre la segmentation dans le cadre de notre projet.

3.2. Dans le pire des cas

Notre second test met notre méthode dans les pires conditions par le programme ci-dessous. En effet, à chaque itération le pointeur change de segment. Cela a pour effet de déclencher notre mécanisme d'exception. De plus, ce déclenchement s'effectue toujours sur la même position dans le code, provoquant ainsi la mise en place du mode << trace >> pour valider l'instruction.
La seconde mesure a été effectuée sur le programme suivant :

```
int global=0;
void main()
{ int local=0, Cpt;
  int *Ptr ;

for (Cpt=0;Cpt<1000000000LU;Cpt++) {
    Ptr=((Cpt&1)?(&local):(&global));
    (*Ptr)++;
  };
};
```

	Nombre de déclenchement d'exceptions	Temps d'exécution
Sans segmentation	0	34,16 secondes
Sans 'trace'	1 000 000 000	43,04 secondes
Avec 'trace'	2 000 000 000	46,37 secondes

Évidemment, les performances en termes de temps d'exécution sont gravement atteintes. Nous pouvons remarquer un surcoût de 26% à 37%. Heureusement que ce type de programme n'est pas courant. D'ailleurs, dans l'ensemble de notre système Isaac, aucun déclenchement du mode << trace >> n'est nécessaire pour résoudre les problèmes liés à la segmentation. Nous évitons ainsi une double exceptions.

4. Travaux connexes

A ma connaissance, aucun compilateur ne se risque à prendre en compte la segmentation de manière transparente, c'est à dire la possibilité de mapper en mémoire n'importe quels programmes ayant la pile et les données dans deux segments distincts. En effet, cette prise en compte sans une modification du code source n'est pas simple. Néanmoins, parmis les compilateurs que nous avons pu tester à ce sujet, l'implantation mise en place par le *Watcom C*[1] version 11 reste intéressante. Il existe l'option */zu* permettant de déclarer à la compilation la séparation du segment de pile avec celui des données. L'utilisation de cette option sur un source en *C ansi* provoque la génération d'un *warning* du type: *"pointer truncated"* à chaque affectation de pointeur.
Exemple :

```
char *Ptr;       /* Notre pointeur en C ansi.                */
char Global;     /* Une variable dans le segment de données. */
```

[1] Copyright by Sybase.

```
void main()
{ char Local;    /* Une variable dans le segment de pile      */

  Ptr = &Local;  /* Génére Warning : Pointer truncated.       */

  (*Ptr) ++;     /* Une utilisation dangeureuse !             */
};
```

Dans notre exemple, le message nous indique que nous affectons un pointeur 48 bits (segment + offset) délivré par `&Local` dans une variable pointeur `Ptr` de 32 bits (uniquement l'offset). Par la suite, l'utilisation de ce pointeur provoque l'arrêt du programme par déclenchement d'exception.
Pour résoudre et corriger cette erreur, le compilateur admet un nouveau type de variable pointeur sur 48 bits. C'est à dire qu'il faut modifier le source en ajoutant `_far` à chaque déclaration de pointeur.
Voici notre exemple acceptant la segmentation :

```
char _far *Ptr;  /* Notre pointeur sur 48 bits (Segment + Offset). */
char Global;     /* Une variable dans le segment de données.       */

void main()
{ char Local;    /* Une variable dans le segment de pile           */

  Ptr = &Local;  /* L'ensemble des pointeurs sont sur 48 bits      */

  (*Ptr) ++;     /* Utilisation sans difficulté                    */
};
```

Ainsi le compilateur *Watcom C* admet la segmentation avec une modification du code source en utilisant un nouveau type pour la déclaration d'un pointeur.
D'autres compilateurs parlent de la gestion de la segmentation. Nous pouvons citer les travaux du compilateur C de *mentor* [2] ou de celui d'*Intel* [1]. Mais nous avons pas pu tester leur implantation par absence de compilateur.
A noter, dans les années 80, Intel développa un langage propriétaire "PLM" qui réalisait du code segmenté. La publication [9] montre une autre approche de l'utilisation de la segmentation et pagination sur processeur Intel.

5. Conclusion

Dans cet article, nous dénonçons et défendons le manque d'utilisation de la segmentation mémoire dans les systèmes d'exploitation. Les avantages de ce mécanisme sont nombreux et nous avons pu souligner certains petits problèmes dans la gestion de la mémoire non segmentée sous Unix (section 1.1). Dans le cadre de notre projet de recherche *Isaac*, son utilisation nous paraît naturelle et indispensable pour la mise en place de techniques de communication (section 1.2).
Dans la pratique, les compilateurs actuels ne prennent pas en compte de manière transparente la segmentation processeur et pire encore, rendent son utilisation impossible. De plus, nous avons vu qu'une modification massive du compilateur risquerait d'atteindre les performances en matière de temps d'exécution d'un programme (section 1.3).
C'est en prenant en compte ces considérations que nous proposons une méthode transparente permettant de résoudre ce problème avec un coût d'exécution négligeable. Celle-ci s'applique sans modification du code généré par le compilateur. Elle est basée sur l'utilisation d'exception système pour résoudre les problèmes de segmentation durant l'exécution d'un programme (section 2).
Quelques rares compilateurs prennent en compte la segmentation du processeur, mais celle ci n'est pas transparente pour le programmeur (section 4).
L'implantation de notre méthode dans le cadre de notre projet nous a permis de valider ces performances et d'atteindre nos objectifs par l'utilisation de la segmentation (section 3).

Bibliographie

1. http://www.intel.com.
2. http://www.mentor.com/embedded/papers/whitepapers/using80x86/.
3. C.T. Clingen & R.C. Daley A. Bensoussan. The Multics Virtual Memory: Concepts and Design. In *Communications of ACM*, volume 15, number 5, pages 308–318, May 1972.
4. R. Hummel. Structure et gestion de la mémoire. In *Processeur et coprocesseur, guide de référence du programmeur PC*, pages 83–104, 1992.
5. J-M. Rifflet. Espace d'adressage virtuel. In *La programmations sous unix*, page 410, 1994.
6. B. Sonntag. `http://www.isaacOS.com`. Site web du système Isaac. En cours de construction., 2000.
7. A. Tanenbaum. La segmentation. In *Systèmes d'exploitation, systèmes centralisés, systèmes distribués*, pages 144–159, 1994.
8. A. Tanenbaum. Partage de mémoire. In *Systèmes d'exploitation, systèmes centralisés, systèmes distribués*, pages 731–739, 1994.
9. Prashant Pradhan Tzi-cker Chiueh, Ganesh Venkitachalam. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *17th ACM Symposium on Operating Systems Principles*, pages 140–153. ACM Press, 1999.
10. D. Ungar and R. Smith. Self: The Power of Simplicity. In *2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 227–241. ACM Press, 1987.
