Rapport Final

Projet de Système d'Exploitation

Manon Barbier, Nour Grati, Guillaume Le Breton,
Youssef Naïm et Hugo Vikström

Responsables : Adèle Villiermet, Brice Goglin

Table des matières

1 Introduction

Ce rapport présente notre travail concernant le projet de Système d'Exploitation. Ce projet s'inscrit dans le cadre de notre deuxième année à l'ENSEIRB-MATMECA et a pour but de construire une bibliothèque de threads en espace utilisateur. Ce document présente les choix que nous avons été amené à faire, les problèmes rencontrés et les solutions envisagées pour notre projet.

2 Organisation de l'équipe

Nous avons dans un premier temps décidé ensemble comment implémenter la structure *thread*, nous nous sommes séparés en deux équipes. Les deux équipes s'étaient divisées le travail pour l'implémentation des fonctions des threads et leurs tests. Sur les dernières séances, les objectifs étaient de relever les performances de nos implémentations, le développement des mutex ainsi que la mise en place d'une préemption pseudo-coopérative.

3 Thread

3.1 Choix d'implémentation

Afin d'implémenter l'interface *thread.h*, nous avions le choix parmi plusieurs macros déjà implémentées pour la gestion de notre liste FIFO de thread dans le fichier *queue.h*. Nous avons choisi de travailler avec la structure de données de *smple queue* pour implémenter les queues dans notre projet. Au stade où nous sommes arrivés, nous n'avons pas eu besoin de rechercher un élément particulier dans une liste donc ce modèle nous convenait en ce qui concerne la complexité.

Nous avons 2 FIFO suivant ce modèle déclarées en tant que variables globales :

- **runq** : elle représente la liste des threads prêts à exécuter du code ;
- **overq** : elle représente la liste des threads ayant terminé le code qu'ils étaient chargés d'exécuter.

3.2 La structure thread

Notre structure thread contient les champs suivants :

```
thread_t id;
ucontext_t context;
ret retval;
struct queue waiting_thread;
int valgrind_stackid;
struct thread* next;
```

Notre structure regroupe un identifiant donné par un entier, un contexte, une valeur de retour de type void* et une FIFO regroupant les threads en attente sur ce thread.

Nous avions fait le choix dans un premier temps de déclarer une 3e FIFO dénommée *sleepq* représentant la liste des threads en attente d'un autre thread mais l'ajout du champ **waiting_thread** l'a rendu inutile. Ce choix nous a paru pertinent puisqu'il permettait le réveil des threads en attente du thread concerné sans avoir à les chercher dans une *sleep queue* contenant tous les threads en attente ce qui appuie notre utilisation du modèle *simple queue*.

Le dernier champ *valgrind_stackid* est utilisé pour la libération de la pile du contexte.

3.3 Les fonctions

Nous avons implémenté les fonctions suivantes :

3.3.1 thread_self et thread_create

En plus des deux queues citées plus haut, nous avons également une variable globale qui est un pointeur sur une structure thread que l'on nommera **current_thread**. C'est grâce à cette variable que nous saurons sur quel thread le code est en train d'être exécuté.

Ainsi, la fonction THREAD_SELF renvoie le champ *id* de **current_thread**.

Lorsque THREAD_CREATE est appelée, on vérifie dans un premier temps si le thread en cours de création correspond au thread du main avec la variable globale **firstThread**. Dans ce cas, on appelle une fonction spécifique au main qui initiialise le thread courant. C'est dans cette fonction qu'on appelle ON_EXIT afin de libérer toute la mémoire allouée pour nos threads à la fin de l'exécution du programme.

Dans le cas où il ne s'agit pas du premier thread, on initialise également ce thread dans un nouveau pointeur que l'on insère dans la FIFO **runq**. Le champ *id* est incrémenté à chaque création de thread et le champ *context* est initialisé via un appel à MAKECONTEXT qui crée un contexte avec la fonction et les arguments à utiliser pour l'exécution de ce contexte. Ces derniers sont fournis en arguments de la fonction THREAD_CREATE.

3.3.2 thread_yield

Le thread qui appelle la fonction THREAD_YIELD passe la main à un autre thread et est placé à la fin de la file des threads qui sont prêts. Pour ce faire, on vérifie dans un premier temps si le thread en question est le premier thread à avoir été appelé depuis le début du programme. Si c'est le cas, on fait appel à la fonction CREATE_MAIN_THREAD. Cette fonction sert à initialiser le thread main comme décrit plus haut.
On vérifie ensuite que la file des threads prêts n'est pas vide. Dans le cas où elle est vide, on sort de la fonction.
Une fois ces vérifications faites, on insère le thread courant à la fin de **runq** puis on sauvegarde le thread courant en le plaçant dans une nouvelle structure thread. Le thread courant devient la tête de **runq** donc on l'enlève. On appelle SWAPCONTEXT. Lors du retour de la fonction, on rétablit le thread courant.

3.3.3 thread_exit

Cette fonction est appelée pour mettre fin à l'exécution d'un thread et se charge de stocker la valeur de retour de la fonction exécutée par le thread dans le champ *retval* prévu à cet effet.

Nous avons essayé plusieurs approches pour choisir la méthode suivante. Étant donné qu'un appel à THREAD_JOIN - que nous verrons dans la partie suivante - sur un thread déjà terminé est possible, nous avons eu recours à une liste **overq** pour garder des références aux threads terminés au cas où on aurait besoin de leur valeur de retour. En premier lieu, notre fonction affecte la valeur prise en paramètre dans le champ *retval* du thread courant puis insère ce thread dans la liste **overq**. Ensuite, elle donne la main au thread se trouvant en tête de **runq** pour s'exécuter.

Il est possible aussi qu'il y ait d'autres threads en attente sur celui qui appelle THREAD_EXIT. Dans ce cas, il faudra les remettre dans la runq.

Cela se fait en parcourant l'attribut **waiting_thread** contenant les threads en attente sur celui-ci. Pour faciliter l'exécution de THREAD_JOIN, on place également la valeur de retour du thread dans chacun des champs *retval* des threads de cette liste.

3.3.4 thread_join

Un thread appelant THREAD_JOIN se met en attente sur le thread voulu et peut stocker la valeur de retour du thread attendu à une adresse spécifiée.

La solution adoptée pour l'implémentation de cette fonction est la suivante : elle commence par un parcours de la liste **overq** pour chercher le thread attendu. Une fois récupéré, elle affecte la valeur de retour disponible à l'adresse donné en argument. Dans le cas où le thread attendu n'est pas dans **overq**, on le recherche dans **runq** et on insère le thread courant dans la liste-attribut *waiting_thread* du thread attendu. La fonction donne ensuite la main au thread suivant de **runq** comme ce qui est implémenté dans la fonction THREAD_YIELD. Une fois revenu dans le contexte du thread appelant la fonction, on est sûr que le thread attendu a terminé puisque THREAD_EXIT a réinséré le thread dans la **runq** grâce à *waiting_thread* du thread attendu. On retrouvera également la valeur de retour du thread attendu dans le champ *retval* du thread courant comme expliqué plus haut. Il suffit donc d'affecter cette valeur à l'adresse spécifiée en argument.

4 Mutex

4.1 La structure thread_mutex_t

Nous avons fait le choix de développer nos mutex sous la structure suivante :

```
typedef struct thread_mutex {
    struct queue mutexq;
    int locker;
} thread_mutex_t;
```

Nous avons décidé que nos mutex avaient besoin d'une file chacun afin de gérer les attentes des threads lors d'une tentative d'entrée dans une section critique. Nous avons décidé d'utiliser le modèle **SIMPLEQ** puisque nous

n'effectuons pas de recherche dans la file. En ce qui concerne l'entier **locker**, il s'agit de l'*id* du thread occupant la section critique.

4.2 Les fonctions

La première fonction THREAD_MUTEX_INIT est chargée de l'initialisation du mutex. Elle initialise la file et l'identifiant **locker**. Pour ce qui est de la fonction THREAD_MUTEX_DESTROY, il s'agit ici simplement de supprimer tous les éléments de la file.

La fonction THREAD_MUTEX_LOCK est chargée de limiter l'entrée d'une section critique à un thread. Pour ce faire, si le lock n'est pas pris, le thread courant affecte son identifiant dans le champ **locker**. Dans le cas contraire, le thread est inséré dans la file d'attente du mutex et le contexte instauré est celui en tête de la *run queue*. Lorsque le contexte du thread appelant sera rétabli, on vérifie si le locker est prenable. Si ce n'est pas le cas, on revient à l'étape d'instauration d'un nouveau contexte.

Dans le cas du THREAD_MUTEX_UNLOCK, si la file d'attente du mutex n'est pas vide, on prend sa tête que l'on insère de nouveau dans la *run queue* et on affectue l'identifiant de ce thread dans le champ **locker**. Ainsi, lorsque son contexte sera instauré via le déroulement de la *run queue*, il pourra prendre le lock. Dans le cas où la file d'attente est vide, on réinitialise simplement le champ **locker** à -1.

5 Preemption pseudo-cooperative

Nous avons choisis d'implémenter une première version d'ordonnancement où la préemption est caché dans les fonctions de notre bibliothèque. Cette ordonnancement se fait grace à un Timer. A chaque fois qu'on fait un changement de contexte, nous réinitialisons le compteur à 0. Ensuite, a chaque fois que le thread fait un THREAD_SELF ou un THREAD_CREATE, on mesure le temps écoulé. S'il est plus grand que 100 microsecondes, on fait un THREAD_YIELD, on continue normalement sinon.

En moyenne, la version avec préemption est plus lente dû aux nombreux changements de contextes. Néanmoins, dans certains exemples, nous avons comme résultat que les threads ayant le moins de travail à exécuter se ter-

minent en premier, peu importe l'ordre dans lequel ils ont été lancé. Ceci vient du fait qu'ils ont plus souvent la main. Un exemple pour observer ce comportement est le fichier d'exemple **preemption_exemple.c** : sans préemption, le thread lancé en premier se termine avant ; avec préemption, le thread avec le temps d'éxécution le plus faible se termine en premier.

6 Bilan des tests et problèmes rencontrés

6.1 Les tests

Nous avons réalisé différents tests. Pour cela, nous avons utilisé la batterie de tests qui nous était proposée et nous avons également créé des tests supplémentaires. Certains de ces tests prennent un argument à la compilation. Nous avons donc implémenté un script qui permet de tracer les courbes qui représentent le temps que mettent les programmes à s'exécuter en fonction de la valeur qu'ils prennent en argument.

6.1.1 La bibliothèque de tests

Nous avons réalisé les tests proposés dans la bibliothèque de tests. Tous les tests passent et réalisent ce que l'on souhaite.

L'usage de valgrind nous a permis de confirmer qu'il n'y avait pas de fuite mémoire et que nous libérions correctement toutes les variables allouées.

Afin de mesurer les performances de nos implémentations, nous les avons comparé avec la biblothèque pthread en travaillant sur une machine munie de 8 processeurs. Nous avons donc comparé le temps que mettait le programme à s'exécuter en fonction du nombre de threads choisis.

Fibonacci Nous avons comparé les performances du test implémentant la fonction de Fibonacci en utilisant notre interface et en utilisant la bibliothèque pthread. On remarque qu'avec notre implémentation, notre calcule peut aller jusqu'à la valeur 20 en un temps raisonnable. A partir de 20 cela devient plus long (on calcule Fibonacci(20) en 4.371953 secondes). La figure 1 en annexe présente le temps d'exécution du test en fonction de la valeur prise en entrée selon que l'on utilise notre implémentation ou la bibliothèque

des pthreads.

On remarque que notre implémentation est plus rapide que celle utilisant la bibliothèque pthread ce qui est normal car nos coûts d'ordonnancement sont plus faibles.

Les tests qui créent une multitude threads Nous avons également testé nos programmes sur create-many, create-many-recursive et create-many-once. Les courbes associées à nos résultats et qui comparent notre imlémentation avec celle des pthreads sont respectivement les courbes 2, 3 et 4 en annexes. Dans les 3 cas on constate que notre implémentation est plus rapide que celle des pthreads.
Pour le test create-many, au delà de 200 threads le temps d'exécution n'est plus proportionnel au nombre de threads.
Pour create-many-once la valeur critique est à 225 threads.

Les tests sur les switchs La courbe représentant nos résultats pour le test switch many est représenté en figure 5 en annexes. On peut voir que notre implémentationest plus rapide que celle avec des pthreads et que la valeur critique se situe à environ 50 threads.

Les mutexs La courbe représentant le test sur les mutex est la figure 6 en annexe. On peut noter qu'à partir d'environ 70 threads, le temps d'exécution n'est plus proportionnel aux nombre de threads.

6.1.2 Les tests supplémentaires

Nous avons implémenté de nouveaux tests qui utilisent une multitude de threads.

Somme des éléments d'un tableau Le premier calcule la somme des éléments d'un tableau très grand. Le nombre de threads utilisé est fixé à la compilation et la taille du tableau est donnée en paramètre à l'exécution.

Tri fusion Le second est le tri fusion d'un tableau très grand. Le nombre de thread est fixé à la compilation et la taille du tableau est donnée en paramètre à l'exécution.

Multiplication de matrices Ce test calcule le carré d'une matrice dont la taille est fixée à la compilation et le nombre de threads utilisés est donné à l'exécution.

Nous n'avons pas réalisé de courbes pour ces tests car nous obtenions des résultats en dessous de la millisecondes même avec de très grands tableaux ou matrices.

6.2 Problèmes rencontrés

Les problèmes rencontrés dans la première phase du projet sont quasiement tous résolus au cours de la deuxième phase. Nous avons arrivé enfin avec notre version actuelle de la bibliothèque à executer tous les exemples sans fuites mémoire et dans un temps d'execution raisonnable.

Nous avons rencontré des problèmes au niveau de la récupèration par les threads des signaux générés par un Timer en élaborant la version de préemption coopértive. Nous ne sommes pas suffisemment investis pour résoudre ce problème à cause de la contrainte du temps.

Lors de la libération des threads à la fin de l'exécution du main, nous avons rencontré des problèmes de valgrind. Dans un premier temps, le problème était des fuites mémoires dûes à la libération des threads de la liste avant de passer à l'élément suivant de la liste. Mais une fois ce problème résolu, beaucoup d'erreurs sont apparus dans valgrind, et seulement lors de l'exécution du test 12-join, et nous n'avons pas réussi à résoudre cela. Néanmoins, nous avons résolu toutes les fuites mémoires, et la libération des ressources se passe correctement hormis ces "warning" de valgrind.

7 Conclusion

Au terme de ce projet, nous avons réussi à implémenter l'interface thread afin que tous les tests fonctionnent. Nous y avons ajouté une préemption

pseudo-copérative et des mutexs afin d'améliorer nos résultats. Avec plus de temps nous aurions pu nous pencher sur un autre modèle de préemption.

8 Annexes

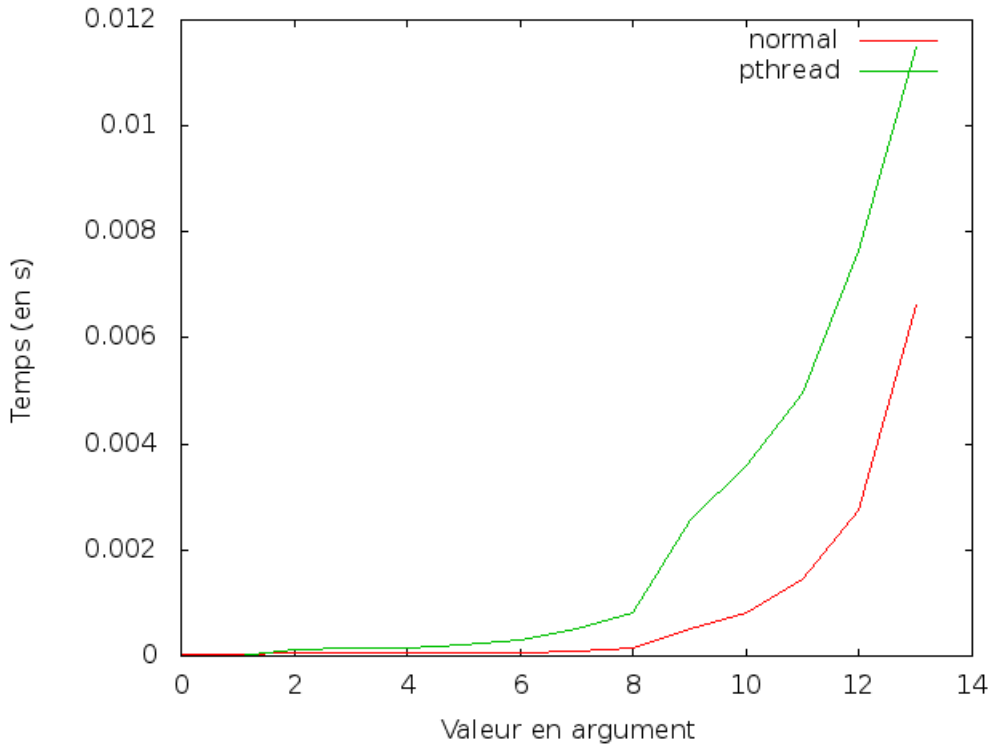

FIGURE 1 – Courbe représentant le temps d'exécution en fonction de la valeur prise en entrée pour le test de Fibonacci

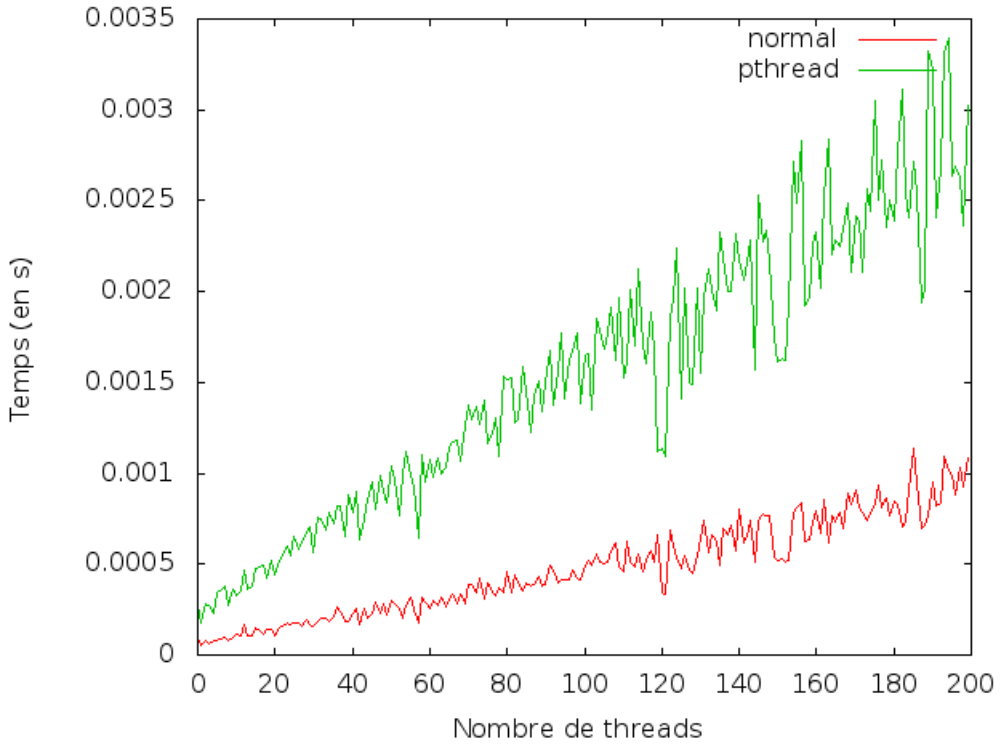

FIGURE 2 – Courbe représentant le temps d'exécution en fonction de la valeur prise en entrée pour le test create-many

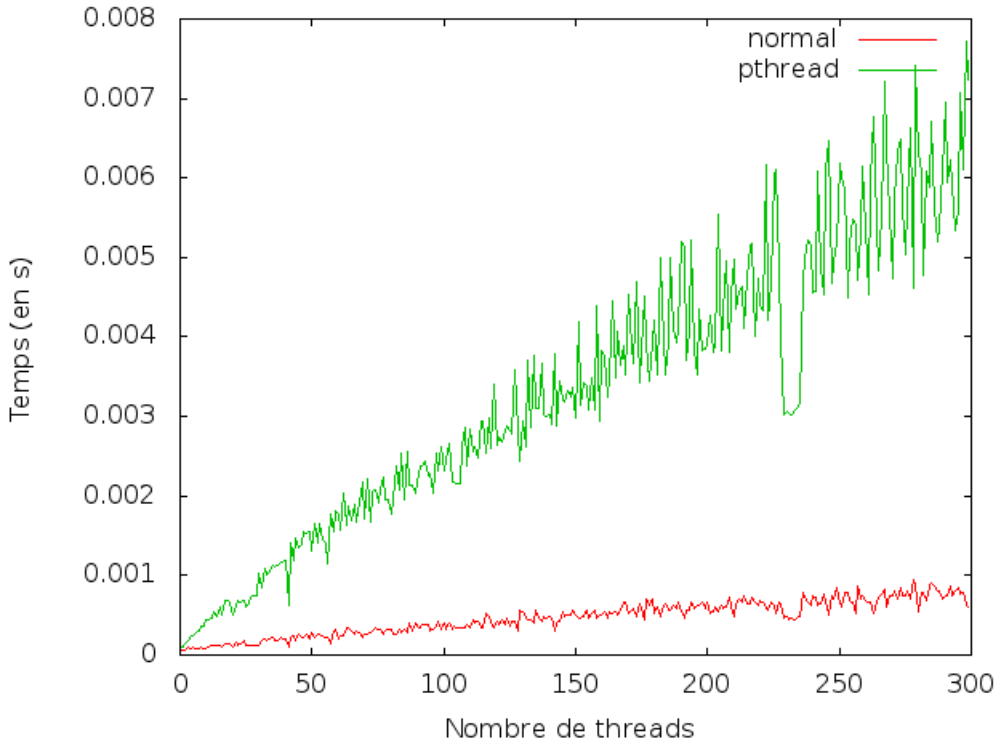

FIGURE 3 – Courbe représentant le temps d'exécution en fonction de la valeur prise en entrée pour le test create-many-recursive

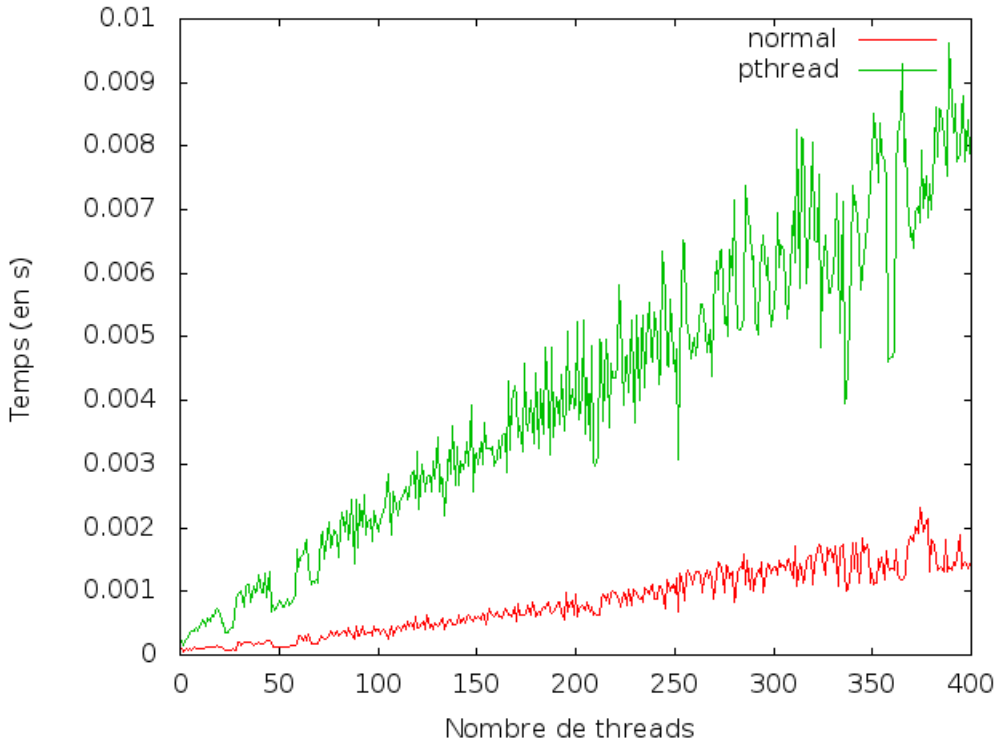

FIGURE 4 – Courbe représentant le temps d'exécution en fonction de la valeur prise en entrée pour le test create-many-once

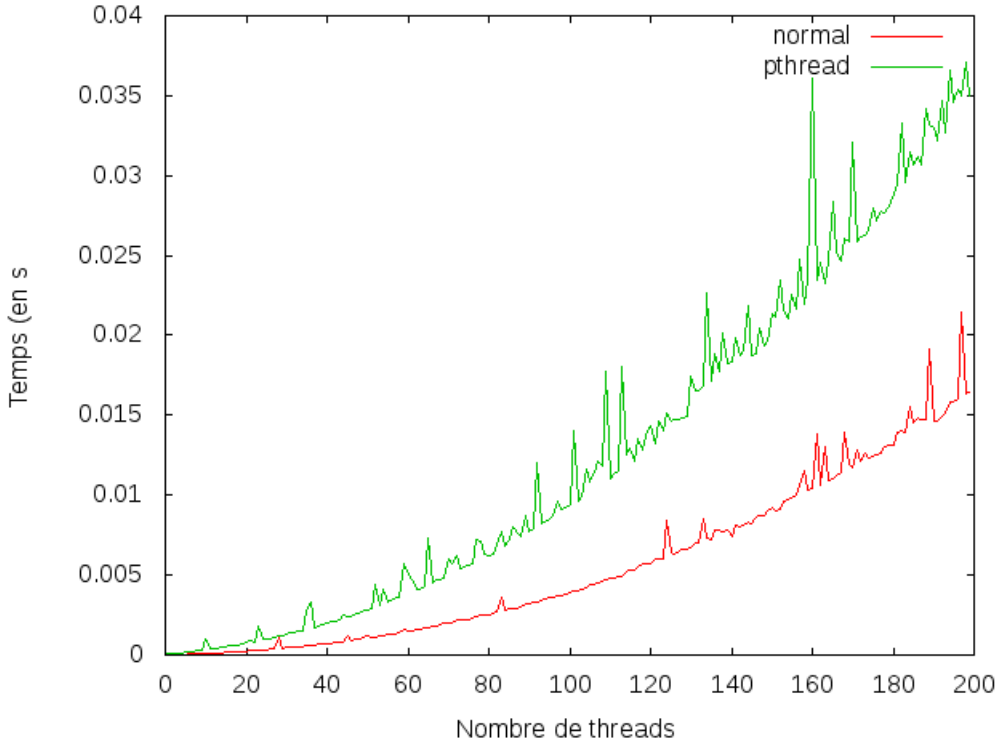

FIGURE 5 – Courbe représentant le temps d'exécution en fonction de la valeur prise en entrée pour le test switch-many

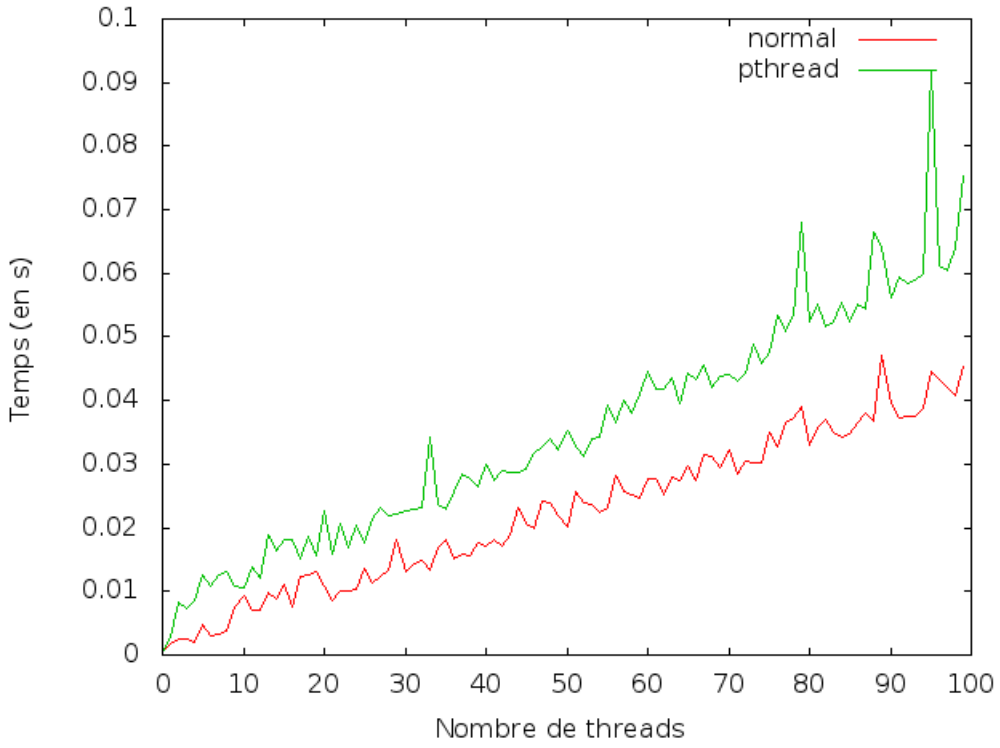

FIGURE 6 – Courbe représentant le temps d'exécution en fonction de la valeur prise en entrée pour le test mutex