Architecture d'un système d'exploitation

TD2

L'ordonnanceur du noyau Linux

Etudiant : Romain Pereira

Encadrants : M. Loussert
M. Taboada

25/10/2018

TD1 : l'ordonnanceur du noyau Linux

Romain PEREIRA

25/10/2018

Table des matières

1 Préambule

Le but de ce TP est de comprendre le fonctionnement d'un ordonnanceur, en étudiant l'ordonnanceur du noyau Linux.

2 Fonctionnement de l'ordonnanceur

Q.1 Nos ordinateurs de bureau dispose généralement d'un seul processeur (avec 1, 2, 4 ou 8 coeurs d'execution).

Lorsque plusieurs processus s'execute sur la machine, il faut donc qu'il y ait un méchanisme de partage du processeur : il est le seul point d'execution pour les processus.

C'est la fonction principale de l'ordonnanceur. Il choisit dans quel ordre plusieurs processus peuvent être executé sur un même processeur.

N.B : l'ordonnanceur est lui même un processus.

L'execution de l'ordonnanceur peut avoir lieu pour répondre aux problèmes suivants :

- Lorsqu'un processus se clone (*fork()*) : lequel du père ou du fils doit avoir la priorité d'éxecution ?

- Lorsque le processus courant s'arrête : quel processus doit prendre la main ensuite

- Lorsqu'un processus bloque sur une entrée/sortie, ou qu'il est en attente : doit t'il continuer de bloquer le processeur alors qu'il ne travaille pas ?

- Lorsqu'une entrée/sortie est interrompu : peut être que le processus a fini son job, ou bien c'est un signal indiquant qu'un processus précèdemment en attente entrée/sortie peut maintenant effectué son travail.

Aussi, on distingue 2 grandes politiques décidant de l'execution de l'ordonnanceur.

- non-pre-emptive : l'ordonnanceur ne tourne que si le processus utilisateur s'arrête, bloque, ou demande explicitement à changer de processus. Si le processus ne rends pas la main pendant 100 ans, l'ordonanneur attendra impassiblement pendant 100 ans.

- pre-emptive : Chaque processus à un temps d'execution fixe. L'ordonnanceur est executé à intervalle régulier, et détermine le prochain prochain processus qui doit être executé sur le prochaine 'créneau'.

Q.2 La fonction *sched_yield* passe la main sur le processeur d'un processus à un autre (ou d'un processus vers lui même s'il reste le plus 'prioritaire').

S'il n'y a pas d'autre processus devant être executer, cette fonction s'arrête.

Q.3 Code modifié

```
[...]
printk("current=%p\n", current);
schedule();
printk("current=%p\n", current);
[...]
```

Compilation de l'image linux modifié et déploiement

```
cd ~/debian_kernel/linux-4.9.30/
make bindeb-pkg
dpkg -i ../linux-image-4.9.30_4.9.30-2_amd64.deb
reboot
```

Programme utilisateur 'main.c'

```
int main() {
    sched_yield();
    return 0;
}
```

Compile et lance le programme utilisateur

```
gcc main.c
./a.out
dmesg
```

Sortie dmesg

```
[  170.029953] current=ffff9337bbe47140
[  170.029956] current=ffff9337bbe47140
```

=> Le pointeur 'current' reste inchangé Ceci est dû au fait que peu de processus tourne sur la machine virtuelle. Le processus prioritaire reste le même entre 2 appels de 'schedule()'

En lançant d'autres processus sur la machine, la valeur du pointeur aurait changé

Q.4 On génère d'abord les 'ctags' pour naviguer plus facilement dans le code source.

```
cd ~/debian_kernel/linux-4.9.30/
ctags -R .
```

Lecture de 'core.c'

```
n ligne  profondeur/schedule()  execution
4898  0  schedule();

3454  1  sched_submit_work(task);
3438  2  ...

//debut boucle : desactive le pr emption
3456  1  preempt_disable();

3457  1  __schedule(false);
3342  2  cpu_rq(); // recuperes la 'run queue' du CPU
3343  2  prev = rq->curr; // enregistre le processus courant dans la run queue
3391  2  next = pick_next_task(rq, prev, cookie); // recuperes la task suivante

// r active la pr emption
3457  1  sched_preempt_enable_no_resched();

3459  1  need_resched();
// tant que 'need_resched()' renvoit 'vrai',
// on boucle sur partir de preempt_disable()
```

Q.5 La fonction principal de l'ordonnanceur est 'schedule()'

```
// choisit le prochain processus      tre   execut
// avant l'appel de cette fonction, un processus tourne
schedule();
// apres l'appel, un autre processus peut tourner
```

Q.6 La préemption est un mecanisme d'ordonnancement qui consiste à donner un temps maximal d'execution à un processus, au dela duquelle l'algorithme d'ordonnancement sera relancé (c.f **??**)

Si la préemption est active lors de l'execution de l'ordonnanceur, ...

Q.7 bench_huge

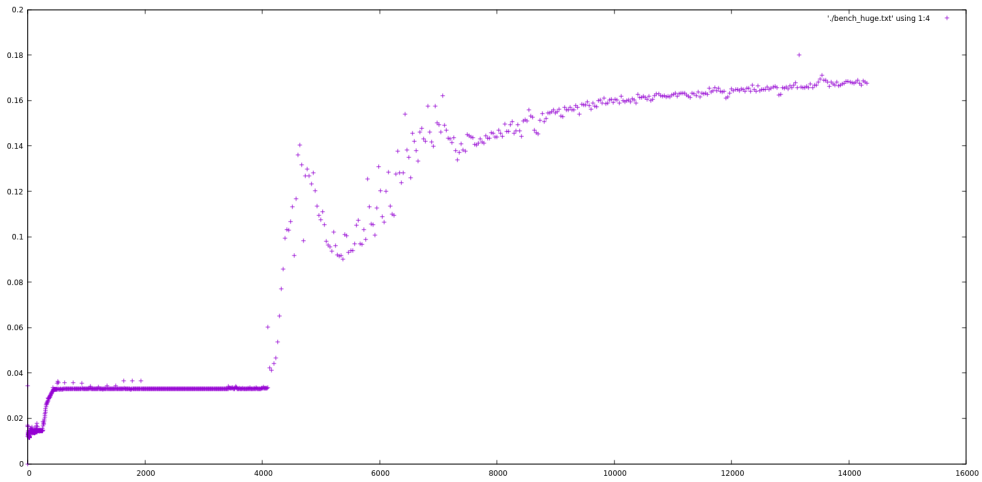

FIGURE 1 – en dessous de 4096ko (capacité du cache **??**), les performances obtenus sont optimales

Q.8 L'utilisation du cache est beaucoup plus optimal avec les huges pages. Au regard de ce qui a été dit précèdemment, le mappage mémoire est plus contigu avec des huge pages, ce qui réduit le risque d'avoir des lignes de cache non-utilisées.

Q.9 Pour un petit processus qui requiert peu de mémoire, les *huge pages* alloue beaucoup de mémoire en surplus. Les processus les plus petit peuvent avoir besoin de moins de $1Ko$.

Avec des pages de taille traditionelle ($4Ko$), on a 25% de la mémoire alloué qui est réellement utilisé.

Avec des huge pages de $2Mo$, on passe à moins de 1% de mémoire alloué utile pour des petits processus.

On peut donc rapidement se retrouver avec une mémoire 'pleine de vide' (comprendre : 100% de mémoire alloué pour un besoin réel de 1%)

3 Création d'un allocateur en mode utilisateur

3.1 Gestion du malloc

Q.10 La fonction *hp_ init* alloue une réserve mémoire, mappé sur un fichier '/mnt/huge/bench' (il faut que des huge pages aient préalablement été monté sur '/mnt/huge').

La fonction *hp_ finalize* libère la réserve mémoire, et supprimes le fichier '/mnt/huge/bench'.

Ce code est destiné à être compilé en bibliothèque dynamique ('shared library').

Au chargement de la bibliothèque par un programme (*dlopen()* en C), la fonction *hp_ init* est appelée.

Lorsque la bibliothèque est déchargé (*dlclose()* en C), la fonction *hp_ finalize* est appelée.

Q.11 La variable **alloue** est un 'accumulateur'. Elle accumule le nombre d'octet alloué dans la réserve.

Ceci permet de savoir où ce situe la prochaine zone allouable. (à l'adresse $reserve.mem + alloue$)

Q.12 *hp_ malloc*

```
static void *__hp_malloc(size_t taille)
```

Cette fonction prends une taille T en paramètre (en octet), et renvoie un pointeur vers une zone mémoire pouvant accueillir jusqu'à T octets. Elle *alloue* T octet à l'utilisateur.

Pour cela, elle renvoie un pointeur à la 1ère zone libre de sa réserve, puis enregistres la prochaine adresse libre à l'aide de l'accumulateur.

De plus, la taille réellement alloué est au moins égal à T, modulo la taille d'alignement mémoire.

Aussi, s'il n'y a plus de mémoire en réserve, la fonction affiche une erreur et renvoie un pointeur $NULL$.

Q.13 implémentation *hp_ malloc*

```
static void *__hp_malloc(size_t taille)
{
    static size_t alloue = 0UL;
    void *ret;

    //La fonction calcule_multiple_align() d termine le plus petit
    //multiple de  l alignement  maximal  gal  ou sup rieur  a.
    //Celle-ci nous permet  d  arrondir  la taille demand e de
    //sorte que les blocs allou s soit toujours correctement align s.
    taille = calcule_multiple_align(taille);

    if (alloue + taille > MEM_SIZE) {
        fprintf(stderr, "Il␣n'y␣a␣plus␣assez␣de␣m moire␣disponible.\n");
        return NULL;
    }
    ret = (void *) (reserve.mem + alloue);
    alloue += taille;
    return ret;
}
```

Q.14 Cette implémentation n'offre pas à l'utilisateur la possibilité de libérer de la mémoire.

Ainsi, si la réserve est pleine, l'utilisateur sera simplement à court de mémoire.

3.2 Gestion du free

Q.15 *recherche_bloc_libre*

```
//        En-t te                                          En-t te
//        <-------------------->                           <-------------------->
//        +----------+----------+------------------+       +----------+----------+------------------+
//        | Taille   | Suivant |    Bloc allou     |       | Taille   | Suivant |    Bloc allou     |
//        +----------+----------+------------------+       +----------+----------+------------------+
//                         +                                ^
//                         |                                |
//                         +--------------------------------+
```

```
static struct bloc *recherche_bloc_libre(size_t taille)
```

Les blocs libres de l'allocateur ont une structure de liste chainée. Cette fonction part du 1er bloc libre, et les parcourt jusqu'à trouver un bloc de taille suffisante (pouvant accueillir au moins *taille* octets.

Si un bloc de taille suffisante est trouvé, elle supprime le bloc-maillon de la liste, et renvoie le bloc trouvé.

Sinon, elle renvoit 'NULL'.

Q.16 implémentation *recherche_bloc_libre*

```
struct bloc *bloc = libre;
struct bloc *precedent = NULL;
struct bloc *ret = NULL;

while (bloc)
{
```

...

Q.17 *hp_malloc* La fonction a le même prototypage et la même utilisation que la fonction 3.1.

Son fonctionnement interne est cependant différent.

Tout d'abord, elle aligne la taille mémoire comme dans 3.1. Ensuite, elle recherche parmis sa structure de liste chainée de bloc libre, si un bloc libre de taille suffisante existe déjà (via *recherche_bloc_libre* 3.2).

Si un tel bloc existe, il est récupéré, sinon, un nouveau bloc est crée dans la réserve puis initialisé.

Finalement, la fonction renvoie l'adresse mémoire destiné à l'utilisateur ($bloc + TAILLE_ENT_ETE$, voir schéma ASCII 3.2)

Q.18 implémentation *hp_malloc*

```
static void *__hp_malloc(size_t taille)
{
    /*
     *      * Alloue un bloc de m moire au moins de taille `taille`.
     *           */

    static size_t alloue = 0UL;
    void *ret;

    assert(taille > 0);

    //La fonction calcule_multiple_align() d termine le plus petit
    //multiple de  l alignement  maximal  gal  ou sup rieur    a.
    //Celle-ci nous permet  d    arrondir    la taille demand e de
    //sorte que les blocs allou s soit toujours correctement align s.
    taille = calcule_multiple_align(taille);

    ret = recherche_bloc_libre(taille); //recherche d'un bloc libre

    if (!ret) // aucun bloc de taille suffisante libre
    {
        if (MEM_SIZE - alloue < TAILLE_EN_TETE + taille)
        {
            fprintf(stderr, "Il n'y a plus assez de m moire disponible.\n");
            return NULL;
        }

        ret = reserve.mem + alloue;
        alloue += TAILLE_EN_TETE + taille;
        bloc_init(ret, taille);
    }

    // renvoie l'adresse sur la zone m moire utilisateur ('juste apr s' l'ent te)
    return (void *) (((char *)ret) + TAILLE_EN_TETE);
}
```

Q.19 *hp_free*

Cette fonction libère le bloc qui contient la mémoire utilisateur passé en paramètre.

Comme pour la fonction *free*(), cette fonction ne fait rien si le pointeur passé en paramètre est *NULL*.

Le bloc libéré est alors chainé à la liste des blocs libres.

Q.20 *hp_free*

ATTENTION : J'ai modifié l'implémentation de cette fonction : elle ajoute le bloc libre en tête de liste chainée (structure FIFO), et non pas en fin comme le code à trou le suggérait (FILO)

Si n est le nombre de blocs déjà libres, la complexité du 'free' aurait été en $O(n)$ avec l'implémentation suggeré, en FIFO, la complexité sera toujours en $O(1)$.

De plus, inséré en tête ou en fin de liste n'a à priori aucunes influences sur les performances du reste de l'allocateur.

```
static void __hp_free(void *ptr)
{
    /*
     *      * Ajoute le bloc fourni    la liste des blocs libres.
     *           */


    if (! ptr)
        return;

    struct bloc * prev_libre = libre;
    libre = (struct bloc *)((char *)ptr - TAILLE_EN_TETE);
    libre->suivant = prev_libre;
}
```

4 Bibliothèque dynamique

Q.21 'Makefile' dans le dossier 'hp_allocator/hp_allocator_malloc_free/SRC/'

```
help:
	@echo "Compile le programme de test"
	@echo "> make a.out"
	@echo ""
	@echo "Compile en biblioth que dynamique"
	@echo "> make shared"

a.out: main.c hp_allocator.c hp_allocator.h
	gcc $^ -o $@ -g

shared: hp_allocator.c hp_allocator.h
	gcc -c -fpic hp_allocator.c -o hp_allocator.o
	gcc -shared -o libhp_allocator.so hp_allocator.o

clean:
	-rm *.o
	-rm libhp_allocator.so
	-rm a.out
```

Q.22 Pour lancer le programme (avec un CWD à la racine du rendu)

Listing 1 – svalat_bench_lib avec la bibliothèque dynamique

```
# compile la biblioth que
make -C hp_allocator/hp_allocator_malloc_free/SRC/ shared

# compile le benchmark et execution avec la biblioth que
make -C svalat_bench_lib/
LD_LIBRARY_PATH=hp_allocator/hp_allocator_malloc_free/SRC/:$LD_LIBRARY_PATH ./svalat_bench_lib/bench_hp_malloc
```

Ce script peut être lancer via l'utilitaire **Q22.sh** à la racine du rendu

ATTENTION : Après étude du benchmark, il y a une coquille dans le TD.

Pour cette partie et la partie 5, la mémoire alloué en réserve (dans 'MEM_SIZE') est insuffisante pour accueilir la mémoire utilisateur ET les en-têtes.

(les benchmark peuvent alloué jusqu'à 14 000Ko, et 'MEM_SIZE' est égal à 14 000Ko, il n'y a donc pas la place pour les en-têtes).

J'ai donc modifié la valeur de 'MEM_SIZE' pour que la réserve soit suffisement large pour accueillir les en-têtes.

```
#include "hp_allocator.h"

#define MEM_SIZE (14UL*1024UL * 1024UL * 2)
#define ALIGNEMENT(type) (offsetof(struct { char c; type t; }, t))
#define TAILLE_EN_TETE (offsetof(struct { struct bloc b; union align u; }, u))
```

(ajout d'un $*2$ ligne 3)

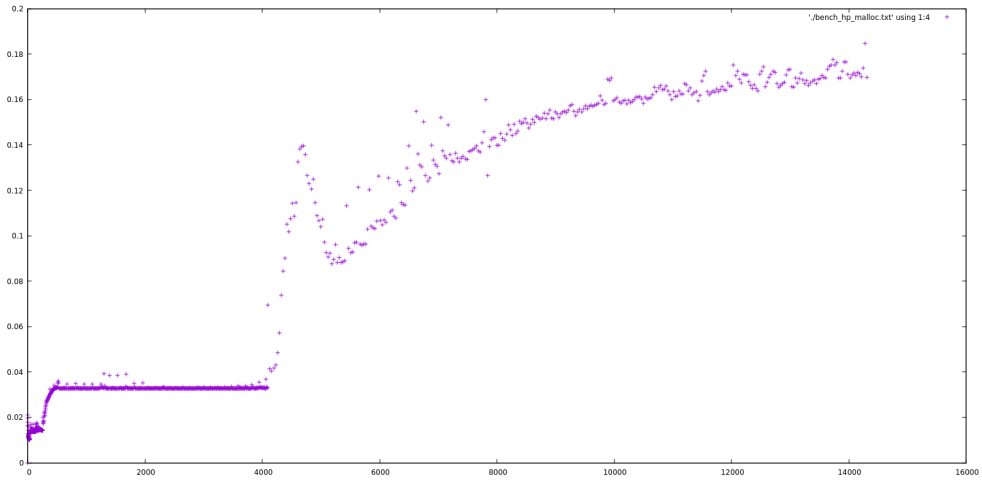

FIGURE 2 – La courbe obtenu est similaire à 2

L'utilisation du cache est optimal.

5 Wrapper de la fonction malloc

Q.23 Pour wrapper les fonctions, le code suivant a été ajouté

Au fichier 'hp_allocator/hp_allocator_malloc_free/SRC/hp_allocator.c'

```
/** malloc wrapper */
void * malloc(size_t taille) {
    return __hp_malloc(taille);
}
void free(void * ptr) {
    return __hp_free(ptr);
}
```

Au fichier 'hp_allocator/hp_allocator_malloc_free/SRC/hp_allocator.h'

```
/* wrapper malloc */
void * malloc(size_t taille);
void free(void* ptr);
```

Pour executer le benchmark 'svalat_bench' sans modifier le code, il suffit sous Linux de modifier l'adresse des fonctions de bibliothèques dynamique (5).

Pour cela, se placer avec un CWD à la racine du rendu, puis entrer :

Listing 2 – svalat_bench_lib avec la bibliothèque dynamique

```
# compile la biblioth que
make -C hp_allocator/hp_allocator_malloc_free/SRC/ shared

# compile le benchmark et execution avec la biblioth que
make -C svalat_bench
LD_PRELOAD=hp_allocator/hp_allocator_malloc_free/SRC/libhp_allocator.so ./svalat_bench/bench_malloc
LD_PRELOAD=hp_allocator/hp_allocator_malloc_free/SRC/libhp_allocator.so ./svalat_bench/bench_huge
```

Ce script peut être lancer via l'utilitaire **Q23.sh** à la racine du rendu

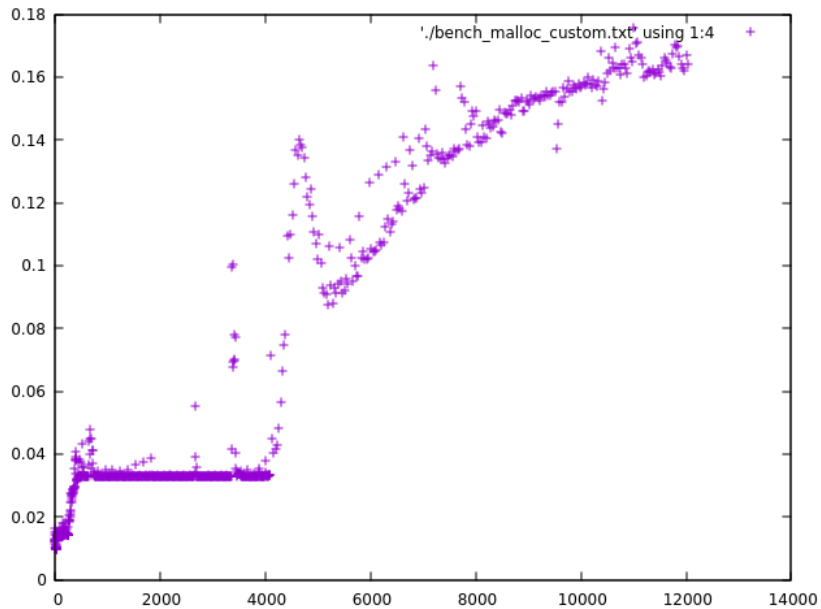

FIGURE 3 – Résultat du benchmark 'bench_malloc' avec mon implémentation de 'malloc'

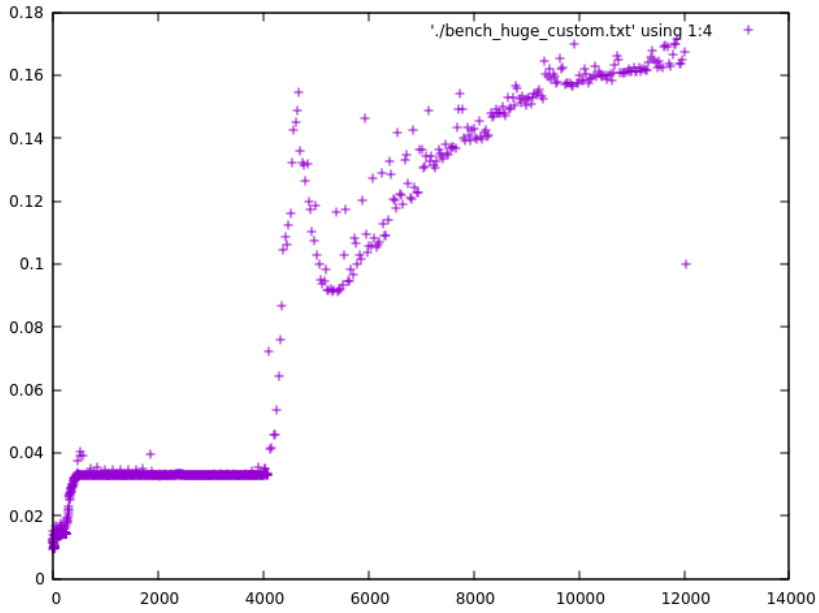

FIGURE 4 – Résultat du benchmark 'bench_huge' avec mon implémentation de 'malloc'

On remarque que l'utilisation du cache optimal dans les 2 cas.

En effet, notre allocateur repose sur les 'huge pages', qui se mappent plus efficacement sur le cache.

6 Références

[1] Sébastien Valat. Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance. Performance et fiabilité [cs.PF]. University de Versaille Saint-Quentin en Yvelines, 2014. Français. <tel-01253537>
https://hal.archives-ouvertes.fr/tel-01253537/document

[2] What Every Programmer Should Know About Memory - Ulrich Drepper
https://people.freebsd.org/ lstewart/articles/cpumemory.pdf

[3] Modern Operating Systems - Andrew S. Tanenbaum, Herbert Bos
https://github.com/concerttttt/books/

[4] 3.2. Cache Lines and Cache Size - Rogue Wave Software
https://docs.roguewave.com/threadspotter/

[5] Stack-overflow
https://stackoverflow.com/questions/262439/create-a-wrapper-function-for-malloc-and-free-in-c

[6] Table de hachage - Wikipédia
https://fr.wikipedia.org/wiki/Table_de_hachage