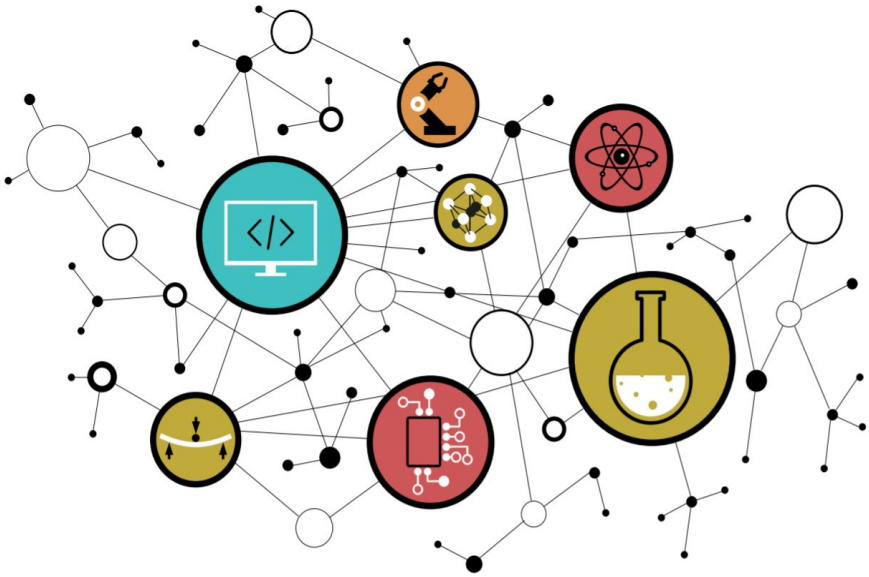
TRAVAUX PRATIQUES

ALLOCATIONS AUTOMATIQUES ET SEGMENT DE PILE



Hugo Descoubes – neku – GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2022

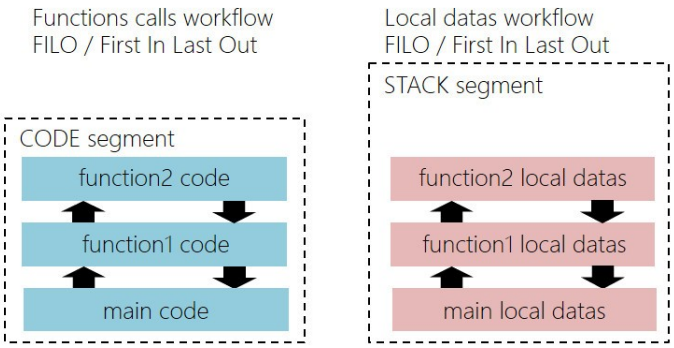
SOMMAIRE

4. ALLOCATIONS AUTOMATIQUES ET SEGMENT DE PILE

4.1. Fonction main
4.2. Variables locales initialisées
4.3. Variables locales non-initialisées
4.4. Appel et paramètres de fonction
4.5. Fonction inline et optimisation
4.6. Limites de la pile
4.7. Synthèse

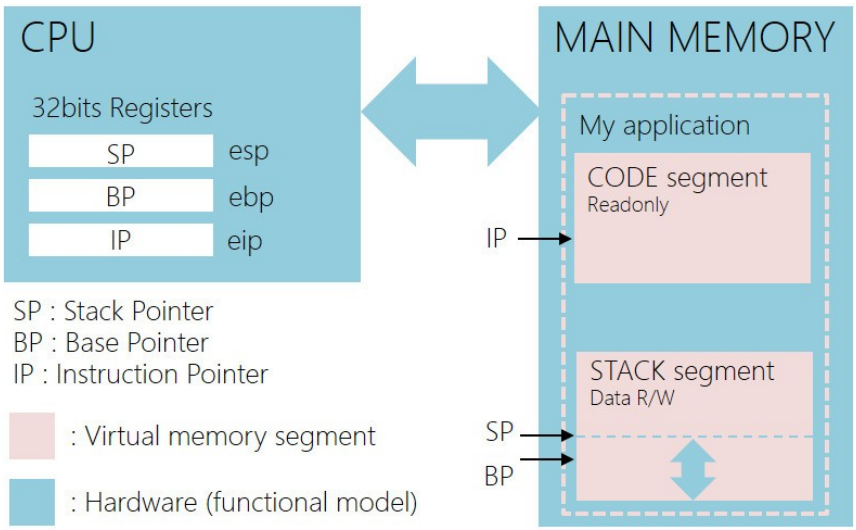
4. ALLOCATIONS AUTOMATIQUES ET SEGMENT DE PILE

```
int main (void)
{
    /* local user code and datas */
    function1();
   return 0;
}

void function1 (void)
{
    /* local user code and datas */
   function 2();
}

void function2(void)
{
    /* local user code and datas */
}
```



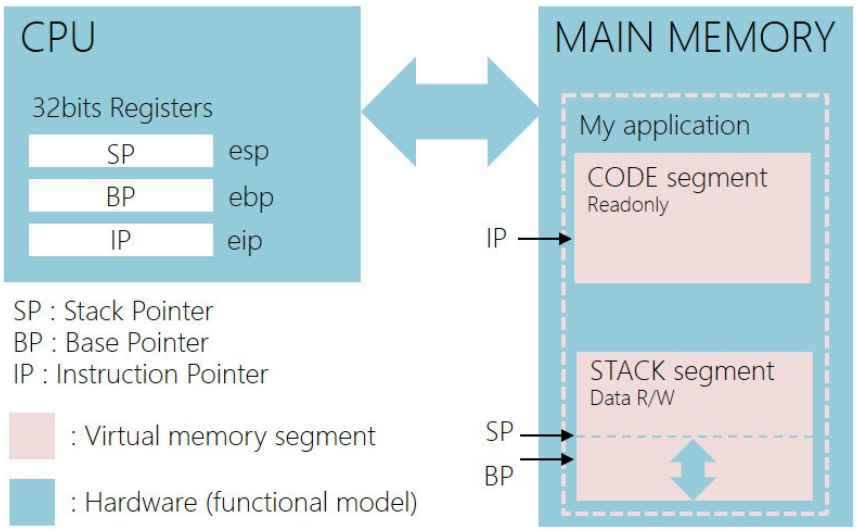
Les allocations de ressources mémoire réalisés dynamiquement à l'exécution durant la manipulation de variables locales se font sur le segment de pile ou *stack*. Ce segment est présent en mémoire principale durant l'exécution d'un programme. *Un segment est une zone logique contigu virtuelle allouée en mémoire principale par le noyau du système avant l'exécution d'un programme*. Rappelons que pour des raisons de robustesse (spatialité des usages), les variables locales sont les variables les plus couramment utilisées dans le monde du logiciel. Contrairement aux variables globales à n'utiliser qu'en cas d'absolue nécessité (ressources partagées).

En langage C, comme dans beaucoup d'autres langages (C++, Java, D, etc), le point d'entrée d'une application est la fonction *main*. De même, si nous réalisons des appels de fonctions imbriqués depuis le *main* (cf. exemple ci-dessus) et si nous souhaitons revenir à la fonction principale de façon conventionnelle, nous aurons à quitter dans l'ordre d'appel toutes les fonctions respectivement appelées. Les appels de fonctions sont gérés telle une pile de papier (LIFO, Last In First Out) et il en va de même pour les variables locales. Les variables locales à une fonction seront allouées automatiquement en entrée de fonction à l'exécution. A l'usage, toutes les variables locales seront stockées dans le segment de pile, qui sera toujours de taille fixe. Chaque application possède sa propre pile. Il existe au minimum autant de piles que de programmes chargés et démarrés (processus) en mémoire principale par le noyau Linux. Par défaut sur ordinateur sous Linux, chaque pile applicative offre 8Mo potentiel d'espace mémoire de stockage. Le segment de code, contenant le code binaire exécutable du programme, est donc spatialement séparé du segment de pile (cf. schéma ci-dessous), contenant uniquement des données accessibles en lecture et écriture. Ce cloisonnement spatial est nécessaire à la robustesse globale de l'ordinateur et est conjointement réalisé et supervisé par l'unité matérielle de pagination (PMMU ou Paged Memory Management Unit) qui est elle même exploitée par le noyau du système.



4.1. Fonction main

Se placer dans le répertoire *disco/stack/* afin d'appliquer la totalité des commandes qui suivent. Nous analyserons de l'assembleur 32bits x86 (option -m32). Par la suite, nous n'analyserons plus que de l'assembleur 64bits x64. L'objectif étant de pouvoir aisément distinguer les 2 technologies et de constater qu'il n'existe pas de grande différence fondamentale à la lecture de script assembleur élémentaire.



Il est important de noter que le segment de pile possédera toujours une taille fixe (8Mo par défaut sous Linux). Sur machine x86 32bits, SP (Stack Pointer) est toujours sauvé dans le registre CPU 32bits *ESP* et pointe toujours le sommet de la pile. BP (Base Pointer ou Frame Pointer) est toujours sauvé dans le registre CPU 32bits *EBP* et est propre à chaque fonction. Tant que nous nous trouvons dans une fonction, ce pointeur ne sera pas modifié et pointera vers la même zone mémoire. IP (Instruction Pointer ou PC ou Program Counter) sera toujours sauvé dans le registre CPU 32bits *EIP* et pointera toujours la future instruction à exécuter. Pour information, par convention, nous représentons toujours un *mapping* mémoire avec les adresses basses en haut de page et les adresses hautes en bas en bas de page (cf. schéma suivant).

- Compiler le fichier *main.c* en s'arrêtant à la phase d'assemblage. Analyser le fichier assembleur généré. S'aider d'internet, de vos voisins de table, de l'enseignant encadrant, etc.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-
protector -fcf-protection=none -Wall -m32 main.c
```

- Compléter (au crayon) le schéma de la pile sur la page suivante en précisant quel est son contenu suite à l'exécution du programme jusqu'à l'instruction *ret* en fin du *main*. Ne pas oublier qu'avant le début de notre programme, le code de la fonction de *startup* s'est exécuté.

- Proposer une réécriture des instructions CISC-like (Complex Instruction Set Computing) *push* et *pop* à l'aide des instructions RISC-like (Reduce Instruction Set Computing) *sub*, *add* et *mov*

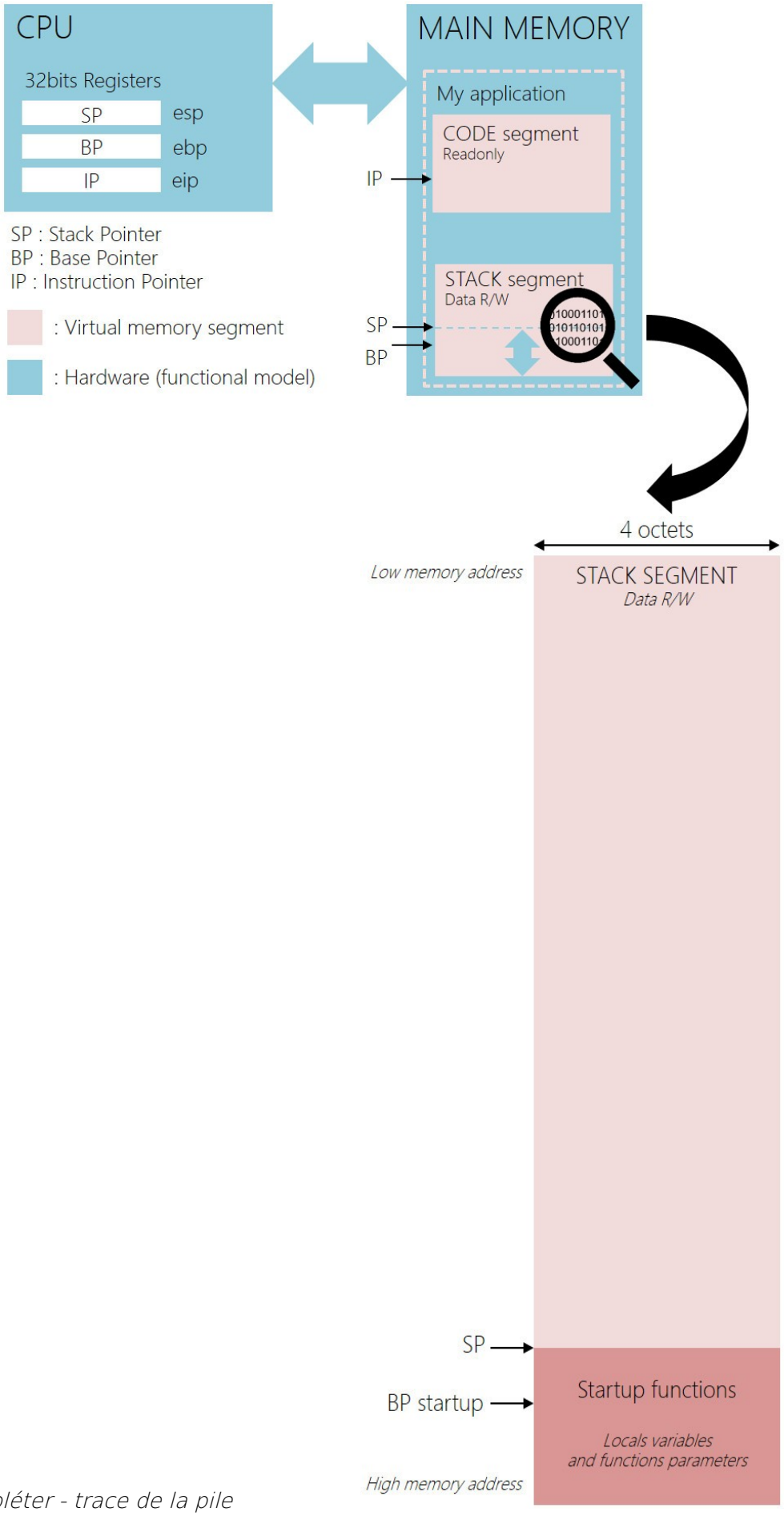


Schéma à compléter - trace de la pile

4.2. Variables locales initialisées

- Ouvrir puis compiler le fichier *local_variable_init.c* en s'arrêtant à la phase d'assemblage. Analyser le fichier assembleur généré.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-
protector -fcf-protection=none -Wall -m32 local_variable_init.c
```

- Compléter (au crayon) le schéma de la pile sur la page précédente en précisant quel est son contenu suite à l'exécution du programme jusqu'à l'instruction *ret* en fin du *main*.
- Préciser les tailles ou empreintes mémoire des variables locales a,b,c et d
- Préciser les adresses relatives des variables locales a,b,c et d
- Combien faut-il de pointeurs, et donc de registres, afin d'adresser et de gérer un nombre quelconque de variables locales ?
- Dé-commenter le *cast* (transtypage) présent dans le programme, compiler et analyser le fichier assembleur de sortie. Analyser le résultat

```
c = (int) a;
```

- Qu'est-ce qu'une extension de signe en arithmétique entière signée Cà2 (Complément à 2) ?
- Qualifier le type de la variable *a* de *const*. Compiler le programme puis interpréter le résultat. Que constatons-nous ?
- Quel est le rôle du qualificateur de type *const* et donc son usage ?

4.3. Variables locales non-initialisées

- Ouvrir puis compiler le fichier *local_variable_uninit.c* en s'arrêtant à la phase d'assemblage. Analyser le fichier assembleur généré.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-
protector -fcf-protection=none -Wall -m32 local_variable_uninit.c
```

- Que constatons-nous ?

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

- Quel est le rôle de ce qualificateur de type *volatile* et donc son usage ? S'aider d'internet

Au regard du système cible (alchimie matérielle et logicielle), le compilateur est susceptible de réarranger des lectures et écritures sur des emplacements mémoire pour des raisons de performances. Les variables qualifiées de *volatile* ne sont pas soumises à ces optimisations (à la compilation comme à l'exécution). Ce mot clé peut notamment être utile en programmation multi-threads ou en programmation événementielle (interruptions, exceptions, etc). 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 volatiles potentielles de la ressource par d'autres entités. Ceci est utilisé dès qu'il s'agit de manipuler des variables critiques comme des ressources partagées (buffer d'échanges, flags, périphériques, etc) 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 voire forcer le compilateur à opérer des traitements privilégiés sur une variable. De façon générale, il s'agit de stratégies de durcissement ou robustification (verrouiller ou forcer un usage) voire d'optimisation (vitesse ou taille).

4.4. Appel et paramètres de fonction

A partir de maintenant, nous analyserons de l'assembleur 64bits compatible pour architectures x64. Retirer l'option -m32 à la compilation. Il s'agit de l'assembleur généré par défaut sur système GNU\Linux 64bits porté sur machine 64bits. Cela vous permettra d'apprécier les différences 32bits/64bits sur des programmes assembleurs élémentaires.

- Ouvrir puis compiler le fichier *function_parameters.c* en s'arrêtant à la phase d'assemblage. Analyser le fichier assembleur généré.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-
protector -fcf-protection=none -Wall function_parameters.c
```

- Compléter (au crayon) le schéma de la pile sur la page suivante en précisant quel est son contenu exhaustif suite à l'exécution du programme jusqu'à l'instruction *ret* en fin du *main*.

Les instructions d'appel de fonction comme *call* empilent également l'adresse de retour sur la pile en plus de réaliser un saut vers le code de la fonction appelée. Ce mécanisme permet d'assurer les appels et surtout les retours de fonctions. L'adresse de retour est l'adresse de l'instruction suivant spatialement le *call* en mémoire programme. Durant l'exécution d'un call, il s'agit donc du pointeur *IP* contenu dans le registre CPU *RIP* pointant toujours l'instruction suivante à exécuter. Ceci permettra après exécution de l'instruction *ret* présent dans la fonction appelée de revenir exactement dans la fonction appelante à l'instruction suivant spatialement en mémoire l'instruction *call*. L'instruction *ret* présente dans la fonction appelée dépile l'adresse de retour précédemment sauvée et la restaure dans le registre d'instruction du CPU (registre *RIP* dans notre cas). Rappelons que sur architecture x86/x64 (32bits/64bits), les registres EIP/RIP sont utilisés par l'étage de *fetch* du CPU afin d'aller chercher en mémoire programme les futures instructions à exécuter. Observons ci-dessous une réécriture en pseudo-code RISC des instructions *call* et *ret* :

CALL called_function_label	PUSH RIP MOV called_function_label, RIP
RET	POP RIP

- Pour le passage d'arguments de type entier, quels sont respectivement les 3 registres CPU utilisés par GCC pour passer les paramètres à une fonction appelée ? Quel est par défaut le registre utilisé pour passer une valeur de retour entière ?

- Quelles sont les adresses relatives des variables *ret_1* (variable locale à function_1) et *a_2* (variable locale à function_2) ? Pourquoi ne sont-elles pas spatialement au même emplacement mémoire sur la pile ?

- Observer la définition de la fonction *function_2* utilisant la syntaxe K&R (Kernighan & Ritchie) originelle du langage C . Au final, qu'est-ce qu'un paramètre de fonction ?

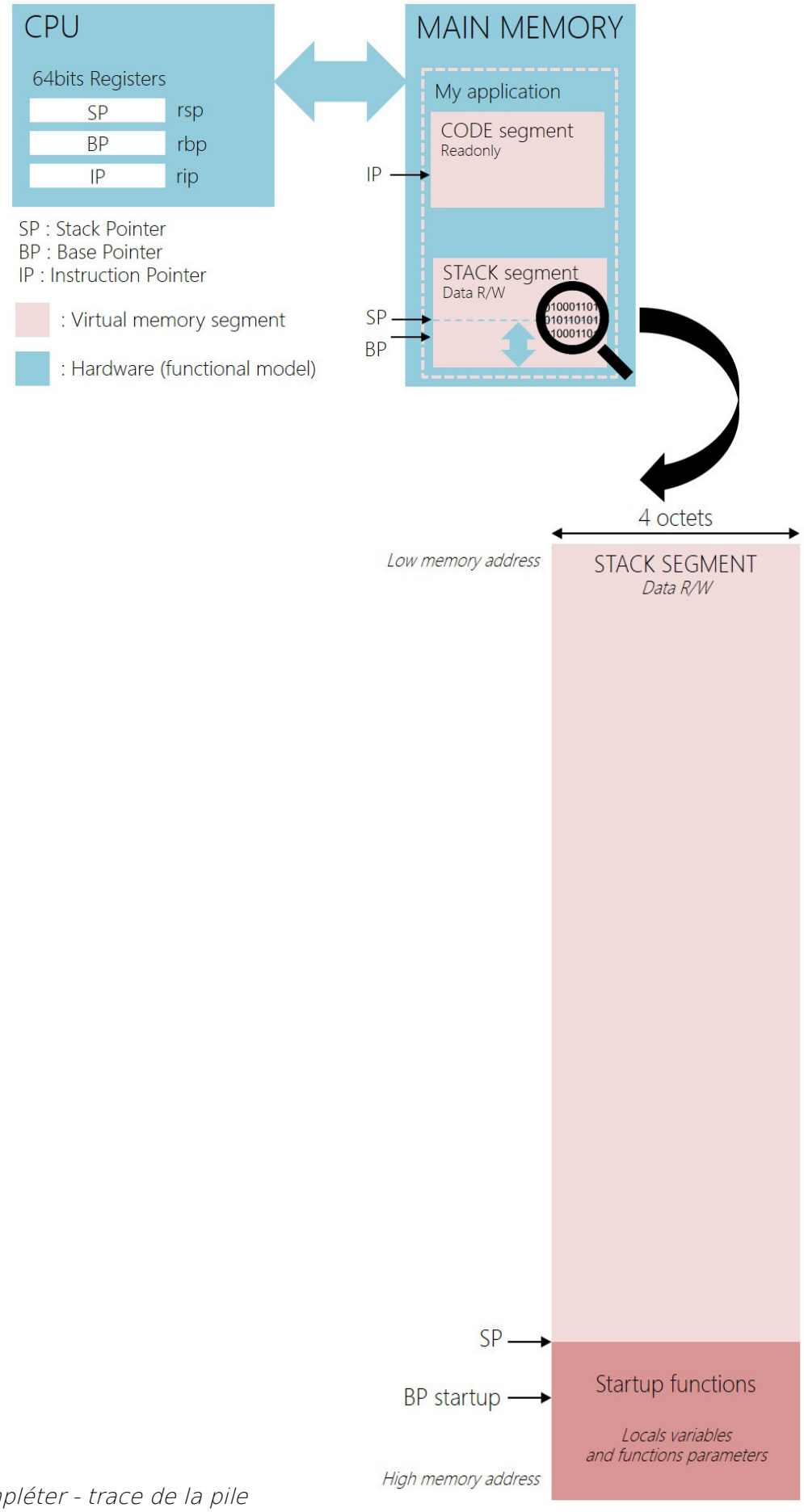


Schéma à compléter - trace de la pile

4.5. fonction inline et optimisation

Nous allons profiter de ce dernier exercice d'analyse de gestion de la pile pour étudier un appel de fonction avec passage d'arguments par pointeur. De même, nous analyserons les effets de quelques optimisations réalisées par GCC à la compilation.

- Ouvrir puis compiler le fichier *function_inlining.c* en s'arrêtant à la phase d'assemblage. Analyser le fichier assembleur généré.

```
gcc -S -fno-asynchronous-unwind-tables -fno-pie -fno-stack-
protector -fcf-protection=none -Wall function_inlining.c
```

- Compléter (au crayon) le schéma de la pile sur la page suivante en précisant quel est son contenu exhaustif suite à l'exécution du programme jusqu'à l'instruction *ret* en fin du *main*.

- Dé-commenter le prototype de la fonction *swap* utilisant la classe de stockage *register* pour la déclaration des paramètres de fonction et commenter le prototype générique. Faire de même au niveau de la définition de fonction. Qualifier également la variable *tmp* dans la fonction *swap* de *register* (à laisser jusqu'à la fin de l'exercice). compiler et analyser le code assembleur de la fonction *swap*. Quel est le rôle et l'intérêt de cette classe de stockage ?

```
//void swap(int* pt_a, int* pt_b);

void swap(register int* pt_a, register int* pt_b);

//inline void swap(int* pt_a, int* pt_b)__attribute__((always_inline));
```

Le mot clé *register* est une classe de stockage demandant (sans l'imposer) à la chaîne de compilation de manipuler les variables ainsi qualifiées par registre CPU et non en mémoire principale par la pile. Ce qualificateur ne peut-être géré que si les ressources matérielles le permettent (nombre de registres disponibles). 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 lectures/écritures avec la pile présente en mémoire principale (technologie lente de transfert DDR sur stockage DRAM en comparaison aux registres en technologie SRAM travaillant à la même fréquence que le CPU).

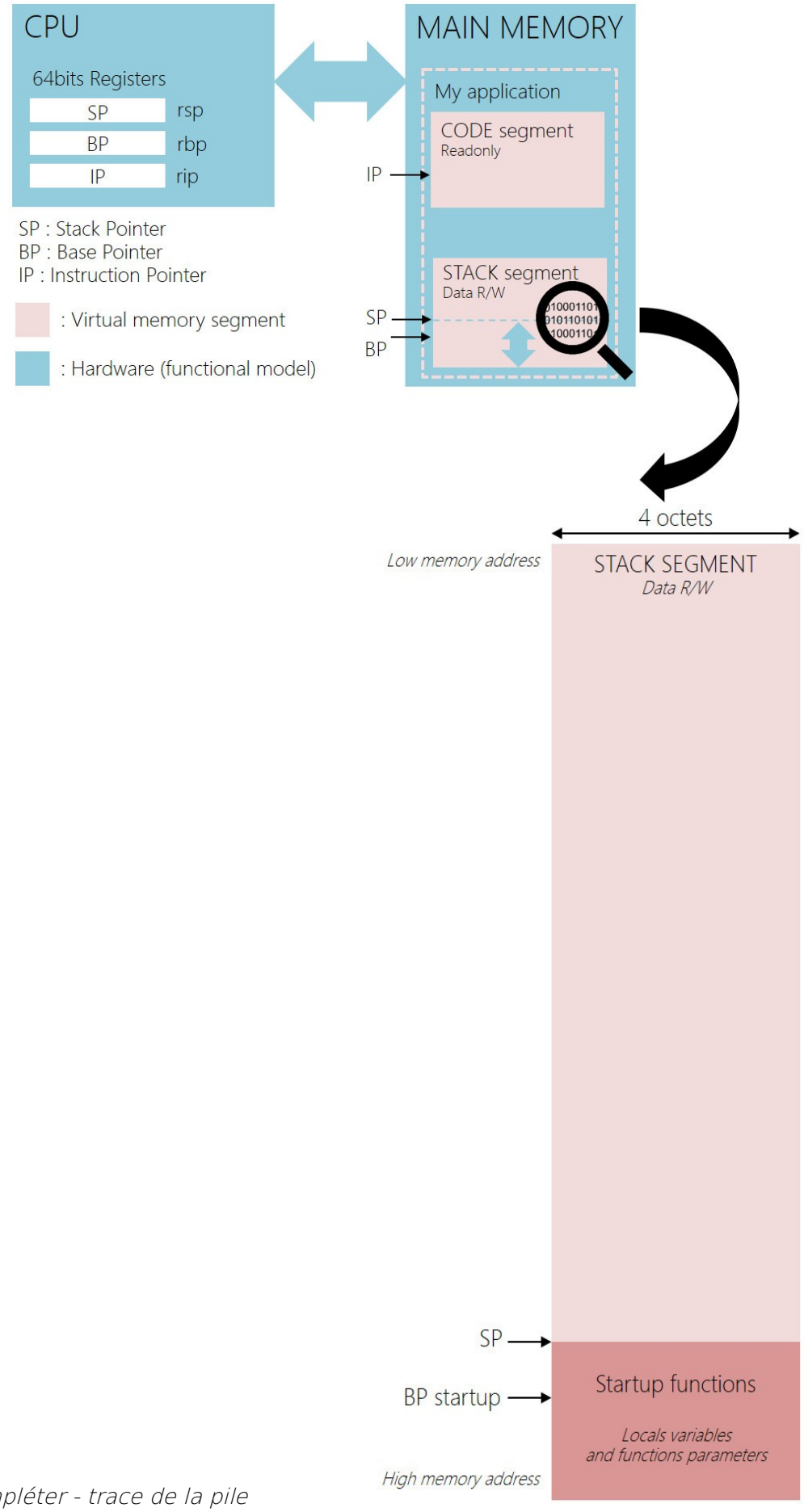


Schéma à compléter - trace de la pile

- Dé-commenter le prototype de la fonction *swap* qualifiée de *inline* (seulement à la déclaration) et commenter les prototypes génériques. Analyser le code assembleur de la fonction *main*. Quel est le rôle du mot clé *inline* arrivé avec la norme C99 ? Pourquoi le code de la fonction *swap* est-il toujours présent dans le programme assembleur et donc à terme dans le *firmware* alors que la fonction n'est plus appelée depuis le *main* ?

```
//void swap(int* pt_a, int* pt_b);

//void swap(register int* pt_a, register int* pt_b);

inline void swap(int* pt_a, int* pt_b) __attribute__((always_inline));
```

- Dé-commenter le prototype de la fonction *swap* qualifiée de *inline* au niveau de la définition de la fonction *swap* et commenter les prototypes génériques. Analyser le code assembleur généré. Verdict ?

```
//void swap(int* pt_a, int* pt_b)
//void swap(register int* pt_a, register int* pt_b)
inline void swap(int* pt_a, int* pt_b)
{
...
}
```

Le mot clé *inline* demande au compilateur d'insérer le code d'une fonction appelée dans le code de l'appelant en retirant l'overhead d'appel de fonction (call, push, pop, ret, etc). Ce mot clé s'applique donc à des fonctions courtes et permet d'augmenter les performances d'un programme. Néanmoins, le code étant dupliqué, en cas d'appels multiples ceci peut impacter la taille du firmware.

- Lever les options d'optimisation de niveau 1 (-O1) de GGC en laissant dé-commentée la fonction qualifiée de *inline* (déclaration et définition). Compiler et analyser la sortie. Observer les décisions radicales prises par GCC.

```
gcc -S -O1 -fno-asynchronous-unwind-tables -fno-pie -fno-stack-
protector -fcf-protection=none -Wall function_inlining.c
```

- Essayer le niveau maximal d'optimisation -O3 et analyser le programme de sortie.

4.6. Limites de la pile

- Nous allons maintenant tester les limites de notre pile applicative. Se déplacer dans le répertoire *disco/except/. C*ompiler le fichier *stack_overflow.c* puis l'exécuter.

```
gcc stack_overflow.c –o stack_overflow

./stack_overflow
```

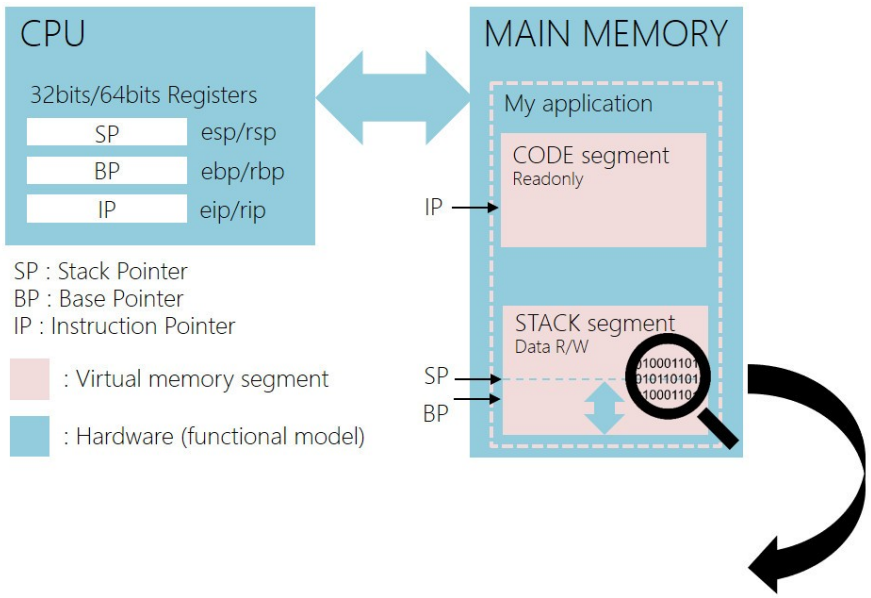
- Quel segment logique mémoire virtuel applicatif a subit un débordement et a causé la fin de notre programme ? Question facile !

- Qu'elle est la taille par défaut de la pile associée à notre programme ? Quelle entité sur la machine fixe et impose la taille de la pile associée à un programme ?

- Dé-commenter la section de code présente dans le programme. Compiler à nouveau le fichier *stack_overflow.c* puis l'exécuter. Qu'elle est la nouvelle taille de la pile associée à notre programme ? Analyser le code dé-commenté.

La fonction *setrlimit* (set application ressources limits) réalise un appel système au noyau Linux en lui demandant d'allouer à notre programme un segment logique virtuel de pile plus large que celui alloué par défaut. Rappelons que Linux est le gestionnaire et le garant du bon fonctionnement des ressources matérielles de la machine ainsi que de ses limites physiques comme logiques. Nous appelons souvent un système d'exploitation un superviseur, sous entendu de l'ordinateur. Tant qu'une application n'est pas trop gourmande en ressource, le kernel Linux s'efforcera de répondre à ses requêtes.

4.7. Synthèse

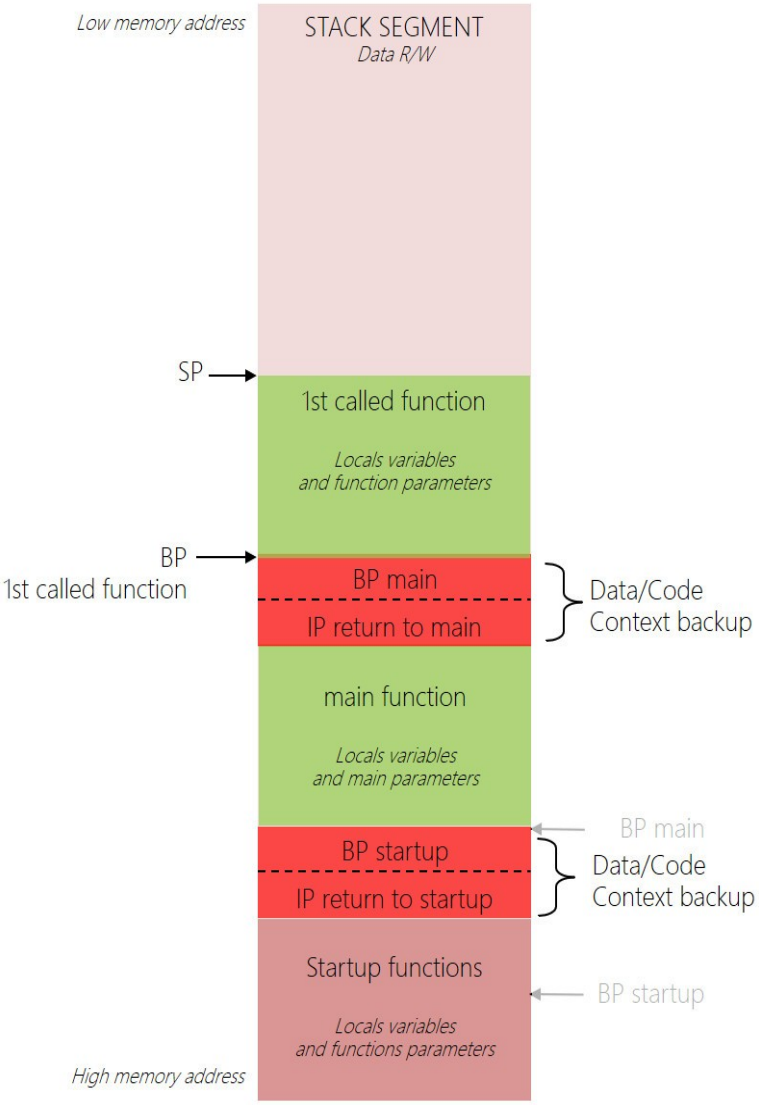


Les processus de gestion des variables locales et des paramètres de fonction (eux même des variables locales paramétrées) sont conjointement réalisés à la compilation par les outils de développement et exploités à l'exécution par la machine.

Variables locales et paramètres de fonction sont allouées dynamiquement à l'exécution suite à l'appel et durant l'entrée dans le code d'une fonction. Les premières lignes de code implémentant une fonction servent donc à sauvegarder le contexte d'exécution de la fonction appelante (BP pour les données et IP pour le code) puis à allouer sur la pile les ressources mémoire données nécessaires à son exécution.

Une fois dans une fonction, les pointeurs BP et SP encapsulent le plus souvent la zone mémoire sur la pile comprenant les variables locales et paramètres propres à cette même fonction. Tant que nous restons dans cette fonction, BP ne sera pas modifié. De ce fait, toutes les variables locales et paramètres de fonction possèdent une adresse mémoire relative au pointeur BP. Un seul et même registre (EBP/RBP 32bits/64bits x86/x64) permet donc indirectement d'adresser un nombre quelconque de variables locales.

Quant à lui, le segment de pile possédera toujours une taille fixe. Sur ordinateur, rappelons qu'un segment est une zone virtuelle contigu allouée en mémoire principale par le noyau du système à l'exécution. Ce segment n'admet aucune existence sur les médias de stockage de masse (HDD, SSD, etc).




notes