BORDEAUX INP Enseirb-Matmeca

Rapport du projet de système d'exploitation

Threads en espace utilisateur

Mohamed Taha SANDI
Mohamed Dyae CHELLAF
Mohamed Reda EL KHADER
Othmane BOUTAHRI
Mohamed Amine ETTAKI

Département Informatique
S8 – Année 2024/2025

Table des matières

1 Introduction et problèmatique **2**

2 Implémentation de la solution **2**
2.1 Structure de données 2
2.2 Fonctionnement de la bibliothèque 3
2.3 Ordonnancement et gestion des files de threads 3
2.4 Implémentation des *Mutex* 3
2.5 Préemption 4
2.6 Priorités 5
2.7 Débordement de pile 5
2.8 Signaux 6

3 Évaluation des performances **7**

4 Amélioration possible **8**

1 Introduction et problèmatique

Ce rapport présente l'avancement du projet de système d'exploitation dont l'objectif est de développer une bibliothèque de threads en espace utilisateur. Les threads sont essentiels pour exploiter la puissance des processeurs multi-cœurs et améliorer les performances des programmes. Cependant, leur gestion par le système d'exploitation peut entraîner des coûts importants, notamment à cause des changements de contexte et des mécanismes de synchronisation lourds.

Pour répondre à ces limites, nous proposons d'implémenter une bibliothèque de threads, avec une interface similaire à celle des pthreads, mais entièrement gérée en espace utilisateur. Ce choix permet un contrôle plus direct sur l'ordonnancement et la gestion des threads, dans le but d'obtenir de meilleures performances.

Le principal défi de ce projet est de concevoir une solution efficace pour la gestion des contextes d'exécution, la planification des threads, la synchronisation et la gestion mémoire.

2 Implémentation de la solution

Cette section détaille notre implémentation de la bibliothèque.

2.1 Structure de données

On a considéré une structure qui représente un thread et qui contient :

- **`ucontext_t uc`** : Contexte d'exécution du thread, qui permet de sauvegarder et restaurer l'état du processeur (registres, pile, etc.) lors des changements de contexte.
- **`void *ret`** : Stocke la valeur de retour de la fonction exécutée par le thread.
- **`int stackid`** : Identifiant associé à la pile d'exécution du thread.
- **`int state`** : Indique l'état courant du thread (par exemple : RUNNING, SLEEPING, FINISHED).
- **`struct thread *joiner`** : Pointeur vers un autre thread qui attend la fin de ce thread.
- **`int is_main`** : Booléen indiquant si ce thread correspond au thread principal du programme.
- **`TAILQ_ENTRY(thread) entries`** : Une macro définissant les champs nécessaires pour inclure cette structure dans une liste doublement chaînée, utilisée ici pour gérer les threads dans une file d'attente.
- **`sigset_t mask`** : Masque des signaux reçus par le thread, pour la gestion des signaux personnalisés.
- **`sigset_t handling_mask`** : Masque des signaux en cours de traitement.
- **`void (*handlers[MAX_SIGNALS])(int)`** : Tableau de pointeurs vers les gestionnaires de signaux associés à ce thread.
- **`int priority`** : Priorité du thread (0 étant la priorité la plus haute), utilisée si l'ordonnancement par priorité est activé.

2.2 Fonctionnement de la bibliothèque

Lors de l'appel à `thread_create`, un nouveau contexte et une pile protégée sont alloués, puis le thread est ajouté à la file d'exécution (ou à la file de priorité selon la configuration). L'**ordonnancement coopératif** s'effectue via `thread_yield`, qui sauvegarde le contexte courant et passe la main au thread suivant, tandis que le mode préemptif utilise un timer et un gestionnaire de signaux pour forcer le changement de contexte. La terminaison d'un thread avec `thread_exit` enregistre la valeur de retour, marque le thread comme terminé et réveille un éventuel thread en attente via `thread_join`, lequel permet aussi de **détecter les situations de *deadlock*** par une chaîne de dépendances. L'initialisation et le nettoyage de la bibliothèque sont assurés automatiquement grâce aux fonctions spéciales `constructor` et `destructor`, qui préparent les files de threads, installent les gestionnaires de signaux et libèrent toutes les ressources à la fin du programme. Enfin, la **synchronisation entre threads** est possible grâce à une interface de mutex, permettant de protéger les sections critiques en bloquant et réveillant les threads selon l'état du mutex.

2.3 Ordonnancement et gestion des files de threads

L'ordonnancement des threads dans notre implémentation repose sur deux listes doublement chaînées : la première, appelée `run_queue`, contient les threads prêts à s'exécuter (état `RUNNING`), tandis que la seconde, `finished_queue`, regroupe les threads ayant terminé leur exécution (état `FINISHED`) mais dont les ressources n'ont pas encore été libérées.

Le passage de la main d'un thread à un autre s'effectue de manière coopérative et en temps constant, sous l'hypothèse que l'appel `swapcontext()` possède une complexité théorique constante. Dans la fonction `thread_yield()`, le thread courant est retiré de la tête de la `run_queue`, réinséré à la fin de celle-ci, puis le premier thread de la file est sélectionné et activé s'il existe. Si aucun autre thread n'est disponible, le thread courant continue son exécution sans changement de contexte.

Un autre point central de l'ordonnancement se situe dans la fonction `thread_join()`, qui traite deux cas principaux. Si le thread cible est en état `SLEEPING` (donc inactif), alors le thread appelant est retiré de la `run_queue`, mis en sommeil, et la main est transférée au premier thread actif de la file. En revanche, si le thread cible est encore en état `RUNNING`, un changement de contexte direct est effectué vers ce thread. Ce dernier, lorsqu'il se termine, appelle `thread_exit()`, ce qui entraîne la réactivation du thread en attente en le réinsérant dans la `run_queue`.

En conclusion, cette gestion explicite des files de threads permet un contrôle fin de l'ordonnancement dans un environnement utilisateur, tout en garantissant des changements de contexte efficaces et prévisibles. Toutefois, le recours à `swapcontext()` impose un surcoût non négligeable en pratique, ce qui peut limiter les performances dans des charges de travail fortement concurrentes.

2.4 Implémentation des *Mutex*

L'implémentation des mutex dans notre bibliothèque repose sur une structure contenant un état interne (`LOCKED`, `UNLOCKED` ou `DESTROYED`) ainsi qu'une file d'attente des threads bloqués, implémentée via une liste doublement chaînée (`TAILQ`) pour permettre une gestion FIFO efficace. Lorsqu'un thread appelle `thread_mutex_lock()`, si le mutex

est libre, il est immédiatement acquis ; sinon, le thread courant est retiré de la file d'exécution et placé dans la file d'attente du mutex, puis mis en état `SLEEPING`. Un changement de contexte est alors effectué via `swapcontext()`, transférant le contrôle à un autre thread actif. Lorsqu'un thread libère un mutex via `thread_mutex_unlock()`, le mutex passe à l'état `UNLOCKED`, et si des threads sont en attente, le premier de la file est réveillé et devient actif, prenant immédiatement le verrou grâce à un `swapcontext()` depuis le thread libérateur. Cette approche garantit l'exclusion mutuelle tout en assurant un ordonnancement coopératif correct entre les threads utilisateur.

2.5 Préemption

Pour assurer un partage équitable du processeur entre les threads, nous avons intégré un mécanisme de préemption basé sur les *timers* POSIX et les signaux. Lorsqu'elle est activée via la macro `PREEMPTION`, un minuteur déclenche périodiquement le signal `SIGUSR1`, auquel est associé un gestionnaire appelant `thread_yield()` :

```
void handler(int signum) {
    if (can_preempt)
        thread_yield();
}
```

L'initialisation du *timer* se fait au démarrage du programme :

```
sev.sigev_notify = SIGEV_SIGNAL;
sev.sigev_signo = SIGUSR1;
signal(SIGUSR1, handler);

value.it_interval.tv_nsec = 10000000;
timer_settime(timerid, 0, &value, NULL);
```

Afin d'éviter une préemption en plein accès critique à des structures de données (comme les files de threads ou les mutex), nous avons introduit deux fonctions permettant d'activer ou de désactiver temporairement la préemption via un drapeau global :

```
void disable_preemption() { can_preempt = 0; }
void enable_preemption()  { can_preempt = 1; }
```

Ces fonctions sont appelées à des endroits stratégiques dans le code, notamment :

- **Avant et après la création de threads** dans `thread_create()`, pour éviter une interruption alors que la pile ou le contexte du thread sont encore en cours d'allocation.
- **Autour des opérations sur les mutex** : dans `thread_mutex_lock()`, `thread_mutex_unlock()` et `thread_mutex_destroy()`, afin d'éviter un changement de contexte pendant qu'un thread est en train d'être bloqué ou réveillé.
- **Dans `thread_yield()`, `thread_join()` et `thread_exit()`** pour protéger les opérations de basculement de contexte et la manipulation des files.

Ce système assure que la préemption ne perturbe jamais les sections critiques, tout en maintenant une réactivité correcte du système. La préemption reste active par défaut dans les zones non critiques, ce qui permet d'ordonnancer automatiquement les threads sans qu'ils aient besoin d'appeler explicitement `thread_yield()`.

Cette approche a permis de valider le test`71-preemption`, tout en évitant des comportements indéfinis ou des corruptions de mémoire. Elle reste simple à maintenir et compatible avec notre planificateur FIFO ou par priorité.

2.6 Priorités

Pour améliorer l'ordonnancement des threads, nous avons ajouté une gestion statique de la priorité. Chaque thread est associé à un niveau de priorité fixe lors de sa création (0 étant la plus haute). L'ordonnanceur parcourt les files de threads de priorité décroissante pour sélectionner le prochain thread à exécuter.

La structure utilisée consiste en un tableau de files d'attente :

```
#define MAX_PRIORITY_LEVELS 3
struct thread_queue run_queues[MAX_PRIORITY_LEVELS];
```

La fonction de sélection choisit le premier thread prêt dans la file la plus prioritaire non vide :

```
thread *choose_next_thread() {
    for (int i = 0; i < MAX_PRIORITY_LEVELS; i++) {
        if (!TAILQ_EMPTY(&run_queues[i]))
            return TAILQ_FIRST(&run_queues[i]);
    }
    return NULL;
}
```

Lors de la création d'un thread, son niveau de priorité est spécifié et il est inséré dans la file correspondante :

```
int thread_create(thread_t *t, void *(*f)(void *), void *arg, int
    priority) {
    ...
    new_thread->priority = priority;
    TAILQ_INSERT_TAIL(&run_queues[priority], new_thread, entries)
       ;
    ...
}
```

Ce mécanisme garantit que les threads les plus prioritaires sont toujours choisis en premier, ce qui permet d'introduire des politiques plus fines d'ordonnancement.

Le système reste simple et déterministe : les priorités ne sont pas modifiées dynamiquement, et il n'y a pas d'inversion de priorité. Cette solution a été suffisante pour valider des tests avec charge concurrente, tout en gardant le code lisible et modulaire.

2.7 Débordement de pile

Pour garantir la détection et la gestion sécurisée d'un dépassement de pile, nous plaçons une page de garde immédiatement après la zone allouée à la pile de chaque thread, puis nous protégeons cette page. Si le programme tente d'accéder (lecture ou écriture) à cette page, cela déclenche un *segfault* que nous interprétons comme un *stack overflow*.

Dans notre implémentation initiale, chaque thread possédait sa pile allouée dynamiquement via `malloc`, ce qui ne permettait pas d'insérer ni de protéger une page de garde.

Nous avons donc remplacé cet appel par `mmap`, de façon à réserver explicitement une région mémoire de taille `STACK_SIZE + PAGE_SIZE`, comme suit :

```
void *stack = mmap(NULL, STACK_SIZE + PAGE_SIZE, PROT_READ |
    PROT_WRITE,
                   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

Grâce à `mmap`, nous contrôlons précisément les permissions et la taille de la zone. Après allocation, nous appliquons `mprotect` sur la page de garde pour interdire toute lecture et écriture :

```
mprotect(stack, PAGE_SIZE, PROT_NONE);
```

La pile étant croissante vers les adresses basses, nous avons réservé la première page (située aux adresses les plus basses) comme page de garde.

Ainsi, si un thread dépasse les limites de sa pile et touche la page protégée, le noyau génère un segfault. Nous interceptons alors ce signal via un gestionnaire. Le gestionnaire de signaux `segv_handler` est ainsi capable d'intercepter tous les signaux `SIGSEGV`, qu'ils soient causés par un dépassement de pile ou par une autre erreur mémoire. Il vérifie ensuite si l'adresse fautive (`siginfo.si_addr`) se situe dans la zone de garde du thread courant. Si c'est le cas, le dépassement de pile est confirmé. Le gestionnaire procède alors comme suit :

- le **thread fautif** est marqué comme **terminé** et retourne une valeur d'erreur,
- il est déplacé vers la file des threads terminés,
- le contrôle est transféré à un autre thread valide.

Cependant, cette implémentation naïve ne fonctionnait pas comme prévu. Après investigation, nous avons découvert que : lorsqu'un débordement de pile se produit, le noyau tente de livrer le signal `SIGSEGV` en utilisant la pile du thread fautif. Il en résulte un double `SIGSEGV` et l'arrêt du processus, sans que notre gestionnaire de signaux soit appelé. Pour éviter ce scénario, nous avons mis en place une pile de secours à l'aide de la fonction `sigaltstack`, et nous avons enregistré notre gestionnaire de signaux avec l'option `SA_ONSTACK`. Cela garantit que le gestionnaire s'exécute sur une pile indépendante pour permettre un traitement sécurisé, même lorsque la pile principale est inutilisable.

2.8 Signaux

Dans le cadre de l'implémentation de la gestion des signaux entre threads, nous avons adopté une approche inspirée du comportement des systèmes Unix/Linux. Chaque thread possède un masque de signaux (`mask`), représenté par un entier dont les quatre premiers bits correspondent aux quatre types de signaux pris en charge (avec `MAX_SIGNALS = 4`). Cette représentation permet de manipuler l'état des signaux en utilisant des opérations binaires simples et rapides, ce qui est beaucoup plus efficace que l'utilisation d'une file d'attente ou d'une liste chaînée. En effet, les opérations binaire, comme l'OR ou l'AND, sont exécutées en quelques cycles processeur. Ce choix reflète le comportement du noyau Linux, où un même signal envoyé plusieurs fois n'est traité qu'une seule fois. En utilisant un bit unique par type de signal, notre implémentation garantit naturellement qu'aucun traitement redondant n'a lieu tant que le signal n'a pas encore été pris en charge.

Pour chaque type de signal, un gestionnaire peut être enregistré via la fonction `setHandler`, qui associe une fonction de traitement spécifique à un signal donné. Lorsqu'un signal est envoyé à un thread via `sendSignal`, le bit correspondant est activé dans

son masque de signaux. La fonction `checkSignals`, appelée régulièrement par le thread lui-même, se charge de vérifier les signaux actifs et d'exécuter les gestionnaires associés. Puisque le thread reprend son exécution immédiatement après l'appel à `swapcontext()` suite aux interruptions, nous avons décidé d'insérer un appel à `checkSignals()` juste après cette reprise, garantissant ainsi que tous les signaux en attente soient traités avant que le thread ne poursuive son exécution normale.

Lorsqu'un thread est préempté pendant l'exécution d'un gestionnaire, il est crucial d'éviter que ce gestionnaire soit relancé à son retour. Pour cela, un second masque `handling_mask` indique quels signaux sont déjà en cours de traitement. Avant d'exécuter un gestionnaire, le signal est marqué comme "en cours", ce qui permet à `checkSignals()` d'ignorer les doublons tant que le traitement n'est pas terminé. Ce mécanisme garantit qu'un même gestionnaire ne peut jamais être exécuté deux fois simultanément, même en cas de signaux multiples ou de préemptions fréquentes.

Une fois le gestionnaire terminé, les bits du signal sont effacés à la fois dans `mask` (qui liste les signaux à traiter) et dans `handling_mask` (qui signale les traitements en cours).

Il est essentiel de protéger l'accès à `mask` et `handling_mask`. Pour cela, nous désactivons la préemption autour des lectures et modifications de ces masques. Cela évite qu'un thread soit interrompu dans une zone critique, ce qui pourrait laisser les masques dans un état incohérent. Enfin, l'appel au gestionnaire lui-même ne doit jamais se faire dans une section critique, pour éviter de bloquer la progression d'autres threads ou signaux pendant son exécution.

3 Évaluation des performances

Cette section propose une analyse comparative des performances entre notre bibliothèque de threads et `pthread`. La figure 1 illustre cette comparaison à travers différents scénarios d'exécution. Les tests ont été réalisés sur une machine équipée de **16 cœurs physiques** et de **32 cœurs logiques**.

Pour les tests `31-switch-many_performance`, `32-switch-many-join_performance` et `33-switch-many-cascade_performance` :

- Les trois premières courbes représentent les performances avec un nombre de **yield** fixé à **40**.
- Les trois suivantes montrent les performances lorsque le **premier argument** est fixé à **1000 threads**.

Analyse générale

Notre bibliothèque présente un temps d'exécution globalement **linéaire par rapport au nombre de threads**, ce qui confirme la stabilité de notre ordonnanceur coopératif dans la majorité des cas. Cependant, deux tests mettent en évidence des limites spécifiques de notre implémentation par rapport à `pthread` : `33-switch-many-cascade` et `62-mutex`.

Le test `33-switch-many-cascade` met en œuvre une chaîne récursive de threads. Chaque thread crée un sous-thread, effectue un nombre décroissant de `thread_yield()` proportionnel à sa profondeur, puis effectue un `thread_join()`. Cette structure provoque un comportement en cascade avec un enchaînement de `swapcontext()` à chaque niveau. Par exemple, pour une exécution avec les arguments `500 40`, notre implémentation effectue environ **5 030 002 appels à `swapcontext()`**. Cette explosion vient du fait que

chaque `yield()` provoque un changement de contexte. Cela introduit un coût inutile, en particulier dans les phases récursives profondes où les contextes sont enchaînés sans gain réel. En comparaison, `pthread` utilise des threads du noyau, gérés de façon plus intelligente par le planificateur Linux. Il amortit les appels à `yield()` et optimise les `join()` via des mécanismes comme `futex`, ce qui lui permet de surperformer très largement sur ce test.

Le test `62-mutex`, en revanche, met en jeu un très grand nombre de threads exécutant des incrémentations concurrentes protégées par des mutex. Chaque thread effectue 1000 itérations, chacune comportant deux appels à `thread_yield()` à l'intérieur de la section critique. Cela conduit à environ **4 001 080 `swapcontext()`** pour un argument de `100`. Cependant, contrairement au test précédent, la structure est plate : tous les threads sont créés dès le départ, travaillent indépendamment et les `join()` sont effectués à la fin. De plus, les appels à `yield()` dans ce contexte sont souvent utiles, car ils permettent un vrai partage du processeur entre threads actifs. Par conséquent, notre bibliothèque reste ici compétitive, avec un écart de performance très faible (de l'ordre de 0.1 à 0.15 seconde selon les machines).

En conclusion, `pthread` surperforme notre bibliothèque lorsqu'il peut exploiter des optimisations noyau dans des structures récursives profondes ou à forte densité de synchronisation hiérarchique. En revanche, notre ordonnanceur coopératif reste très efficace dans les situations plates, parallèles et régulières, malgré une utilisation intensive de `swapcontext()`.

4 Amélioration possible

Support du multi-cœur : Pour exploiter pleinement les processeurs multi-cœurs, une amélioration envisageable serait d'autoriser l'exécution simultanée de plusieurs threads utilisateur. Actuellement, notre ordonnanceur est mono-threadé : un seul thread s'exécute à la fois. Une solution consisterait à utiliser plusieurs `pthread`s internes, chacun gérant un thread utilisateur sur un cœur distinct.

Cela nécessiterait plusieurs adaptations majeures :

- **Synchronisation** : les structures partagées comme `run_queue` devraient être protégées par des verrous ou opérations atomiques.
- **Sécurité des sections critiques** : toutes les opérations sensibles (mutex, `join()`, etc.) doivent être rendues thread-safe.
- **Affinité processeur** : il serait possible de lier un thread utilisateur à un ou plusieurs cœurs via `sched_setaffinity()` pour mesurer l'effet du parallélisme.
- **Répartition de charge** : pour éviter les déséquilibres, des stratégies comme le *work stealing* ou des files locales/globales pourraient être utilisées.

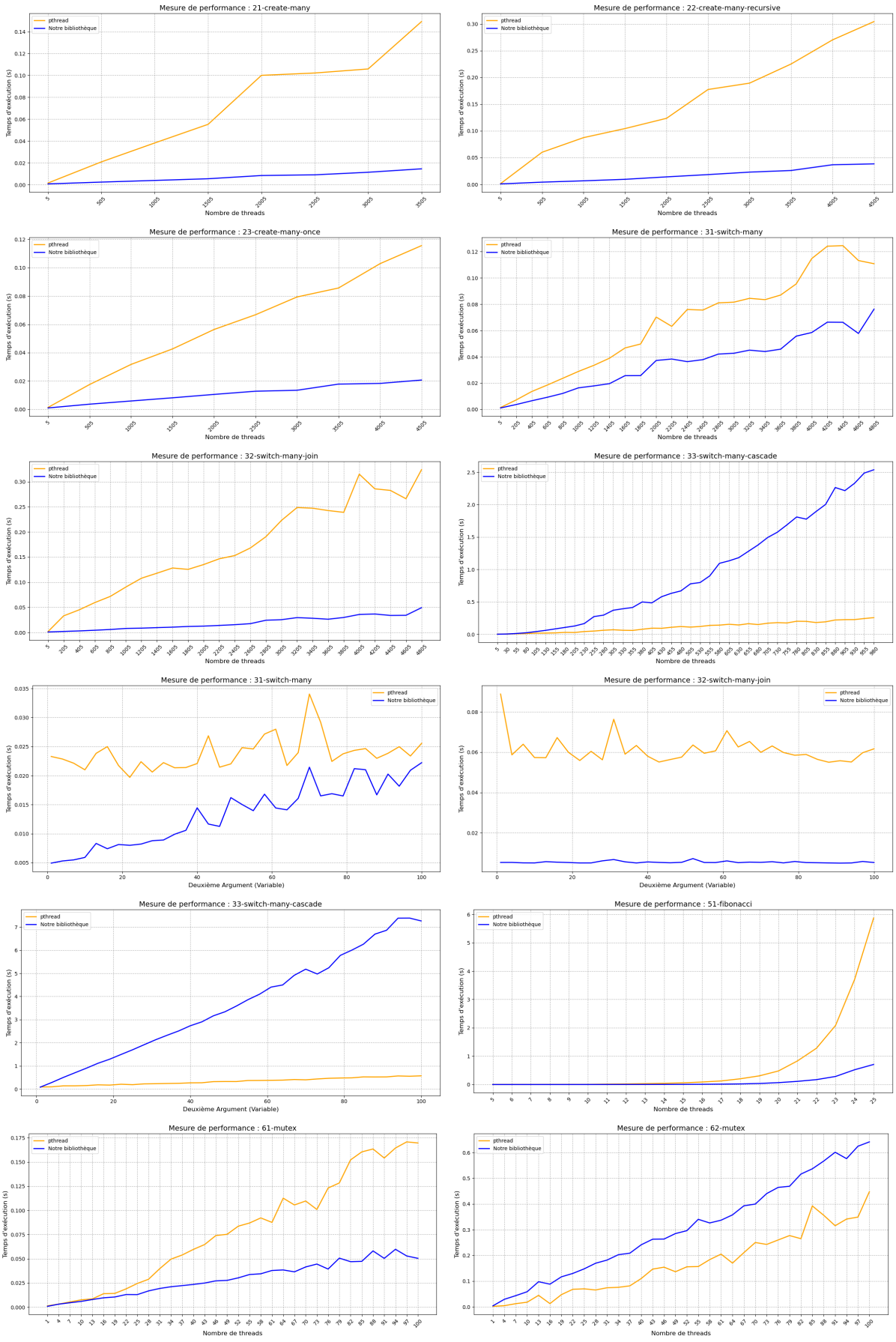


FIGURE 1 – Mesures de performance sur différents benchmarks avec notre bibliothèque et `pthread`