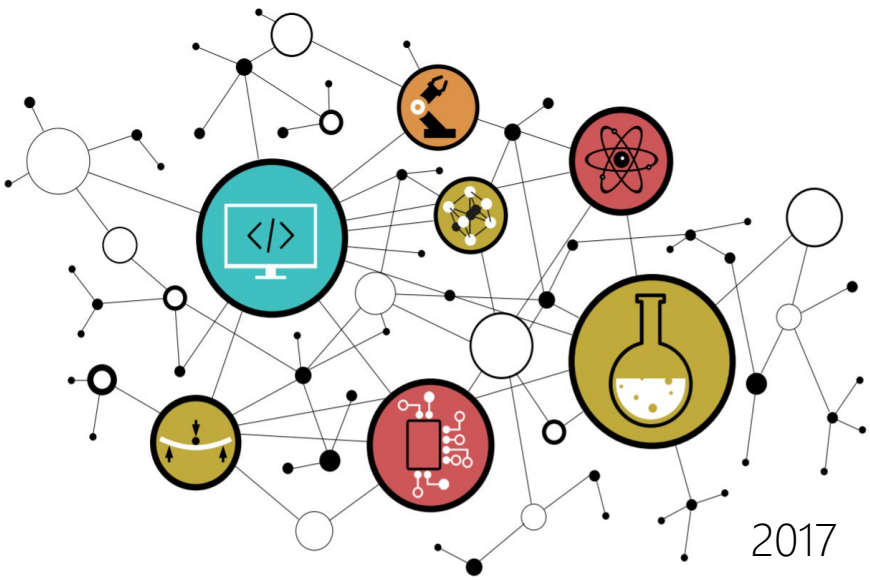
ENSI CAEN

ARCHITECTURES ET TECHNOLOGIES DES ORDINATEURS

TRAVAUX PRATIQUES



2017

SOMMAIRE

SEQUENCEMENT

- ***Travail préparatoire :*** lire le préambule et réaliser l'exercice 1. Un rapport de synthèse de 3 pages sera à déposer sur la plateforme d'enseignement avant l'arrivée en première séance

- ***Séance n°1 :*** exercice 2 jusqu'au 2.3 inclus, le reste à la maison et préparer vos questions pour la séance suivante

- ***Séance n°2 :*** jusqu'au 2.4 inclu, le reste à la maison et préparer vos questions pour la séance suivante

- ***Séance n°3 :*** jusqu'au 3.2 inclu, le reste à la maison et préparer vos questions pour la séance suivante

- ***Séance n°4 :*** jusqu'au 4.2 inclu, le reste à la maison et préparer vos questions pour la séance suivante

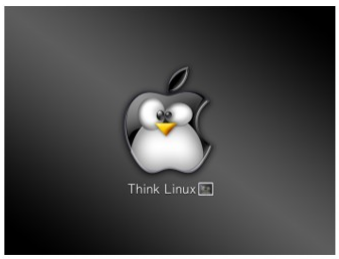
- ***Séance n°5 :*** le plus loin possible !

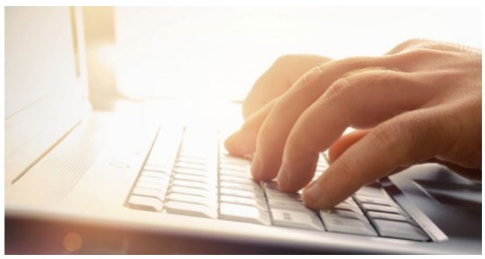
PREAMBULE

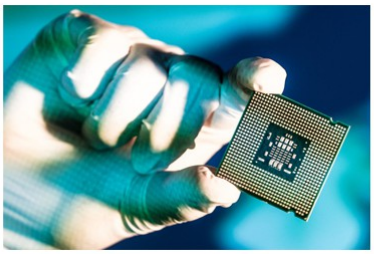

notes

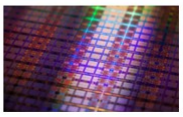
PREAMBULE

Cet enseignement ainsi que la trame de travaux pratiques associée a pour objectif de poser quelques bases dans compréhension des architectures matérielles actuelles réalisant l'exécution de programme quel qu'en soit le langage de programmation à la source. L'idée étant de comprendre plus finement les processus d'exécution de code et de gestion des ressources de stockage réalisés conjointement par l'architecture processeur, le système d'exploitation et les outils de compilation.







Cette trame s'appuie sur les enseignements de logiques, d'outils de développement logiciel et de programmation en langage C. Le langage C est favorisé dans cet enseignement car il reste suffisamment proche du matériel pour comprendre les liens entre langages de haut niveau et langages natifs bas niveaux supportés par les architectures processeurs cibles. Toutes les illustrations s'appuieront sur l'analyse de programmes simples écris en C. Rappelons que bien que maintenant ancien, le langage C reste encore à notre époque un langage de programmation majoritairement rencontré dans les couches basses des systèmes et sur les architectures à ressources limitées voire spécialisées (noyaux, systèmes d'exploitations, bibliothèques spécialisées, systèmes embarqués …). De plus, à travers ses différents standards, le langage C continu toujours d'évoluer à notre époque (normes C ANSI ou C89 en 1989, ISO C90 en 1990, C99 en 1999 et C11 en 2011).

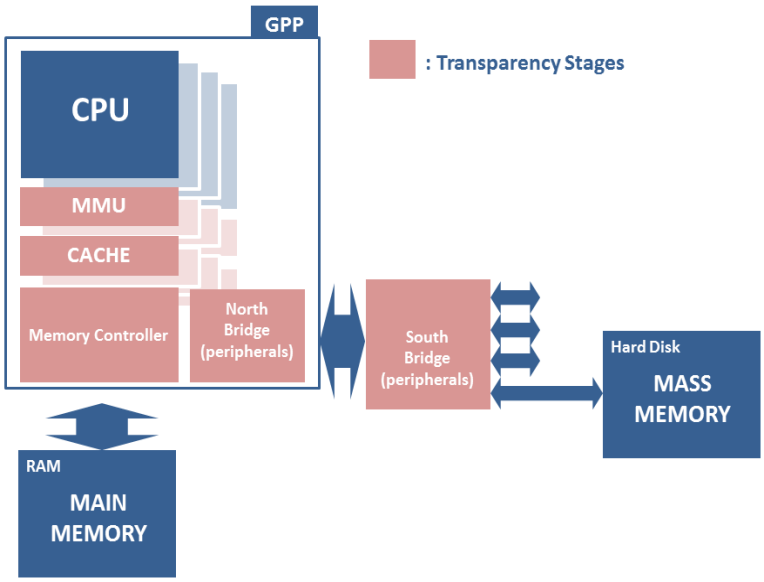
Cette trame de travaux pratiques est avant tout une séquence d'analyse de programmes. Nous nous intéresserons aux impacts de nos développements sur la qualité et le fonctionnement du code généré. L'objectif opérationnel étant d'élargir la capacité à durcir un programme quelque soit l'architecture cible voire à l'optimiser en tenant des contraintes imposées par celle-ci. Nous nous attarderons notamment sur les stratégies de gestion mémoire et une bonne compréhension et maîtrise des segments mémoire associés à notre applicatif :

- *Allocations automatiques et gestion de la pile*
- *Allocations statiques et fichier ELF*
- *Allocations dynamiques et gestion du tas*

Nous profiterons de cet enseignement pour découvrir des mots clés, qualificateurs de types et classes de stockage non vues en enseignement d'initiation au langage C. Nous aurons l'occasion d'observer leurs impacts à l'exécution et nous affinerons notre connaissance des outils de développement : *const*, *register*, *restrict*, *volatile*, *inline*, *static* ...

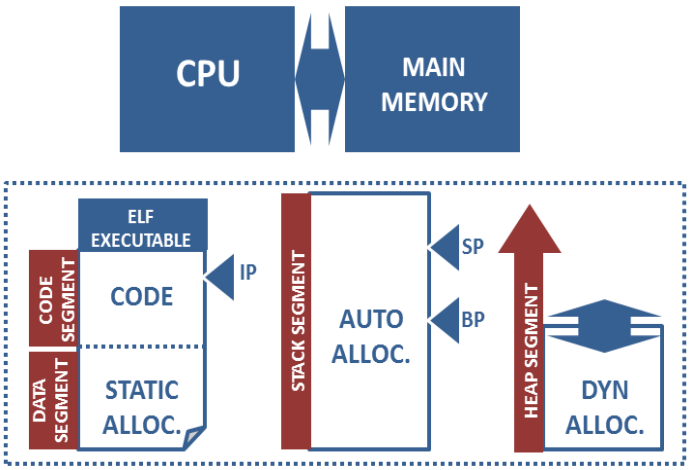
Architecture matérielle réelle

Rappelons que l'architecture matérielle réelle sur laquelle nous travaillons est complexe. Néanmoins, la plupart des étages en jeux sont présents afin d'accélérer le mécanisme d'exécution ou de virtualiser l'accès aux ressources et sont dans tous les cas transparents pour le développeur.



Architecture matérielle virtualisée

Une fois chargé en mémoire principale, virtualisé et lancé sur un CPU par le système, le modèle de la machine vu du programme et donc du développeur reste relativement simple. Modèle d'exécution de J. Von Neuman (1946). La compréhension d'une représentation en segments logiques de code et de données de notre applicatif permet alors une réelle expertise dans la réalisation de programmes durcis.



Archive de travail

Avant de commencer la trame de travaux pratiques, télécharger sur la plateforme moodle de l'ENSICAEN dans la section *download* l'archive nommée *gpp* (arborescence de TP dans le répertoire *gpp/tp/x86x64*). Accès libre à la plateforme en lecture, néanmoins, bien penser à s'inscrire avant tout dépôt éventuel. Effectuer l'extraction sur votre bureau :

```
# tar  -xvf   gpp.tar   -C   <dir_path>
```

Outils de développement

La trame de travaux pratiques sera réalisée sur système GNU\Linux, les différents sources seront donc compilés sous GCC (GNU Collection Compiler, http://gcc.gnu.org/). Les chaînes de compilation GCC-like restent majoritairement rencontrées, notamment dans le domaine de l'embarqué. Dans un premier temps, l'assembleur généré durant les phases d'analyse de code devra être compatible avec les architectures IA-32 (assembleur 32bits compatible x86). Puis, nous analyserons de l'assembleur 64bit compatible pour architectures x64.

```
# gcc   -Wall   -m32  [options]   fileName.c          ----> assembleur 32bits x86
# gcc   -Wall   [options]   fileName.c               ----> assembleur 64bits x64
```

Nous vous encourageons d'ailleurs à utiliser vos propres machines pour réaliser ces TP. Si vous souhaitez réaliser la trame sur vos machines personnelles 64bits, il vaut faudra installer les packages suivant, offrant notamment le support des bibliothèques système 32bits.

```
# sudo apt-get install gcc-multilib watch
```

Documentation

Vous aurez probablement à vous documenter sur le fonctionnement de certaines instructions assembleur. Bien respecter la chronologie suivante. S'aider avant tout d'internet, puis rechercher vos réponses dans les documents de référence Intel (répertoire de TP */x86/doc/intel/*, jeu d'instructions dans le volume 2), et enfin demander à l'enseignant référent. Ne pas hésiter à s'aider de vos voisins de table afin de confronter vos idées. Liens officiels vers les ressources Intel ci-dessous :

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

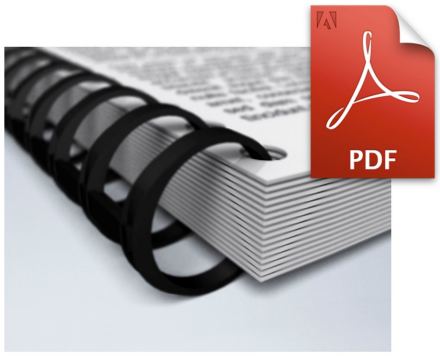
COMPILATION

EDITION DES LIENS


notes

1. COMPILATION ET EDITION DES LIENS

Cette partie est à réaliser à la maison avant l'arrivée en première séance de TP. Il s'agit d'une trame d'analyse du processus de compilation ainsi que d'édition des liens. Elle sert de support à la séquence de cours traitant du sujet. A la fin de cet exercice (~1h), vous aurez à réaliser et à déposer sur la plate-forme, dans le dépôt associé à votre groupe de TP, un rapport de 2-3 pages expliquant le processus de compilation, d'édition des liens et de chargement d'un programme. S'aider de schémas afin d'illustrer vos propos. Que votre document soit progressif et pédagogique. Concernant le dépôt du fichier, respecter l'extension et le nommage suivant *gpp-homework-2016-<name>-<surname>.pdf* (tout en minuscule)



Se placer dans le répertoire */x86/toolchain/* pour l'application des commandes suivantes. Le fichier *README.txt* contient la séquence d'exécution complète. Analyser le fichier source présent dans le répertoire */x86/toolchain/src/hello.c*. Compiler le projet et analyser la sortie.

```
# gcc -Wall -m32 -I./inc/ ./src/hello.c -o ./build/bin/hello
# ./build/bin/hello
```

1.1. Préprocesseur

- Compiler à nouveau le même projet en s'arrêtant à l'étage de préparation du code pour la compilation (preprocessing). Analyser la sortie.

```
# gcc -E -Wall -m32 -I./inc/ ./src/hello.c > ./build/misc/hello.i
```

- Jouer sur la valeur de la macro MINIMAL présente dans le fichier d'en-tête */x86/toolchain/inc/hello.h* puis répéter l'opération précédente.

- Rappeler le travail du pré-processeur, et préciser les principales commandes élémentaires appliquées dans le cadre de cet exercice.

1.2. Analyse et génération de code natif

- Compiler le programme en partant du code déjà préparé par le préprocesseur et en s'arrêtant à l'étage d'assemblage. Analyser la sortie.

```
# gcc -S -Wall -m32 ./build/misc/hello.i -o ./build/misc/hello.s
```

- Bien observer qu'un fichier assembleur est avant tout un fichier texte. Nous pouvons développer en assembleur. Nous rencontrons ce type de développement à notre époque dans certains cas spécifiques (optimisation de code à l'exécution, diminution de l'empreinte mémoire d'applicatif sur système contraint, bibliothèques spécialisées, hacking …)

1.3. Assembleur

- Assembler le fichier assembleur */x86/toolchain/build/misc/hello.s* et achever ainsi le processus de compilation. Le fichier source résultant est un fichier binaire au format ELF 32bits. Nous ne pouvons plus utiliser un éditeur de texte afin d'afficher son contenu. Il nous faudra utiliser un utilitaire dédié (*objdump* ou *readelf*). Analyser la sortie. Bien avoir lu la partie du cours traitant des fichiers ELF.

```
# as --32 ./build/misc/hello.s -o ./build/obj/hello.o
# readelf -h ./build/obj/hello.o
# objdump -Sx ./build/obj/hello.o
```

- Le fichier */x86/toolchain/build/obj/hello.o* est un fichier ELF 32bits. Préciser son type ainsi que l'architecture cible après lecture de son en-tête.

1.4. Édition des liens

- Finaliser la génération d'un fichier exécutable par l'édition des liens puis exécuter le programme. Analyser la sortie. Nous étudierons plus en détail l'édition des liens par la suite.

```
# gcc -Wall -m32 ./build/obj/hello.o -o ./build/bin/hello
# readelf -h ./build/bin/hello
# objdump -Sx ./build/bin/hello
# ./build/bin/hello
```

- Le fichier */x86/toolchain/build/bin/hello* est un fichier ELF 32bits. Préciser son type.

1.5. Exécutable minimal

- Nous allons maintenant chercher à obtenir un programme binaire exécutable minimal. Pour ce faire, nous allons jouer avec l'étage d'édition des liens et réaliser celle-ci en appelant directement le linker. Donner la taille actuelle du fichier binaire de sortie alors que notre programme ne fait que quelques dizaines d'octets.

```
# ls -l build/bin
```

La fonction *main* n'est jamais le premier point d'entrée réel d'un programme. Tout programme C débute par un source générique d'amorçage obligatoirement chargé d'appelé la fonction *main*. L'entrée typique sur système GNU\Linux est la fonction ou label *_start*. Nous verrons par la suite que cette entrée peut être renommée. Ce ou ces fichiers d'amorçage système restent toujours les mêmes utilisés à l'édition des liens et sont pré-compilés pour une architecture cible donnée.


```
1  .section .text
2  .global _start
3
4  _start:
5      mov     $0, %ebp
6      push    %ebp
7      mov     %esp, %ebp
8      call    main
9  _trap:
10     jmp     _trap
11     pop     %ebp
12     ret
```

- Nous allons utiliser un fichier d'amorçage minimal. Ouvrir le fichier assembleur */x86/toolchain/build/startup/crt0.s*, analyser puis expliquer sont travail.

- Assembler le nouveau fichier d'amorçage et l'ajouter manuellement à l'édition des liens en appelant directement le linker. Valider l'exécution du programme. A ce stade là, nous ne pourrons plus utiliser de bibliothèques linkées dynamiquement. Bien placer la macro MINIMAL à 1. Tous les programmes suivants de la trame de TP n'utilisant aucune fonction dépendante de bibliothèques dynamiques peuvent être compilées jusqu'à édition des liens en utilisant ce fichier d'amorçage.

```
# as --32 ./build/startup/crt0.s -o ./build/obj/crt0.o
# ld -melf_i386 ./build/obj/crt0.o ./build/obj/hello.o -o ./build/bin/hello
# readelf -h ./build/bin/hello
# objdump -Sx ./build/bin/hello
# ls -l build/bin
# ./build/bin/hello
```

- Observer la taille et le contenu du fichier ELF binaire de sortie. Préciser sa nouvelle taille.

- En analysant plus précisément les options passées à gcc, notamment concernant l'édition des liens, en déduire les noms des fichiers binaires d'amorçage utilisés par défaut par le système et préciser leurs noms (préfixés par *crt*).

```
# gcc -v -Wall -m32 ./build/obj/hello.o -o ./build/bin/hello
```

1.6. Linker script minimal

- Nous pouvons constater à cette étape que l'empreinte du programme binaire exécutable est optimale. Néanmoins, l'éditeur de liens continu à architecturer le fichier ELF de sortie avec un grand nombre de sections génériques maintenant inutiles. Le linker utilise un fichier texte nommé *linker script* (extension .ld), lui permettant de définir l'ossature du fichier ELF exécutable de sortie. Observer le linker script utilisé par défaut par le linker (riche en informations).

```
# gcc  -m32 -Wl,--verbose
```

- Ouvrir maintenant le fichier */x86/toolchain/build/script/linker_scrip_minimal.ld* et analyser sont contenu. Bien constater qu'il s'agit d'un élagage du linker script utilisé par défaut.

```
1  OUTPUT_FORMAT("elf32-i386")
2  OUTPUT_ARCH(i386)
3  ENTRY(_start)
4
5  SECTIONS
6  {
7      . = SEGMENT_START("text-segment", 0x400000) + SIZEOF_HEADERS;
8      .text :
9      {
10         *(.text)
11     }
12     .rodata :
13     {
14         *(.rodata)
15     }
16     .data :
17     {
18         *(.data)
19     }
20     .bss :
21     {
22         *(.bss)
23     }
24     /DISCARD/ : {
25         *(.comment)
26         *(.note.GNU-stack)
27         *(.eh_frame)
28     }
29 }
```

Ce nouveau script ne propose plus que les sections accessibles au développeur. S'aider de la documentation officielle et de la page wikipedia traitant des fichiers ELF (https://en.wikipedia.org/wiki/Executable_and_Linkable_Format et https://sourceware.org/binutils/docs/ld/Scripts.html). Il s'agit d'ordres et d'une ossature spécifiée à l'éditeur de liens lui permettant de définir la segmentation logique du binaire exécutable de sortie.

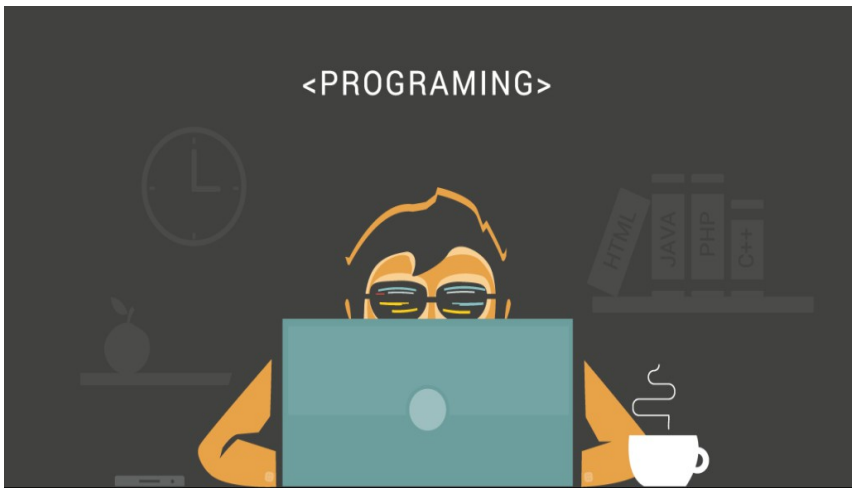
- Forcer le linker à utiliser ce script minimal et observer maintenant le contenu et la taille du binaire de sortie. Valider l'exécution du programme. Préciser la nouvelle taille du binaire exécutable.

```
# ld -melf_i386 -T ./build/script/linker_script_minimal.ld ./build/obj/crt0.o
./build/obj/hello.o -o ./build/bin/hello
# objdump -Sx ./build/bin/hello
# ls -l ./build/bin
# ./build/bin/hello
```

- Dernière rapide optimisation, nous allons nettoyer le fichier ELF (commande *strip*) notamment en concaténant les différentes sections de même nature afin de former des segments de code et de données. Observer maintenant le contenu et la taille du binaire de sortie. Valider l'exécution du programme. Préciser la nouvelle taille du binaire exécutable.

```
# strip ./build/bin/hello
# objdump -Sx ./build/bin/hello
# ls -l ./build/bin
# ./build/bin/hello
```

Voilà, l'exercice est maintenant terminé. Vous venez d'obtenir probablement l'un des programmes C les plus légers que l'on puisse faire tourner sur un système GNU\Linux 32bits et sur architecture x86. Rustique, mais il s'exécute !



ALLOCATIONS AUTOMATIQUES GESTION DE LA PILE


notes

2. ALLOCATIONS AUTOMATIQUES ET GESTION DE LA PILE

Dans cette nouvelle partie, nous allons nous intéresser aux mécanismes d'allocations mémoire sur la pile (variables locales hors qualifiées de static, paramètres de fonction, adresses de retour de fonction ...). Rappelons que la pile ou stack étudiée est propre au programme en cours d'analyse et a été allouée par le système au chargement du programme. Elle se situe en mémoire principale.

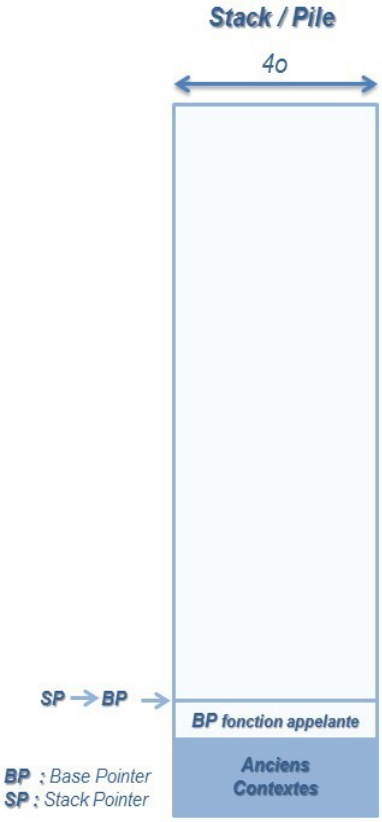
2.1. Fonction main

- Éditer puis compiler le programme présent dans le répertoire de TP */x86x64/stack/main.c* (cf. ci-dessous) en s'arrêtant à la phase d'assemblage. Dans un premier temps, nous analyserons de l'assembleur 32bits x86 (option -m32) pour passer par la suite à de l'assembleur 64bits x64.

```
# gcc  -Wall  -m32  -S  main.c
```

- Comme ci-dessous, insérer le code assembleur généré à votre fichier source C sous forme de commentaires puis Interpréter son fonctionnement.

```
int main (void) {
/* pushl  %ebp : sauvegarde le contexte de la fonction appelante
    movl %esp, %ebp : préparation à l'allocation de ressources mémoire
pour le contexte de la fonction courante ou fonction main  */
return 0 ;
/* movl  $0, %eax : passage par registre de la valeur de retour
    popl %ebp : restaure le contexte de la fonction appelante
    ret : dépile l'adresse de retour de la fonction appelante, la charge dans
le pointeur d'instruction puis quitte la fonction main */
}
```



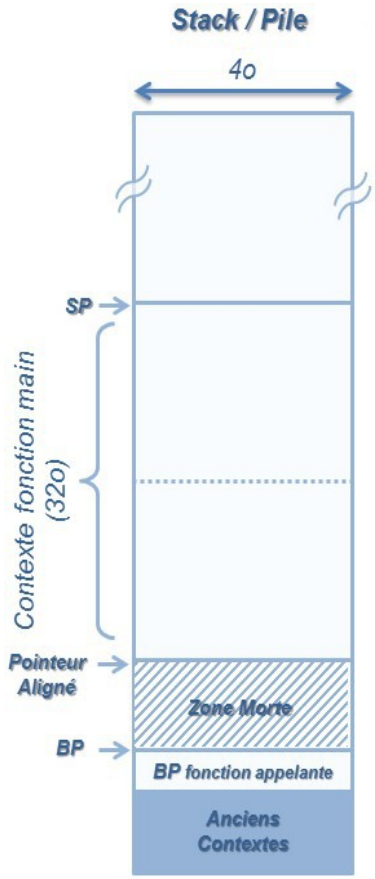
- Analyser graphiquement l'usage fait de la pile par votre programme.
- Quel est le travail réalisé par l'instruction *push* ?
- Proposer une autre écriture des instructions *push* et *pop* via les instructions *sub*, *add* et *mov*

Les directives assembleur *.cfi* (Call Frame Information) sont des extensions proposées par GAS (GNU Assembler). Elles peuvent potentiellement être utilisées par les outils de debug dans une optique de trace et de déroulement de la pile en proposant un complément d'information sur le contexte d'exécution de la procédure courante. Analyse poussée du contenu de la pile. *Dans une optique d'analyse logique du code, ne pas s'y intéresser*. Pour les retirer à la compilation, ajouter l'option *-fno-asynchronous-unwind-tables*

2.2. Variables locales non initialisées

- Éditer puis compiler le programme *local.c* en s'arrêtant à la phase d'assemblage.

```
int main (void){
/* pushl %ebp
    movl %esp, %ebp
    andl $-8, %esp : alignement mémoire du pointeur de sommet de pile.
adresse générée de valeur multiple de la taille d'un path CPU
    subl $32, %esp : allocation de 32o sur la pile (multiple de 16o) */

            char foo;
            unsigned int bar;

return 0;
/* movl  $0, %eax
    popl  %ebp
    ret */
}
```



- Que constatons-nous ?

- Ajouter le ''qualifier'' ou qualificateur de type *volatile* devant chaque déclaration, compiler le programme puis interpréter le résultat.

- Analyser graphiquement l'usage fait de la pile par votre programme.

Le ''qualifier'' ou qualificateur de type *volatile* force le compilateur à n'opérer aucune optimisation sur les variables ainsi déclarées et laisse alors la porte ouverte à des modifications potentielles de la ressource par d'autres entités de l'applicatif (exemple des applications multi-tâches ou multi-threads). Ceci est extrêmement utilisé dès qu'il s'agit de manipuler des variables critiques (ressources partagées) et que nous sommes amenés à lever les options d'optimisation à la compilation. *Un qualificateur de type doit être vu comme une directive de compilation sciemment écrite par le développeur afin d'aiguiller voir forcer la chaîne de compilation à opérer des traitements privilégiés sur la variable*. De façon générale, il s'agit de stratégies de durcissement ou d'optimisation de code (vitesse ou taille).

2.3. Variables locales initialisées

- Éditer puis compiler le programme *init.c* en s'arrêtant à la phase d'assemblage.

```
int main(void){
        char foo = 1;
        unsigned int bar = 2;

        foo = (char) bar;
return 0;
}
```

- Insérer le code assembleur généré à votre fichier source C sous forme de commentaires puis Interpréter son fonctionnement.

- Représenter graphiquement au *crayon gris* l'usage fait de la pile par votre programme. Faire évoluer le schéma précédent.

- Préciser les adresses relatives des variables locales

- Qualifier la variable *bar* de *const*. Compiler le programme puis interpréter le résultat.

- Le qualificateur de type *const* n'est qu'une protection à la compilation dédié à certaines variables ou pointeurs initialisés accessibles qu'en lecture seule. Qualifier une variable de *const* n'impacte en rien sa localisation en mémoire et permet de durcir l'applicatif en cours de développement en laissant le compilateur s'assurer qu'une ressource accessible en lecture seule le reste d'un point de vu logiciel.

- Constater que durant un transtypage d'un format entier long vers un format entier plus court, la chaîne de compilation ne préserve que les bits les moins significatifs ou bits de poids faibles (Lsb ou Less Significants Bits). Ce point peut complexifier grandement l'implémentation d'algorithmes au format entier (format à virgule fixe).

Vous constaterez que la chaîne de compilation effectue à votre place des alignements mémoire. Vous ne rencontrerez ce type d'optimisation que sur processeur vectoriel. Ceci se manifeste par des espaces mémoires vides des quelques octets présents entres vos différentes allocations de structures de données en mémoire. *Tout pointeur aligné, possède une valeur multiple de l'alignement réalisé et peut ouvrir l'accès au compilateur à l'utilisation d'instructions de chargement et de sauvegarde mémoire plus rapide en temps d'exécution.*

Il s'agit de mécanismes d'optimisation matériels permettant au CPU de minimiser les accès mémoires aux données. Dans une optique d'optimisation, ceci peu éviter des défauts d'alignement mémoire (programmation vectorielle) et l'utilisation d'instructions dédiées à la lecture/écriture de données alignées (souvent plus lentes).

La taille de ces alignements mémoire est le plus souvent liée à la taille d'un mot ''vectoriel'' CPU. D'une architecture CPU à une autre, cette taille diffère. Sur architecture vectorielle x86/x64 récente, un mot CPU fait 128bits (16o) depuis l'arrivée des extensions SIMD SSE (chemins/bus larges vers la mémoire donnée L1D). Les alignements mémoire de structures de données se feront donc le plus souvent via des adresses de valeurs multiples de 16 (modulo 16o).

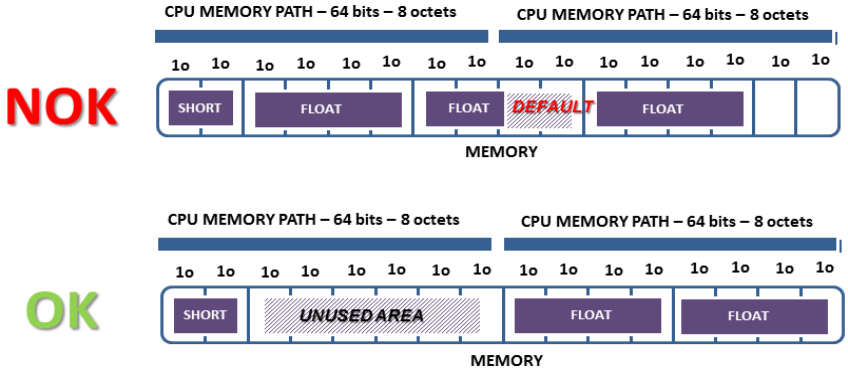
De quoi dépend la taille d'un mot CPU :

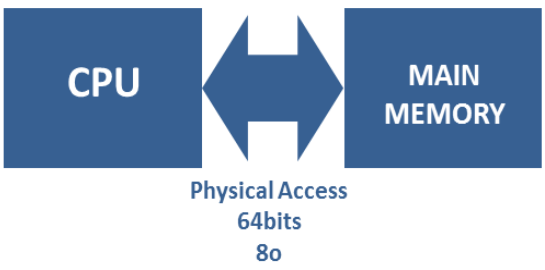
- *Taille des path vers la mémoire donnée de niveau 1 ou L1D (le plus souvent un cache) :* Exemple de la famille coreiX sandyBridge, 128bits vers le cache programme L1P, 2x128bits load et 1x128bits store vers le cache donnée L1D

- *Taille des lignes de cache :* tailles multiples d'un mot vectoriel CPU

Prenons ci-dessus un exemple de jeu de données, les premières non-alignées et les secondes alignées modulo 8 octets. Nous supposerons que les bus ou path entre CPU et niveau mémoire L1D possèdent une taille de 64bits ou 8o :

```
short foo ;
float bar[3] ;
```





2.4. Appel de fonction

- A partir de maintenant et jusqu'à la fin du TP, nous analyserons de l'assembleur 64bits compatible architecture x64. Assembleur généré par défaut sur système GNU\Linux 64bits.

```
# gcc   -Wall   -S  main.c
```

- Éditer puis compiler le programme *function.c* en s'arrêtant à la phase d'assemblage.

```
/*  privates prototypes */
int inc (int bar);

/* program entry point */
int main (void){
        int foo ;

        foo = inc (1);
return 0;
}

/* function incrémentant la valeur d'entrée */
int inc(int bar) {
return bar +1;
}
```

- Insérer le code assembleur généré à votre fichier source C sous forme de commentaires puis Interpréter son fonctionnement.

- Déclarer maintenant à la volée la variable locale *foo*. Que constatons-nous ?

```
int foo = inc (1);
```

Attention ce type de déclaration à la volée n'est pas supporté par toutes les chaînes de compilation C (exemple de la norme C89 également nommée C ANSI). Vous aurez d'ailleurs de mauvaises surprises durant les enseignements de systèmes embarqués si vous utilisez ce type de déclaration (exemple de la toolchain de Microchip pour architecture PIC18 ne supportant que les normes C89-ANSI et partiellement la norme C99). *Prenez dès maintenant pour habitude en langage C de gérer vos déclarations de variables locales en début de fonction, notamment dans l'embarqué et si vos projets futurs peuvent impliquer des portages sur de nouvelles architectures cibles*

- Observons le programme désassemblé (traduction binaire vers assembleur) correspondant au binaire exécutable après compilation et édition des liens. Intéressons-nous aux sections de code correspondant à notre programme d'analyse. Bien constater que les références symboliques (ex : *inc*) sont remplacées à l'édition des liens.

```
00000000004004d6 <main>:
  4004d6:       55                      push   %rbp
  4004d7:       48 89 e5                mov    %rsp,%rbp
  4004da:       48 83 ec 10             sub    $0x10,%rsp
  4004de:       bf 01 00 00 00          mov    $0x1,%edi
  4004e3:       e8 0a 00 00 00          callq  4004f2 <inc>
  4004e8:       89 45 fc                mov    %eax,-0x4(%rbp)
  4004eb:       b8 00 00 00 00          mov    $0x0,%eax
  4004f0:       c9                      leaveq
  4004f1:       c3                      retq

00000000004004f2 <inc>:
  4004f2:       55                      push   %rbp
  4004f3:       48 89 e5                mov    %rsp,%rbp
  4004f6:       89 7d fc                mov    %edi,-0x4(%rbp)
  4004f9:       8b 45 fc                mov    -0x4(%rbp),%eax
  4004fc:       83 c0 01                add    $0x1,%eax
  4004ff:       5d                      pop    %rbp
  400500:       c3                      retq
  400501:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  400508:       00 00 00
  40050b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

```
# gcc  -Wall  function.c  -o  function
# objdump  -S  function > disassembly.txt
```

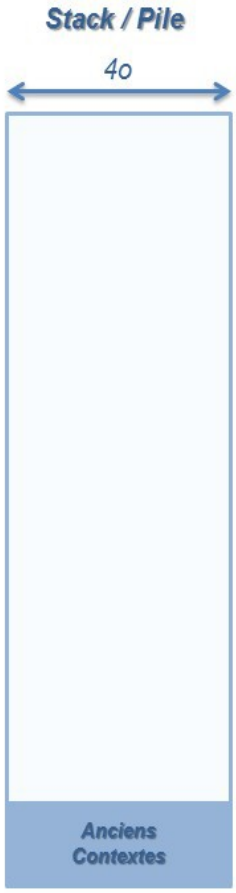
- L'instruction *call* réalise deux opérations. Elle modifie le pointeur programme (écriture dans PC ou IP). Elle sauvegarde également l'adresse de retour sur la pile et modifie donc SP. Dans le listing ci-dessus, quelle est la valeur de l'adresse de retour ?

- Quelle est l'adresse de la fonction *inc* ?

- Bien observer le code binaire (opcode) équivalent à l'instruction *call inc*, que constatez-vous ? Quel est l'opcode de l'instruction *call* pour un adressage relatif 32bits (cf. doc Intel) ?

- Proposer une autre écriture des instructions *call* et *ret* via les instructions *push*, *pop* et *jump*.

- Représenter graphiquement au *crayon gris* l'usage fait de la pile par le programme. Compléter le schéma ci-dessous.



- Observer les mécanismes de gestion de la variable locale (ou paramètre, c'est la même chose !) présente dans la fonction *inc. Critiquer les choix opérés par la chaîne de compilation.*

- Écrire à la main une solution assembleur plus performante (sans toucher au code généré pour la fonction main).

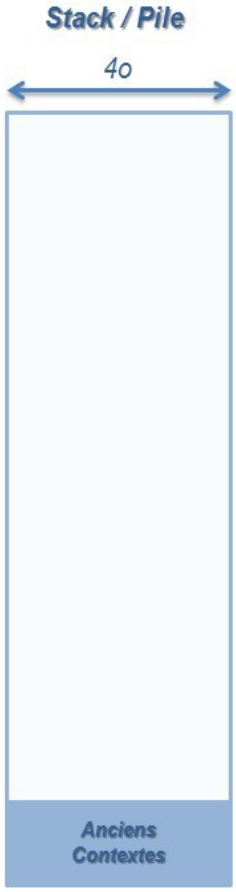
- Modifier le prototype de la fonction par *char inc (register char bar).* Interpréter le code générer et en déduire le rôle de cette classe de stockage.

Le mot clé *register* est une *classe de stockage* demandant (sans l'imposer) à la chaîne de compilation de manipuler, si les ressources matérielles le permettent, les variables ainsi qualifiées par registre CPU et non par la pile. Il s'agit d'un mécanisme simple d'optimisation permettant de limiter légèrement l'empreinte mémoire d'un programme mais pouvant augmenter significativement ses performances en évitant des allez/retours en lecture/écriture avec la pile présente en mémoire principale (DDR SRAM).

- Représenter graphiquement au *crayon gris* l'usage fait de la pile par le programme. Compléter le schéma ci-contre.

- Étudions maintenant un mot clé arrivé avec la norme *ISO C99* du langage C et pouvant être très puissant si bien utilisé, les fonctions *inline*. Modifier les prototypes à la déclaration et la définition de la fonction *inc* comme ci-dessous, compiler et analyser le code résultant ?

```
/* privates prototypes */
inline int inc( int bar) __attribute__((always_inline));

...

inline int inc( int bar) {
return bar+1;
}
```



- Quels sont les avantages et inconvénients de l'*inlining* de fonction ?

- Proposer une solution équivalente à l'aide de *macro fonction*.

Pour information, sachez qu'une fonction peut également être déclarée avec le mot clé *static*. Le cas échéant, si votre chaîne de compilation le supporte, elle implémentera les mécanismes suivants :

- Inlining éventuel de la fonction (élimine l'overhead inhérent à un appel de fonction)
- Utilise sinon des saut relatif courts plutôt que des sauts absolus longs.
- N'empile que les registres strictement nécessaire à la fonction (peut être différent de la convention par défaut utilisée par la toolchain)
- Limite la portée de la fonction au source courant
- Selon la chaîne de compilation, peut également supprimer toute trace de la fonction dans l'exécutable de sortie

2.5. Limites de la pile

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86x64/except/grow.c*

```
int main( void ){
    // allocations sur la pile jusqu'à débordement
    main();

    return 0 ;
}
```

- Que constatons-nous ? De quel segment logique s'agit-il ?

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/stackOverflow.c*

- Qu'elle est la taille actuelle de la pile associée au programme ?

- Interpréter le code source de ce fichier puis le modifier de façon à obtenir une pile de *16Mo*. Tester et vérifier les modifications apportées.

ALLOCATIONS STATIQUES

FICHIERS ELF


notes

3. ALLOCATIONS STATIQUES ET FICHIER ELF

Dans cette nouvelle partie, nous allons nous intéresser aux allocations statiques de ressources, c'est à dire aux allocations réalisées à la compilation contrairement aux allocations automatiques (pile/stack) et dynamiques (tas/heap) qui sont allouées à la volée durant l'exécution. Nous nous focaliserons donc sur les sections et segments présents dans le fichier ELF binaire exécutable. Une allocation statique garantie la non modification de l'emplacement mémoire d'une ressource du début à la fin de l'exécution du programme courant puisque déjà présente dans le fichier exécutable de sortie. D'où la dénomination de *static*.

3.1. Variables globales

- Éditer puis compiler le programme présent dans le répertoire de TP */x86x64/static/global.c* en s'arrêtant à la phase d'assemblage.

```
/*  privates declarations */
static char foo[1000000] = {'a'};

/* program entry point */
int main(void){
        *foo = 'A';
return 0;
}
```

- Insérer le code assembleur généré à votre fichier source C sous forme de commentaires puis Interpréter son fonctionnement.

- Compiler le programme précédent en allant jusqu'à l'édition des liens incluse, puis observer la taille du fichier exécutable de sortie. Comment expliquer le résultat obtenu ?

- En utilisant l'utilitaire *objdump* du package *binutils* du projet GNU, observer le contenu du fichier ELF exécutable après compilation et édition des liens. Observer, après désassemblage du binaire, l'adresse du premier élément du tableau présent dans la section *.text* label *main*. Pourquoi l'adresse générée n'est-elle pas une adresse absolue mais relative à IP (Instruction Pointer) ?

```
# gcc  -Wall  global.c  -o  global
# objdump  -S  global
```

- Durant le développement d'une application, en cas d'absolue nécessité concernant l'allocation de larges ressources mémoire, quelle solution est à préconiser afin de garder une gestion optimale des ressources mémoire de la machine ?

- Sur quels types de machines peut-on utiliser sans risque la fonction standard de libération de ressources mémoire *free* ?

Il faut savoir que durant l'édition des liens, toute chaîne de compilation C travaille de façon très ordonnée et structure l'espace mémoire virtuel/logique de l'application en segments (regroupement de sections) ne pouvant se chevaucher (aussi appelé Virtual Memory Area ou VMA sous Linux). Prenons l'exemple de quelques unes des principales sections : *.text* (pour le code), *.data*, *.bss*, *.rodata* … Après compilation, nous pouvons observer la table des sections en utilisant l'utilitaire *objdump* de *binutils* (option h, comme header sections) :

- *gcc foo.c -o foo*
- *objdump -h foo*

Dans les langages compilés comme le langage C, une donnée est soit allouée statiquement à la compilation soit allouée dynamiquement/automatiquement à l'exécution. Par essence, les langages interprétés ne peuvent qu'allouer des ressources durant l'exécution (allocations dynamiques) :

- *Allocations statiques :* allocations à la compilation et structuration des familles de variables en sections (variables globales ou statiques initialisées, non-initialisées , constantes …). Les allocations statiques de ressources (variables et codes) sont présentes dans le fichier exécutable de sortie (fichier ELF). Principales sections de données développeur. Adressage relatif à IP :
  - *.data :* section mémoire où se situe les données statiques initialisées
  - *.bss :* section mémoire où se situe les données statiques non initialisées
  - *.rodata :* données statiques initialisées en lecture seule (read-only)
- *Allocations automatiques :* allocations/libérations à l'exécution sur la pile ou stack : contextes d'exécution courant (encapsulé entre BP et SP, variables locales, paramètres de fonctions) et sauvegardes des contextes (adresses de retour des fonctions, contextes d'exécution des fonctions - BP)
- *Allocations dynamiques :* allocations/libérations à l'exécution sur le tas ou heap (malloc, realloc, free ... et variantes)

Pour information, ne jamais utiliser la fonction *free*, amenant une fragmentation de la mémoire physique, sur un processeur ne possédant pas de PMMU (unité de pagination). *Par exemple sur un MCU, nous n'effectuerons que des allocations statiques et automatiques.*

3.2. Variables locales statiques

- Éditer puis compiler le programme *static.c* en s'arrêtant à la phase d'assemblage.

```
int main(void){
        static char foo = 'a';
        foo++;
return 0;
}
```

- Insérer le code assembleur généré au fichier source C sous forme de commentaires puis Interpréter son fonctionnement.

- La variable *foo* est-elle gérée par la pile ? Dans quelle section du binaire de sortie est rangée cette variable (changer sa valeur et effectuer plusieurs compilations afin de répondre à la question) ? S'aider de l'utilitaire *objdump* (*objdump -s <exec_name>*)

- Pourquoi une variable locale statique peu préserver les valeurs qui lui ont été précédemment affectées durant les appels antérieurs de la fonction dans laquelle elle a été déclarée ?

- Nous allons maintenant utiliser l'utilitaire *strip* (décaper/enlever) de *binutils*, très utilisé dans le domaine de l'embarqué. Compiler votre fichier source jusqu'à l'obtention d'un fichier exécutable puis observer son empreinte mémoire. ''Stripper'' le fichier puis observer à nouveau son empreinte mémoire. Expliquer en quoi le processus d'exécution de ce binaire sera plus efficace ?

```
# gcc -Wall -m32 static.c -o static
# strip ./static
```

- Une variable locale *static* possède une portée limitée à la fonction courante. Ce point n'est géré qu'à la compilation. Physiquement et spatialement, ces variables sont très proches des variables globales.

- Le package *binutils* ou *GNU Binutils* est un ensemble d'outils GNU dédiés à l'analyse et la manipulation de fichiers binaires ELF et DWARF. Le service *strip* permet de nettoyer le contenu de fichier binaire en retirant notamment les informations inutiles (debug, références symboliques, concaténation de sections, etc) et en concaténant notamment certaines sections (table des sections allégée). *gcc* peut stripper à la compilation via l'option *-s*.

3.3. Chaînes de caractères

- Éditer puis compiler le programme *string.c* en s'arrêtant à la phase d'assemblage. Ajouter la commande *-fno-stack-protector* à la compilation, sans quoi le compilateur insérera du code pour la protection de débordement de buffer (*-fstack-protector*)

```
/* program entry point */
int main(void){
        char* pfoo = "foo";
        char bar[] = "bar";

return 0;
}
```

- Insérer le code assembleur généré à votre fichier source C sous forme de commentaires puis Interpréter son fonctionnement.

- Comment expliquer la taille de la variable *pfoo* ?

- Comment est allouée la chaîne de caractères *''toto''* ?

- Comment est allouée la chaîne de caractères *''titi''* ? Ma phrase précédente a-t-elle un sens ?

- En utilisant l'utilitaire *objdump* du package *binutils* du projet GNU, observer le contenu du fichier ELF exécutable après compilation et édition des liens du fichier *string.c* . Dans quelle section a été rangée la chaîne de caractères *''toto''* ? Est-ce normal ? Justifier votre réponse.

```
# gcc  -Wall  -m32  string.c  -o  string
# objdump  -s  string
```

Dans le cadre d'allocations statiques, bien constater que toutes les allocations sont réalisées à la compilation et sont donc présentes dans le fichier exécutable de sortie (fichier ELF dans notre cas).

ALLOCATIONS DYNAMIQUES

GESTION DU TAS


notes

4. ALLOCATIONS DYNAMIQUES ET GESTION DU TAS

Dans cette partie, nous allons nous intéresser aux variables allouées dynamiquement sur le tas ou heap. Cette solutions est à préconiser dès que votre applicatif nécessite de large ressources mémoires. *Ne jamais oublier de libérer les ressources allouées après usage.*

4.1. Gestion du Tas

- Éditer puis compiler le programme */x86x64/heap/heap.c* en s'arrêtant à la phase d'assemblage . Ce code étant très simple, analyser rapidement l'assembleur généré.

```
#include  <stdlib.h>

/* program entry point */
int main(void){
        char *pfoo;

        pfoo = (char *) malloc (100 * sizeof(char));
        *(pfoo + 7) = 1;
        free (pfoo);

return 0;
}
```

- Comparer une adresse allouée sur le tas à un adresse allouée sur la pile. Constater qu'il s'agit d'espaces mémoire virtuels distincts (affichage via printf d'un pointeur sur la pile et sur le tas).

4.2. Limites du Tas

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86x64/except/heapOverflow.c*. Ce programme permet de tester la taille limite du Tas.

- Qu'elle la taille limite théorique du Tas sous système Linux (information système) ?

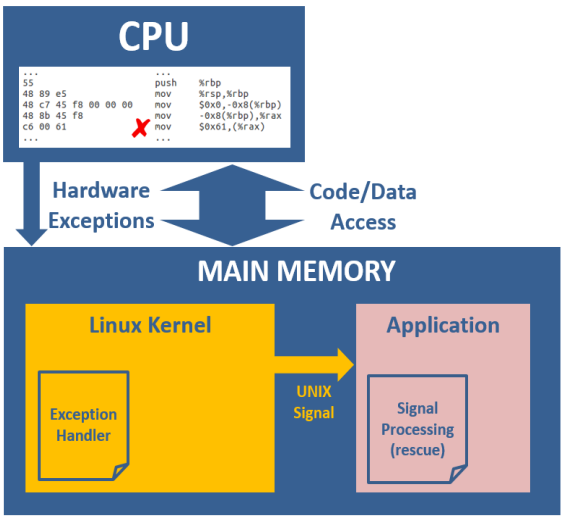
EXCEPTIONS MATERIELLES

SIGNAUX UNIX


notes

5. EXCEPTIONS MATERIELLES ET SIGNAUX UNIX

Dans cette nouvelle partie, nous allons nous intéresser aux exceptions matérielles générées par l'ensemble CPU/MMU. Une exception est un événement matériel synchrone généré par le CPU (synchrone au regard du fonctionnement d'un CPU dont les traitements restent synchronisés sur une référence d'horloge, et non au regard de la probabilité d'occurrence). Ces événements sont relevés par le CPU lorsque celui-ci détecte une voire plusieurs conditions prédéfinies durant l'exécution d'une instruction (violation de privilège, division flottante par zéro, accès illégal en mémoire …). Rappelons les traitements réalisées par l'ensemble CPU/MMU et le systèmes d'exploitation lorsque qu'une exception matérielle survient :



- Le programme en cours d'exécution génère une erreur connue du CPU. Les familles et types d'exceptions sont toutes prédéfinies et connues de l'architecture.

- Le CPU stoppe l'exécution du programme en cours et appel une procédure enfouie et endormie du système. Une sauvegarde du contexte CPU (registres internes) est également réalisée et peut être accessible depuis un applicatif en espace utilisateur. Pour les curieux, il s'agit de la fonction *do_page_fault* dans le cas de Linux (présente dans le fichier */arch/<cpu>/mm/fault.c* du système de fichier du kernel Linux)

- La procédure d'exception appelée (ou exception handler) est chargée de relever le type de défaut et de générer, si l'exception le permet, un signal logiciel UNIX à destination du processus ayant généré le défaut.

- Si le processus ne capte et ne traite pas le signal UNIX, il est alors tué par le système. Ceci assure un *cloisonnement des défauts et la stabilité du système*.

- Si le processus gère le signal UNIX, votre applicatif peut alors tenter d'acquitter le défaut (restaurer un contexte CPU viable par exemple) ou de réaliser un traitement spécifique assurant une robustification ou durcissement de votre applicatif (message d'erreur, redémarrage de l'application, mode dégradé, fichier de log ...).

5.1. Lecture seule

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/readonly.c*. Ce programme force la génération d'une exception matérielle de type *Fault*.
- Interpréter et expliquer le résultat obtenu. Quel type de signal UNIX a été envoyé au processus à la source du défaut ? Préciser le segment concerné ?
- Durant les exercices précédents, vous avez été amenés à forcer des débordements de pile et de tas. Quelles types et familles d'exceptions et de signaux ont été levés dans les deux cas (s'aider du cours) ?

5.2. Défaut d'alignement

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/buserror.c*.
- Interpréter et expliquer le résultat obtenu. Quel type de signal UNIX a été envoyé au processus à la source du défaut ?

5.3. Division par zéro

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/divzero.c*.
- Interpréter et expliquer le résultat obtenu. Quel type de signal UNIX a été envoyé au processus à la source du défaut ?

5.4. Debug

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/sigtrap.c*. Ce programme force la génération d'une exception matérielle de type *Trap*. Dans certains cas, ce type d'exception peut-être utilisé par les outils de debug.
- Interpréter et expliquer le résultat obtenu. Quel type de signal UNIX a été envoyé au processus à la source du défaut ?

5.5. Pointeur nul

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/nullptr.c*. Ce programme force la génération d'une exception matérielle de type *Fault*.
- Interpréter et expliquer le résultat obtenu. Quel type de signal UNIX a été envoyé au processus à la source du défaut ?

5.6. Signal UNIX

- Éditer, compiler puis exécuter le programme présent dans le répertoire de TP */x86/except/ackdefault.c*. Ce programme force la génération d'une exception matérielle de type *Fault*.
- Analyser le résultat obtenu.
- Dé-commenter la partie propre à la configuration et la gestion de signaux UNIX pour votre application (gestion du signal UNIX SIGSEGV). Compiler, exécuter puis analyser le résultat obtenu.
- Sachant que le contexte CPU est sauvé dans la structure *context->uc_mcontext* déclarée dans le fichier d'en-tête *ucontext.h* (se documenter sur internet pour répondre à cette question) :
    - Identifier le registre à la source du défaut (désassemblage de votre fichier exécutable)
    - Observer puis restaurer un contexte CPU viable depuis le handler de signal
    - Acquitter le défaut puis rendre la main au processus
    - Proposer l'affichage suivant depuis le shell

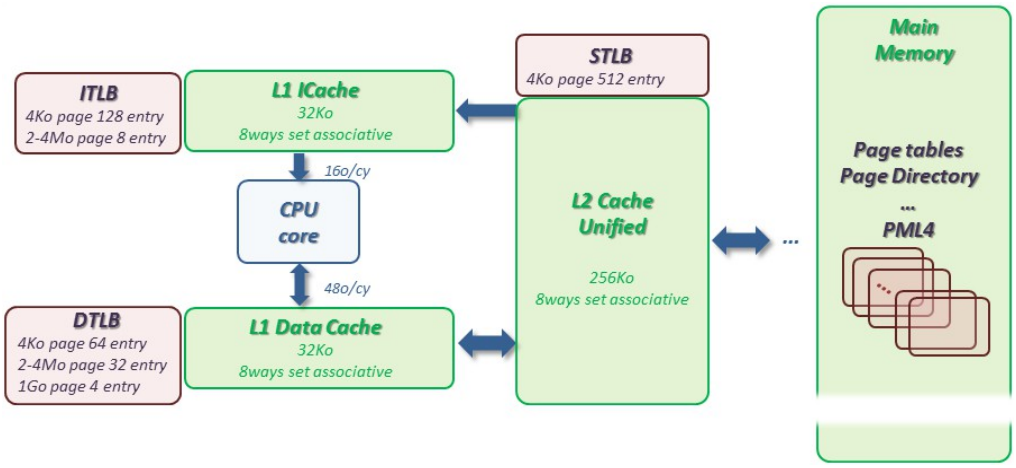
```
hardware exception has been generated

 *** signal handler
 *** current cpu context eax=0 (dereferenced pointer)
 *** new cpu context eax=0x?????? (referenced pointer)
 *** resume processus

exception has been acknowledged, tmp = 'a'
```

UNITE DE PAGINATION


notes

6. UNITE DE PAGINATION

Dans cette ultime partie, nous allons nous intéresser au travail réalisé par la MMU (Memory Management Unit). Rappelons que sur architecture x86-x64 la MMU réalise deux grands traitement sur les adresses virtuelles, la segmentation et la pagination. Néanmoins, nous avons pu voir en cours que l'étage de segmentation est transparent à notre époque (segmentation logique réalisée à la compilation par les outils et au chargement par le système). De ce fait, les MMU sont souvent nommées PMMU (Paged MMU) à notre époque.



Exemple d'architecture des TLB's
sur la micro-architecture Intel Sandy Bridge

- Rappeler le travail réalisé par la PMMU ou l'unité de pagination ?

- Rappeler les granularités possibles de la mémoire physique ? Ces granularités sont des services proposés par le processeur et doivent ensuite être gérées par le système d'exploitation.

- Rappeler ce qu'est une TLB ?

- Analysons le travail réalisé par la fonction *malloc*. Analyser le projet présent dans le répertoire *mmu* puis le compiler. Sachant que cette application travail sur des tableaux de grande taille, nous allons constater qu'en appelant la fonction malloc le système prendra l'initiative d'allouer des *huge pages* de 2Mo plutôt qu'un grand nombre de pages de 4Ko.

  Estimer par le calcul le nombre de huge pages de 2Mo nécessaires au stockage de nos différentes matrices et l'empreinte mémoire équivalente en Ko ? Conditions : SIZE = 2048 dans le fichier *bench.c*

- Le fichier */proc/meminfo* permet de visualiser quelques statistiques de l'usage fait de la mémoire par le système (fichier mis à jour en temps réel par le système). Lancer la commande *watch* ci-dessous permettant de répéter toutes les secondes l'affichage du contenu du fichier *meminfo* puis lancer le script *perf.sh* . Constater l'allocation mémoire de huge pages (champ *AnonHugePages*, les autres champs sont obsolètes) et valider le calcul de la question précédente.

```
# watch -n 1 ''cat /proc/meminfo | grep Huge''
```

- Pour les étudiants travaillant sur leur machine personnelle (droit *root* exigé), nous allons maintenant forcer le système à désactiver l'allocation de *huge pages* puis constater la différence de performance entre les deux stratégies d'allocation.

- Modifier la configuration par défaut du système.

```
# sudo chmod 666 /sys/kernel/mm/transparent_hugepage/enabled
# echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

- Lancer le script *perf.sh* et constater le nombre de dTLB-load-misses :

- Restaurer la configuration par défaut du système, lancer le script *perf.sh* et constater le nombre de dTLB-load-misses :

```
# echo always > /sys/kernel/mm/transparent_hugepage/enabled
```

- Comparer également les temps d'exécution du programme dans les deux cas de figure et constater une dégradation des performances liée à un mauvais usage de la PMMU par le système.

- La question qui suit est issue d'un problème rencontré par un industriel partenaire. En effet, pour certains de ses algorithmes, il pouvait constater un grand manque de déterminisme avec des écarts dans les temps d'exécution allant jusqu'à 200% après plusieurs exécutions successives et des temps d'attente de plusieurs minutes. Son programme implémenté initialement la première solution.

  Les deux exemples qui suivent réalisent l'allocation dynamique de la même empreinte mémoire (à quelques % près).

Solution 1

```
#define  SIZE  1024

int **pVect;
int i ;

pVect = (int **) malloc(sizeof(int*)*SIZE);

for(i=0; i<SIZE; i++)
     pVect[i] = (int *) malloc(sizeof(int)*SIZE);
```

Solution 2

```
#define  SIZE  1024

int *pMatrix;

pMatrix = (int *) malloc(sizeof(int*)*SIZE*SIZE);
```

- En quoi la première solution est plus intéressante dans le cadre d'algorithmes implémentant du calcul matriciel ?

- Expliquer le travail réalisé par le système dans le premier cas et justifier en quoi la seconde solution est plus performante.