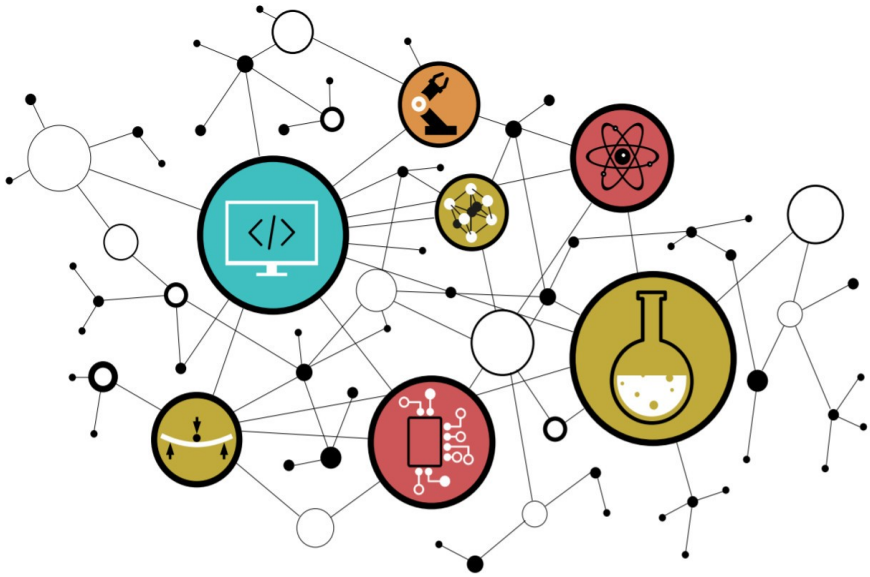
creative commons

TRAVAUX PRATIQUES

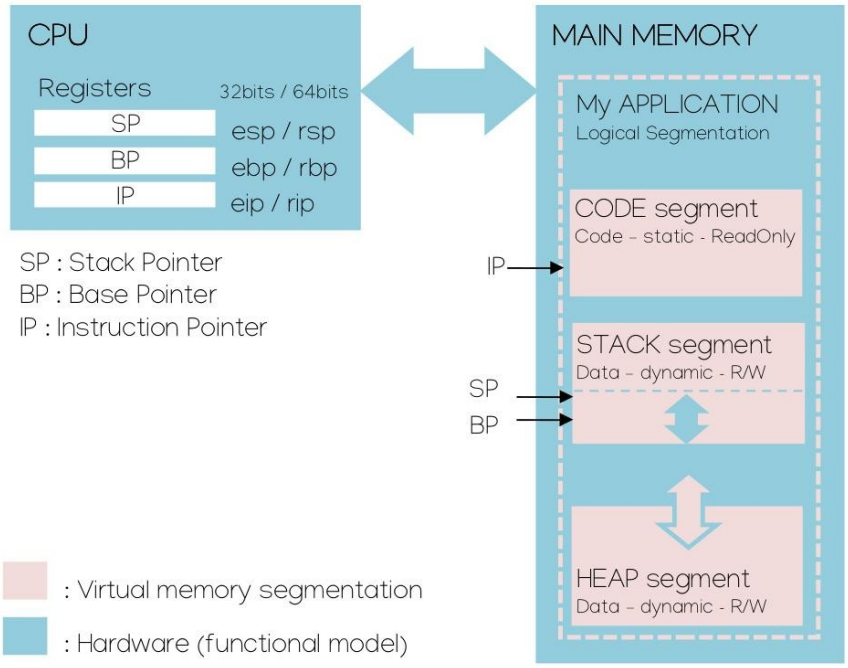
ALLOCATIONS DYNAMIQUES ET SEGMENT DE TAS



Hugo Descoubes – neku – GNU\Linux 5.8 - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2021

SOMMAIRE

6. ALLOCATIONS DYNAMIQUES ET SEGMENT 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 précédents 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 (cf. schéma ci-dessus) possède une taille extensible pouvant aller jusqu'aux limites physiques des ressources de stockage en mémoire principale de la machine (mémoire principale DDR SDRAM + potentiellement SWAP système sur média de stockage de masse HDD/SSD/etc). 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*). Tant que l'application est active en mémoire principale, l'espace demandé restera alloué. Une fois la ressource mémoire utilisée, bien penser à libérer les allocations précédentes avec appel de la fonction *free* (à la responsabilité du développeur). 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 MCU, DSP, MPPA, etc (sans MMU). 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). 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 le bug de l'application.

6.1. Gestion du tas

- Se placer dans le répertoire *disco/heap*. Compiler le fichier *heap.c* jusqu'à l'édition des liens incluse et exécuter le programme. Récupérer le PID (Process Identifier) du programme dans une nouvelle console et observer le mapping mémoire du programme. Constater que les 3 pointeurs affichés via le *printf* dans le programme *heap.c* pointent vers 3 segments mémoire distincts (CODE, STACK et HEAP). *Pour faciliter l'analyse, nous compilerons le programme en statique (-static) en incluant donc dans le firmware de sortie le code binaire de la fonction malloc et des fonctions systèmes dépendantes (évite les dépendances avec la bibliothèque partagée standard du C ou libc).*

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

./heap

[CTRL] + [c]
```

```
ps -a

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

- Qu'elle est la taille du segment de pile ?

- Qu'elle est la taille du segment de tas ?

- Qu'elle est la taille du segment de code (seul segment statique dont les propriétés permettent l'exécution, propriétés --x-) ? *Constater que chaque segment en mémoire principale possède une taille multiple de 4Ko, la taille d'une page gérée par l'unité de pagination MMU.*

- Compiler le fichier *heap.c* en s'arrêtant à l'assemblage. Analyser le programme assembleur généré. *A ce stade de l'enseignement, vous devez pouvoir être capable d'analyser des script assembleur de ce type. Si c'est le cas, ce que j'espère, bravo, du chemin a été fait !*

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

6.2. Limites du tas

- Se placer dans le répertoire *disco/except*. Compiler le fichier *heap_overflow.c* jusqu'à l'édition des liens incluse et l'exécuter. Analyser le programme et observer les limites du tas. Comparer aux informations proposées par le système.

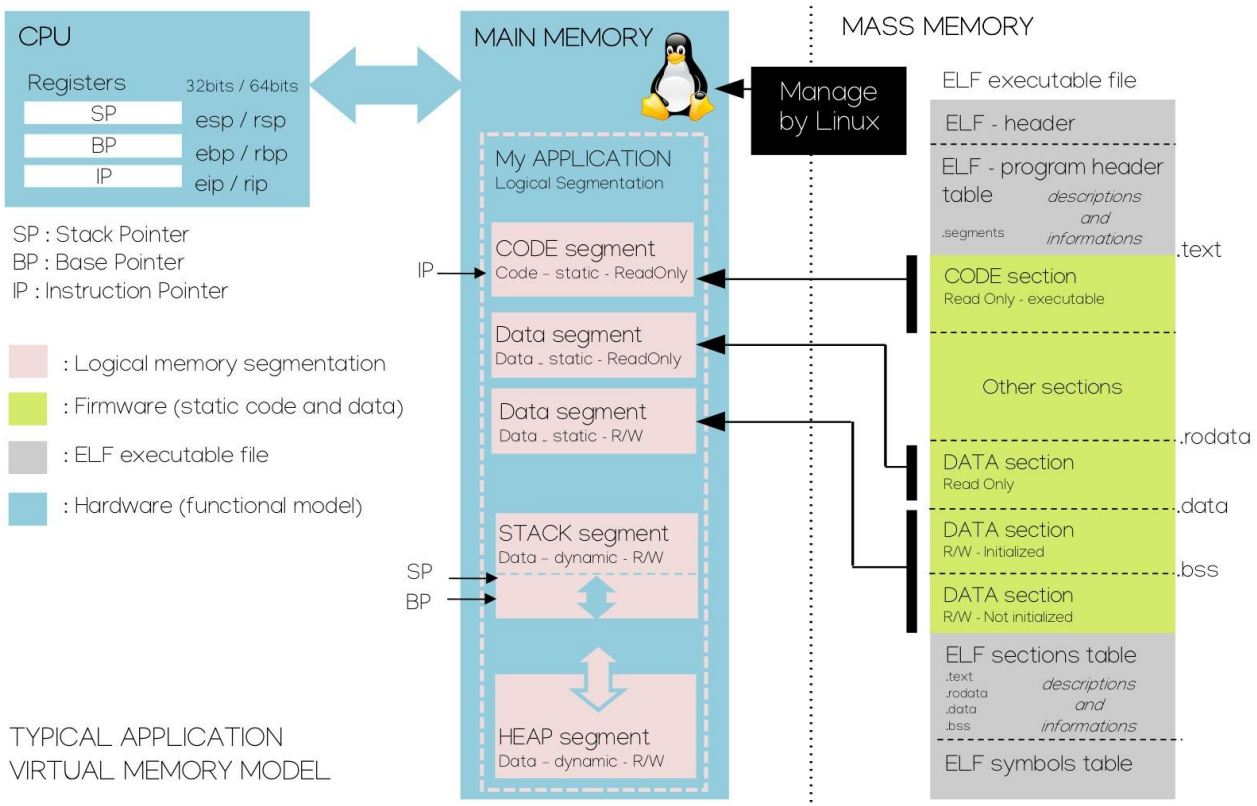
```
gcc heap_overflow.c -o heap_overflow

./heap_overflow

free

cat /proc/meminfo
```

6.3. Synthèse globale sur les stratégies d'allocations



Le schéma ci-dessus rappel 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, etc) est chargée de traduire le programme d'un langage source (C, C++, D, etc) en langage machine binaire (x86, x64, ARM, MIPS, etc). Le format de fichier standard d'encapsulation de *firmware* sur système Unix-like est le format ELF (COFF, PE sous windows, etc). Le *firmware* est découpé en sections, des zones logiques séparant l'information (code et donnée) en partie de même natures et propriétés (code, donnée, lecture seule, lecture/écriture, exécutable, etc).

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 différentes 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, etc) 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 *Segmentation fault (core dumped)* sera retourné par le système et l'application sera retiré (*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 allocation dynamiques sur tas. Chaque type d'allocation possède un segment mémoire dédié.


notes