Université de Bretagne Occidentale

Projet de fin d'étude

Les accès mémoire en pratique sous linux

Monitoring mémoire pour Linux embarqué

Par :
Jugurtha BELKALEM

Superviseur 1 :
Jalil boukhobza

Superviseur 2 :
Stéphane RUBINI

5 mars 2018

Table des matières

1 Introduction

L'évolution des semi-conducteur au cours des 50 dernières années a propulsé les mémoires à base de silicium SRAM, DRAM et plus récemment Flash vers les densités actuelles(plusieurs gigabits disponibles sur une seule puce) à un coût par bit en baisse constante.
Avec l'avènement des applications multimédias, la demande de stockage de données non volatiles a fortement augmenté. Cependant cela exige :

— Réduire la consommation énergétique.
— Besoin de plus de fiabilité(les recherches de Yoongu Kim ont démontré l'échec de l'intégration à grande échelle à maintenir la fiabilité[1]).

Au cours des deux dernières décennies, les chercheurs ont commencé à étudier le problème pour trouver des mémoire alternatifs.

Certaines de ces mémoires existent déjà[2] et sont très prometteuses(**Figure 1 par Tom Moxon** [1]).

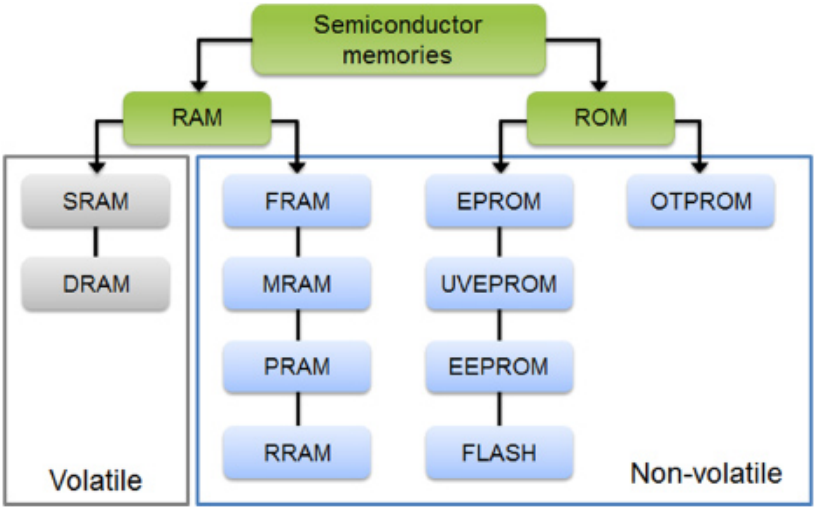

FIGURE 1 – Mémoire volatile et non-volatile [source : Tom Moxon]

le laboratoire Lab-STICC veut intégrer ses mémoires dans des systèmes embarqués, ainsi que de développer des heuristiques pour réduire la consommation énergétique qui se basera sur les ***accès mémoire***.

2 Problématique

La gestion de la mémoire est l'une des parties les plus complexes et importantes du noyau. Cette dernière se caractérise par un fort besoin de coopération entre le processeur et le noyau car les tâches à effectuer les obligent à collaborer de très près.
En plus de gérer sa propre mémoire, le noyau doit également gérer la mémoire des processus de l'espace utilisateur. Linux est un système d'exploitation de mémoire virtuelle, et donc la ressource de la mémoire est virtualisée parmi les processus sur le système.

Plusieurs questions peuvent faire objet d'analyse :

— L'organisation d'un processus dans la mémoire virtuelle ?
— Le mappage des pages vers des cadres de la mémoire ?
— Le nombre de référence à un cadre (un cadre peut être partagé) à un instant t ?
— Les cadres les plus utilisés ?

Pour comprendre ces mécanismes, nous devons explorer différents outils permettant de tracer les ***« accès mémoire »****, de les classer et de les caractériser selon différentes métriques(efficacité, intrusivité, facilité d'utilisation et portabilité).*

1. Une introduction aux mémoire est donnée par Tom Moxon : `https://github.com/PatternAgents/Electronics_One_Workshop/wiki/Memory-Circuits`

3 Gestion de la mémoire sous linux

Le noyau représente l'espace d'adressage d'un processus avec une structure de données appelée *descripteur mémoire*[3]. Ce dernier est représenté par « **struct** mm_struct »et défini dans *<linux/mm_types.h>*.

```
struct mm_struct {
  struct vm_area_struct * mmap;   /* liste des VMAs(zone mémoire virtuelle) */
  struct rb_root mm_rb; /* liste des VMAs - arbre rouge et noire */
  struct vm_area_struct * mmap_cache; /* dernière VMA trouvée avec find_vma */

  pgd_t * pgd; /* pointeur vers le 'page global directory' */
  atomic_t mm_count;       /* nombre de référence à "struct mm_struct" */
  int map_count;          /* nombre de VMAs */

  struct list_head mmlist;     /* Liste des mm_struct */
}
```

Les zones de mémoire sont souvent appelées zones de mémoire virtuelle (abréviations VMA), Dans le noyau Linux.

Chaque zone est représentée par une structure « vm_area_struct »qui est défini dans *<linux/mm_types.h>*.

```

struct vm_area_struct {
  struct mm_struct * vm_mm; /* L'espace d'adressage à qui le processus appartient. */
  unsigned long vm_start;   /* début de l'espace adressage vm_mm. */
  unsigned long vm_end;   /* fin de l'espace adressage vm_mm. */

  /* liste chainée des VMs par processus, triées par adresse */
  struct vm_area_struct *vm_next, *vm_prev;

  pgprot_t vm_page_prot;     /* permission d'accès VMA. */
  unsigned long vm_flags;    /* Flags */

  struct rb_node vm_rb;
  /* Opération possible sur la VM */
  const struct vm_operations_struct *vm_ops;
}
```

Le champ *vm_ops* dans la structure *vm_area_struct* pointe vers la table des opérations associée à une zone de mémoire donnée, que le noyau peut invoquer pour manipuler le VMA. Vm_area_struct agit comme un objet générique pour représenter tout type de zone de mémoire, et la table des opérations décrit les méthodes spécifiques qui peuvent fonctionner sur cette instance particulière de l'objet.

```
struct vm_operations_struct {
  /* Appélée lors de la création de la VM */
  void (*open)(struct vm_area_struct * area);
  /* Appélée lors de la libération de la VM */
  void (*close)(struct vm_area_struct * area);

  /* Gestion des défauts de page */
  int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);

  /* notification sur une page read-only qui va devenir Write
   * SIGBUS sera retourné dans le cas d'une erreur*/
  int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);

}
```

La fonction *fault* permet de faire le lien entre l'espace d'adressage virtuelle et la mémoire physique, en d'autre termes, elle s'occupe de remplir la table des pages.

La **Figure 2** résume le mécanisme de gestion mémoire sous linux.

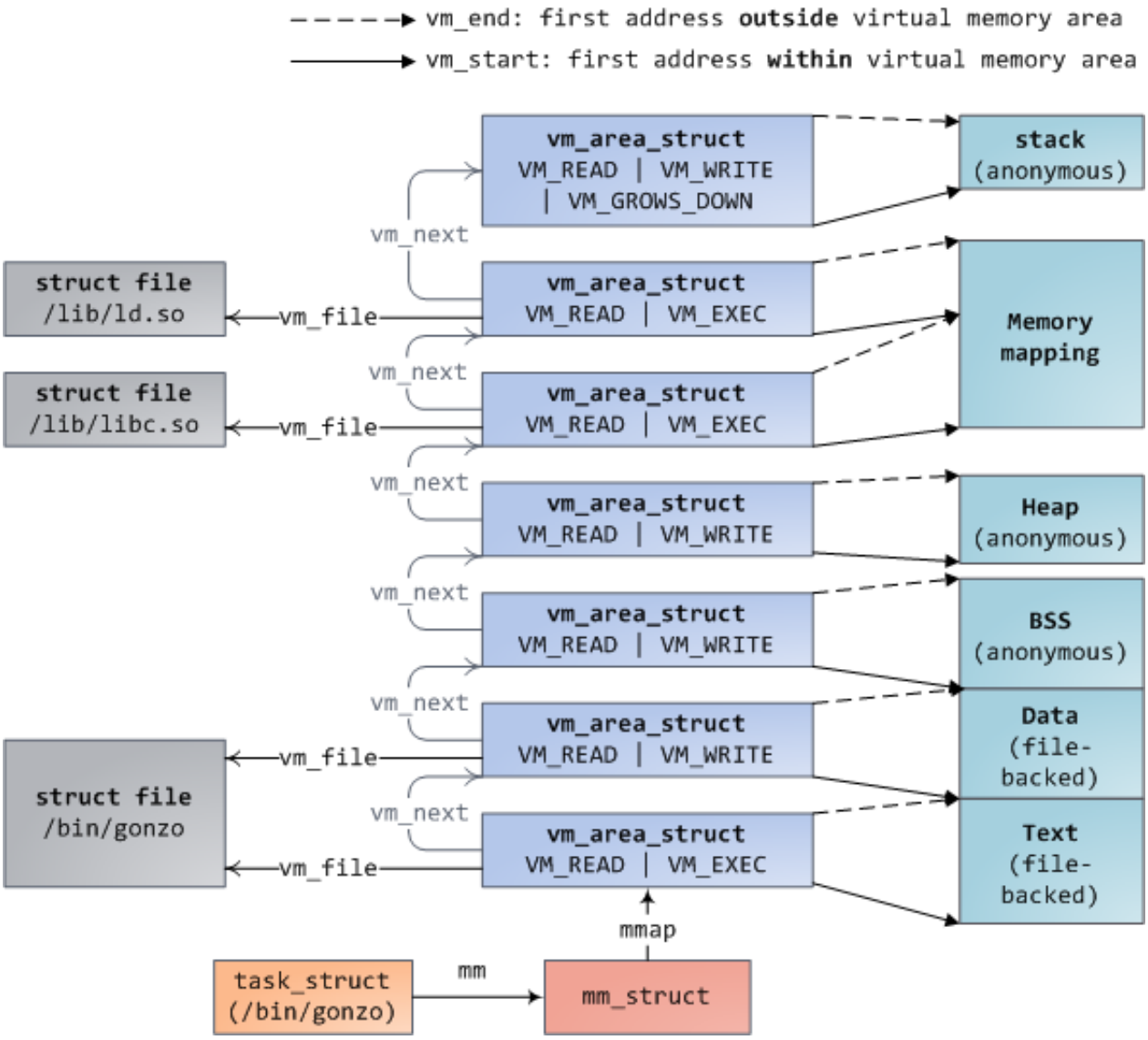

FIGURE 2 – Gestion de mémoire sous linux - source : manybutfinite.com par Gustavo Duarte

4 Comprendre les accès mémoire

4.1 Suivre les accès à la mémoire

4.1.1 Les défauts de page

Lorsqu'un processus tente de visiter une adresse virtuelle, le système d'exploitation vérifie (voir La **Figure 3**) :

1. TLB (Translation Lookaside Buffer) : car la page peut être mise en cache.
2. Table de pages : si la page n'existe pas dans le TLB, l'OS vérifie dans la Table des pages et s'assure que le bit *valide* est activé.
3. Défaut de pages : Un défaut de pages est généré(**Figure 3**) si la page ne peut être retrouvé dans le TLB ou et la table des pages.

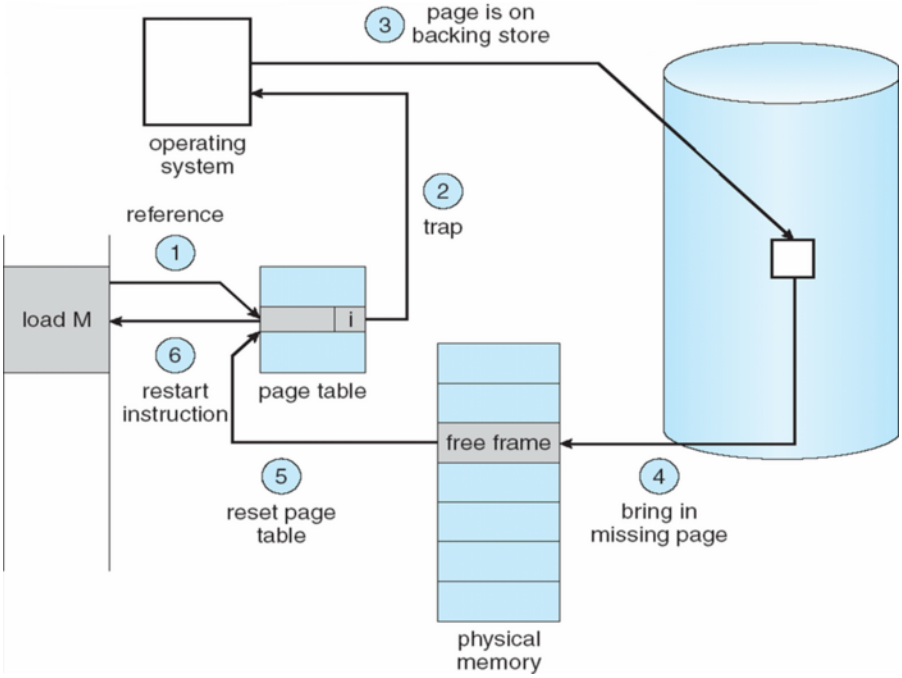

FIGURE 3 – Gestion des défauts de pages - source : Operating system concepts, 9th edition

Les Performances de la pagination à la demande sont coûteuses[4] en terme de temps, le *temps d'accès effectif* à la mémoire est calculé avec :

$$\text{temps d'accès effectif} = (1 - p) * ma + p * pft$$

— **p** $(0 \leqslant p \leqslant 1)$ **:** taux de défaut de page.
— **ma :** temps d'accès à la mémoire.
— **pft :** temps total de pagination : temps nécessaire pour charger la page du disque à la mémoire.

4.1.2 Types de défaut de pages

Les défauts de pages(voir l'annexe 2 pour comptabiliser les défauts de pages) peuvent être de trois types :
— **Mineur :** la page est chargée en mémoire au moment où l'erreur est générée, mais n'est pas marquée dans l'unité de gestion de la mémoire comme étant chargée en mémoire.
— **Majeur :** la page n'est pas chargée en mémoire au moment de la faute. Le système doit la charger du disque.
— **Invalide :** référence à une adresse qui ne fait pas partie de l'espace d'adressage virtuel, ce qui signifie qu'il ne peut y avoir de page en mémoire correspondant à celle-ci.

4.1.3 Le fichier maps

Le système de fichiers /proc contient un système de fichiers illusoire. Il n'existe pas sur un disque (le noyau le crée en mémoire). Il est utilisé pour fournir des informations sur le système (à l'origine sur les processus, d'où le nom *proc* qui signifie *process*) [2]. L'un des fichiers de /proc est le fichier maps(/proc/pid/maps) qui nous permet de voir l'organisation de l'espace virtuelle d'un processus.

2. le fichiers et des répertoires les plus importants sont expliqués de proc sont détaillés sur ce lien : `https://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html`

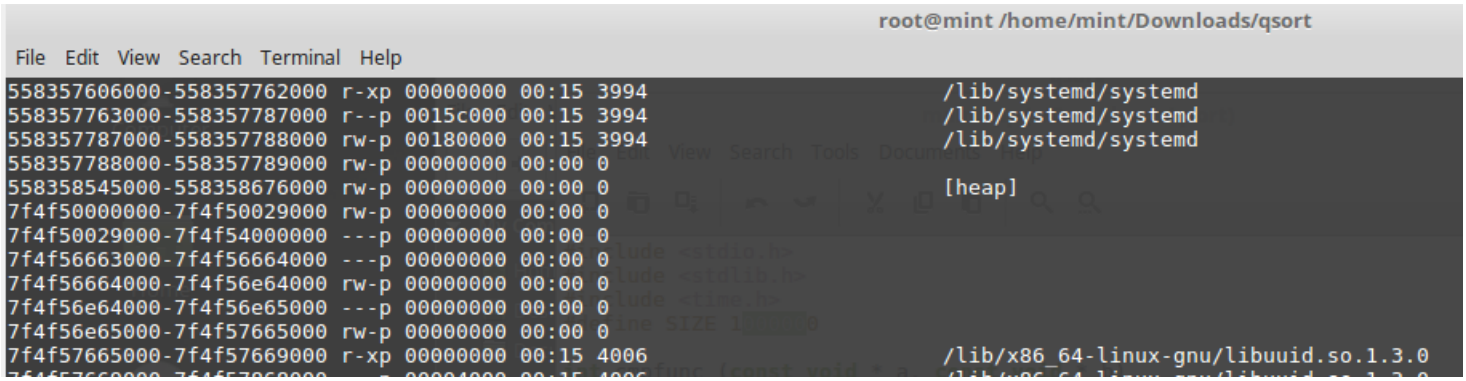

FIGURE 4 – Organisation de l'espace d'adressage virtuelle du processus init

Les trois premières colonnes montrent les adresses virtuelles de début et de fin et les autorisations pour chaque mappage. Les autorisations sont : r = read, w = write, x = execute, s = shared et p = private (copy on write).

Si le mappage est associé à un fichier, le nom de fichier apparaît dans la dernière colonne et les colonnes quatre, cinq et six contiennent le décalage depuis le début du fichier(Offset), le numéro de périphérique du bloc et l'inode du fichier.

4.1.4 L'interface pagemap

C'est un ensemble d'interfaces du noyau (depuis 2.6.25) qui permet aux programmes utilisateur d'examiner les tables de pages et les informations associées en lisant les fichiers dans le dossier */proc*[3].

- **/proc/pid/pagemap :** Ce fichier permet à un processus de l'espace utilisateur de déterminer à quelle cadre physique est mappée chaque page virtuelle.
- **/proc/kpagecount :** Ce fichier contient un décompte de 64 bits du nombre de fois que chaque page est mappée, indexée par PFN.
- **/proc/kpageflags :** Ce fichier contient un ensemble d'indicateurs(flags) de 64 bits de chaque page, indexé par PFN.

Problème : Ces fichiers sont binaires, donc il n'est pas possible de les lire avec un éditeur de texte.

Solution : Il est possible d'écrire notre propre parseur[4] ou prendre une « solution sur étagère ». Le script python « v2pfn »de *JEFF LI* (le code est disponible sur cette page : `https://blog.jeffli.me/blog/2014/11/08/pagemap-interface-of-linux-explained/`) permet de parser les trois fichiers /proc/pid/pagemap, /proc/kpagecount, /proc/kpageflags et d'afficher un rapport sur la page passé en paramètre.

- **Processus init :** Trouver l'association de l'adresse virtuelle 0x560426151000 avec le cadre mémoire correspondant.

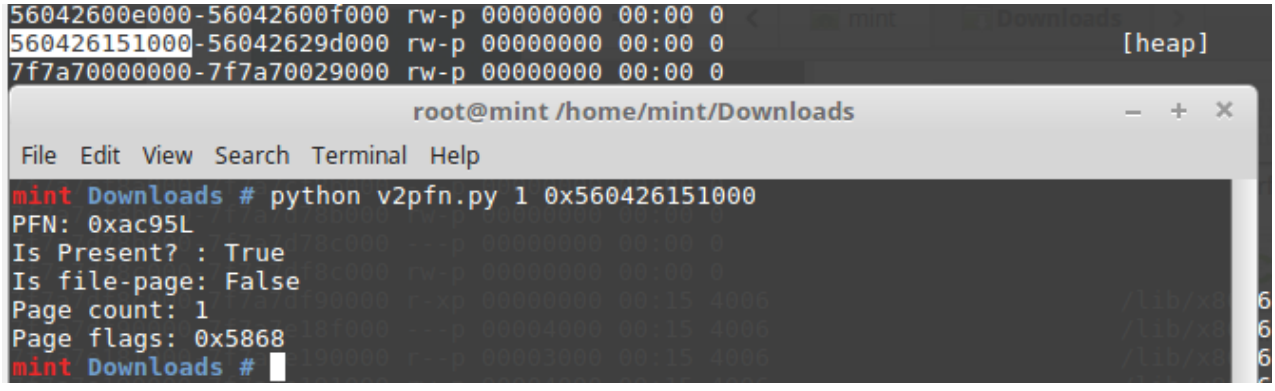

FIGURE 5 – Mapping d'une addresse virtuelle à un cadre mémoire - processus init

3. La documentation officielle est disponible sur ce lien : `https://www.kernel.org/doc/Documentation/vm/pagemap.txt`
4. Parseur du pagemap en C : `https://stackoverflow.com/questions/17021214/how-to-decode-proc-pid-pagemap-entries-in-linux`

La page est mappée vers le PFN(Page Frame Number) à 0xac95L.

— **firefox :** Trouver l'association de l'adresse virtuelle 0x55c4a9aab00 avec le cadre mémoire correspondant.

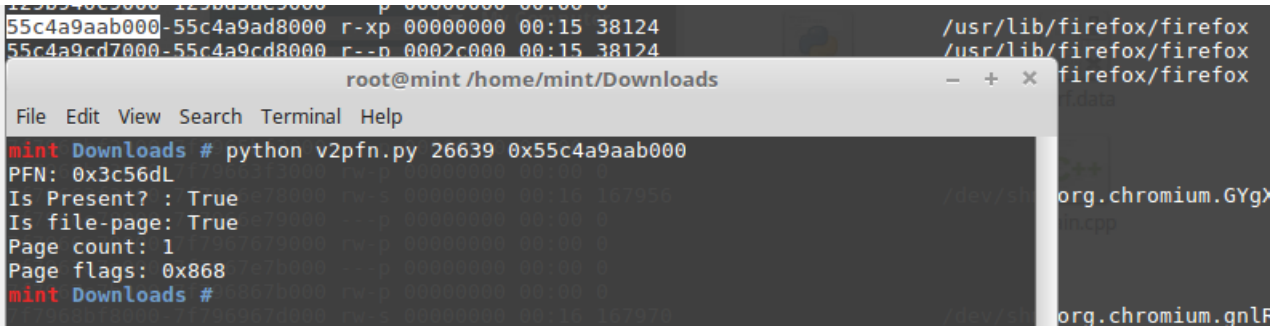

FIGURE 6 – Mapping d'une addresse virtuelle à un cadre mémoire - firefox

Le champs *Is file-page* indique que l'adresse est chargée à partir d'un fichier(/usr/lib/firefox/firefox).

4.2 Les outils sous linux

Pour comprendre les différents outils qui existent sous linux, Brendan D. Gregg nous propose une classification `www.brendangregg.com/Slides/Velocity2015_LinuxPerfTools.pdf` très détaillé durant sa conférence « Build resilient systems at scale »en 2015.

Type	Caractéristiques des outils
Observabilité	Regarder l'activité(voir **l'annexe 1**). Sûr, généralement, en fonction des frais généraux de ressources(vmstat, free, top, .., etc.)
Benchmarking	Test de chargement. Attention : les tests de production peuvent provoquer des problèmes dus à la contention(lmbench, perf bench, sysbench, memtester, ...,etc.)
Configuration	Changement. Danger : les modifications pourraient nuire à la performance, maintenant ou plus tard avec charge(ulimit, sysctl, ...,etc.)
Statique	Vérifiez la configuration. Devrait être sûr

Nous allons prendre comme exemple, les outils « free »et « vmstat »(**Figure7**).

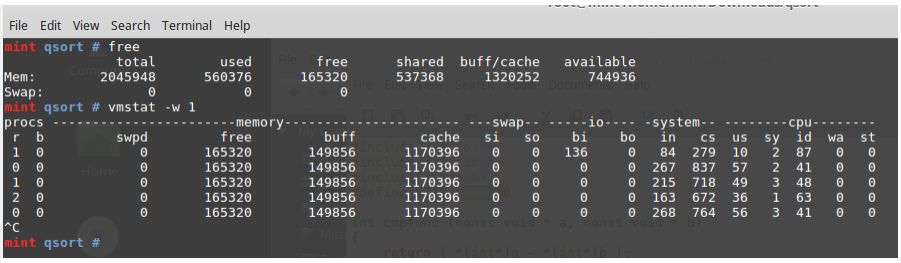

FIGURE 7 – Outils d'observabilité - free et vmstat

1. l'outil free : les champs renvoyés par la commande sont les suivants :
   — *total* représente la quantité de mémoire physique totale.
   — *used* représente la quantité de mémoire physique utilisée.
   — *free* représente la quantité de mémoire physique inutilisée.
   — *shared* est obsolete n'est plus utilisé.
   — *buffers et cached* est une mémoire qui peut être rendue disponible à tout moment.
2. l'outil vmstat : les champs renvoyés par la commande sont les suivants :
   — *free, buff et cache* sont similaire à free.
   — *swpd* montre la quantité de swap utilisée.

Cependant, il existe une autre catégorie d'outils dont le but est d'analyser le comportement du système en mettant en place de l'instrumentation au niveau du noyau[5](ustd, uprobes, tracepoints, kprobes).

4.2.1 Les traceurs d'événements

Brendan Gregg définit le traçage en 4 étapes :
- Comprendre les USDT, tracepoints, kprobes, uprobes.
- Apprendre à utiliser les traceurs principaux : ftrace, perf_events (perf, perf tools), eBPF.
- Tester d'autres traceurs(qui ne sont pas inclus dans linux) : SystemTap, LTTng, ktap, sysdig.
- Conscience de ce que le traçage peut accomplir.

dans cette partie, nous allons étudier le premier point :

- Un point de trace est quelque chose que nous compilant dans notre programme. Quand quelqu'un utilisant notre programme veut voir quand ce point de trace est touché et extraire des données, on peut « activer »le point de trace pour commencer à l'utiliser. les points de trace se divise en deux : *USDT*(Userland Statically Defined Tracepoints) et *tracepoints* pour attacher de l'instrumentation du coté utilisateur et noyau respectivement.
- Une sonde est une modification dynamique du code assembleur à l'exécution afin d'activer le traçage. C'est très puissant, car on peut activer une sonde sur littéralement toute instruction dans le programme qu'on trace. Les sondes de divise en deux aussi : *uprobes* et *kprobes* pour attacher dynamiquement de l'instrumentation du coté utilisateur et noyau respectivement.

Le tableau de la **figure 8** résume les différences entre les points d'instrumentation [5].

	Static	Dynamic	Kernel Tracing	Userland Tracing
Tracepoints	✓		✓	
Kprobes		✓	✓	
Uprobes		✓		✓
USDT	✓			✓

FIGURE 8 – Comparaison entre les tracepoints et les kprobes

Traceurs ou profileur ?
- **profileur :** fournit un inventaire sur les événements(exemple : il y'a eu : 154 allocations de pages virtuelles, 50 accès vers la mémoire).
- **traceur :** fournit la chronologie des événements.
  => firefox-11116 : adresse 0x7ff8e3dcf000, accès écriture PFN=0x7dfa0 à t= 000000
  => firefox-11116 : adresse 0x7ff8e3ddf000, accès lecture PFN=0x7cfa0 à t= 000010
  => firefox-11116 : adresse 0x7ff8e3dcf000, accès lecture PFN=0x7dfa0 à t= 000080
  => firefox-11116 : adresse 0x7ff8e3dcf000, accès écriture PFN=0x7dfa0 à t= 000101

Les traceurs que nous allons tester sont : *Ftrace*, *Perf* et *systemTap*.

5. Cette page résume les différences entre les points d'instrumentation : `http://nanxiao.me/en/brief-differences-between-tracepoints-kprobes-uprobes-usdt/`

4.2.2 Ftrace

Ftrace est le traceur officiel du noyau Linux (depuis la version 2.6.27) [6] créé par *Steven Rostedt*[7].
L'utilisation de Ftrace est fastidieuse[6][7], Steven Rostedt propose « trace-cmd »comme outil Front-end de Ftrace.
Remarque : Trace-cmd est intégré seulement avec certaines distributions((Vous pouvez le télécharger depuis le dépôt git : `https://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git`)).

Trace-cmd est simple à utiliser, cependant deux étapes sont requises pour avoir un rapport de la trace :

1. **Enregistrer la trace :** par exemple pour tracer les fonctions d'allocation mémoire sur l'application *xclock*. La trace sera sauvegardé par défaut dans un fichier *trace.dat*.

 # trace-cmd record -e 'kmem :mm_page_alloc' -e 'kmem :mm_page_free' xclock

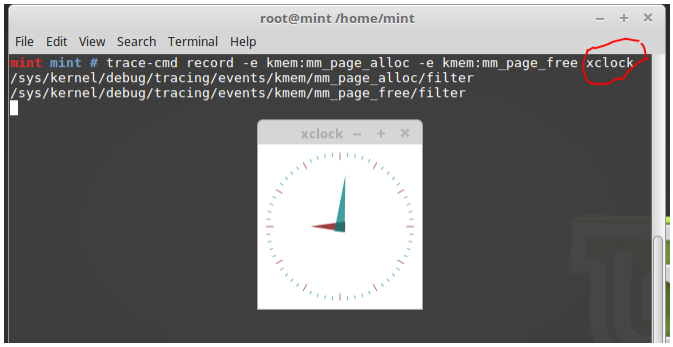

FIGURE 9 – trace-cmd trace les fonctions d'allocation mémoire de xclock

2. **Lire le rapport :** Par défaut, trace-cmd parse le fichier *trace.dat* (l'option -i sert a spécifier un fichier différent à lire).

 # trace-cmd report

```
xclock-12128 [000] 95574.822449: mm_page_alloc:        [FAILED TO PARSE] pfn=0x116a4 order=0 gfp_flags=21102794 migratetype=1
xclock-12128 [000] 95574.822458: mm_page_alloc:        [FAILED TO PARSE] pfn=0x116a5 order=0 gfp_flags=21102794 migratetype=1
xclock-12128 [000] 95574.822687: mm_page_alloc:        [FAILED TO PARSE] pfn=0x116a6 order=0 gfp_flags=21135562 migratetype=1
xclock-12128 [000] 95574.822745: mm_page_alloc:        [FAILED TO PARSE] pfn=0x3c20f order=0 gfp_flags=16777728 migratetype=0
xclock-12128 [000] 95574.822766: mm_page_free:         page=0x3c20f pfn=246287 order=0
xclock-12128 [000] 95574.822794: mm_page_alloc:        [FAILED TO PARSE] pfn=0x116a7 order=0 gfp_flags=21135562 migratetype=1
xclock-12128 [000] 95574.822849: mm_page_alloc:        [FAILED TO PARSE] pfn=0x116a8 order=0 gfp_flags=21135562 migratetype=1
xclock-12128 [000] 95574.822853: mm_page_alloc:        [FAILED TO PARSE] pfn=0x116a9 order=0 gfp_flags=21135562 migratetype=1
xclock-12128 [000] 95574.822859: mm_page_alloc:        [FAILED TO PARSE] pfn=0x116aa order=0 gfp_flags=21135562 migratetype=1
xclock-12128 [000] 95574.823179: mm_page_alloc:        [FAILED TO PARSE] pfn=0x3c20f order=0 gfp_flags=24150208 migratetype=0
```

FIGURE 10 – rapport de la trace sur xclock - trace-cmd

4.2.3 Perf

Les outils perf(qui a également été appelé Performance Counters pour Linux (PCL), événements perf Linux (LPE), ou perf_events) sont intégrés dans le noyau Linux depuis la version 2.6.31. Ce traceur utilise des compteurs matériels pour profiler l'application.
La façon la plus simple d'utiliser perf (est similaire a trace-cmd) :

1. **Sauvegarder la trace :** La trace sera sauvegardé par défaut dans un fichier *trace.dat*.

 # perf record -e 'kmem :mm_page_alloc' -e 'kmem :mm_page_free' ./smiuc

6. La documentation officiel de ftrace est sur le lien : `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/trace/ftrace.txt`
7. La présentation de Ftrace par Steven Rostedt est disponible sur ce lien : `https://www.youtube.com/watch?v=2ff-7UTg5rE`

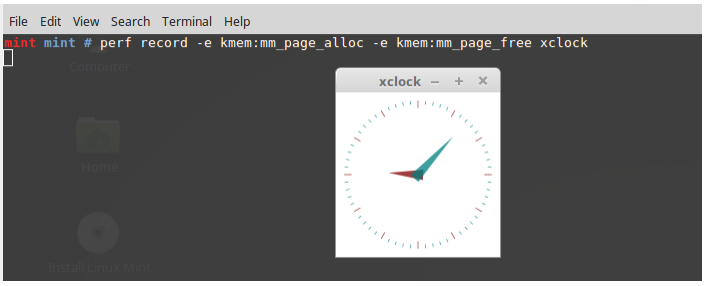

FIGURE 11 – Perf trace les fonctions d'allocation mémoire de xclock

2. **Lire le rapport :** Par défaut, Perf parse le fichier trace.dat (l'option -i sert a spécifier un fichier différent à lire). Vous pouvez sélectionner l'un des événements pour voir son détail(voir l'annexe 5)

 # perf report

FIGURE 12 – rapport de la trace sur xclock - Perf

4.2.4 SystemTap

Systemtap est un outil qui permet aux développeurs et aux administrateurs d'écrire et de réutiliser des scripts simples pour examiner en profondeur les activités d'un système Linux *en direct*.

Remarque : SystemTap est très optimisé avec les distributions « Red-Hat », les dernières versions ne sont pas compatible (des corrections sont apportées au Bugs), la version SystemTap2.5 fonctionne toujours bien(disponible sur ce lien : `https://sourceware.org/systemtap/ftp/releases/`) [8].

1. SystemTap vérifie le script par rapport à la bibliothèque tapset existante (normalement dans /usr/share/-systemtap/tapset/ pour tous les tapsets utilisés) SystemTap remplacera alors tous les tapsets localisés par leurs définitions correspondantes dans la bibliothèque tapset.
2. SystemTap traduit ensuite le script en C, exécutant le compilateur C du système pour en créer un module noyau. Les outils qui exécutent cette étape sont contenus dans le package SystemTap.
3. SystemTap charge le module, puis active toutes les sondes (événements et gestionnaires) dans le script.
4. Lorsque les événements se produisent, leurs gestionnaires correspondants sont exécutés.
5. Une fois la session SystemTap terminée, les sondes sont désactivées et le module noyau est déchargé.

8. Pour une version plus spécifique à votre noyau : `https://pkgs.org/download/systemtap`

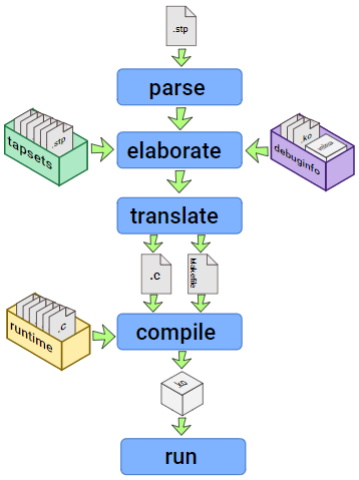

FIGURE 13 – Cycle de vie d'un script systemTap

N.B : SystemTap a besoin de certaines dépendances, des pages de guide d'installation sont disponible sur `https://blog.jeffli.me/blog/2014/10/10/install-systemtap-in-ubuntu-14-dot-04/` ou `https://wiki.ubuntu.com/Kernel/Systemtap`.

4.2.5 Suivre les accès mémoire en utilisant des outils automatisés

— **SystemTap :** Le script de l'annexe 3 permet de tracer les accès mémoire en temps réel.

```
root@m2lse-VirtualBox:/home/m2lse# stap showmem.stp
vminfo: Page fault at 0xb77b0000 on a read..Major fault
pool: Page fault at 0xb6300000 on a write..Minor fault
pool: Page fault at 0xb6300000 on a write..Minor fault
unity-scope-vid: Page fault at 0xa348db4 on a write..Minor fault
gnome-terminal: Page fault at 0x926a20c on a write..Minor fault
gnome-terminal: Page fault at 0x9266000 on a write..Minor fault
gnome-terminal: Page fault at 0x9267010 on a write..Minor fault
gnome-terminal: Page fault at 0xbfdad000 on a write..Minor fault
gnome-terminal: Page fault at 0x9268018 on a write..Minor fault
gnome-terminal: Page fault at 0x9269000 on a write..Minor fault
gnome-terminal: Page fault at 0x927220c on a write..Minor fault
gnome-terminal: Page fault at 0x9273214 on a write..Minor fault
gnome-terminal: Page fault at 0x926b00c on a write..Minor fault
gnome-terminal: Page fault at 0x9274a1c on a write..Minor fault
^Cgnome-terminal: Page fault at 0xb6d830c8 on a read..Major fault
gnome-terminal: Page fault at 0xb6cff2ae on a read..Major fault
gnome-terminal: Page fault at 0x926c014 on a write..Minor fault
gnome-terminal: Page fault at 0x9275224 on a write..Minor fault
root@m2lse-VirtualBox:/home/m2lse#
```

FIGURE 14 – Tracer les accès mémoire en temps réel - SystemTap

— **Monitoring des accès à distance avec FTrace :** Il peut y avoir des situations où il y'a un besoin de tracer un système embarqué(Raspberry PI, Beagle-bone black, i.MX6, ...,etc.) ou une machine avec très peu d'espace disque. C'est là que le traçage par réseau est pratique.
Dans cette exemple, nous allons tracer les accès des fonctions d'allocations mémoire *kmalloc* et dé-allocation

mémoire *kfree*.
La configuration se fait en deux étapes[8] :

1. **Coté PC :**

 # trace-cmd listen -p 8890

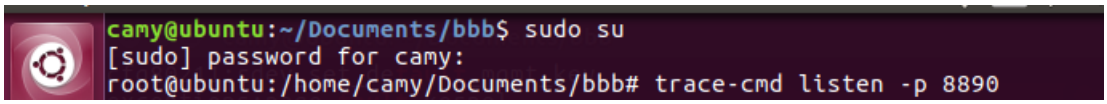

FIGURE 15 – Lancement du listener trace-cmd du coté PC

2. **Coté carte embarqué :** le contenu de *ji.sh* est le suivant :

 # ./trace-cmd record -e 'kmem :mm_page_alloc' -e 'kmem :mm_page_free' -N <IP-addr-PC> :8890

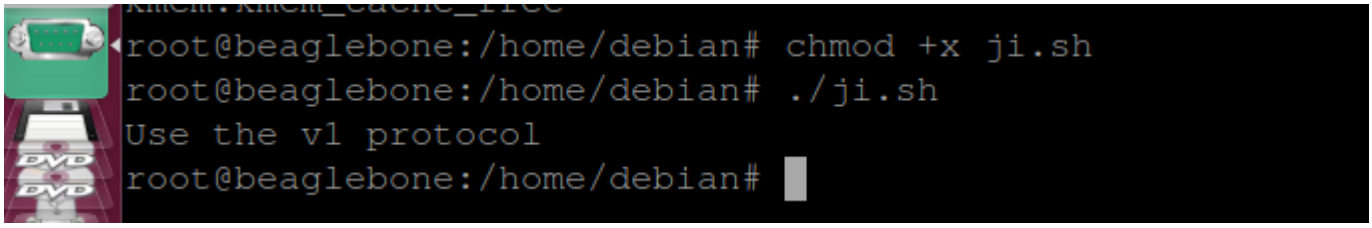

FIGURE 16 – Lancement du client trace-cmd du coté Beaglebone

3. **Le PC reçoit la trace et l'enregistre :**

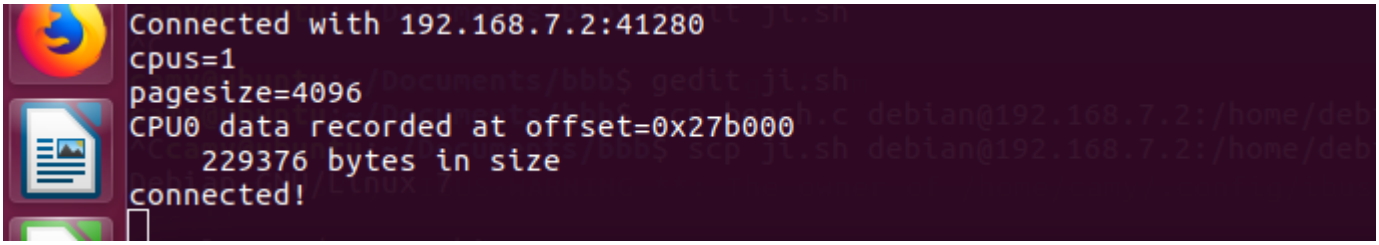

FIGURE 17 – Réception de la trace du coté PC

4. **Afficher le rapport du coté PC :**

 # trace-cmd report

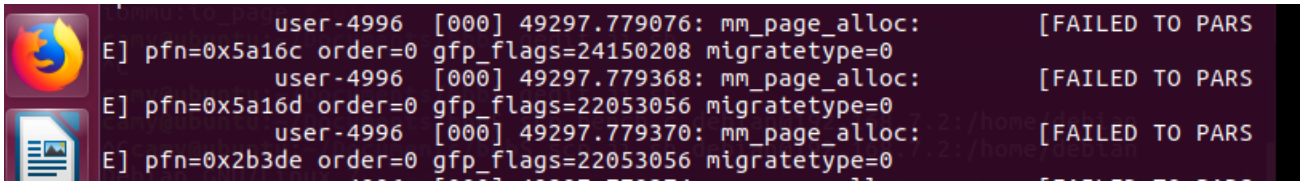

FIGURE 18 – Rapport de la trace - trace-cmd

Voici comment lire la trace :
— user-4996 : c'est la paire nom_processus-PID_processus
000 : le processeur sur lequel s'exécute le programme
— 49297.779076 :c'est le timestamp.
— mm_page_alloc : fonction qui s'est exécuté.
— pfn(page frame number) : cadre de la mémoire.

5 Méthode d'évaluation d'outils

5.1 Métrique de comparaison d'outils

5.1.1 Intrusivité Mémoire

Il existe plusieurs métriques pour mesurer la quantité de mémoire utilisée par un processus(ou un outil)[9].

— **Vss (Virtual set size) :** quantité totale de mémoire mappée par un processus.

— **Rss (Resident memory size) :** la somme de la mémoire qui est mappée aux pages physiques de la mémoire.

En 2009, « Matt Mackall »a examiné le problème de la comptabilisation des pages partagées dans la mesure de la mémoire des processus et a ajouté deux nouvelles indicateurs :

— **Uss (Unique set size) :** C'est la quantité de mémoire qui est affectée à la mémoire physique et qui est unique à un processus ; elle n'est partagé avec aucun autre.

— **Pss (Proportional set size) :** Cela fractionne la comptabilisation des pages partagées qui sont affectées à la mémoire physique entre tous les processus qui les ont mappés.

5.1.2 Intrusivité temps d'exécution

Les statistiques de la comparaison doivent inclure :

— Le temps d'exécution d'un programme sans/avec le traceur (utiliser des outils avec la précision de gettimeofday()).

— Comptabilisation du nombre de changement de contexte sans/avec le traceur.

5.2 Outils de profiling

5.2.1 Outils de base

— **time :** La commande time permet de déterminer la durée d'exécution d'un processus. La **Figure 19** montre la durée d'exécution de mon programme RNCS(Robot Navigation Control/Simulation), un simulateur de trajectoire pour un robot de navigation.

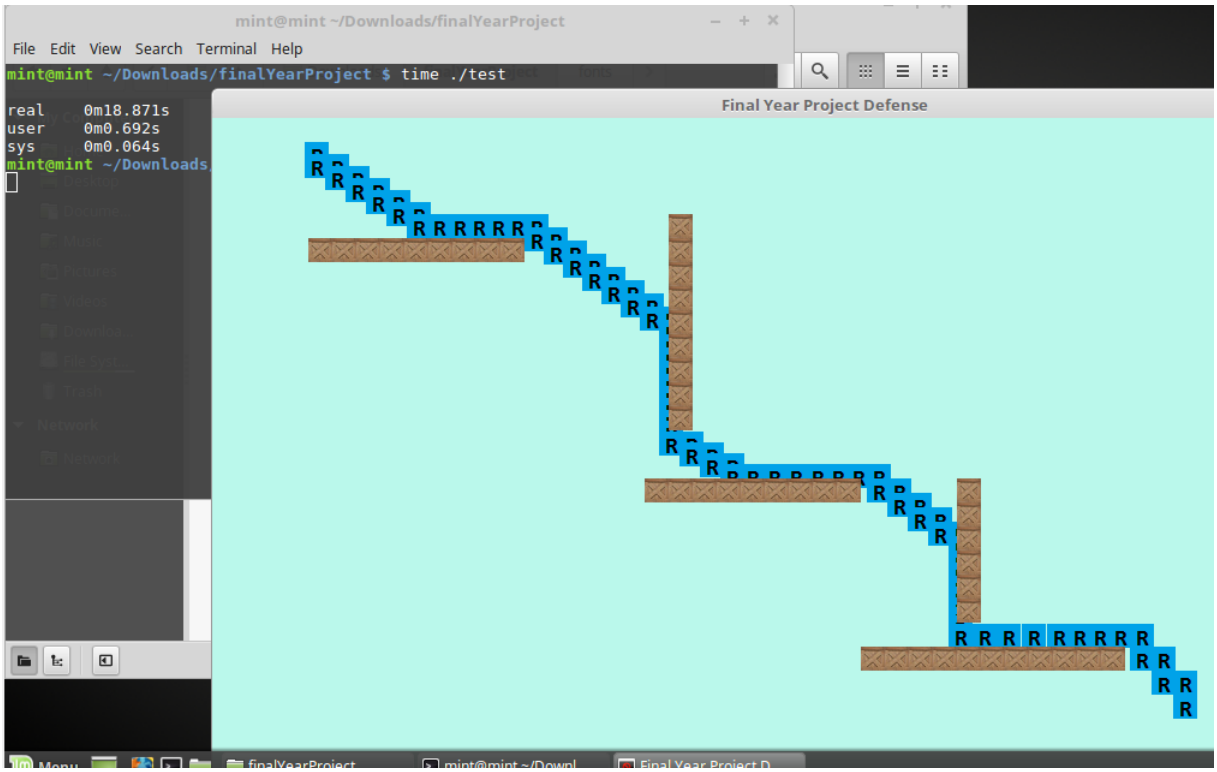

FIGURE 19 – Commande Time

Les différents champs représentent :

1. **real :** temps du début à la fin de l'appel. C'est tout le temps écoulé, y compris les tranches de temps utilisées par d'autres processus et le temps que le processus passe bloqué.
2. **user :** Ceci est le temps CPU réel(mode utilisateur) utilisé dans l'exécution du processus.
3. **sys :** Ceci est le temps CPU réel(mode noyau) utilisé dans l'exécution du processus.

— **/usr/bin/time :** Permet de mesurer le peak Rss (Resident memory size) du début du programme jusqu'à la fin. Elle permet aussi d'avoir des statistique sur les défauts de pages(mineur et majeur) et les changements de contexte(**Figure 20**).

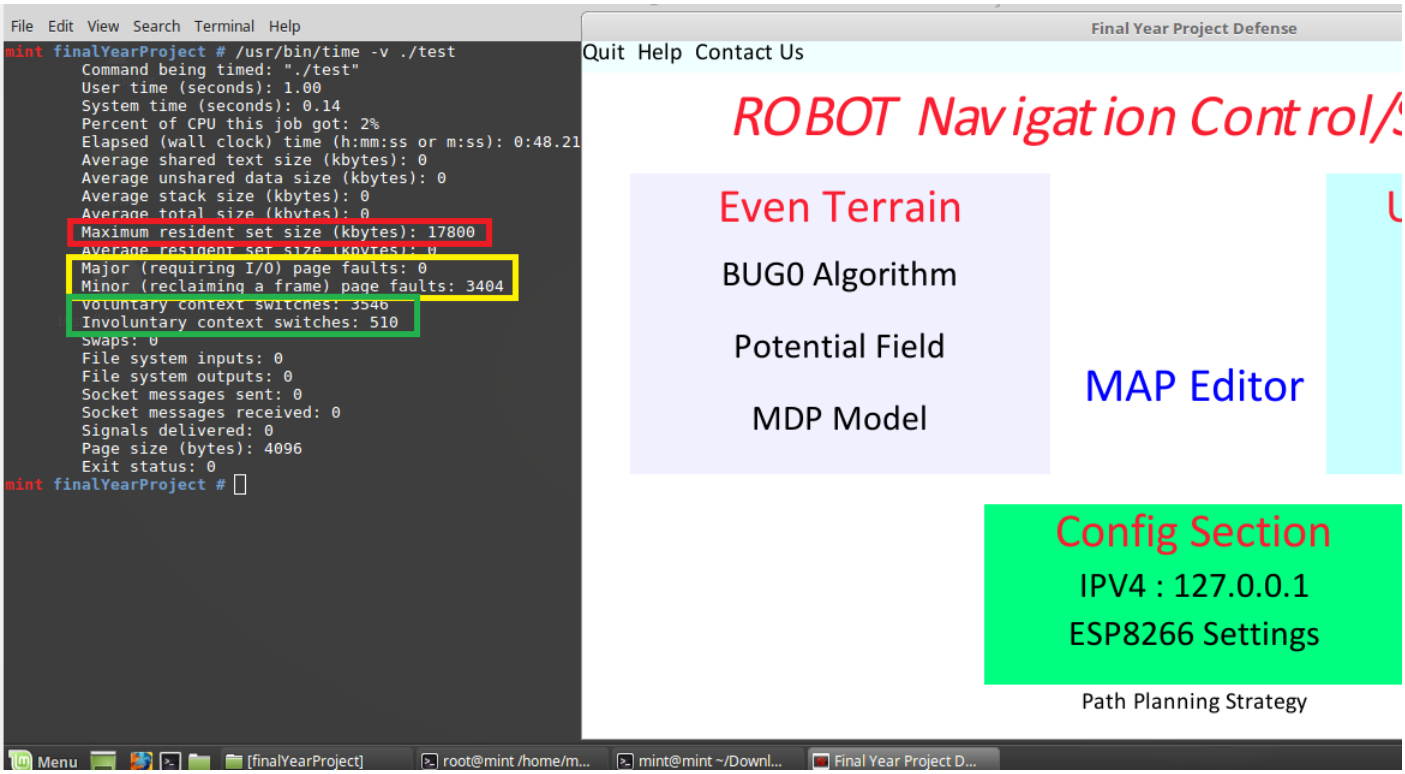

FIGURE 20 – Commande /usr/bin/time

5.2.2 Avoir plus de précision sur les temps d'exécution

— **Gprof :** Gprof est un programme de profilage qui recueille et organise des statistiques sur un programme. Gprof ne capture pas d'échantillons à partir de threads autres que le thread principal d'un processus multi-thread, et il n'échantillonne pas l'espace du noyau, ce qui limite son utilité(la **Figure 21** indique l'échec de GPROF à calculer le temps d'exécution de mon programme).

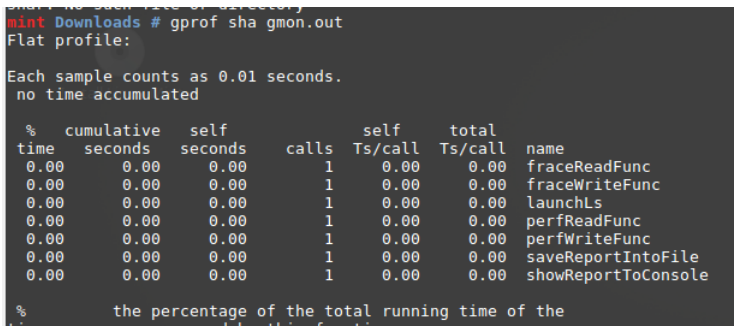

FIGURE 21 – Les limitations de GProf

— **Gperftools :** un ensemble d'outils fournis par Google visant à analyser et à améliorer les performances des applications multi-thread. Ils offrent un profileur de CPU, une implémentation de malloc rapide, un détecteur de fuite de mémoire et un profileur de tas [9].

— **Valgrind/callgrind :** fournit les résultats les plus précis et convient bien aux applications multithread. Valgrind conduit à une exécution lente de l'application(voir l'annexe 4).

— **Perf :** Pour profiler une application, nous devons enregistrer des informations sur une exécution.

 # perf record programme [options_programme]

par exemple :

— pour enregistrer les résultats de Perf dans un fichier « assembly.eddi » :

 # perf record myGame assembly.eddi

— Puis il suffit de lire le rapport :

 # perf report

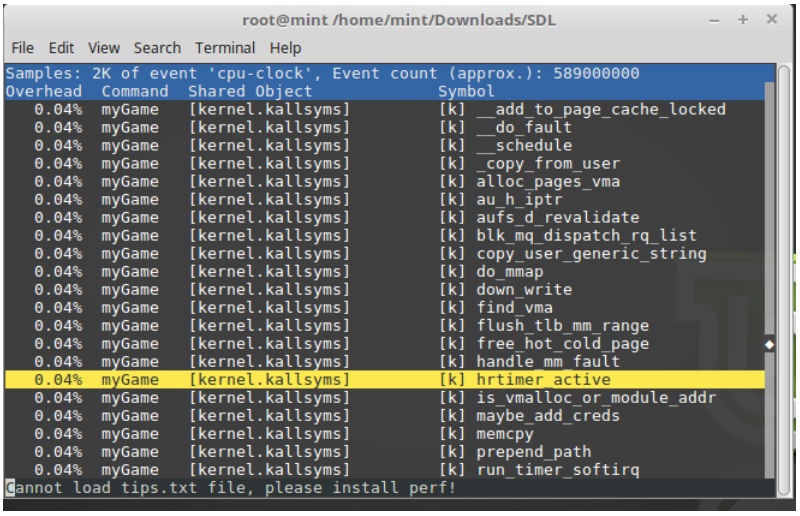

FIGURE 22 – Rapport de Mesure de la granularité d'exécution d'un programme avec Perf

— **OProfile :** OProfile est un profileur statistique pour les systèmes Linux, capable de profiler tout le code courant à une faible intrusivité.
On peut utiliser *OProfile* pour profiler les performances d'un seul processus ou sur l'ensemble d'un système. Il supporte les gestionnaires d'interruption matériels et logiciels, les modules de noyau, le noyau, les bibliothèques partagées et les applications[10].

5.3 Outils d'analyse des performances de la mémoire

5.3.1 lmbench

lmbench est une suite de microbenchmarks ANSI/C simples et portables pour UNIX/POSIX. En général, il mesure deux caractéristiques clés : la *latence* et la *bande passante*.
Les sources de lmbench sont disponible sur ce lien : `http://www.bitmover.com/lmbench/get_lmbench.html`, cependant il existe un programme *lmbench-run* [10] qui est un outils front-end(pas la peine de passer par les sources).
Après quelques essais sur deux machines différente deux choses sont constatés :

— configuration compliqué et prend beaucoup de temps(la documentation dit que les paramètres par défauts sont bon)

— lmbench-run requière beaucoup d'espace de stockage(la documentation ne fait pas référence au chiffre exacte).

Aprés quelques secondes du lancement de lmbench, les deux machines se sont planté

5.3.2 perf bench

Ce n'est pas un nouvelle outil, c'est l'une des options qu'offre *Perf*(voir section 4.2.3). Cette outils test la globalité du système. la maniére la plus simple d'utiliser perf est :

9. Pour apprendre à utiliser gperfools : `https://github.com/gperftools/gperftools/wiki`
10. lmbench-run est disponible sur ce lien : `http://manpages.ubuntu.com/manpages/artful/man1/lmbench-run.1.html`

 # perf bench all

FIGURE 23 – Perf bench compare les différentes implémentations de memcpy et memset

On voit bien que ca renvoie le temps d'exécution de memcpy et memset avec différente implémentation.

Ce lien explique le fonctionnement de Perf bench : `https://perf.wiki.kernel.org/index.php/Tutorial`

6 Etude comparative des outils

6.1 Etude comparative

— **Classification basique :**

Outils	Outil Front-end	Nombre de Tracepoints	Traçage par réseau	Facilité d'utilisation	Portabilité
Ftrace	Trace-cmd	1228	Oui	Facile	inclus
Perf	perf	1217	Non	Facile	inclus
SystemTap	/	Pas d'option pour lister le nombre	Non	Moyenne* (basé sur des scripts)	SystemTap-2.5 fonctionne bien

* : SystemTap est plus facile que eBPF (basé sur une variante du langage C).

— **Classification élaborée :** les outils *time* et */usr/bin/time* ont été utilisés sur un algorithme de tri *Quick sort* avec et sans traceurs, Les tests se sont déroulés sur une machine virtuelle linux(mint-18.3-cinnamon-64bit avec 2GB RAM , 1 CPU(2.00GHZ)). La charge du système est mesurée avant de commencer le test(**Figure 24**)

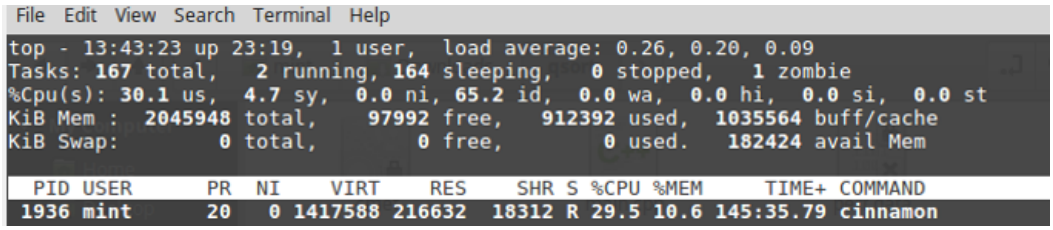

FIGURE 24 – Charge de la machine linux avant le test

La valeur du « load average »est loin de 1, ce qui veut dire que le système n'est pas chargé. on voit aussi qu'il y'a assez de mémoire pour effectuer l'expérimentation.

1. Quick sort avec 500.000 éléments

Outils	Temps d'exécution	Max Mémoire RSS (kB)	Défauts de page mineurs
qsort (sans traceur)	3.983	5000	1034
qsort (avec trace-cmd)	19.654	5016	1199
Lecture trace trace-cmd	2m26.764s	18856	15384
qsort (avec Perf)	5.743	11720	3258
Lecture trace Perf	2.174	13564	2333

On voit clairement que Ftrace(Trace-cmd) est plus intrusive que Perf.

Les résultats ci dessous peuvent s'expliquer par la quantité d'information capturée et enregistrée par les deux traceurs. La **Figure 25** montre la taille des fichiers de traces de Ftrace et Perf pour cette algorithme de tri *Quick Sort*.

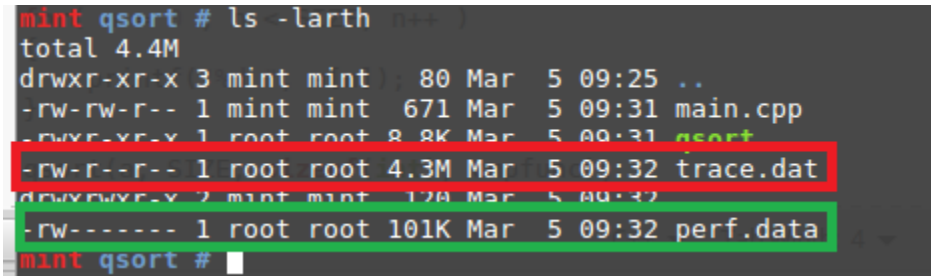

FIGURE 25 – Taille de la trace Ftrace et Perf

On peut voir de la **Figure 25** que la trace Pef(perf.data) prend 101Ko et Ftrace(trace.dat) prend 4.3Mo(x43.6 fois plus grand que Perf).

La charte suivante visualise mieux cette différence :

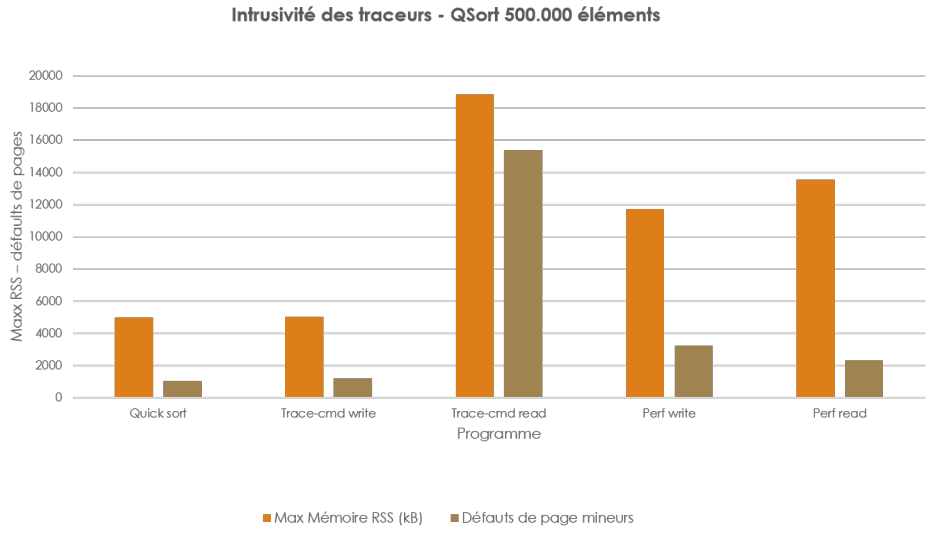

FIGURE 26 – Graphe de comparaison de l'intrusivité des outils avec qsort 500.000 éléments

2. Quick sort avec 1.000.000 éléments : On peut vérifier les résultats déja obtenus avec plus d'éléments.

Outils	Temps d'exécution	Max Mémoire RSS (kB)	Défauts de page mineurs
qsort (sans traceur)	6.300	9252	2014
qsort (avec trace-cmd)	35.913	9252	2178
Lecture trace trace-cmd	4m16.082s	18936	23897
qsort (avec Perf)	10.567	12244	4317
Lecture trace Perf	3.004	15772	2811

On peut constater que les temps d'exécution ont presque doublé par rapport au *quick sort avec 500.000 éléments*.
On peut aussi compter le nombre d'entrées enregistrées par les deux traceurs(**Figure 27**), On voit que Trace-cmd enregistre une plus grande quantité d'information que Perf.

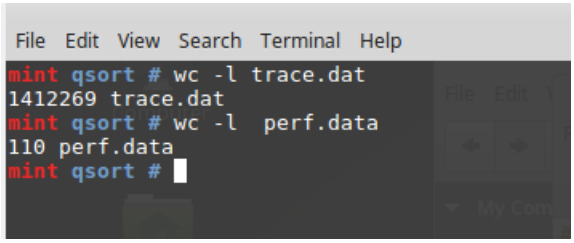

FIGURE 27 – Nombre d'entrées enregistrées par les deux traceurs

La charte suivante visualise mieux cette différence :

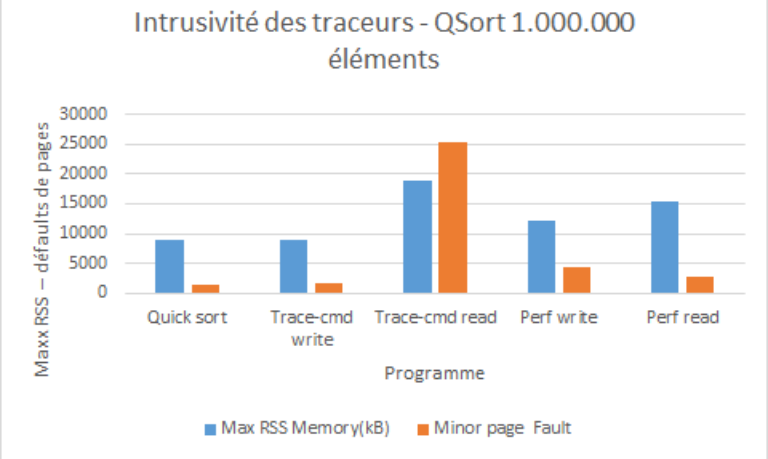

FIGURE 28 – Graphe de comparaison de l'intrusivité des outils avec qsort 1.000.000 éléments

7 Conclusion

Les traceurs sont différents dans plusieurs aspect :
— SystemTap est le seul traceur d'événements à temps réel(puisque c'est un module donc pas de changement de contexte), le format du rapport de la trace est défini par l'utilisateur.
— Ftrace peut faire du traçage réseau (il faut utiliser la même version de trace-cmd sur les deux machine), Ftrace sauvegarde beaucoup d'informations.

— Perf est un profileur est un traceur (peut aussi tracer en temps réel mais les résultats sont moins précis que SystemTap à cause des changements de contexte), Perf est plus léger que Ftrace et cause moins d'intrusivité.
— SystemTap2.5 est la version SystemTap qui fonctionne bien sur linux, cependant les dernières versions fonctionnent parfaitement sur RED-Hat.
— SystemTap est le plus puissant des traceurs car il permet aussi de modifier le comportement d'une fonction et ses arguments(si on compile le script en mode GURU avec l'option -g).
— d'autres outils qui sont très attendus comme : eBPF et ktap(qui va utiliser la même machine virtuelle que eBPF).

Ftrace, Perf et systemTap peuvent tracer n'importe quelle fonction par le billet des Tracepoints et kprobes.
Tous ces outils nous ont permis de tracer les fonctions d'allocations mémoire puis les défauts de page. Seulement il reste encore à tracer les accès mémoire, les *cache miss* et la réclamation de pages(avec deux listes maintenues par linux « Active »et « innactive »qui fait usage de l'algorithme *second chance algorithm*) serai de bonnes pistes pour continuer.

Ces outils sont puissants, et peuvent causer une instabilité du système ou son arrêt, tous les tests doivent se faire des machines virtuelle.

Appendices

1 Un outil d'observabilité personalisé

Sous Linux, la fonction **sysinfo()** remplit une structure *sysinfo* avec les statistiques système.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysinfo.h>

int main(int argc,char *argv[]){
  struct sysinfo info;
  sysinfo(&info);
  printf("\033[31m Uptime(Since system booting) : %ld \n",info.uptime);
  printf("\033[32m Load average : 1 minutes = %ld, 5 minutes = %ld, 15 minutes = %ld\n",info.
  loads[0],info.loads[1],info.loads[2]);
  printf("\033[33m Total usable main memory size : %ld kB\n",info.totalram/DIV);
  printf("\033[34m Available memory size : %ld kB\n",info.freeram/DIV);
  printf("\033[35m Amount of shared memory  : %ld kB\n",info.sharedram/DIV);
  printf("\033[36m Memory used by buffers : %ld kB\n",info.bufferram/DIV);
  printf("\033[31m Total swap space size : %ld kB\n",info.totalswap/DIV);
  printf("\033[33m swap space still available : %ld kB\n",info.freeswap/DIV);
  printf("\033[32m Number of current processes : %d\n",info.procs);
  printf("\033[34m Total high memory size : %ld kB\n",info.totalhigh/DIV);

  printf("\033[35m Available high memory size : %ld kB\n",info.freehigh/DIV);
  printf("\033[36m Memory unit size in bytes : %d\n",info.mem_unit);

  return EXIT_SUCCESS;
}
```

Cette page `http://nadeausoftware.com/articles/2012/09/c_c_tip_how_get_physical_memory_size_system` écrite par *Dr. David R. Nadeau* exlique la façon d'écrire un outil d'observabilité cross-platform.

2 Comptabiliser les défauts de pages

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>

int main(int argc,char *argv[]){
  int retourUsage;
  struct rusage usage;
  retourUsage = getrusage(RUSAGE_SELF, &usage);
  if (retourUsage == -1) {
    perror("Erreur dans la fonction getrusage()\n");
  } else {
    //Comptabilser les défauts de pages
    printf ("Nombre de défaut de pages majeur %ld\n", usage.ru_majflt);
    printf ("Nombre de défaut de pages mineur %ld\n", usage.ru_minflt);
  }

  return EXIT_SUCCESS;
}
```

3 Tracer les accès mémoire en temps réel - systemTap

```
#!/usr/bin/stap
```

```
2
3   global current_fault // variable globale
4
5   probe vm.pagefault{ // début de la fonction fault
6     if (write_access){
7       type="write.."
8     } else {
9       type="read.."
10    }
11    current_fault[tid()] = sprintf("%s: Page fault at %p on a %s",execname(), address, type)
12  }
13  probe vm.pagefault.return { // retour de la fonction fault
14    print(current_fault[tid()])
15    delete current_fault[tid()]
16    if (fault_type == 1){
17      printf("Out of memory - OOM\n")
18    } else if (fault_type == 2){
19      printf("Bad memory access - SIGBUS\n")
20    } else if (fault_type == 0) {
21      printf("Minor fault\n")
22    } else {
23      printf("Major fault\n")
24    }
25  }
```

4 Profilage d'un programme avec valgrind

Valgrind permet de faire un profiling très précis, même en incluant les fonctions des bibliothèques partagées(**Figure 29**).

Incl.	Self	Called	Function	Location
100.00	0.01	(0)	0x000011d0	ld-2.15.so
49.28	0.01	1	0x08048470	shar
48.80	0.03	1	(below main)	libc-2.15.so: libc-start.c
47.87	0.01	1	main	shar
47.24	0.35	1	_dl_start	ld-2.15.so: rtld.c, dynamic-lin...
46.87	0.22	1	_dl_sysdep_start	ld-2.15.so: dl-sysdep.c, dl-sys..
46.62	0.50	1	dl_main	ld-2.15.so: rtld.c, dynamic-link.
37.81	0.03	1	saveReportIntoFile	shar
35.70	10.33	4	_dl_relocate_object	ld-2.15.so: dl-reloc.c, dl-machi.
32.36	0.01	1	fopen@@GLIBC_2.1	libc-2.15.so: iofopen.c
32.35	0.02	1	__fopen_internal	libc-2.15.so: iofopen.c
31.87	0.01	1	malloc	libc-2.15.so: malloc.c
31.86	0.01	1	malloc_hook_ini	libc-2.15.so: hooks.c
31.28	0.11	1	ptmalloc_init	libc-2.15.so: arena.c
31.16	31.15	1	_dl_addr	libc-2.15.so: dl-addr.c
28.44	9.61	99	_dl_lookup_symbol_x	ld-2.15.so: dl-lookup.c
18.74	11.22	99	do_lookup_x	ld-2.15.so: dl-lookup.c
10.07	3.31	9	vfprintf	libc-2.15.so: vfprintf.c, printf-p..
6.68	2.61	95	check_match.8846	ld-2.15.so: dl-lookup.c
6.56	0.03	1	showReportToConsole	shar
6.14	0.06	8	printf	libc-2.15.so: printf.c
4.84	0.06	3	_dl_catch_error	ld-2.15.so: dl-error.c
4.73	4.73	215	strcmp	ld-2.15.so: strcmp.S

FIGURE 29 – Profiling du code avec valgrind

si je sélectionne la fonctionne « main de mon code », le détail d'execution de chaque fonction appelée par main est alors affiché(**Figure 30**).

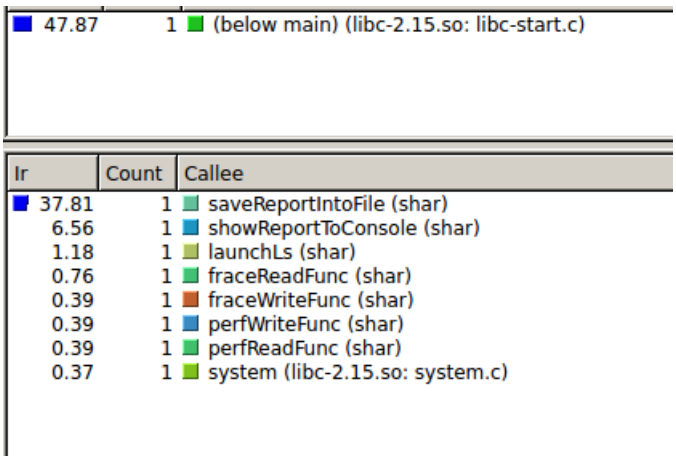

FIGURE 30 – détail d'une fonctions et ses sous-fonctions

5 Détails de la trace avec Perf

La **figure 12** de la **section 4.2.3** montre la fenêtre de statistique de Perf en mode « profileur », il suffit de sélectionner une entrée pour voir sa trace(la chronologie des événements).

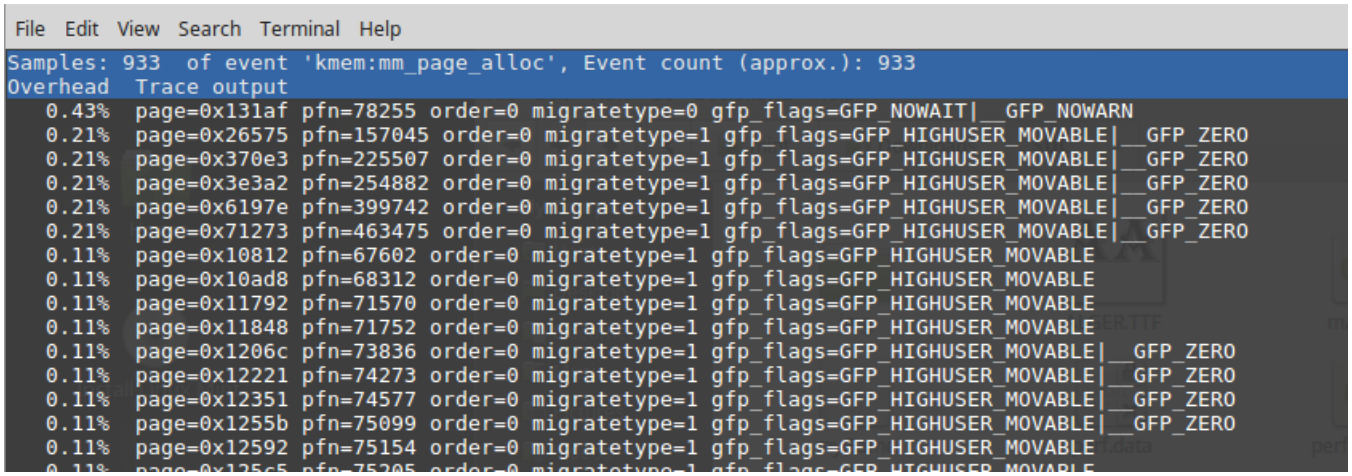

FIGURE 31 – Utiliser Perf en mode Traceur

6 Quantité de mémoire utilisée par le noyau

La mémoire utilisée par le noyau est la somme de : Slab, KernelStack, PageTables, VmallocUsed du fichier */proc/meminfo*(voir La **figure 32**).

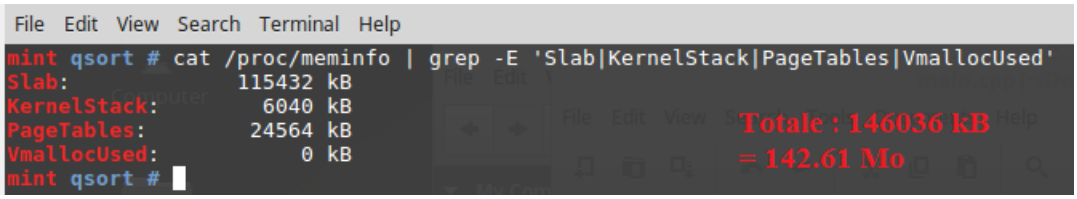

FIGURE 32 – Mesurer la mémoire utilisé par le noyau

Bibliographie

[1] Yoongu Kim1Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them : An experimental study of dram disturbance errors. *researchgate*, page 12, jun 2014.

[2] Seungbum Hong, Orlando Auciello, and Dirk Wouters. *Emerging Non-Volatile Memories*. Springer New York Heidelberg Dordrecht London, 2014.

[3] Robert Love. *Linux Kernel Development, 3rd edition*. Pearson Education, Inc, 2010.

[4] Abraham SILBERSCHATZ, Peter Baer GALVIN, and Greg GAGNE. *Operating system concepts, 9th edition*. Wiley Publishing, Inc, 2013.

[5] Pierre Ficheux. Introduction à ftrace. `http://www.linuxembedded.fr/2011/03/introduction-a-ftrace/`, Mar 2011.

[6] Steven Rostedt. Debugging the kernel using ftrace - part 1. `https://lwn.net/Articles/365835/`, Dec 2009.

[7] Steven Rostedt. Debugging the kernel using ftrace - part 2. `https://lwn.net/Articles/366796/`, Dec 2009.

[8] Pierre Ficheux. Debugging realtime application with ftrace. `https://fosdem.org/2018/schedule/event/debugging_tools_ftrace/attachments/slides/2670/export/events/attachments/debugging_tools_ftrace/slides/2670/Ftrace.pdf`, Feb 2018.

[9] Chris Simmonds. *Mastering Embedded Linux Programming*. Packt Publishing Ltd, 2015.

[10] IBM. *Getting started with OProfile*. IBM, 2016.