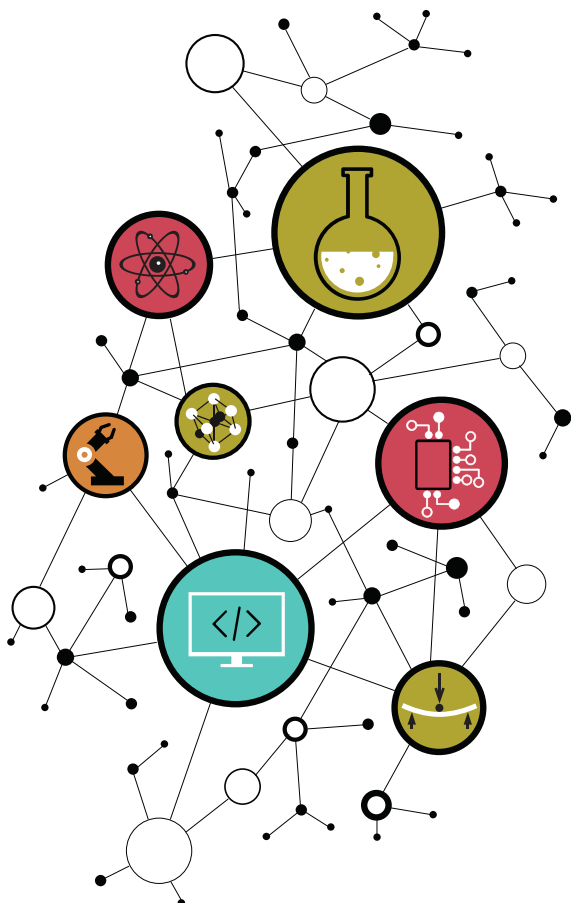


PARTIE 5

ALLOCATION AUTOMATIQUE ET SEGMENT DE PILE

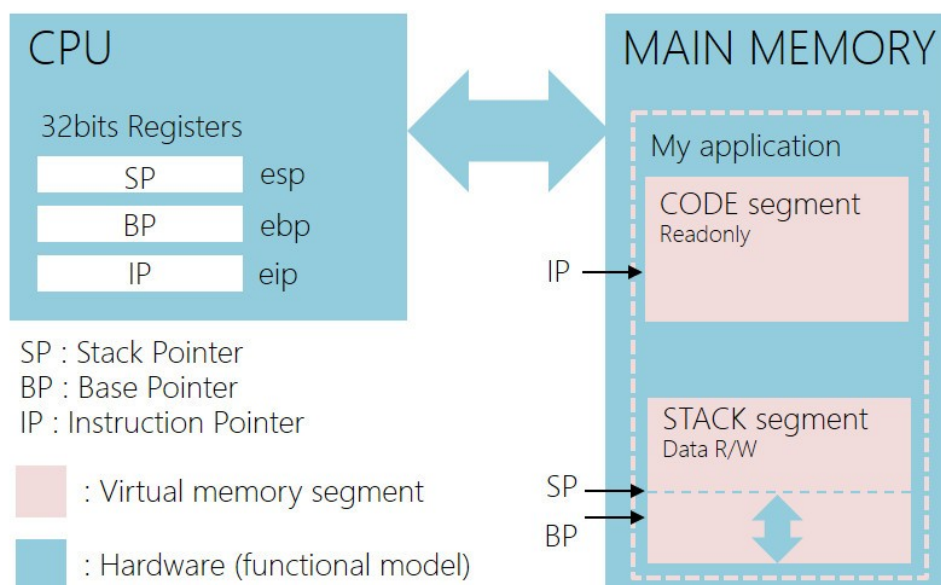


I. Segments de code et de pile

Nous avons vu avec les chapitres précédents que les instructions (sous forme binaire) et certaines données (variables globales, variables locales statiques et chaînes de caractères) sont **allouées statiquement à la compilation** et sont donc stockées dans le fichier binaire au format ELF (dans les sections `.text`, `.data`, `.bss` et `.rodata`). D'autres données (notamment les variables locales non-statiques) sont quant à elles **allouées dynamiquement pendant l'exécution** du programme. Dans ce chapitre, nous nous intéresserons aux allocations de ressources mémoire réalisées 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 contigue virtuelle allouée en mémoire principale par le noyau du système avant l'exécution d'un programme. Par exemple le segment **code** contient le code binaire des instructions du processus en cours d'exécution. Il est construit en mémoire principale au lancement du programme, à partir de la section `.text` du fichier exécutable. Il contient donc les mêmes instructions et la taille du segment de code dépend du nombre d'instructions. Le segment de **pile** (ou **stack**) qu'on étudiera dans ce chapitre ne contient que des données accessibles en lecture et en écriture. Sa taille est toujours fixe (8 MB par défaut sous système Linux). D'autres segments existent, nous les aborderons dans de futurs chapitres.

Ces segments sont spatialement séparés les uns des autres. 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.



Le segment de *stack* est référencé par deux pointeurs. Le **Stack Pointer SP** pointe toujours vers le sommet de pile (il évolue donc fréquemment), et le **Base Pointer BP** (aussi appelé *Frame Pointer*) indique le début de la zone mémoire utilisée par la fonction courante (il n'évolue donc qu'à chaque appel ou retour de fonction). Pour les CPU 32-bit x86, ces pointeurs sont respectivement stockés dans les registres CPU `esp` et `ebp`. Pour les CPU 64-bit, ces pointeurs sont respectivement stockés dans les registres CPU `rsp` et `rbp`.

Le **Instruction Pointer IP** (parfois appelé *Program Counter PC*) est un pointeur dédié au segment de *code*, désignant la prochaine instruction à exécuter. Sa valeur est sauvée dans le registre `eip` (pour CPU 32-bit x86) ou `rip` (pour CPU 64-bit x86).

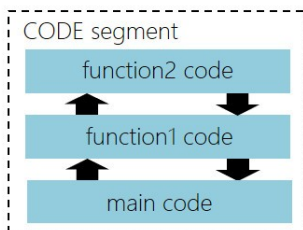
En langage C, comme dans beaucoup d'autres langages (C++, Java, D, etc), le point d'entrée d'une application est la fonction `main()`. De même, si nous réalisons des appels de fonctions imbriqués depuis le `main()` (cf. exemple ci-dessous) et si nous souhaitons revenir à la fonction principale de façon 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. À 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 8 MB potentiel d'espace mémoire de stockage.

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

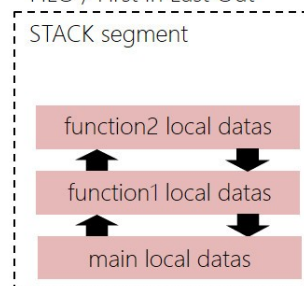
void function1 (void)
{
    /* local user code and datas */
    function 2();
}

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

Functions calls workflow
FILO / First In Last Out



Local datas workflow
FILO / First In Last Out



II. Fonction main

Pour suivre ce chapitre de TP, placez-vous dans le répertoire `disco/stack/`.

Compilez le fichier `main.c` vous arrêtant à la phase d'assemblage.

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

Affichez le code assembleur généré, puis expliquez chacune des instructions en vous aidant de la documentation Intel ([tp/doc/architectures-software-developer-manuals-vol2.pdf](https://www.intel.com/docs/tp/doc/architectures-software-developer-manuals-vol2.pdf)¹⁴).

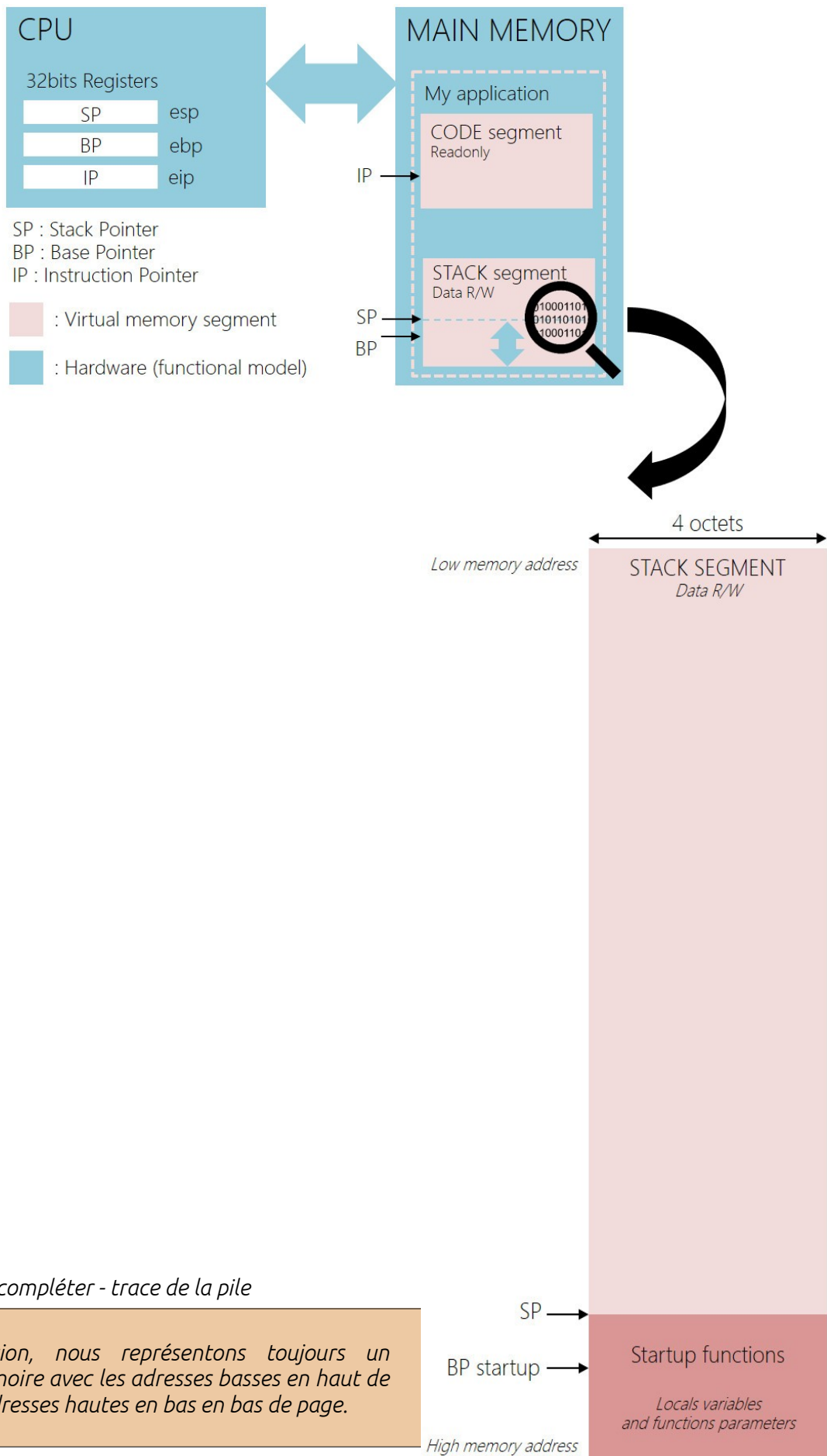
Complétez instruction par instruction le schéma de la pile sur la page suivante en précisant quel est son contenu suite à l'exécution du programme jusqu'à l'instruction `ret` en fin du `main`. Ne pas oublier qu'avant le début de notre programme, le code de la fonction de `startup` s'est exécuté.

Proposez 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`.

À partir de maintenant, il faudra avoir deux niveaux de lecture d'un code assembleur.

- 1) Comprendre ce que fait littéralement l'instruction (ex : `pushl %ebp` copie le contenu du registre `ebp` en sommet de pile).
- 2) Comprendre la finalité de l'instruction (ex : `pushl %ebp` sauvegarde la valeur actuelle du registre `ebp` sur la pile).

¹⁴ Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B & 2C). Intel



III. Variables locales initialisées

Compilez le fichier `local_variable_init.c` en vous arrêtant à la phase d'assemblage.

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

Analysez le fichier assembleur généré et complétez (au crayon) le schéma de la pile sur la page suivante en précisant son contenu suite à l'exécution du programme jusqu'à l'instruction `ret` en fin du `main`.

Précisez les tailles ou empreintes mémoire des variables locales `a`, `b`, `c` et `d`.

Précisez 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 ?

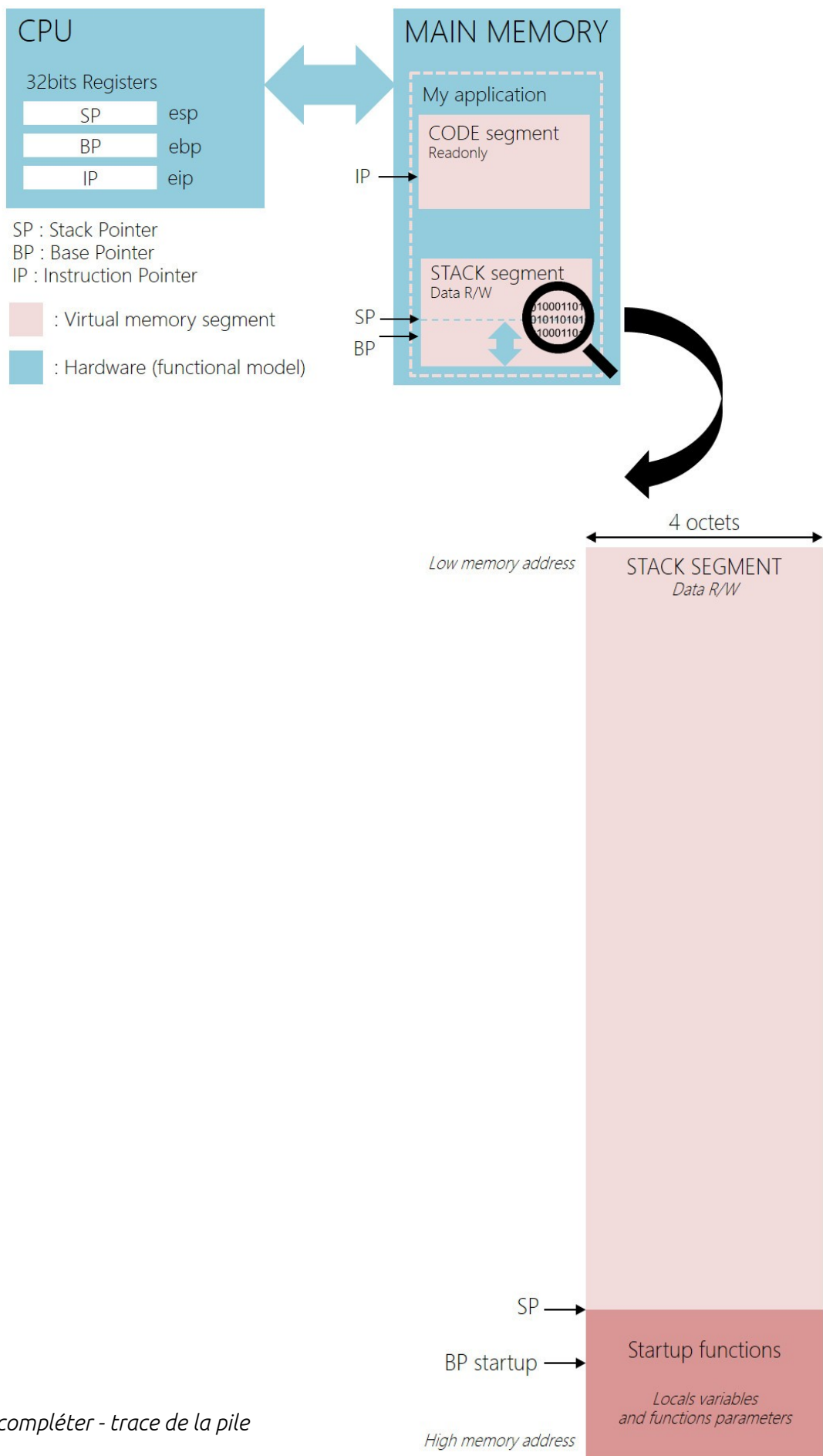


Schéma à compléter - trace de la pile

Dé-commentez le *cast* (transtypage) présent dans le programme, compiler et analyser le fichier assembleur de sortie. Analysez le résultat.

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

Qu'est-ce qu'une extension de signe en arithmétique entière signée Cà2 (Complément à 2) ? Aidez-vous de l'instruction **MOVSX** dans la documentation Intel.

Modifiez le fichier et qualifiez le type de la variable **a** de **const**. Compilez le programme puis interprétez le résultat. Que constatez-vous ?

Modifiez le fichier et qualifiez le type de la variable **c** de **const**. Compilez le programme puis interprétez le résultat. Que constatez-vous ?

Quel est le rôle du qualificateur de type **const** et donc son usage ?

IV. Variables locales non-initialisées

Ouvrez et compilez le fichier `local_variable_uninit.c` en vous arrêtant à la phase d'assemblage.

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

Analysez le fichier assembleur généré, que constatez-vous ?

Ajoutez le qualificateur de type `volatile` devant chaque déclaration, compilez le programme puis interprétez le résultat. Quel nouvelle instruction est apparue ? Quels sont les deux niveaux de lectures que vous accordez à cette instruction ?

Quel est le rôle de ce qualificateur de type `volatile` et donc son usage ?

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).

V. Appel et paramètres de fonction

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

Ouvrez et compilez le fichier `function_parameters.c` en s'arrêtant à la phase d'assemblage.

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

Analysez le fichier assembleur et complétez instruction par instruction le schéma de la pile sur la page suivante. Précisez son contenu exhaustif suite à l'exécution du programme jusqu'à l'instruction `ret` en fin du `main`.

L'instruction `call` réalise un appel de fonction : elle stocke l'adresse de retour sur la pile et effectue un saut vers le code de la fonction appelée.

L'adresse de retour est celle de l'instruction située spatialement après le `call` en mémoire programme. Or pendant l'exécution du programme le *Instruction Pointer* IP (dont la valeur est stockée dans le registre `eip/rip`) contient l'adresse de la future instruction à exécuter. C'est à dire qu'au moment de l'exécution du `call`, le registre `eip/rip` contient l'adresse de retour de fonction. Empiler cette valeur en sommet de pile permet de pouvoir revenir à cette instruction en fin de fonction appelée.

Pour effectuer un saut vers le code de la fonction appelée, l'adresse visée sera écrite dans le *Instruction Pointer*.

L'instruction `ret` présente dans la fonction appelée dépile l'adresse de retour précédemment sauvée et la restaure dans le registre d'instruction `eip/rip` du CPU. 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

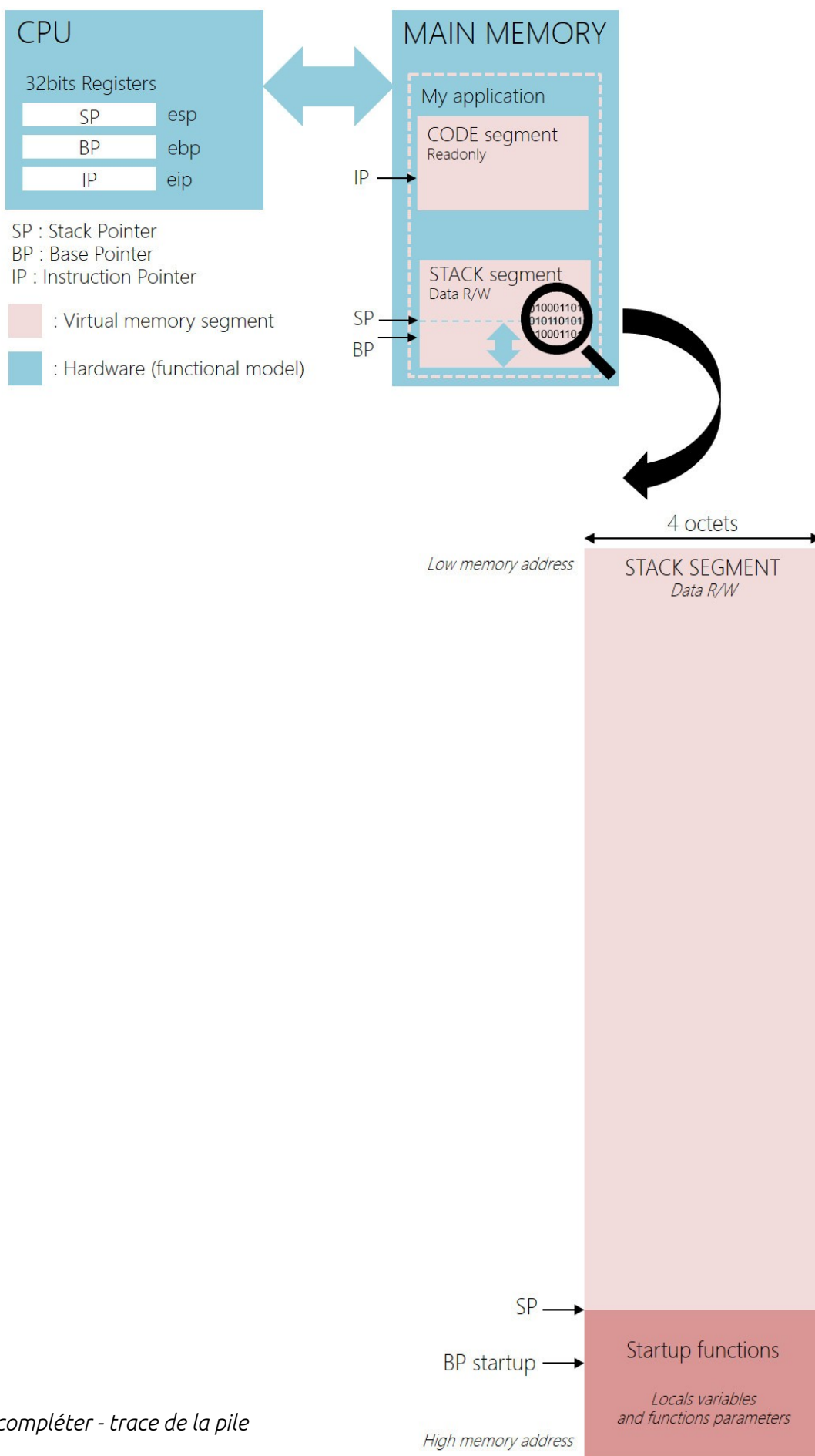


Schéma à compléter - trace de la pile

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 ? Notez qu'il s'agit d'une stratégie proposée par GCC, cela ne dépend pas directement du langage ou de l'architecture CPU.

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 ?

Observez 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 ?

VI. 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.

Ouvrez puis compilez le fichier `function_inlining.c` en s'arrêtant à la phase d'assemblage.

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

Analysez le fichier assembleur généré. Complétez le schéma de la pile sur la page suivante en précisant son contenu exhaustif suite à l'exécution du programme jusqu'à l'instruction `ret` en fin du `main`.

Dé-commentez le prototype de la fonction `swap` qui utilise la classe de stockage `register` pour la déclaration des paramètres de fonction, et commentez le prototype générique. Faites de même au niveau de la définition de fonction, et qualifiez également la variable `tmp` dans la fonction `swap` de `register` (à laisser jusqu'à la fin de l'exercice). Compilez et analysez 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 qualificatif 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).

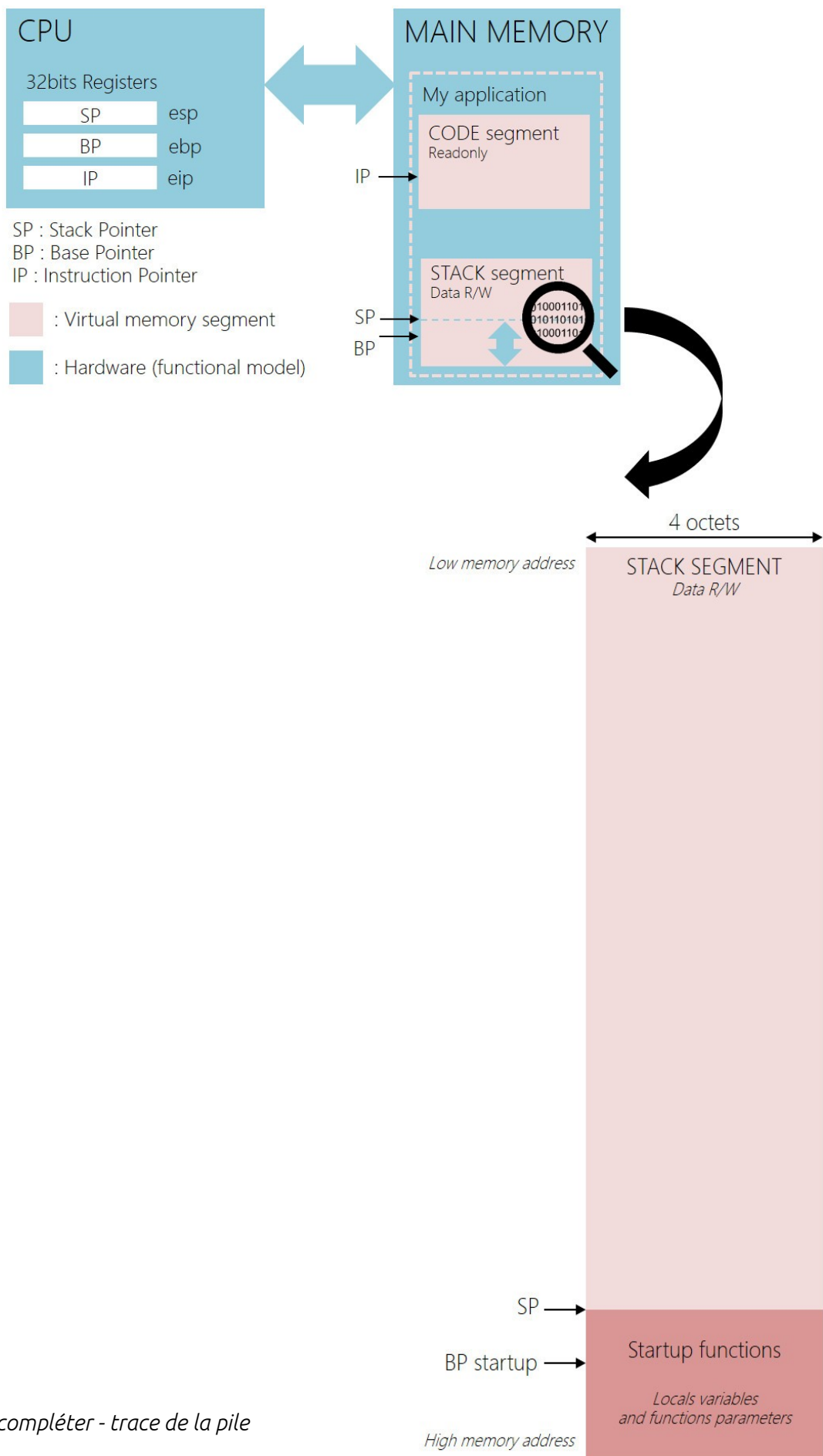


Schéma à compléter - trace de la pile

Dé-commentez le prototype de la fonction `swap` qualifiée de `inline` (seulement à la déclaration) et commentez les prototypes génériques.

```
//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));
```

Compilez et analysez 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` ?

Dé-commentez le prototype de la fonction `swap` qualifiée de `inline` au niveau de la définition de la fonction `swap` et commentez les prototypes génériques. Analysez le code assembleur généré.

```
//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.

Laissez votre code tel quel (fonction `swap()` déclarée et définie avec le qualificateur `inline`) et ajoutez l'option d'optimisation de niveau 1 (`-O1`) de GCC. Compilez et analysez la sortie. Observez 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
```

Essayez le niveau maximal d'optimisation `-O3` et analysez le programme de sortie.

VII. Limites de la pile

Nous allons maintenant tester les limites de notre pile applicative. Placez-vous dans le répertoire `disco/except/`. Compilez le fichier `stack_overflow.c` puis exécutez-le.

```
gcc stack_overflow.c -o stack_overflow
./stack_overflow
```

Le segment de pile a débordé. Expliquez l'erreur indiquée.

Quelle 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-commentez la section de code présente dans le programme. Compilez à nouveau le fichier `stack_overflow.c` puis exécutez-le. Quelle est la nouvelle taille de la pile associée à notre programme ? Analysez le code dé-commenté.

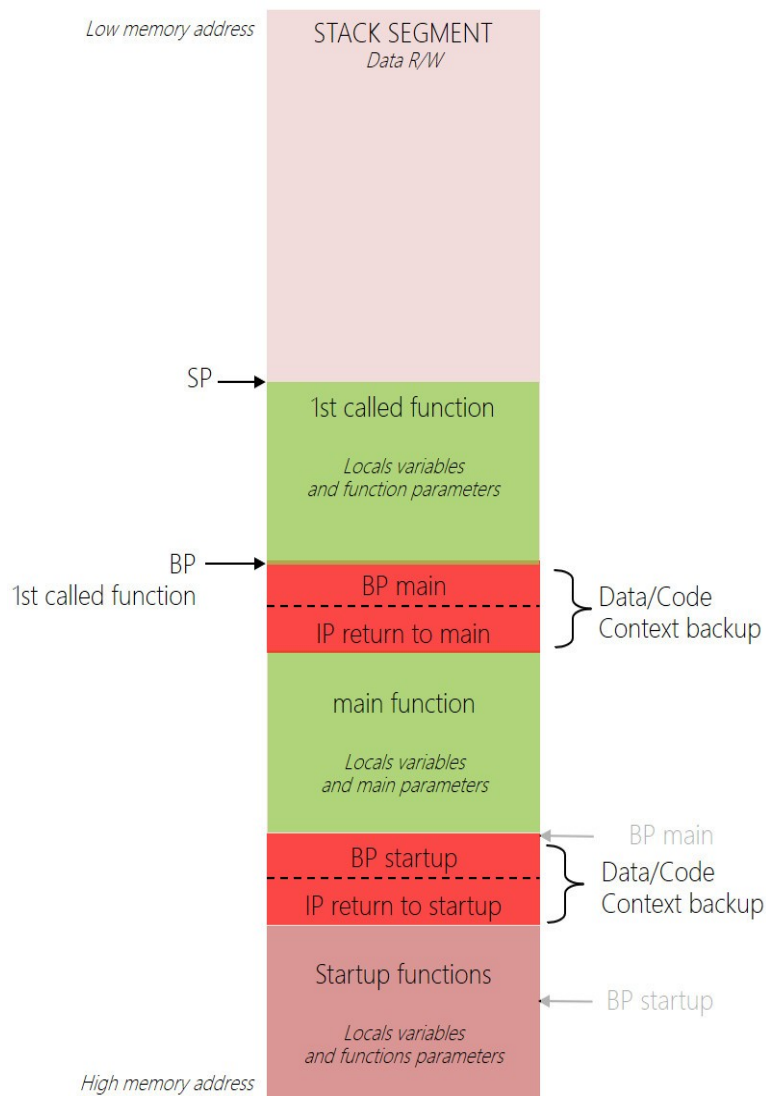
La fonction `setrlimit` (*set application resources limits*) réalise un appel système au noyau Linux en lui demandant d'allouer à notre programme un segment logique virtuel de pile plus large que celui alloué par défaut. Rappelons que Linux est le gestionnaire et le garant du bon fonctionnement des ressources 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.

VIII. 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 alloués 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** 32-bit/64-bit x86/x64) permet donc indirectement d'adresser un nombre quelconque de variables locales.



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 ou SSD).

