Rapport final

Projet de Système d'exploitation

BARBARIN Paul COMITI Santinu DUCAMP Simon
MORENO CARPIO Kenzo MOREL Mathieu

S8 — 2023-2024

Table des matières

1 Introduction **1**

2 Fonctionnement général **1**
2.1 Structure et politique d'ordonnancement des threads 1
2.2 Pseudo-algorithmes de chaque opération sur les threads 2
2.3 Gestion de la mémoire 6

3 Analyse des performances **6**
3.1 Complexité des algorithmes implémentés 7
3.2 Comparaisons empiriques avec *pthread* 8

4 Les mutex **11**
4.1 Fonctionnement de nos mutex 11
4.2 Analyse des performances des mutex 12

5 Les signaux **14**
5.1 Leur implémentation 14

5.2 Discussion et critique de l'approche 15

6 Détecter les débordements de pile **15**

1 Introduction

Dans ce rapport, vous trouverez un résumé de notre projet de système d'exploitation, consistant en la construction d'une bibliothèque de threads en espace utilisateur. Au sein de ce projet, nous avons par équipe de 5 été amenés à manipuler les concepts de système d'exploitation vus en cours tout le long de notre seconde année à l'ENSEIRB-Matmeca, durant les UE de programmation système et de systèmes d'exploitation. Nous articulerons notre rapport en plusieurs parties, en prenant soin de déduire une partie uniquement pour expliquer la complexité des algorithmes que nous avons implémentés. Nous passerons par ailleurs en revue les principaux sujets de développement, à savoir tout d'abord le fonctionnement général mais également les fonctionnalités plus spécifiques telles que l'implémentation de la gestion des mutex.

2 Fonctionnement général

Cette section détaille le fonctionnement général de notre bibliothèque de threads.

Il était demandé à ce que nos threads se basent sur la spécfication des *pthread* et implémentent une interface similaire, mais aient à contrario de ces derniers un fonctionnement en mode utilisateur.

Nos threads s'appuient sur la bibliothèque *ucontext*, qui permet de manipuler des contextes, c'est-à-dire des *snapshot* de la pile du programme à un moment donné. Il faut définir une politique d'ordonnancement de ces threads (FIFO, LIFO, par priorité, aléatoire, collaboratif ou non, etc) et une structure qui implémente cette dite politique et qui permet de passer d'un thread à un autre.

Nous nous aussi basés sur la spécfication des *pthread*s et avons munis de notre bibliothèque du même jeu de codes d'erreurs pour les mêmes cas.

2.1 Structure et politique d'ordonnancement des threads

Notre bibliothèque de threads suit une politique FIFO comme demandé dans le sujet, bien qu'un objectif secondaire proposait d'intégrer d'autres politiques. Afin de respecter cette politique FIFO, on emploie une liste doublement chaînée à deux bouts pour stocker les threads, qui fait office de file.

Nous avons écrit dans thread_queue.h une structure wrapper pour manipuler les opérations sur les listes de threads. L'implémentation de cette *thread queue* utilise de manière sous-jacente la bibliothèque TAILQ de *FreeBSD*. Notre module wrapper permet de transformer la simple file en file circulaire en reliant début et fin de file, et en créant une variable globale qui indique le thread actif. Les éléments de cette file circulaire sont des *threadblock*, qui stockent comme information :

— Contexte du thread (la "pile" sur laquelle va s'exécuter le thread)

— Statut du thread
— Valeur de retour du thread

2.2 Pseudo-algorithmes de chaque opération sur les threads

Cette sous-section décrit les pseudo-algorithmes de chacune de nos opérations standards sur les threads. Il faut garder à l'esprit que ces pseudo-algorithmes ne respectent ni la syntaxe, ni certaines sémantiques C (comme le retour d'argument par effet de bord).

```
thread_create(fonction, argument){

    threadblock = initialise_threadblock(fonction, argument);
    threadblock.contexte = alloue_contexte();

    retourne threadblock;
}
```

Cette fonction est très simple en apparence, mais elle pose une difficulté conceptuelle : pour tout thread créé, on alloue dynamiquement sa pile, et on alloue dynamiquement son *threadblock*, or le thread qui a lancé la fonction *main*, que l'on appellera par la suite **main thread**, a un statut particulier n'a pas été initialisé par cette fonction. On détaille dans la sous-section 2.3 dédiée à la gestion de mémoire ce statut particulier du *main thead*.

Erreurs :
— EAGAIN si le programme ne dispose pas assez de mémoire pour créer le thread

```
thread_yield(){
    si file_est_vide() :
        retourne;

    actuel = thread_self();
    prochain = defiler();

    enfiler(actuel);

    actuel.statut = READY;
    prochain.statut = RUNNING;

    swapcontext(actuel, prochain);
}
```

C'est ici qu'on observe la première occurence d'un changement de contexte. C'est d'ailleurs une des deux seules responsabilité qu'à la fonction *thread yield* : passer

du contexte actuel au contexte du prochain thread, et faire tourner la file circulaire.

Erreurs :

sched_yield ne définit pas d'erreurs dans sa documentation. Cependant on utilise les contextes dans notre implémentation, qui peuvent échouer. En cas d'erreurs de *swapcontext*, on retourne -1.

```
auto_exit(thread_func){
    thread_func();
    thread_exit();
}

@auto_exit
thread_exit(valeur_retour){

    actuel = thread_self();
    actuel.valeur_retour = valeur_retour;
    actuel.statut = TERMINATED;

    si est_join(actuel):
        enfiler(actuel.join_par);

    si file_est_vide():
        si actuel == main_thread:
            exit(SUCCESS);
        sinon :
            setcontext(main_thread);
    sinon:
        prochain = defiler();
        si actuel == main_thread:
            swapcontext(actuel, prochain);
        sinon :
            setcontext(prochain);

}
```

La fonction *thread exit* est appelée quand un thread a fini de s'exécuter. Soit via un appel manuel, soit automatiquement à la fin de la fonction de thread grâce à notre wrapper *auto exit*. Dans cette fonction, le thread terminé change son statut et met à jour sa valeur de retour. Là entre en jeu une décision en tableau double-entrées pour le thread :

	Je suis main	Je ne suis pas main
File vide	Je quitte le programme	Je rend la main à main pour qu'il quitte
File non vide	Je **swap** vers le prochain pour garantir de revenir à main	Je **set** vers le prochain pour garantir de ne pas revenir sur moi

TABLE 1 – Exemple de tableau avec saut de ligne, lignes entre les cellules et du padding

Cette décision permet de garantir que, dans le cas où le programme s'est déroulé sans erreurs, la main revienne bien à la *main stack* en fin de programme. Voire pourquoi dans la section 2.3.

Une autre responsabilité de *exit* est la remise en file du thread qui le join, dont on parle plus bas dans *thread join*.

Pas de code d'erreurs, *exit* est une fonction void.

```
thread_join(cible, valeur_retour){

    actuel = thread_self();
    cible.join_par(actuel);

    si statut(actuel) != TERMINATED :
        attente_passive(actuel);

    verification_deadlock_join();

    valeur_retour = cible.valeur_retour;

    si cible != main_thread :
        delete cible.contexte
        delete cible
}
```

La fonction *thread join* est plus complexe parce qu'elle cumule beaucoup de responsabilités. On remarque premièrement que c'est elle qui a la responsabilité de détruire le thread qu'elle join, et pas *thread exit* comme on pourrait en avoir l'intuition. Cela est dû au fait que, si un thread essaie de se détruire pendant *thread exit*, alors il va "scier la branche sur laquelle il est assis" et ne pourra jamais repasser la main à un autre thread. Et s'il décide de passer la main en premier, il ne pourra pas se libérer. C'est pourquoi la libération est faite dans *join* (on rappelle qu'un thread quitté par *thread exit* doit obligatoirement être *join* ou *detach* par l'utilisateur, et inversement, c'est décrit dans l'interface des *pthread*).

La seconde responsabilité intéressante et la vérification de deadlocks de join. Elle procède selon ce pseudo-algorithme :

```
verification_deadlock_join(){
    si cible.statut == WAITING :
        next = cible.join_par;
        tant que next != null :
            si next.join_par == cible : retourne ERROR;
            next = next.join_par
        attente_passive(); // SUCCESS
}
```

Concrètement, si on remarque que le thread cible du *join* est en attente, c'est que lui aussi essaie de *join* un autre thread, et que possiblement un cycle de join (*deadlock*) s'est mis en place. On remonte donc la chaîne de *join*, et si on retombe sur le thread cible, alors il y a *deadlock*.

Finalement, la dernière responsabilité de *join* est la mise en attente passive du thread actif. Une mise en attente active correspondrait à *yield* en boucle tant que le thread cible n'a pas quitté. L'attente passive est plus fine : elle retire le thread qui *join* de la file des threads à exécuter, en espérant que le thread cible *exit* et le renfile plus tard (ce qui est défini dans le contrat des threads : si un thread en *join* un autre, cet autre doit *exit*). Le code de l'attente passive ressemble donc beaucoup au code de *yield*, la remise en file en moins :

```
passive_wait(){
    si file_est_vide() :
        retourne;

    actuel = thread_self();
    prochain = defiler();
    // Pas de thread enfile !

    actuel.statut = READY;
    prochain.statut = RUNNING;

    swapcontext(actuel, prochain);
}
```

Erreurs :

- EDEADLK si la fonction détecte un deadlock
- EINVAL si la fonction essaie de join un thread non-joignable (dans notre cas, un thread déjà join)

2.3 Gestion de la mémoire

Le premier élément à prendre en compte dans la gestion de la mémoire a à voir avec le coeur du projet : les contextes. Chaque contexte définit en effet une pile d'exécution pour le thread, mais ces contextes doivent être allouées dynamiquement sur tas. Ils sont alloués sur *create* et libérés sur *join*. Une exception est la pile du main qui doit rester intacte pour pouvoir exécuter la fin de programme, et qui n'est d'ailleurs pas allouée sur tas puisque c'est la **pile du programme entier**. Pour permettre à Valgrind de suivre les erreurs de contexte, on munit chaque thread d'un id de stack Valgrind.

L'autre élément que l'on alloue dynamiquement dans notre programme est le *threadblock* que l'on souhaite manipuler comme élément de liste chaînée. Ce threadblock est aussi libéré sur *join*. Dans le cas particulier du main, on souhaite :

- Lui allouer dynamiquement un threadblock
- Faire cette allocation dynamique automatiquement en début de programme (on ne *create* jamais le main)
- Ne pas libérer ce threadblock sur *join*, sinon on ne pourrait pas réaccéder à son contexte pour la fin de programme
- En conséquence, libérer automatiquement son threadblock en fin de programme (sans libérer sa pile qui n'est pas sur le tas, on le rappelle).

C'est là qu'entrent en jeu les mécanismes de *constructor* et *destructor* en C. Ce sont des attributs de **compilateur** (GCC) qui indiquent à ce dernier d'exécuter certaines fonctions dans des périodes de vie particulières du programme. Le constructeur est appelé avant la fonction *main*, et le destructeur après avoir quitté *main* ou avoir appelé *exit*. Ces outils sont les outils standards qui permettent l'initialisation statique et la libération automatique de ressources dans une bibliothèque dynamique C.

On utilise de notre côté le constructeur pour allouer le threadblock du main, et charger son contexte dans ce threadblock, et le destructeur pour libérer ce threadblock. Il est pertinent de noter qu'à la fois le constructeur et le destructeur sont exécutés sur la pile principale du programme, donc la pile du *main thread*. Enfin, dans le cas où le dernier thread à s'exécuter n'est pas le main, alors son contexte sera aussi libéré dans le destructeur.

3 Analyse des performances

Dans cette section, nous mesurerons et interprèteront les performances de notre bibliothèque de threads, autour de deux grands axes :

- Les complexités théoriques en temps de nos algorithmes, en fonction du nombre de threads en file ou autre variable.
- Les temps d'exécution empiriques de nos algorithmes, en comparaison avec la bibliothèque *pthread*.

3.1 Complexité des algorithmes implémentés

Les complexités asymptotiques de nos fonctions sur les threads dépendent en grande partie de la complexité asymptotique des primitives sur les files de threads, puisque ces fonctions reposent sur ces primitives :

- thread_self : tqueue_get_current
- thread_create : tqueue_push (enfiler)
- thread_yield : tqueue_get_current, tqueue_pull (défiler), tqueue_set_current, tqueue_push
- thread_join : tqueue_get_current, tqueue_pull (via passive wait), tqueue_get_main
- thread_exit : tqueue_get_main, tqueue_push, tqueue_is_empty, tqueue_pull, tqueue_set_current

Or, les complexités asymptotiques en temps de nos primitives sont, en fonction du nombre de threads en pile :

- tqueue_get_$*$: $O(1)$, grâce à des variables globales
- tqueue_set_$*$: $O(1)$, grâce à des variables globales
- tqueue_push : $O(1)$. Notre implémentation utilisant une liste doublement chaînée, l'insertion se fait en tête par accès direct à un pointeur (via TAILQ_INSERT_HEAD).
- tqueue_pull : $O(1)$. Notre implémentation utilisant une liste doublement chaînée, la suppression se fait en queue par accès direct à un pointeur (via TAILQ_LAST puis TAILQ_REMOVE). Le chaînage double garantit la suppression en temps constant : on accède au dernier élément via un pointeur, puis on accède à l'élément précédant via le champ *prev* qui permet de définir ce précédant en prochain élément de queue de file. Cependant ce n'est pas obligatoire, il existe des implémentations de file via une liste simplement chaînée qui permettent à la fois d'enfiler et de défiler en temps constant.

Ainsi, nos opérations sur les threads héritent de ces complexités de primitives (en fonction du nombre de threads en file) :

- thread_self : $O(1)$
- thread_create : $O(1)$
- thread_yield : $O(1)$
- thread_join : $O(1)$
- thread_exit : $O(1)$

Et on devrait s'attendre à retrouver ces complexités asymptotiques dans la section 3.2

Cependant, certaines nuances sont à apporter. Dans le cas d'une détection de *join deadlock* par exemple, notre fonction remonte le chaînage formé par le champ *joined_by*, et est donc en $O(n)$ avec n le nombre de threads présents dans la chaîne de join. Cependant ce surcoût est négligeable, les *join deadlock* restant une situation extrême, qui plus est un *join deadlock* avec un grand nombre de threads chaînés.

Aussi, un poids sur le coût en temps de nos algorithmes peut être apporté non pas via une complexité asymptotique, mais un coût en cycles par des opérations lourdes. C'est le cas par exemple des appels aux méthodes de contextes. Un exemple de surcoût évident est l'utilisation d'une attente active plutôt qu'une attente passive, c'est-à-dire un thread qui ferait un *yield* dès qu'on tombe sur lui. Du point de vue exécution du programme, il est équivalent que le thread en attente *yield* ou ne prenne pas la main (les attentes actives et passives mènent donc à la même exécution), cependant la différence de coût en temps est très élevée, car l'appel à *swapcontext* fait dans *yield* est très lourd.

Enfin, d'autres mécanismes liés à la compilation peuvent influer sur les performances de nos threads : la compilation avec les options `-OX` qui permet au compilateur d'optimiser certaines opérations, la compilation avec `-g` qui permet à *gdb* de faire une exécution instruction par instruction du programme mais diminue les performances, la compilation avec `-fPIC`, nécessaire pour rendre le code positionnellement indépendant et donc construire une bibliothèque dynamique, mais pouvant impacter négativement les performances.

3.2 Comparaisons empiriques avec *pthread*

L'analyse à priori des complexités de nos algorithmes n'est pas suffisante pour mesurer les performances en temps de notre bibliothèque de threads. Nous proposons dans cette section une analyse empirique sur le jeu de tests qui nous a été fourni, en faisant varier certains paramètres et en comparant avec les performances de la bibliothèque *pthread*.

Les temps d'exécution sont tous mesurés en microsecondes μs

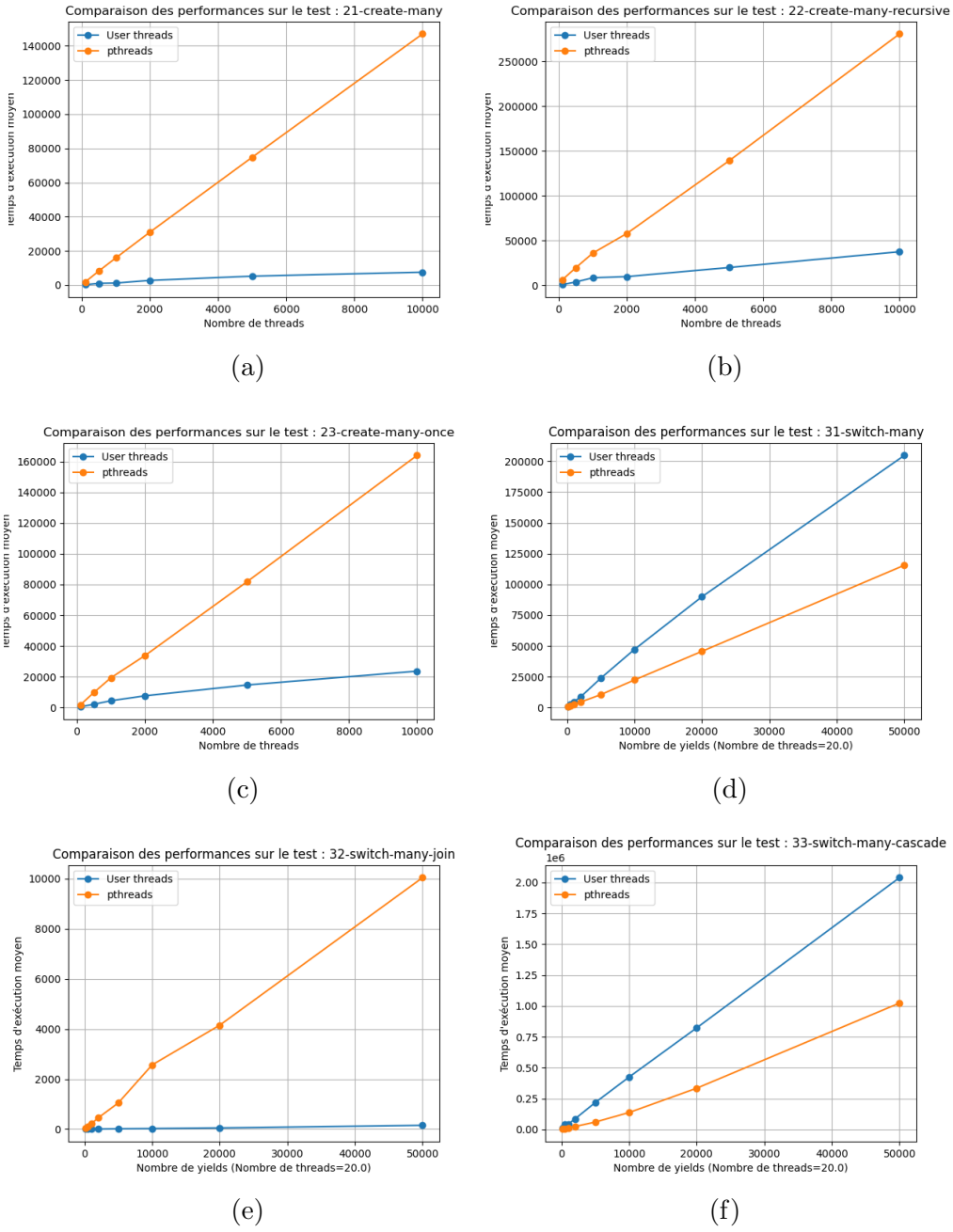


FIGURE 1 – Comparaisons des performances entre nos threads et les pthreads sur les tests des opérations de base sur les threads

Le permier élément que l'on remarque est la linéarité de toutes les courbes, que ce soit en fonction notre bibliothèque ou pthread, en fonction du nombre de threads pour les 3 premiers tests ou du nombre de yields pour les suivants.

Cela vient corroborer notre analyse des complexités : le temps d'exécution, donc

le nombre d'instructions, est proportionnel au nombre de threads. Cela est permis car nos opérations sur les threads ont une complexité constante, donc exécutent un nombre constant d'instructions par appel. Supposons que l'opération *create* ou *yield* était linéaire ($O(n)$) en fonction du nombre de threads en file ; alors l'augmentation du nombre de threads en paramètre aurait fait augmenter le nombre de threads à tester, mais aussi le nombre de threads en file, ce qui aurait conduit à une courbe à l'allure quadratique.

Le deuxième élément à souligner est que notre bibliothèque est plus performante sur les tests de création de thread :

- Sur le test 21, pthread montre un coefficient directeur de 15 contre < 0.5 pour notre bibliothèque
- Sur le test 22, pthread montre un coefficient directeur de 27.8 contre 4.2 pour notre bibliothèque
- Sur le test 23, pthread montre un coefficient directeur de 16.6 contre 2.5 pour notre bibliothèque

C'est dû au fait que la création de thread dans la bibliothèque pthread requiert des opérations plus lourdes de mise en place, dont notamment des appels systèmes relativement coûteux comme `clone3` (appel détecté avec *strace*).

En revanche, on observe que notre bibliothèque est moins performante sur les *yield* sans *join* (tests 31 et 33).

- Sur le test 31, pthread montre un coefficient directeur de 2.26 contre 4.2 pour notre bibliothèque
- Sur le test 33, pthread montre un coefficient directeur de 41.6 contre 20 pour notre bibliothèque

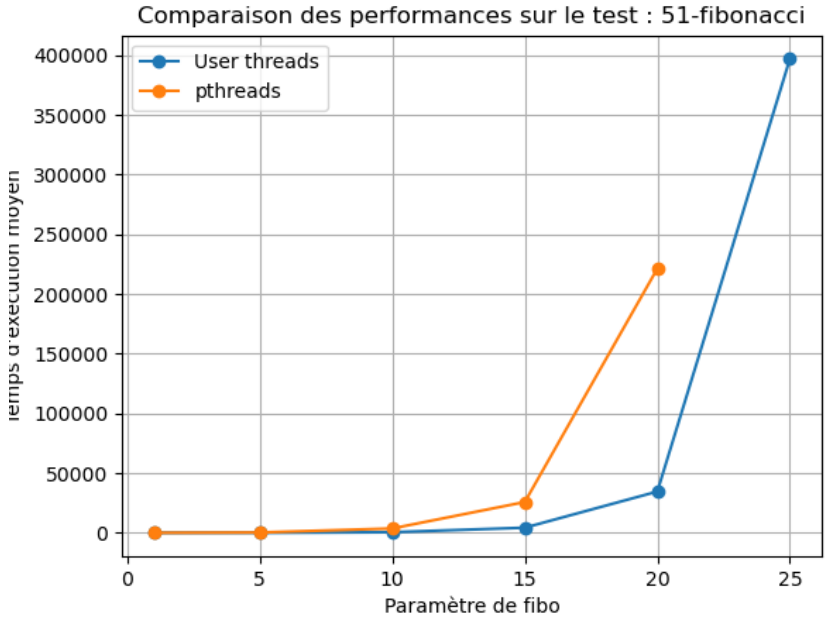
Notre hypothèse est que l'utilisation de *swapcontext* (qui fait aussi un appel système) est beaucoup plus lourde que l'appel système `sched_yield` fait par *pthread*. Nous n'excluons pas non plus la possibilité d'un mécanisme intelligent sur `sched_yield` fait par le système qui permet de réduire son coût en temps.

De manière surpenante, notre bibliothèque est plus performante sur les *yield* avec *join* :

Sur le test 32, pthread montre un coefficient directeur de 0.2 contre un coefficient directeur négligeable pour notre bibliothèque.

Nous supposons que *pthread* utilise des mécanismes sous-jacents lors du *join* qui le rend particulièrement lent (mais toujours en temps constant). En particulier, peut-être que la gestion de threads détachés faits par pthread, et qui n'est pas prise en charge par notre bibliothèque peut causer ce surcoût.

Le test de fibonacci servait à mesurer la performance en temps de notre bibliothèque sur un problème informatique concret : calculer la suite de Fibonacci.



On observe que la courbe est exponentielle en fonction de l'entrée de fibo, car nous calculons la suite de fibonacci récursivement, avec un algorithme connu pour avoir une complexité temporelle en $O(2^n)$ (n le paramètre passé à fibo).

Secondairement, on voit que notre bibliothèque est plus performante que *pthread* pour ce problème, notamment grâce à nos meilleurs performances sur le *create* et le *join*. Il aurait été pertinent de rajouter un graphique avec une échelle logarithmique pour pouvoir comparer l'ordre de grandeur de nos performances face à *pthread*, nous ne pouvons pas statuer sur ce point dans l'état actuel.

4 Les mutex

Un des premiers objectifs que nous avons réalisé est l'implémentation de *mutex* conformes à l'interface des *mutex* de *pthread*. Le jeu de tests allant de 61 à 64 a permis de vérifier le bon fonctionnement de ces mutex.

4.1 Fonctionnement de nos mutex

Nos *mutex* se basent sur un système de *ownership*. Lorsqu'un mutex vide est verrouillé par un *thread*, celui-ci en devient le propriétaire, et le mutex garde en mémoire cette information. Les prochains *threads* qui essaieront de le vérouiller se verront mis en attente passive (retirés de la liste des threads en attente) et gardés en mémoire comme bloqués par le mutex.

Lorsque le mutex est libéré (par son propriétaire), le mutex récupère le prochain

thread de la liste des bloqués et l'enlève de cette file ; si c'est `NULL`, il met son propre propriétaire à `NULL` (il est donc déverouillé), sinon il met ce prochain thread en propriétaire.

La fonction thread_mutex_init sert à initialiser la structure de mutex et mettre son état à "initialisé". Puisqu'aucune allocation dynamique n'est faite, thread_mutex_destroy sert uniquement à détecter une éventuelle erreur de mutex non déverouillé (conforme à la spécfication des mutex pthread) et à remettre le mutex en état non-initialisé.

De la même manière que pour les opérations standards sur les threads, nous avons fait attention à rester le plus conforme possible à la spécfication des *mutex pthread* quant aux valeurs de retour d'erreur :

- `EINVAL` si le mutex est verrouillé sans être initialisé
- `EDEADLK` si le mutex est re-verrouillé par le même thread
- `EPERM` si un thread non propriétaire tente de déverouiller le mutex
- `EBUSY` si un thread essaie de détruire un mutex encore vérouillé

4.2 Analyse des performances des mutex

Nous utilisons le jeu de tests fourni (tests 61-64) afin de comparer nos performances sur les opérations de mutex face à pthread.

Les temps d'exécution sont tous mesurés en microsecondes μs

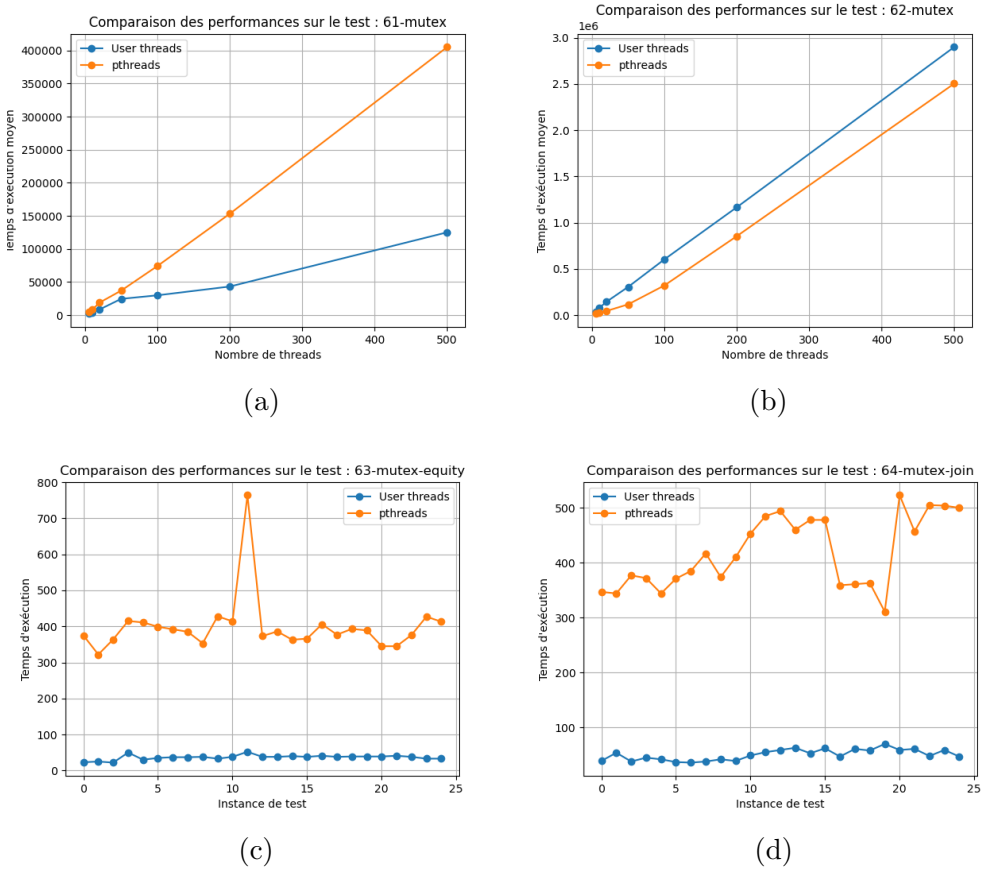


FIGURE 2 – Comparaisons des performances entre nos threads et les pthreads sur les tests des opérations de mutex

On observe que nos mutex sont plus performants que les mutex pthread sur le test *61-mutex* mais moins performants sur le test *62-mutex*.

- Sur le test 61, pthread montre un coefficient directeur de 800 contre 300 pour notre bibliothèque
- Sur le test 62, pthread montre un coefficient directeur de 5500 contre 5500 pour notre bibliothèque

Le test *61-mutex* ne crée qu'un seul mutex alors que le test *62-mutex* en crée plusieurs. Nous n'arrivons pas à expliquer avec certitude pourquoi nos mutex passent de plus performants à moins performants entre ces deux tests. Une hypothèse serait la façon dont pthread gère ses mutex : peut-être qu'il utilise des effets de cache ou de "vectorisation" des mutex, que ces derniers sont stockés dans une structure de donnée globale qui permet de gagner du temps quand on manipule plusieurs mutex simultanément. On note cependant que nos performances pour le test 62 sont dans le même ordre de grandeur que pthread.

Sur les tests d'équité et de join, on ne teste pas par proportionnalité à une valeur

mais par juste par instance de test. On observe alors que notre code est signicativement plus rapide que le code avec pthread (ordre de grandeur différent). On peut expliquer cela par le surcoût des opérations *create* et *join* dans pthread.

Dans notre implémentation précédente, sur thread_mutex_unlock notre mutex enlevait tous les threads de la liste et mettait sa file de bloqués à `NULL`. Cette implémentation était particulièrement inefficace en temps puisque le prochain thread qui allait arriver sur le mutex allait **de toute manière verrouiller le mutex**, et que les threads suivants allaient de même **se rebloquer sur le même mutex**. On se retrouvait alors alors avec une opération de déverouillage en $O(n)$ au lieu de $O(1)$ en fonction du nombre de thread bloqués. Or les tests de performances 61 et 62 font un verrouillage/déverouillage par thread : alors que l'allure actuelle de notre courbe est linéaire en fonction du nombre de threads, la précédente était quadratique et bien en dessous des performances de pthread.

5 Les signaux

5.1 Leur implémentation

L'implémentation de la gestion des signaux pour les threads est assez proche de celle des processus. Deux signaux sont disponible : `SIGKILL` et `SIGUSR`.
Chaque thread, lorsqu'il est créé, est initialisé avec un tableau de pointeurs vers ses gestionnaires de signaux.
`tsignal(int sig, void * (*func)())` personnalise le gestionnaire de signaux pour le thread courant. Nous avons prit soin d'empêcher l'utilsateur de personnaliser un signal `SIGKILL`, comme pour les signaux entre processus dans le standard posix.
Lors de la réception d'un `SIGKILL`, `void * handle_tsigkill()` appelle directement `thread_exit(NULL)`
`tsignal_send(thread_t * th, int sig)` permet d'envoyer les signaux, elle change le bit approprié du bitmask stocké dans le threadblock
Enfin, `tsignal_handling_header()` s'execute à chaque reprise en main par le thread, vérifie le masque de bits, et appelle les gestionnaires appropriés.
Pour implémenter ce comportement dans un monde parfait, il existerait une fonction `swapcontext` prenant en argument une fonction d'en-tête et l'executant juste après le changement de contexte. Malheuresement, une telle fontion n'existe pas et nous avons du faire autrement.
Le `makecontext` initiale lors de la création du thread nous permet de glisser un appel à cette fonction, en l'englobant dans une fonction `auto-exit` avec la fonction d'execution du thread.
Par la suite, C'est le thread lui-même qui place un appel à l'en-tête juste après un `swapcontext`, de sorte qu'il reprendra son execution par la fonction d'en-tête.

Ainsi la fonction se comporte comme une "fonction d'en-tête", sans vraiment vraiment en être une.

5.2 Discussion et critique de l'approche

L'implémentation de `SIGKILL` pose un problème : puisque tout les threads doivent être join, si `t1` est tué par `t3` avant d'avoir join `t2`, la mémoire allouée à `t2` ne sera jamais libérée. Pour résoudre ce problème, nous avions pensé à plusieurs solutions

- **le main adopte les threads non join.** Problème :il n'y aurait plus d'erreur lorsque l'utilisateur oublie de join un thread.
- **Le main adopte les enfants des threads tués :** Cette approche se rapproche de la manière dont sont géré les processus dans le système, lorsque `init` adopte les processus orphelins. On suppose ici que le main thread a un statut particulier et ne peut être tué comme les autres threads. De plus, on suppose également que les threads parents sont en général ceux qui join leurs enfants. Ce n'est pas forcément le cas pour les threads. De plus, en pratique, leur implémentation impacterait certainement les performance (lors d'un join, le main thread devrait supprimer le thread joined de son registre des thread à adopter)

Finalement, nous avons conclus que la gestion des join et des signaux ensemble était la responsabilité de l'utilisateur.
Autre point : dans notre code, le main thread a un statut particulier, on l'utilise via le destructeur pour libérér la queue, et c'est à lui qu'on rend la main dans certains cas lors d'un `thread_exit`. Nous avions donc installé une protection sur le main pour éviter l'envois de signaux avec en tête l'idée de poursuivre le développement d'une des deux solutions mentionnées plus haut, avant de réaliser notre erreur. Nous avons préféré garder cette version, bien qu'imparfaite, que d'essayer de gérer le `SIGKILL` du main, dans le temps qu'il nous restait pour finir le projet. Peut-être qu'une manière de résoudre le problème serait de transferer le statut de main au thread qui join le main lors de l'envois d'un `SIGKILL` au main.

6 Détecter les débordements de pile

La détection des débordements de pile est un des derniers objectifs que nous avons décidé d'essayer de réaliser, il n'a de ce fait pas été entierment implémenté, mais nous pensons qu'il est quand même interessant de présenter ce que nous avons commencé à faire et nos démarches.
L'idée était de trouver un moyen de gérer le cas du débordement de pile de thread, en les détectant à l'aide de *mprotect* et *sigaltstack*, et en supprimant le thread concerné. Nous avons aussi utilisé *sigemptyset*.

Bien que usuellement dans un espace mémoire de processus, les piles des threads

se remplissent "vers le bas", comme on peut le voir sur la figure 3. C'est a dire que l'espace à protéger devrait être entre *adresse_de_début_de_pile - taille_de_page* à *adresse_de_début_de_pile - 1.*

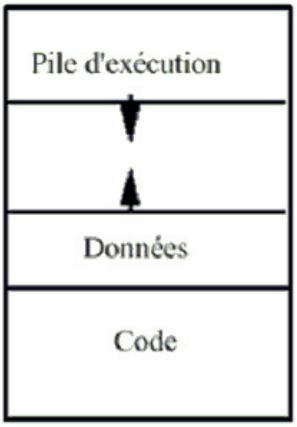


FIGURE 3 – Représentation de l'espace mémoire d'un processus

Or dans ce cas, un dépassement de pile était systématiquement détecté, même quand cela n'était pas le cas. Cela doit être du à une erreur de notre part.
Nous avons donc supposé que dans notre projet, les piles se remplissaient "vers le haut". Ainsi, l'espace à protéger fut entre *adresse_de_début_de_pile + taille_de_pile* à *adresse_de_début_de_pile + taille_de_pile + taille_de_page - 1.*

Tout d'abord, nous avons décidé d'ajouter une nouvelle méthode *page_aligned_malloc* prenant en paramètre une taille de mémoire à allouer. En effet, une des problématiques rencontrée est celle de l'utilisation de mprotect.
Pour marcher correctement, ce dernier doit commencer sa protection sur un début de page, ou une adresse mémoire étant un multiple de 4096. Ainsi, lors de l'allocation de la mémoire de notre pile, il fallait la décaler de telle sorte qu'elle commence sur le début d'une page. Ainsi, la fin de la pile tombera sur la fin d'une page, etant donné que sa taille est aussi un multiple de 4096.
Pour cela, au lieu d'allouer seulement la taille de la pile, on alloue *stack_size + page_size * 2*, on calcule le premier "début de page", et on le retourne. Ainsi, on est sur d'avoir notre pile qui tombe pile poil sur le début d'une page mémoires. Il est nécessaire d'ajouter deux fois la taille d'une page à la taille de la pile pour eviter que l'on ne protège un espace non déjà reservé.

Pour que plus tard le free se déroule correctement, nous avons décidé de stocker l'adresse de base du malloc à l'adresse précédant celle calculée. Ainsi, lors du free, on peut facilement récupérer la vrai adresse a libérer.

Voici une représentation de la manière dont l'allocation de la pile est géré pour cet objectif sur la figure 4 :

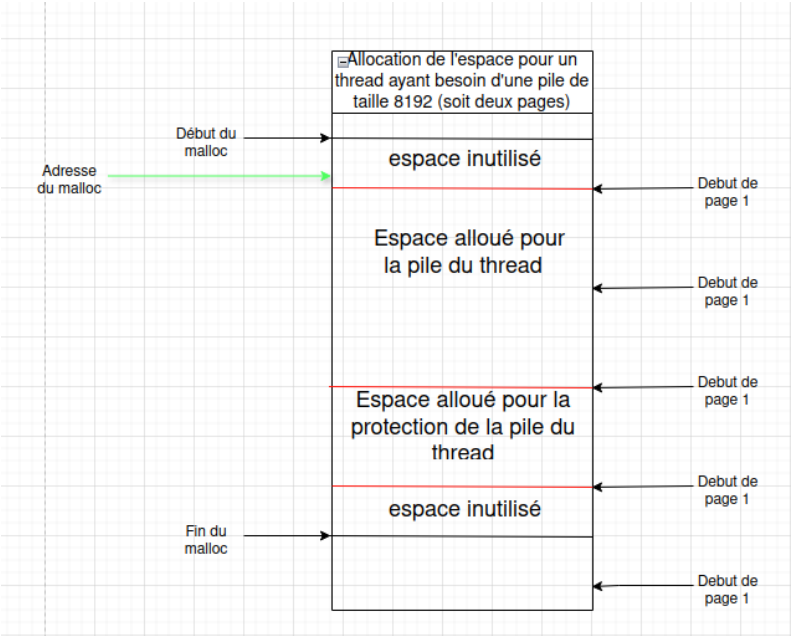


FIGURE 4 – Gestion de l'allocation de la mémoire pour une pile de thread

On initialisera alors *mprotect* a la limite de la pile, sur une taille de 4096, une page étant le minimum possible, en PROT_READ, qui envera donc un signal si le programme essaye d'écrire en dehors de cette pile.

Ce signal sera alors réceptionné par *sigaction* que nous avons initialisé avec une méthode *handler* qui vérifie que c'est bien un signal provenant d'un *mprotect* à cause d'une violation de prottection, et alors met fin au thread courant ayant fait un dépassement de pile, qui sera alors appelé a la réception du signal envoyé si cela arrive.

Mais pour pouvoir réceptionner ce signal, il faut de la place dans la pile. C'est là qu'intervient *sigemptyset*, une pile alternative qui permettra au code de continuer a tourner même avec un dépassement de pile, tout en respectant la protection. Ainsi, on peut envoyer et receptionner ce signal sans soucis.

Nous avons rencontré un soucis qui nous a fait décider de laisser les avancées de cet objectif sur une branche alternative *protect* plutôt que sur la branche principale. En effet, lors du test de *fibonacci*, plusieurs envois de signaux par mprotect

sont effectués, dont certains reconnus comme des dépassements de piles. Aussi, nous n'avons pas entierement finalisé la fonction handler, car le dépassement de pile de notre test *91-stack-overflow* ne semble pas faire l'erreur escomptée (un dépassement de pile).