Bordeaux INP
ENSEIRB
MATMECA

Systèmes d'exploitation
Rapport de projet
Équipe M2-E4

12 mai 2021

Clément Préaut
Baptiste Lecat
Tom Moënne-Loccoz
Aurélien Moinel
Fabien Savy

Introduction

Ce rapport vise à présenter le travail réalisé par l'équipe M2 - E4 pour le projet de Systèmes d'exploitation. Il s'agit de développer, en langage C, une bibliothèque de *threads* en espace utilisateur, dont l'interface est celle proposée par la bibliothèque de référence *POSIX thread* (`pthread`).

Nous commencerons par détailler les choix effectués concernant les structures de données pour la manipulation des *threads*, avant d'expliquer le fonctionnement de la version initiale de notre bibliothèque. Ensuite, nous détaillerons les fonctionnalités supplémentaires implémentées par rapport à la version de base. Enfin, nous comparerons les performances de notre bibliothèque entre différentes versions et avec celle de référence.

1 Structures de données

L'objectif de la bibliothèque étant de manipuler des *threads*, nous avons commencé par raisonner sur la manière dont ceux-ci devraient être représentés en mémoire. Nous avons ensuite choisi des structures adaptées pour stocker ces *threads* tout au long de l'exécution.

1.1 Représentation des *threads*

Nous commençons par distinguer les informations essentielles que notre bibliothèque doit posséder sur un *thread*, pour les regrouper au sein d'une structure montrée dans le Code 1, qu'on pourrait désigner sous le nom de *thread control block*.

```c
/** Représentation interne d'un thread */
typedef struct uthread {
    ucontext_t context;
    void *retval;
    ustate_e state;
    uthread_t* waited_by;
    int valgrind_stackid;
    STAILQ_ENTRY(uthread_node) next;
} uthread_t;
```

Code 1: Le *thread control block* pour l'implémentation initiale

En suivant les indications du sujet, on se propose d'utiliser les fonctions de manipulation des contextes définies dans `ucontext.h`, qui permettent de maninpuler des contextes d'exécution qui serviront à créer des *threads*. Ces contextes (structures de type `ucontext_t`) contiennent les informations nécessaires à l'exécution (notamment un pointeur vers la pile du *thread* considéré). Ainsi, nous devons stocker pour chaque *thread* un `context` qui représente son état d'exécution à un instant donné.

Ensuite, nous souhaitons pouvoir récupérer la valeur de retour d'un *thread* t_1 lorsque la fin de son exécution est explicitement attendue par un autre *thread* t_2 pour pouvoir continuer sa propre exécution. Pour cela, on se munit d'un champ `retval` de type `void *` dans la structure représentant un *thread*.

Nous avons ensuite besoin d'un moyen pour connaître l'état d'un *thread* à tout moment de son cycle de vie. Cette information permet notamment de connaître l'état d'un *thread* en temps constant et de détecter certaines erreurs qui pourraient survenir lors de l'exécution des *threads*. Pour cela, nous utilisons l'énumération `ustate_e` dont les valeurs possibles sont les suivantes :

— `RUNNING` : Le thread est en cours d'exécution.

— `READY` : Le thread se trouve dans la file d'attente pour être exécuté.

— `DONE` : Le thread a terminé de s'exécuter.

— `WAITING` : Le thread est en attente d'un autre thread (ou d'un outil de synchronisation).

En outre, nous verrons par la suite que la fonction `thread_join` place un *thread* t_1 en attente d'un autre *thread* t_2. Le champ `waited_by` permet à t_2 de garder en mémoire le thread t_1 qui l'attend pour lui rendre la main à la fin de son exécution.

Afin d'éviter que Valgrind ne lève des erreurs de type `invalid read`, il est nécessaire de lui indiquer la pile utilisée par le *thread* dans son contexte d'exécution. Pour cela, on enregistre la nouvelle pile grâce à la macro `VALGRIND_STACK_REGISTER(start, end)`, qui renvoie un identifiant stocké dans le champ `valgrind_stackid`. Le désenregistrement de la pile s'effectue grâce à la macro `VALGRIND_STACK_DEREGISTER(id)` avant de libérer cette même pile.

Enfin, le champ `next` est imposé par l'utilisation de la bibliothèque `Queue BSD` (sous-section 1.2) qui permet de stocker les *threads*. Il fait référence au *thread* suivant dans la liste.

1.2 Représentation de la file de *threads*

Pour stocker les *threads* prêt à être exécutés et maintenir un ordre dans leurs exécutions respectives, nous avons mis en place une file de *thread* : la `run_queue`. En particulier, le *thread* en tête de cette file doit toujours être celui qui est en cours d'exécution. Par opposition, nous nous efforçons de ne pas garder des *threads* en attente ou terminés dans cette file. Cela nuirait aux performances dans la mesure où la recherche d'un nouveau *thread* à exécuter nécessiterait de consulter toute une portion de la file.

Toujours d'après les indications du sujet, nous avons étudié les différentes implémentations de listes proposées par la bibliothèque `Queue BSD`. Nous avons décidé d'utiliser la structure « file d'attente simplement chaînée » `STAILQ`, car elle s'avère être la plus équilibrée en termes de performances au vu des opérations que nous devons réaliser.

Tout d'abord, la `STAILQ` permet d'ajouter en temps constant des entrées à la fin de la file, ce qui n'est pas le cas pour une liste simplement chaînée (complexité linéaire) et demande plus de temps pour la liste doublement chaînées. C'est d'autant plus important que cette opération sera effectuée de façon récurrente lors des appels à la fonction `thread_yield()` (sous-section 2.2). Bien que les opérations de la file d'attente simplement chaînée soient, d'après la documentation, 20% plus lentes que celles de la liste simplement chaînée, cette perte est compensée par l'efficacité de l'insertion en queue, une opération qui sera fréquemment utilisée. En effet, une insertion en fin de liste simplement chaînée oblige celle-ci à parcourir l'ensemble de ses éléments, ce qui engendre un surcoût d'exécution.

Enfin, les avantages supplémentaires proposés par les listes de données doublement chaînées pour d'autres opérations plus rares, comme la suppression d'un élément en temps constant, ne sont pas suffisants au vu des pertes de performances obtenues sur les opérations plus courantes (100% plus lentes que leur homologue simplement chaînée selon la documentation).

2 Implémentation

Nous nous intéressons maintenant à l'implémentation des fonctions de l'interface `thread.h`. Pour chacune, nous faisons l'hypothèse que les structures sont dans un état cohérent en entrée tout en nous efforçant de maintenir cet état durant l'exécution des différentes opérations jusqu'au retour du résultat spécifié.

2.1 La fonction `thread_self`

Pour la fonction `thread_self`, puisque le *thread* en cours d'exécution est toujours en tête de la `run_queue`, il suffit de retourner le pointeur vers sa structure (son identifiant), qui est accessible en temps constant. Puisque cette fonction ne demande aucun changement sur l'état des *threads*, aucune modification des structures de données n'est nécessaire.

2.2 La fonction `thread_yield`

Cette fonction vise à laisser un autre *thread* s'exécuter à la place de celui en cours d'exécution. Par construction, la `run_queue` contient tous les *threads* pouvant être exécutés et en particulier, le *thread* t en cours d'exécution à sa tête. Pour que t laisse un autre *thread* s'exécuter à sa place, un appel à `swapcontext` est effectué sur le `context` du second *thread* de la `run_queue`, en sauvegardant le `context` courant de t dans la structure du *thread* en tête de la `run_queue`.

Cela produit l'effet escompté pour `thread_yield` mais ne garantit pas le maintien de nos structures. En effet, un tel appel naïf ne maintiendrait pas l'hypothèse que le *thread* en cours d'exécution se trouve en tête de la `run_queue`. Pour cela, avant l'appel à `swapcontext`, nous retirons la tête de la `run_queue` et nous la replaçons à la fin de cette dernière. Puisque le *thread* lancé par `swapcontext` correspond au deuxième *thread* de l'ancienne `run_queue`, il se trouve en tête de la nouvelle `run_queue` à la fin de `thread_yield`.

2.3 La fonction `thread_join`

Nous partons de l'hypothèse que les *threads* ayant terminés leurs exécutions et n'ayant pas déjà été le sujet d'un appel à `thread_join` se trouvent dans l'état `DONE`. Si le *thread* t_2 attendu n'est pas dans cet état, le *thread* t_1 réalisant l'appel à `thread_join` doit se mettre en attente.

Nous souhaitons que cette attente soit passive, c'est-à-dire que t_1 ne soit pas remis en exécution par la suite alors que t_2 n'a pas encore terminé. Pour implémenter une telle attente, nous retirons le *thread* t_1 de la `run_queue` et nous plaçons son adresse dans le champ `waited_by` du *thread* t_2, qui est alors responsable de la remise du *thread* t_1 dans la `run_queue` lorsqu'il terminera.

Quoi qu'il en soit, lorsque le *thread* t_1 reprend la main (ou s'il ne l'a pas perdu), le *thread* t_2 a terminé son exécution. Nous supposons que t_2 a placé sa valeur de retour dans son champ `retval` lors de sa terminaison. En conséquence, nous copions cette valeur vers une variable temporaire que nous retournerons à la fin de l'appel à `thread_join`. Avant de terminer cet appel, nous libérons les ressources allouées au *thread* t_2 ?

L'implémentation décrite ici fait apparaître un cas limite lorsque deux *threads* lancent un appel pour rejoindre un même *thread* t_2. Cette situation peut poser de nombreux problèmes auxquels nous ne répondons volontairement pas. En effet, il est à la charge de l'utilisateur de ne pas employer plusieurs fois la fonction `thread_join` sur un même *thread*. De plus, puisque cette fonction est en charge de libérer la mémoire, il faudrait mettre en place une structure complexe et coûteuse pour éviter qu'un double appel engendre une double libération de la mémoire (interdite). A l'origine, nous avions implémenté une telle structure sous la forme d'une `done_queue` mais nous avons changé d'implémentation pour des objectifs de performance.

2.4 La fonction `thread_create`

Dans le cadre de la création de *threads*, il faut distinguer deux cas : la création d'un *thread* par un autre grâce à `thread_create` et la création du tout premier *thread* (le `main_thread`) qui réalise les opérations de la fonction `main` et ne nécessite pas d'appel à `thread_create`.

2.4.1 Création d'un *thread* par un autre

Tout d'abord, pour la création d'un *thread* t_1 standard, nous allouons dynamiquement un *thread control block* et une pile. Ensuite, nous initialisons le contexte avec la fonction `makecontext` sur la fonction donnée en argument de `thread_create`. Cette fonction est d'abord « emballée » dans un appel de `thread_exit`. Ainsi, si la fonction passée en argument de `thread_create` ne termine pas par un appel à `thread_exit`, elle retournera dans la fonction *wrapper* qui appellera à son tour `thread_exit` sur le retour fourni. De cette manière nous garantissons que tous les *threads* standards terminent par un appel à `thread_exit`.

Pour terminer, t_1 est placé en tête de la `run_queue` tandis que le *thread* t_2 appelant est placé à sa queue. On effectue ensuite un appel à `swapcontext` pour passer l'exécution à t_1 nouvellement créé tout en sauvegardant le progrès du contexte appelant. L'appel à `thread_create` respecte donc bien l'interface tout en maintenant la cohérence de nos structures.

L'ordonnancement choisi ici vise à empêcher le monopole du temps de calcul par une sous-partie de l'algorithme en replaçant le *thread* appelant à la queue de la file. Ce choix favorise le partage des ressources entre les *threads* au détriment cependant de la localité temporelle du cache. D'autres ordonnancements auraient cependant pu être envisageable. Par exemple, sans le définir explicitement, nos tests manuels nous indiquent

que la bibliothèque pthread semble généralement prendre la décision inverse en maintenant l'exécution du *thread* appelant et en plaçant le *thread* généré dans les derniers à être exécutés.

2.4.2 Initialisation d'un `main_thread`

L'initialisation du `main_thread` est particulière dans la mesure où il s'agit de la création du premier *thread*. La fonction `thread_create` ne peut donc pas être utilisée car elle repose sur un échange de contexte avec l'appelant. De plus, puisque la fonction à exécuter en tant que *thread* est le `main`, il faut nécessairement que la création ait lieu avant son appel dans la séquence d'exécution. Pour cela, nous faisons usage de l'attribut GCC `constructor` qui nous permet d'exécuter une fonction avant l'appel du `main`.

Dans ce constructeur, nous initialisons la `run_queue` ainsi qu'un *thread* t_{main} en tête de celle-ci, qui correspondra à l'appel du `main`. Pour pouvoir le différencier des autres *threads*, nous stockons son identifiant particulier dans une variable globale `main_thread`. Contrairement aux autres, sa `stack` a déjà été allouée par le système : il n'est donc pas nécessaire de lui en allouer une dynamiquement. De plus, t_{main} n'a pas besoin d'un contexte initialisé par une fonction à travers `makecontext` puisque la fonction `main` s'exécute directement après la terminaison du `constructor`.

2.5 Terminaison des *threads*

Nous nous intéressons maintenant à la terminaison des *threads* durant l'exécution d'un programme.

2.5.1 Terminaison d'un *thread* qui n'est pas le dernier

Considérons pour commencer le cas où il reste au moins un autre *thread* non terminé. Le *thread* terminant peut être un *thread* « normal » ou le `main_thread`.

Pour un *thread* « normal », la situation est identique qu'il termine « naturellement » (c'est-à-dire que sa fonction retourne) ou de façon « standard » (à l'aide de la fonction `thread_exit`). Le comportement est le même car la fonction lancée au sein d'un *thread* est toujours *wrappée* dans un appel à `thread_exit`. Ainsi, dans tous les cas, la fonction `thread_exit` est appelée.

Cette dernière stocke la valeur de retour fournie en paramètre (la valeur placée dans le `return` pour une terminaison naturelle) dans le champ `retval` du *thread*. Ensuite, le champ `waited_by` est consulté pour vérifier si un *thread* attend le *thread* terminant (dans le cas contraire, ce champ vaut `NULL`). Le cas échéant, le *thread* attendant est réinséré dans la `run_queue`. Ensuite, le *thread* terminant lance un changement de contexte irréversible (`set_context`) sur le premier *thread* de la `run_queue`.

Dans le cas du `main_thread`, s'il s'agit d'une terminaison « standard » le comportement est le même que pour un autre *thread* à l'exception du changement de contexte qui est réversible (`swap_context`) et qui est suivi par la libération d'un *thread* particulier nommé le `last_thread_standing`. Si la terminaison est naturelle, comme évoqué plus haut, il est impossible d'envelopper l'appel à la fonction `main` comme pour les *threads* standards, ce qui implique la terminaison du programme entier. D'ailleurs, le même comportement est observé avec la bibliothèque pthread. De façon similaire à la fonction `constructor`, nous utilisons l'attribut GCC `destructor` pour enregistrer une fonction qui sera toujours exécutée en sortie de programme. Celle-ci est chargée de libérer la mémoire du `main_thread`. Ainsi, que le `main_thread` soit le dernier *thread* ou non, s'il retourne « naturellement » les effets sont équivalents et se soldent par un appel au destructeur. Pour le restant des *threads* n'ayant pas encore été l'objet d'un appel à `thread_join`, et n'ayant donc pas été libéré, nous ne libérons pas leur mémoire.

2.5.2 Terminaison du dernier *thread*

Dans le cadre maintenant de la terminaison du dernier *thread*, nous devons à nouveau faire la distinction entre un *thread* « normal » ou le `main_thread`.

Pour un *thread* « normal », il doit libérer son propre espace mémoire et notamment sa pile d'allocation ce qu'il ne peut évidemment pas faire depuis cette dernière. Il doit donc rendre la main au `main_thread` qui a terminé avant lui. Or, nous avons vu tout à l'heure que lorsque le `main_thread` termine sans être le dernier *thread*, il donne la main juste avant de libérer le `last_thread_standing`. Ainsi, le dernier *thread* peut écrire son adresse dans la globale `last_thread_standing` pour être libérée par le `main_thread`. Ce dernier se libère ensuite tout seul par la fonction `destructor` et l'ensemble de la mémoire est libérée (à l'hypothèse que tous les *threads* sauf le dernier ont été rejoints).

Enfin, si le `main_thread` est le dernier à terminer, quelle que soit la manière dont il est terminé, seul le code du `deconstructor` doit être lancé pour libérer sa propre mémoire. La fonction `thread_exit` n'engendre donc aucune action dans ce cas. Ici encore, sous l'hypothèse que tous les autres *threads* ont été l'objets d'un appel à `thread_join`, l'ensemble de la mémoire allouée dynamiquement a été libérée.

3 Fonctionnalités avancées

3.1 Deadlock de `join`

La fonction `thread_join` peut bloquer l'exécution certains *threads* sans possibilité d'y revenir. Il s'agit d'une situation de *deadlock*. Cela arrive quand des *threads* forment un cycle d'attente. Par exemple, si le *thread* t_1 attend le *thread* t_2 et inversement.

Pour éviter cette situation, nous nous proposons d'ajouter un champ `waiting` contenant un *thread* dans la structure du *thread control block* qui vaut `NULL` par défaut. Lors d'un appel à la fonction `thread_join`, le *thread* réalisant l'appel change la valeur de son champ `waiting` pour qu'il prenne l'adresse du *thread* attendu. Cette valeur est remise à `NULL` lorsque le *thread* attendu termine son exécution et réinsère le *thread* attendant dans la `run_queue`.

Ainsi, tout *thread* en attente par la fonction `thread_join` donne accès au *thread* attendu par son champ `waiting`. Nous sommes donc en mesure de remonter une chaîne de *threads* en attente les uns des autres pour vérifier si cette dernière ne forme pas un cycle.

Pour prévenir l'apparition de cycle, nous effectuons alors une remontée de la chaîne d'attente depuis le *thread* attendu. Si lors de cette remontée, nous trouvons le *thread* réalisant l'appel, nous en déduisons que sa mise en attente conduirait à un *deadlock*. L'appel est donc avorté et une valeur d'erreur est retournée.

Cet algorithme demande de remonter la chaîne d'attente à chaque appel à la fonction `thread_join`. Elle lui apporte donc dans le pire des cas un surcoût linéaire, le coût étant à l'origine constant cela peut paraître désagréable. Pour cette raison, nous proposons à la compilation d'activer ou non cette vérification.

3.2 Signaux

Nous nous intéressons maintenant à la gestion des signaux au sein de notre implémentation. Il s'agit de permettre aux *threads* de s'envoyer et de traiter des signaux, indépendants des signaux systèmes « réels ».

Pour cela, nous avons besoin d'une structure de données adaptées pour stocker des ensembles de signaux. Nous avons choisi d'utiliser un entier comme un masque de 32 bits, la présence d'un signal i étant représentée par la valeur 1 pour le $i^{\text{ème}}$ bit. En s'appuyant sur l'interface `sigset_t` de Linux, nous avons implémenté des fonctions permettant de manipuler ces ensembles.

Nous avons commencé par permettre l'utilisation de `thread_kill` pour envoyer un signal à un *thread* en particulier, et à assurer leur traitement par le *thread* ciblé. Pour cela, nous avons augmenté le *thread control block* d'un tableau de pointeurs de fonctions gestionnaires et d'un masque de signaux en attente (`pending`). Lors d'un appel à `pthread_kill`, l'appelant se charge simplement d'ajouter le signal à l'ensemble. C'est le *thread* ciblé, lors de son réveil, qui se charge d'examiner les signaux en attente : le cas échéant, il retire le signal de l'ensemble et appelle le gestionnaire associé. La mise en place d'un gestionnaire s'effectue avec la fonction `thread_signal`.

Nous avons ensuite ajouté la possibilité pour un *thread* de bloquer certains signaux en spécifiant un masque. Comme précédemment, on ajoute un ensemble de signaux à la structure représentant un *thread*. La fonction `thread_sigprocmask` se contente de copier le contenu de l'ensemble fourni dans celui du *thread*, tandis que la vérification au réveil s'assure maintenant que le signal n'est pas bloqué avant de le traiter.

Enfin, nous avons implémenté la capacité d'attendre passivement un signal, avec la fonction `thread_sigwait`. Pour cela, on ajoute à la structure un dernier ensemble de signaux attendus, ainsi qu'un champ permettant de stocker le signal qui a réveillé le *thread*, la fonction devant retourner cette valeur. Lors d'un appel à cette fonction, on retire le *thread* de la `run-queue`. Il ne reste alors plus qu'à modifier le comportement de `thread_kill` pour réveiller le *thread* ciblé si le signal est attendu.

3.3 Débordement de pile

Nous nous proposons ici de gérer les débordement de la pile allouée à un *thread*. Lorsque que le programme tente d'accéder à une zone en dehors de la pile, ce dépassement doit être détecté et le *thread* terminé.

3.3.1 Protection de la mémoire

Pour prévenir ce problème, nous souhaitons fixer une adresse limite après laquelle l'écriture et la lecture seront proscrites. L'objectif est ainsi d'éviter que les *threads* utilisent de la mémoire située en dehors de l'espace qui leur a été alloué. Nous supposons pour cela que l'utilisateur fera une progression incrémentale dans les adresses de la mémoire et ne sautera pas complètement l'espace protégé.

Il est alors crucial de remarquer que les *threads* emploient leur espace alloué de la même manière qu'une pile standard, de façon décroissante. Cela signifie qu'ils débutent leurs écritures à la dernière adresse de l'espace alloué et remontent jusqu'à l'adresse original d'allocation. Contre-intuitivement, l'espace à protéger est donc le début de l'espace mémoire alloué par l'appel à la fonction `malloc`.

On utilise la fonction `mprotect` pour protéger le début de la pile du *thread*. On indique à `mprotect` le *flag* `PROT_NONE` qui signifie que la zone protégée n'est accessible ni en lecture, ni en écriture. Si le programme tente d'accéder à cette zone, le signal `SIGSEGV` est envoyé par le système pour signaler le dépassement.

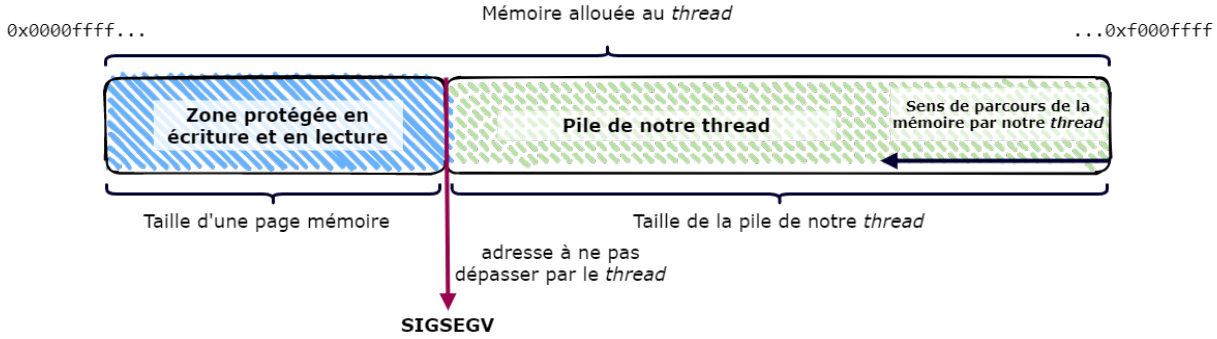

Figure 1: Schéma de protection de la mémoire

3.3.2 Gestion du signal de dépassement

Pour réaliser le comportement attendu en cas de dépassement, on installe un gestionnaire pour capturer le signal `SIGSEGV`. Celui-ci doit pouvoir exécuter une fonction alors que la pile mémoire utilisée est saturée. Pour résoudre cette problématique, on alloue une pile mémoire à part, dédiée uniquement au gestionnaire de signal. Une fois le signal reçu, le programme quitte la pile du *thread* pour exécuter le gestionnaire dans sa pile réservée.

Le gestionnaire est mis en place dans le constructeur de `thread.c` : on alloue d'abord dynamiquement de la mémoire pour la pile, puis on met en place le gestionnaire de signal avec `sigaction`, en mettant dans la structure le `sa_flag` à `SA_ONSTACK`, pour relier l'appel du gestionnaire à l'utilisation de la pile alternative.

Une fois le signal `SIGSEGV` reçu, on appelle le gestionnaire est appelé. Celui-ci appelle à son tour la fonction `thread_crash`, qui fonctionne sur le même principe que `thread_exit` pour stopper le *thread* courant, à la différence qu'elle place le *thread* dans l'état `KILLLED`. Cet état particulier indique que le *thread* ne s'est pas terminé normalement, ce qui permet à `thread_join` de signaler une erreur en retournant `-1`.

3.4 Fonctions de synchronisation

L'usage de plusieurs *threads*, principalement sous un système de préemption suppose une exécution non-déterministe du code complet du point de vue de l'utilisateur de notre interface. Cependant, certains codes ne peuvent tolérer les variations qui en résultent. L'utilisateur est alors amené à utiliser des fonctions de synchronisation pour empêcher certains scénarios d'exécutions. Nous fournissons deux ensembles de telles fonctions relatives à deux objets : le `mutex` et le `semaphore`.

Le `mutex` est un outil de synchronisation fonctionnant comme un verrou. Ainsi, si plusieurs *thread* souhaitent se saisir d'un même `mutex`, seul le premier peut l'acquérir jusqu'à ce qu'il rende la main sur ce dernier. Le `mutex` permet ainsi la protection d'une variable partagée pouvant être le sujet de lectures et d'écritures.

Pour mettre en place notre mutex, nous proposons une structure comportant l'adresse du thread propriétaire ou NULL si le mutex n'appartient à aucun *thread*. Ce champ permet de garantir que seul le *thread* propriétaire du mutex puisse le libérer. La structure comporte aussi une STAILQ du nom de waiting_queue qui stocke et ordonne les *threads* en attente du mutex.

Ainsi, lorsqu'un *thread* tente de s'approprier un mutex avec la fonction thread_mutex_lock, il vérifie d'abord que le champ owner est à NULL, c'est-à-dire que le mutex n'appartient à personne. S'il est libre, le *thread* se marque comme propriétaire du mutex en plaçant sa propre adresse dans le champ owner et termine l'appel à la fonction. Dans le cas contraire, le *thread* se retire de la run_queue et se place à la fin de la waiting_queue du mutex. Il effectue ensuite un changement de contexte avec le premier *thread* de la run_queue. Ainsi, l'attente du *thread* est passive et n'oblige pas un retour intempestif à l'exécution sans espoir que le mutex se soit libéré.

Lorsque le *thread* propriétaire du mutex le libère à travers la fonction thread_mutex_unlock, il réveille le premier *thread* de la waiting_queue, si cette dernière n'est pas vide, en le replaçant dans la run_queue. Pour garantir que le *thread* en question réussira à s'approprier le mutex cette fois-ci, nous avons décidé de directement faire de lui le propriétaire du mutex. Ainsi, à moins que la waiting_queue ne soit vide lors de l'appel à thread_mutex_unlock, le mutex n'est pas dans un état libéré au retour l'appel. Il a seulement changé de propriétaire.

Ce choix n'est pas toujours celui implémenté, mais son absence peut entraîner des situations « injustes » où un *thread* t_1 terminant son attente dans la waiting_queue est replacé dans la run_queue mais ne s'exécute pas immédiatement. Il peut alors arriver qu'un autre *thread* t_2, par exemple celui venant de relâcher le mutex, le reprenne immédiatement. Ainsi, lorsque t_1 tentera à nouveau de s'approprier le mutex, il sera contraint de se replacer à la fin de la waiting_queue sans jamais s'être réapproprié le mutex.

Dans notre implémentation, cette situation ne pourrait pas arriver car le *thread* réinséré dans la run_queue est placé en propriétaire du mutex ce qui empêche les autres *threads* de se l'approprier avant lui.

La semaphore est un objet similaire à la différence que la notion de propriétaire est remplacée par un compteur de liberté. Tous les *threads* sont libres d'incrémenter ou décrémenter ce compteur. Cependant, un *thread* souhaitant le décrémenter lorsqu'il se trouve à 0 doit attendre qu'un autre *thread* l'incrémente. L'implémentation de la semaphore est donc exactement homologue à celle du mutex à cette différence près.

3.5 Préemption

Le dernier objectif avancé que nous avons réalisé concerne la préemption. Pour cette implémentation, nous utilisons un système de signaux et de *timers*. L'objectif est d'exécuter un autre *thread* dès qu'un temps associé au *thread* courant est écoulé, par le biais d'une fonction annexe.

Tout d'abord pour réaliser ce *timer*, nous avons opté pour l'utilisation de la structure itimerval et de la fonction setitimer. Celles-ci permettent de configurer la levée de signaux avec une précision de l'ordre de la microseconde, contrairement à la fonction alarm ne proposant qu'une précision à la seconde. L'implémentation actuelle ne propose pas de politique particulière quant aux tranches de temps allouées pour chacun de nos *threads*. Ainsi, tous les *threads* disposent d'une *timeslice* d'exécution de 100 ms avant d'être préempté, ce qui correspond à la valeur par défaut allouée par Linux.

Ensuite, pour exécuter notre fonction annexe, nous devons être en mesure de capturer le signal SIGPROF, un signal levé lorsqu'un *timer* arrive à son terme. Pour cela, nous utilisons un gestionnaire de signal scheduler qui réalise un appel à la fonction thread_yield afin de placer le processus courant à la fin de la run_queue.

Néanmoins, nous ne pouvons pas nous permettre de préempter à n'importe quel moment lors de l'exécution d'un *thread*. Plus particulièrement, lorsque nous réalisons des opérations sur la run_queue, nous devons nous assurer que celles-ci ne soient pas interrompues sous peine de porter atteinte au comportement du programme. En effet, si l'on considère l'opération thread_yield, on se rend compte que l'on ne peut pas préempter l'opération entre le moment où l'on retire le thread en tête de la run_queue et celui où nous le replaçons à la fin de celle-ci. De manière plus générale, nous utilisons la méthode d'ordonnancement « Round-Robin » des *threads* : plus précisément, lorsqu'un *thread* n'a plus de temps alloué pour s'exécuter, il se met en fin de file et attend que l'ensemble des autres *threads* se placent après lui pour pouvoir s'exécuter de nouveau.

4 Comparaison de performances

Le but ici est de comparer les performances de notre implémentation avec le fonctionnement standard des *threads*. Pour cela, nous choisissons de mesurer le temps d'exécution des programmes. Pour que la comparaison soit équitable, nous verrouillons les programmes sur un seul processeur, en utilisant la commande `taskset -c`, car notre bibliothèque ne permet pas de répartir les *threads* sur plusieurs processeurs à l'inverse de `pthread`.

4.1 Méthodologie

Afin de réaliser nos tests, nous faisons varier le nombre de *threads* sur une plage et un pas d'itération donné. Plus précisément, on exécute plusieurs fois le programme de test afin de moyenner les résultats pour une configuration donnée. Ces programmes de tests s'exécutent en utilisant la bibliothèque `pthreads`, notre bibliothèque de *threads*, ainsi qu'une variante facultative augmentée de la préemption ou de la détection de *deadlock* de `join`. De cette manière nous pouvons comparer les différentes implémentations en affichant les résultats obtenus sous forme de courbes.

Pour tester notre implémentation sur des algorithmes différents, nous avons ajouté de nouveaux tests par rapport à ceux fournis. Ceux-ci ont pour but de solliciter l'implémentation, et à terme de la comparer avec la bibliothèque de référence. Au total, quatre tests ont été implémentés s'appuyant sur le principe « diviser pour régner » :

— `52-tri-fusion` : tri d'une liste par fusion

— `53-sum-tab` : somme des éléments d'un tableau d'entiers

— `54-quick-sort` tri rapide d'un tableau d'entiers

— `55-closest-points` : recherche de la plus petite distance entre deux points parmi un ensemble de points

— `57-karatsuba` : multiplication d'entiers arbitrairement grands

4.2 Résultats

Nous présentons maintenant les résultats issus de cette étude de performances.

Nous nous intéressons tout d'abord (Figure 2) aux performances sur les programmes de tests fournis qui créent et détruisent des *threads* selon diverses stratégies. Dans l'ensemble, nous pouvons remarquer que notre implémentation est plus efficace que `pthread`, ce qui était déjà le cas lors du rendu intermédiaire. Nous remarquons également que la préemption n'apporte qu'un léger surcoût, sauf dans le cas du test `33-switch-many-cascade`.

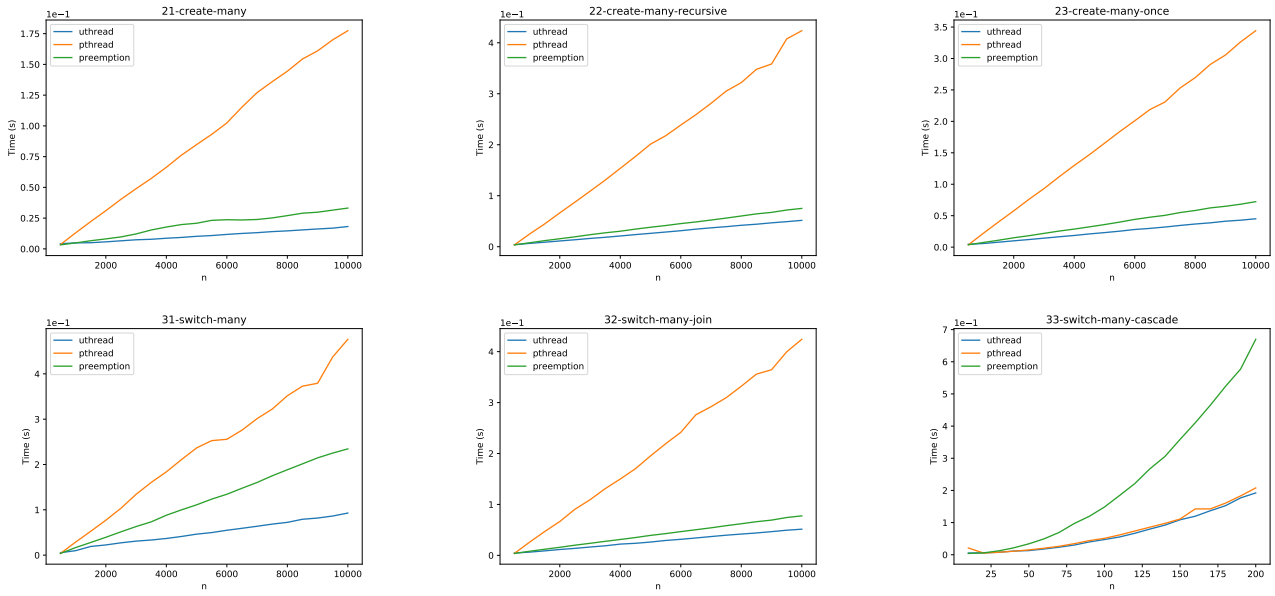

Figure 2: Comparaison des temps d'exécution avec `pthread` et `uthread` préempté sur différents tests (nombre de yields = 10)

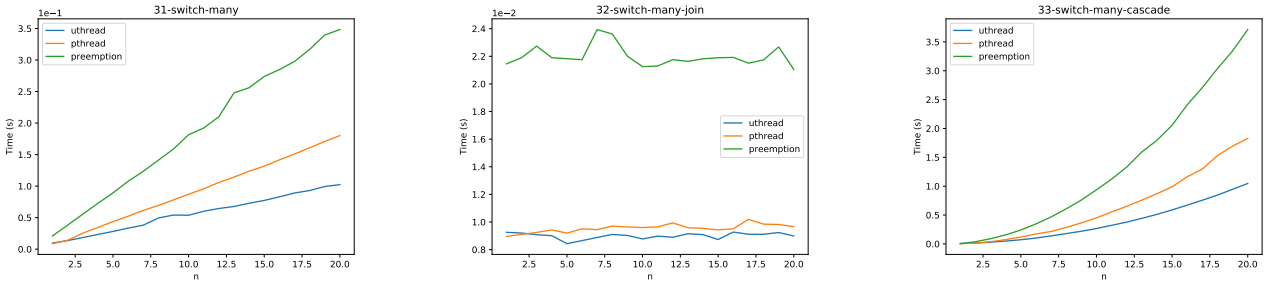

Figure 3: Comparaison des temps d'exécution avec `pthread` et `uthread` préempté sur différents tests (nombre de yields = 10000)

La Figure 3 met en évidence le surcoût à l'exécution lié à la préemption sur des tests réalisant beaucoup d'appels à la fonction `thread_yield`. Nous pensons que celui est lié aux nombreuses opérations de masquage et démasquage des signaux pour l'ensemble du processus.

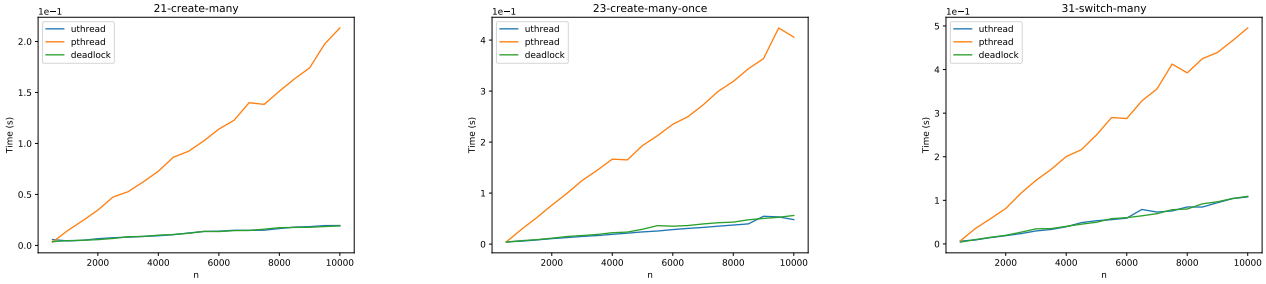

Figure 4: Comparaison des temps d'exécution avec `pthread` et `uthread` détectant les *deadlocks* sur différents tests

Nous observons sur les graphes de la Figure 4 que les performances avec et sans la recherche de `deadlock` dans `thread_join` sont relativement similaires. Cela peut paraître étonnant dans la mesure où la vérification de `deadlock` apporte un surcoût linéaire à chaque appel de la fonction `thread_join` selon le taille de la chaîne d'attente du *thread* attendu. Cependant, sur l'intégralité des tests, les chaînes d'attente sont en réalité de longueurs assez faibles. Qui plus est, la remontée de la chaîne se fait en un seul test et un déréférencement donc la constante multiplicatrice de notre complexité reste relativement faible.

En absence de préemption, et à condition d'avoir un mécanisme d'ordonnancement des *threads* simple et déterministe, il aurait été possible de générer un test particulier démontrant plus précisément les inconvénients en performance de cette vérification. Cependant, le mécanisme de préemption étant en développement lors de l'implémentation de notre algorithme de vérification, nous avons décidé de prioriser la production d'autres objectifs.

Finalement, les courbes tracées sur les tests supplémentaires permettent uniquement de conclure que notre implémentation est plus performante que la bibliothèque de référence.

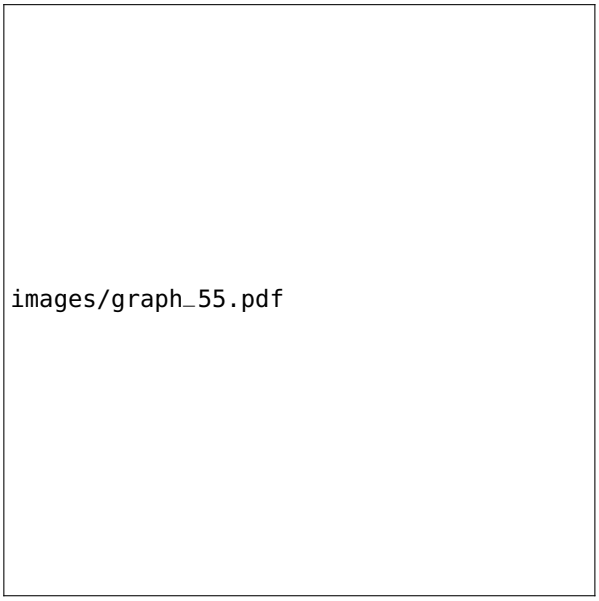

Figure 5: Comparaison pour recherche la plus petite distance

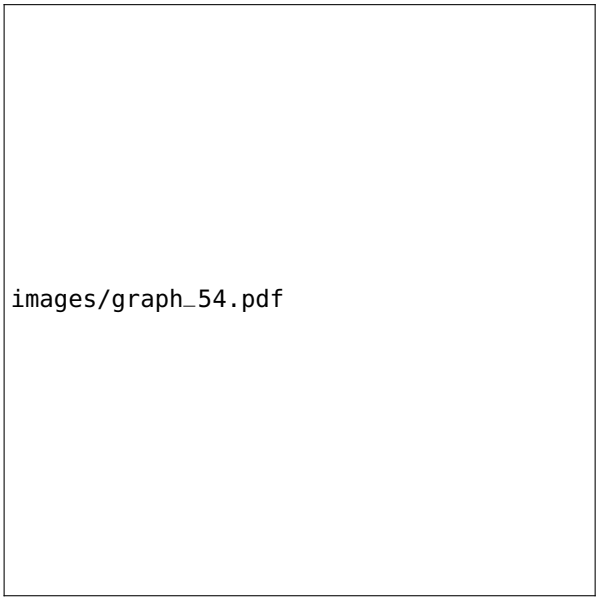

Figure 6: Comparaison pour le tri rapide

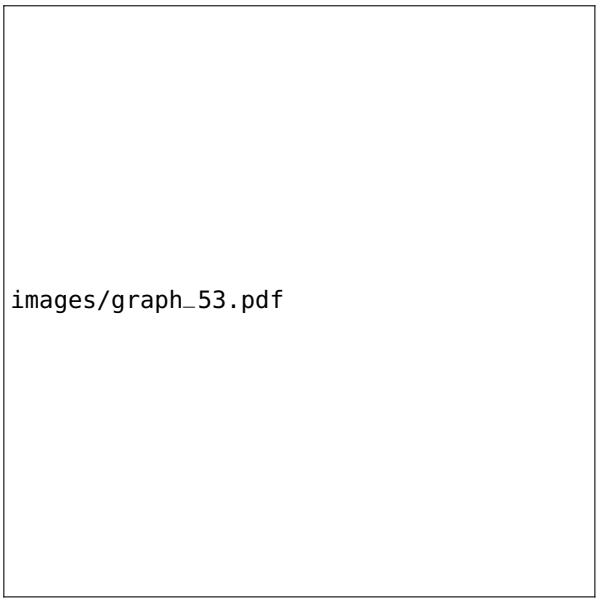

Figure 7: Comparaison pour la somme d'un tableau

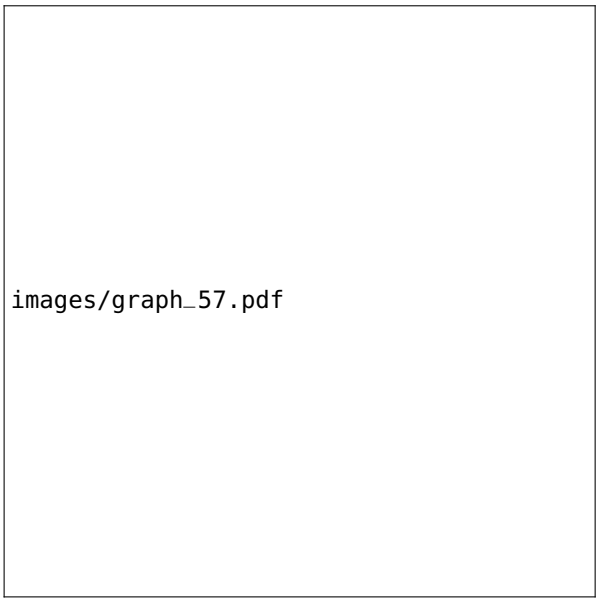

Figure 8: Comparaison pour la multiplication d'entiers

Conclusion

Finalement, à la fin du déroulement de ce projet, nous disposons d'une implémentation fonctionnelle de la bibliothèque de *threads* et avons implémenté un nombre satisfaisant d'objectifs avancés : signaux, détection du *deadlock* de `join`, détection du débordement de pile, fonctions de synchronisation et préemption.

L'une des premières améliorations à mettre en oeuvre concernerait la réunification de la détection de débordement de pile vers notre programme principal. Celle-ci pouvant occasionnellement présenter des instabilités lors de son exécution, la fusionner avec le programme principal le rendrait instable à son tour.

Ensuite, nous pourrions améliorer la gestion des priorités des *threads*. En effet, la préemption actuelle utilise la *timeslice* pour chaque *thread* et, par conséquence, ne permet pas d'établir de priorité entre eux. Pour ce faire, nous pourrions demander à l'utilisateur de préciser une priorité lors de la création d'un *thread* : il suffirait alors de multiplier la *timeslice* par un facteur proportionnel à la priorité demandée.

Enfin, nous pourrions également envisager de mettre en place un support des machines multicoeurs, c'est-à-dire utiliser plusieurs *threads* noyau pour l'exécution de nos *threads* utilisateurs.