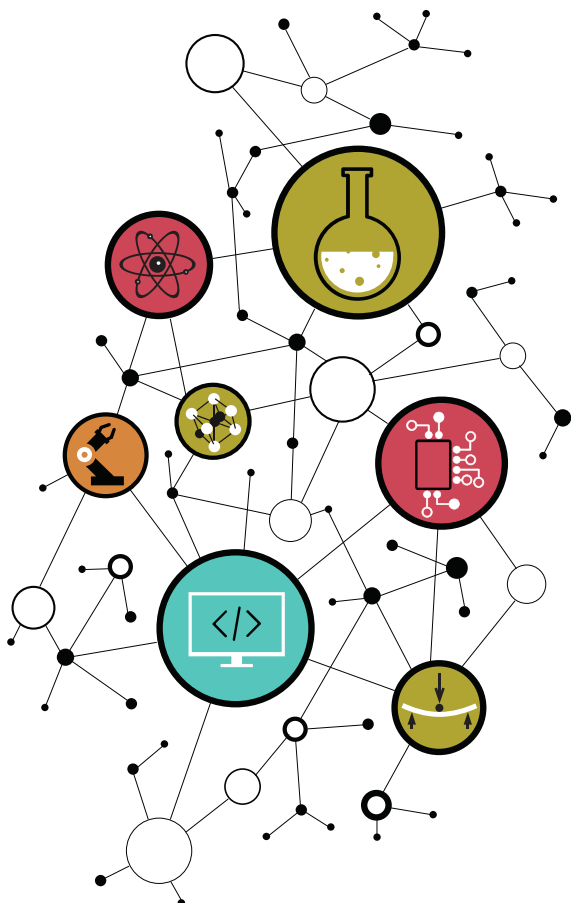


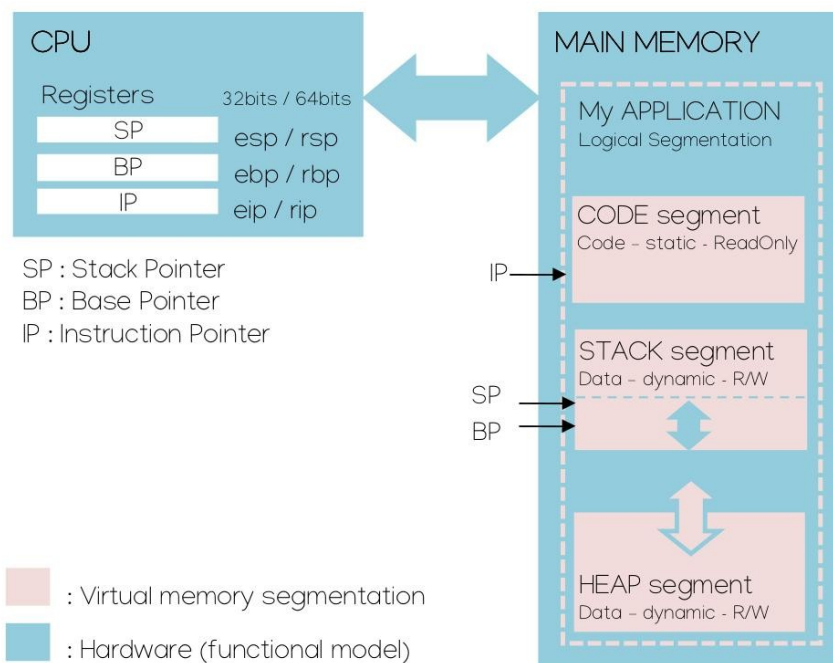
PARTIE 6

ALLOCATION DYNAMIQUE ET SEGMENT DE TAS



I. Segments de code, de pile et de tas

Le **tas** (ou **heap**) est l'un des deux segments mémoire utilisé pour les allocations dynamiques durant l'exécution d'un programme. Nous avons découvert le premier dans les exercices du chapitre précédent à travers l'analyse de la gestion des variables locales et des allocations dynamiques sur la pile aussi nommées allocations automatiques. Ces allocations sont nommées ainsi car l'allocation mémoire est réalisée automatiquement à l'exécution durant l'entrée dans le code d'une fonction.



Contrairement à la **pile** ou **stack** qui possède une taille fixe, le tas possède une taille extensible pouvant aller jusqu'aux limites physiques des ressources de stockage en mémoire principale de la machine (mémoire principale DDR SDRAM + potentiel SWAP système sur média de stockage de masse HDD/SSD). Les allocations dynamiques sur le tas sont exécutées explicitement à la demande du programme sous forme de requêtes envoyées au noyau du système (Linux).

Ces requêtes d'allocations mémoire se font par appels de la fonction **malloc** (*memory allocation*) ou de ses variantes (**calloc**, **realloc** et **aligned_alloc**, ou encore **new** en C++). Tant que l'application est active en mémoire principale, l'espace demandé restera alloué. Une fois la ressource mémoire utilisée, il est de la responsabilité du développeur de libérer les allocations précédentes avec appel de la fonction **free** (ou **delete** en C++). Ceci permet de libérer de l'espace pour les autres applications actives sur la machine. Le phénomène de zones mémoires précédemment allouées non libérées se nomme fuites mémoire et reste des erreurs assez courantes dans le monde du logiciel.

Il est important de noter que la fonction **free** est probablement l'une des fonctions les plus risquées à l'usage du langage C, notamment pour une application dans le domaine des systèmes embarqués sur processeur sans MMU (MCU, DSP, ...). En effet, allouer successivement des ressources mémoire dynamiquement sur le tas puis libérer certaines de ces zones amène une fragmentation du tas (zones mortes non utilisées). Il faut éviter d'utiliser la fonction **free** sur un processeur ne possédant pas de MMU (*Memory Management Unit*) et ne pouvant garantir au noyau du système d'exploitation une virtualisation de la gestion mémoire. Sans quoi, la fragmentation précédemment citée sera inévitable et conduira vers un comportement erratique puis vers un bug certain de l'application.

II. Process en exécution

Placez-vous dans le répertoire `disco/heap/`. Compilez `heap.c` et exécutez le programme.

```
gcc -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector -fcf-protection=none
-Wall -static heap.c -o heap
```

```
./heap
```

```
[CTRL]+[c] pour arrêter l'exécution du processus
```

Pendant que le processus est en cours d'exécution, ouvrez une nouvelle console pour récupérer le **PID (*Process Identifier*)** du-dit processus. Observez ensuite son *mapping* mémoire.

```
ps -a
```

```
cat /proc/<pid_of_heap_program>/maps
```

Quels éléments sont affichés par les trois `printf()` du programme `heap.c` ?

Dans quel segment mémoire est théoriquement situé chacun de ces éléments ?

Observez le *mapping* mémoire du processus et ses segments. Cela confirme-t-il les réponses précédentes ?

Quelle est la taille du segment de pile ? Quelle est la taille du segment de tas ? Quelle est la taille du segment de code (seul segment statique dont les propriétés permettent l'exécution, propriétés `r-x-`) ? Chaque segment en mémoire principale possède une taille multiple de 4 KB, la taille d'une page gérée par l'unité de pagination MMU.

Compiler le fichier `heap.c` en s'arrêtant à l'assemblage. Analysez le programme assembleur généré. À ce stade de l'enseignement, vous devez pouvoir être capable d'analyser des script assembleur de ce type.

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

III. Fuites mémoire

Les mauvaises gestions de la mémoire représentent un très (très) grand nombre de sources d'erreurs de logiciels en cours d'exécution. Pour limiter ces erreurs il est nécessaire que le développeur ne cause aucune **fuite mémoire** en s'assurant de la libération des données qui ont été allouées mais qui ne sont plus nécessaires. Un outil indispensable pour tout développeur est **valgrind**, installable sur n'importe quelle distribution (déjà installé sur les machines de l'école).

Compilez le fichier **memory_leak.c**, exécutez-le une première fois et constatez qu'il n'y a pas de message d'erreur lors de l'exécution. Exécutez l'exécutable cette fois-ci par l'intermédiaire de **valgrind** et observez le résultat.

```
gcc memory_leak.c -o memory_leak
./memory_leak

valgrind --leak-check=full ./memory_leak
```

Qu'indique la sortie de **valgrind** ? Quelle est la cause des fuites mémoire ?

Modifiez le fichier **memory_leak.c**, recompilez et relancez l'exécution jusqu'à résolution complète des fuites mémoires.

Compilez le fichier **memory_leak_in_array.c**, exécutez-le une première fois et constatez qu'il n'y a pas de message d'erreur lors de l'exécution. Puis ré-exécutez au travers de **valgrind**.

```
gcc memory_leak_in_array.c -o memory_leak_in_array
./memory_leak_in_array
valgrind --leak-check=full ./memory_leak_in_array
```

Qu'indique la sortie de **valgrind** ? Quelle est la cause des fuites mémoire ?

Modifiez le fichier **memory_leak_in_array.c**, recompilez et relancez l'exécution jusqu'à résolution complète des fuites mémoires.

Même si l'exécution du processus ne provoque pas directement d'erreur, les fuites mémoires répétées (en quantité par un processus, et ce sur plusieurs processus) mèneront inévitablement à un bug. Un exemple très connu est le bug de la sonde spatiale Mars Pathfinder (1997), dont les fuites mémoires bloquaient les tâches critiques provoquant un redémarrage en boucle du système. Un bug à plusieurs millions de Dollars, corrigé à distance.

Valgrind est un outil puissant d'analyse, permettant de cibler les données perdues (allouées mais non libérées à la fin de l'exécution du processus). Il indique notamment les fonctions en cause ainsi que la quantité de données perdues (nombres et tailles des blocs).

IV. Limites du tas

Pour étudier les limites du segment de tas, placez-vous dans le répertoire `disco/except/`.

Ouvrez le moniteur système (ou *system monitor*) et observez l'onglet « *Resources* » pendant l'exécution du programme.

Compilez le fichier `heap_overflow.c` jusqu'à l'édition des liens incluse et l'exécutez-le.

```
gcc heap_overflow.c -o heap_overflow
./heap_overflow
```

Analysez le programme et observez les limites du tas. Comparez aux informations proposées par le système.

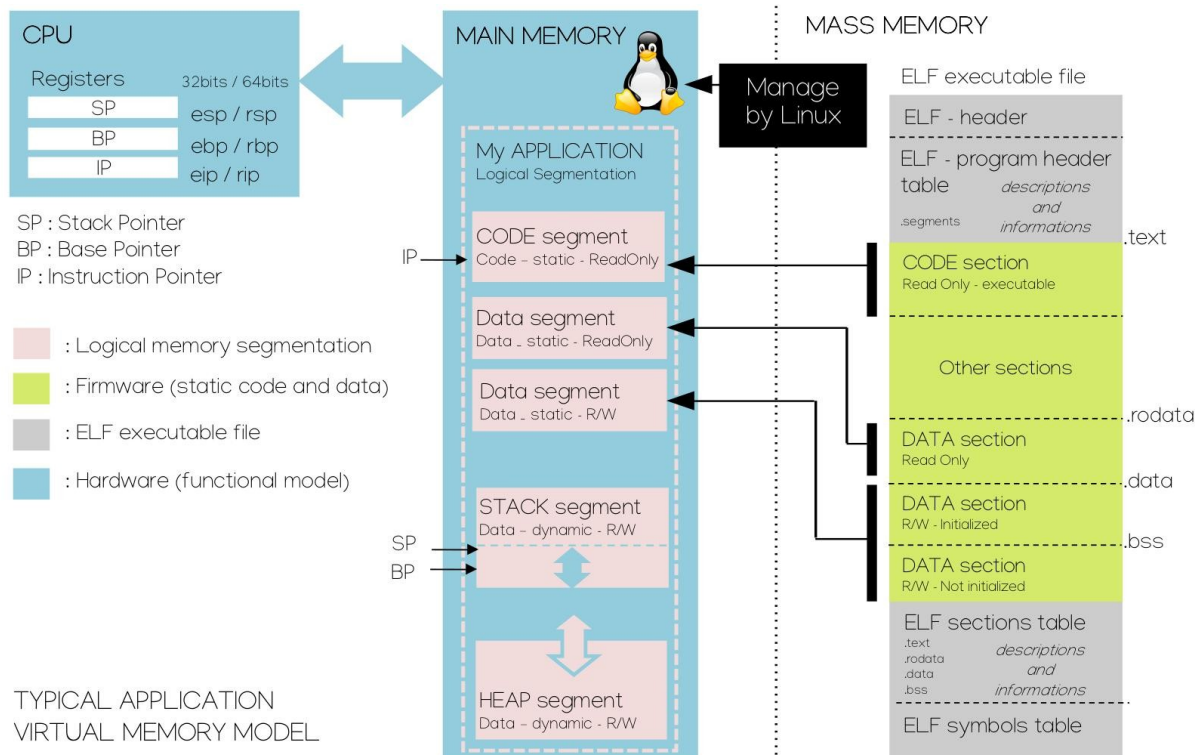
```
free
cat /proc/meminfo
```

La capacité du segment de tas est limité par la **mémoire principale** du système. Cette mémoire principale inclut évidemment la **RAM**, mais également le **swap**. Ce dernier est une partition d'échange présente sur le support de stockage de masse (HDD ou SSD), qui fait office d'extension de la RAM. Ainsi quand la RAM est pleine, le *swap* entre en jeu pour mettre à disposition une portion supplémentaire d'espace. L'avantage est donc d'étendre la quantité maximale de données allouables, mais avec en contrepartie de piètres performances en terme de débit des données manipulées.

Le segment de tas peut donc s'étendre au maximum sur toute la RAM et tout le *swap*, en prenant en compte ce qui est déjà alloué sur la mémoire principale (système d'exploitation, autres processus, autres segments du processus, ...).

V. Synthèse globale sur les stratégies d'allocations

Le schéma ci-dessous rappelle et synthétise une grande partie des nombreux points abordés dans cet enseignement et cette trame de TP. Maintenant vous le savez, le superviseur de la machine à la gestion des ressources matérielles est le noyau du système d'exploitation (Linux, GNU Hurd, XNU, etc). Il est notamment le gestionnaire de la mémoire.



Compilation et édition des liens

Après développement d'un programme logiciel (*software*), la chaîne de compilation (GCC, Clang, ICC, ...) est chargée de traduire le programme d'un langage source (C, C++, ...) en langage machine binaire (x86, x64, ARM, MIPS, ...). Le format de fichier standard d'encapsulation de firmware sur système Unix-like est le format **ELF**. Le *firmware* est découpé en **sections**, des zones logiques séparant l'information (code et données) en partie de même natures et propriétés (code, donnée, lecture seule, lecture/écriture, exécutable, ...).

Allocation des segments et chargement en mémoire par le noyau

Si l'utilisateur demande l'exécution d'un programme, le noyau analyse alors le contenu du fichier exécutable (application à exécuter) grâce aux différents en-têtes et tables du format ELF. Il alloue alors des segments mémoire logiques contigus (virtualisation) ne pouvant se chevaucher (*Virtual Memory Area* ou VMA sous Linux) de tailles et propriétés adaptées en mémoire principale. Une fois les allocations réalisées, par copie il charge du média de stockage de masse (HDD, SSD, MMC) les sections statiques du firmware vers les segments associés en mémoire principale (DDRx SDRAM). Une fois cette opération faite, il donne la main à l'application en commençant par exécuter le code des fonctions de *startup*. L'application pourra ensuite s'exécuter dans le respect des limites des segments mémoire alloués par le système. Sinon un *Segmentation fault* (*core dumped*) sera retourné par le système et l'application sera retirée (*kill*) de la mémoire principale.

Allocations mémoire et segments associés

En résumé, il existe donc 3 types d'allocation de ressource mémoire sur les langages compilés : les allocations statiques, les allocations dynamiques sur la pile (ou automatiques) et les allocations dynamiques sur tas. Chaque type d'allocation possède un segment mémoire dédié.

