SORBONNE UNIVERSITE

Projet SAR

# Ordonnancement de processus avec prise en compte de la synchronisation

*Auteurs :*
Pierre-Loup Gosse
Salim Toufik Nedjam

*Encadrants :*
Redha Gouicem
Julien Sopena

8 juin 2020


# Table des matières

# 1 Introduction

Un des rôles les plus importants pour un système d'exploitation est l'exécution de plusieurs tâches simultanément, c'est ce qu'on appelle le multitasking. Le processeur de la machine est une ressource partagée par l'ensemble des tâches qui s'y exécutent. L'objectif est donc que chaque tâche puisse y avoir accès pour de s'exécuter convenablement. D'un point de vue utilisateur l'alternance de l'exécution des tâches doit être imperceptible.

C'est l'ordonnanceur, scheduler en anglais, qui s'occupe de cette tâche critique. Il a pour but premier de rendre le multitasking le plus efficace possible en utilisant le moins de ressources et en respectant les trois principes clés qui sont : Safety, Liveness et Fairness.

L'exécution parallèle des tâches sur le processeur implique parfois à celles-ci de se synchroniser, cela peut se faire grâce aux verrous. Un verrou est une ressource partagée où son détenteur bloque les autres tâches qui souhaiteraient le prendre.

Un problème se présente quand plusieurs processus souhaitent accéder à une même ressource critique protégée par un verrou. En effet, si le processus, qui détient le verrou, empêche un ou plusieurs processus d'avancer il est important de le faire passer en priorité dans les élections. Ainsi, en favorisant le processus propriétaire du verrou, ce dernier pourra arriver plus rapidement au point de libération de la ressource, et ainsi débloquer les autres processus en attente.

Les objectifs de notre projet ont donc été les suivants : optimiser les décisions de l'ordonnanceur pour apporter une élection plus favorable aux processus détenant un verrou.

Nous verrons tout d'abord dans ce rapport les différents aspects de la glibc qui permettent de gérer la synchronisation des tâches. Ensuite, nous aborderons les différents points qui font le lien entre la glibc et le kernel, notamment pour transmettre l'identité du processus propriétaire du verrou. Nous introduirons le scheduler et son fonctionnement, ainsi que les différents scénarios qui permettent de résoudre le problème. Enfin, nous présenterons notre implémentation et les benchmarks. Pour conclure nous discuterons sur les améliorations et les limites de notre solution.

# 2 Background

## 2.1 User : glibc et mutex

La glibc est une librairie qui offre un panel de fonctionnalités. Celle des verrous (mutex) nous intéresse particulièrement : elle permet d'abstraire aux programmeurs l'utilisation complexe des verrous côté kernel.

Il existe plusieurs types de mutex proposés à l'utilisateur, voici une liste non exhaustive :

1. TIMED : type par défaut, prend le verrou de manière bloquante.
2. RECURSIVE : ne tente pas de prendre le verrou si celui-ci est déjà détenu par le processus courant.
3. ADAPTIVE : correspond à la méthode de prise de verrou 'trylock', au bout d'un certain nombre d'échec la prise de verrou se fait bloquante.

Le choix du type de mutex se fait lors de l'initialisation du verrou. On peut combiner ces types externes avec des types internes qui permettent de modifier le comportement du verrou jusqu'au kernel, notamment les types :

1. ROBUST : type par défaut, si le propriétaire du verrou meurt sans le relâcher la prochaine demande à se verrou sera accepté sans appel système.
2. PI (Priority Inheritance) : la priorité du processus détenant le verrou est augmenté à la plus grande priorité parmi les processus bloqués sur le verrou.

Ces deux types insèrent le *pid* du propriétaire du verrou directement dans la valeur du verrou. Ainsi, lors des appels systèmes le kernel peut connaître le *pid* du propriétaire du verrou avec une simple opération de flag sur la valeur passée.

En tenant compte de ces éléments on peut donc en déduire que la modification de la partie glibc n'est pas nécessaire. En effet, le *pid* du propriétaire du verrou étant déjà communiqué il nous suffit de le récupérer côté kernel. Ainsi, cela permet à notre future solution de ne pas dépendre de modification ou de décoration d'une fonctionnalité de la glibc, et d'éviter des dépendances dur à maintenir entre la glibc et le kernel

## 2.2 Kernel : futex

La glibc peut communiquer avec le kernel à travers des appels systèmes pour effectuer certaines tâches. Un appel système est coûteux, ce qui force la glibc à réduire son utilisation.

Considérons un processus A et B, où A détient un mutex et B souhaite l'acquérir. Observons le comportement de la glibc et du kernel.

Lorsqu'un processus va essayer de prendre un mutex la glibc va faire un test and set atomique sur sa valeur. En fonction de cette dernière la gblic déterminera s'il est nécessaire de faire un appel système.

Dans un premier temps A prend le mutex, sa valeur vaut 0. A peut donc immédiatement l'acquérir et placer la valeur 1 pour indiquer que le mutex est occupé.

Lorsque B souhaitera prendre le mutex il constatera une valeur à 1, B doit donc s'endormir. Cette opération nécessite un appel système vers le kernel. Avant l'appel, B prend soin de placer la valeur 2 dans le mutex afin d'indiquer que des processus attendent sur ce dernier.

Dans l'appel système effectué par B, l'adresse virtuelle de l'espace utilisateur qui contient la valeur du mutex est passée en paramètre. Cette adresse permet au kernel de récupérer la valeur du mutex, et avec une opération bit à bit, le pid du propriétaire. Le kernel utilise l'adresse pour générer et identifier le futex d'une manière unique grâce à une `futex_key`. Grâce à cela le kernel endort le processus B et le représente par une structure `futex_q` qui attend sur le futex.

Lorsque A relâche le mutex il constatera une valeur à 2, lui indiquant qu'un ou plusieurs processus attendent sur le mutex. Il est donc nécessaire de faire un appel système pour réveiller ces processus. Le kernel réveillera les processus associés aux `futex_q`. Si la valeur du mutex aurait été de 1 alors A aurait placé la valeur 0 sans effectuer d'appel système.

Le processus B est réveillé et deviendra le propriétaire du mutex.

Grâce à ce mécanisme la glibc fait des appels système uniquement si d'autres processus attendent sur le mutex.

# 3 Scheduler

Depuis sa version 2.6.23 le kernel utilise le Completely Fair Scheduler (CFS), il est aujourd'hui le scheduler par défaut. Le but du CFS est de proposer un ordonnancement idéal pour le multitasking, où la puissance du CPU est partagée entre les processus. Or ce modèle n'est pas possible, un CPU ne peut exécuter qu'un flot d'exécution à la fois. Dans cette partie nous introduirons les mécanismes mis en œuvre par le CFS pour imiter un multitasking parfait.

## 3.1 L'arbre rbtree

Le CFS fonctionne sur un système d'arbre : le `rbtree`. L'arbre permet de créer une timeline sur les futures tâches à exécuter. Il est composé de `sched_entity`, ces entités regroupent une ou plusieurs tâches, permettant ainsi de faire des lots d'exécution. Chaque entité a un compteur du temps d'exécution virtuel passé dans le CPU nommé `vruntime`. Cela permet au scheduler d'organiser son arbre d'exécution : les entités sont réparties par ordre croissant à partir de leur `vruntime`, de gauche à droite. Ainsi, à gauche de l'arbre se trouve l'entité avec la valeur `vruntime` la plus faible, c'est cette entité qui sera choisie lors de l'élection par le scheduler.

## 3.2 Poids et timeslice

L'équité est le maître mot du scheduler CFS. Chaque tâche se voit attribuer un poids (`load`) calculé dépendamment des autres tâches exécutables et de leur priorité. Le CFS introduit un `target time`, cette valeur correspond aux temps CPU que doit se partager les tâches exécutables en fonction de leur poids. Le temps d'exécution attribué sur le CPU pour chaque tâche est nommé `timeslice` et est calculé à partir de son poids et du `target time`. Ainsi, une tâche avec une priorité plus élevée aura un poids plus important et donc un `timeslice` plus grand. Prenons un exemple, deux tâches de même priorité doivent se partager un `target time` de 10 ms. Les deux tâches ont donc le même poids, le CFS attribuera aux tâches un `timeslice` de 5 ms. Si nous prenons maintenant deux tâches où leur différence de priorité est de 5, soit par exemple 10 et 15, avec un `target time` de 20 ms, le poids diffère et

la tâche la moins prioritaire se verra attribuer 1/3 du `target time` soit une `timeslice` de 5 ms contre 15 ms pour l'autre tâche.

## 3.3 Horloge virtuelle

Le cœur du fonctionnement du CFS se repose sur son horloge virtuelle : la `virtual clock`. Cette horloge permet au scheduler de mesurer un temps d'exécution virtuel sur le CPU pour chaque entité. Cette valeur est stockée dans la variable `vruntime`. L'entité élue se voit mettre à jour ses statistiques fréquemment, c'est ici que sa valeur `vruntime` est mise à jour. Le principe est le suivant : à chaque fois qu'une entité s'exécute dans le CPU le timestamp est sauvegardé dans la variable `exec_start`, à chaque mise à jour de `vruntime` la différence entre le timestamp actuelle et `exec_start` est calculée dans la variable `delta_exec`, la valeur de cette dernière sera divisé par le poids de l'entité puis ajouté au `vruntime`. Ainsi, plus la priorité est élevée plus la valeur ajoutée au `vruntime` sera faible et par conséquent l'entité avancera lentement vers la droite de l'arbre (Figure 1).

Pour prioriser une entité nous pouvons influencer son `vruntime` pour la garder à gauche de l'arbre et assurer une réélection rapide. Étant donné que la valeur `vruntime` se base sur la priorité de l'entité nous avons décidé d'influencer la priorité des processus, et les utiliser comme un levier pour dire au scheduler de favoriser l'élection du processus qui détient le verrou. Pour cause, la modification du scheduler aurait demandé d'étudier et de maîtriser tout le code et les détails du scheduler, afin d'être sûr que nos modifications ne compromettent pas son fonctionnement.

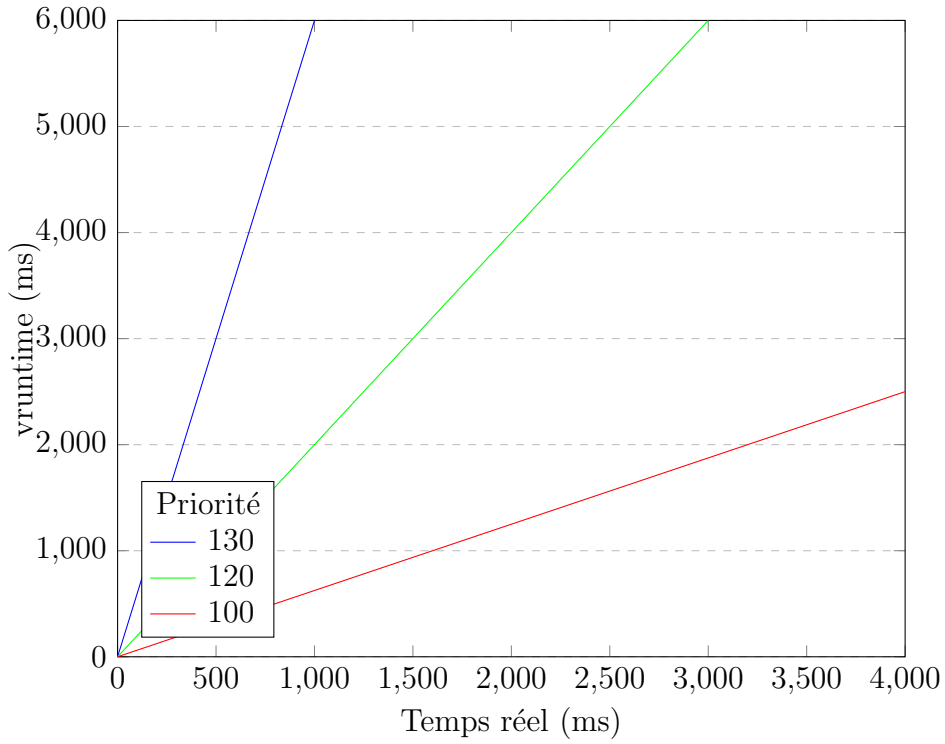


FIGURE 1 – Relation entre le vruntime et le temps réel en fonction de la priorité

# 4 Scénario

Dans la suite du rapport nous considérons avoir trois tâches : A, B et C. L'objectif présenté par les scénarios est de favoriser la ou les tâches qui détiennent un verrou pour débloquer rapidement le fil d'exécution des autres tâches potentiellement en attente, tout en ne dénaturant pas les principes clés du CFS.

## 4.1 Monocœur

Considérons un monoprocesseur où les tâches A et C ont une priorité de 120 et B une priorité de 100. La tâche A détient un verrou. Le taux de CPU attribué aux tâche A et C sera le même, cependant A doit être la tâche cible de notre modification, pour que celle-ci soit favorisé lors de l'élection par rapport à C. Cependant, la modification apportée ne doit pas être une gêne pour l'avancement de B, qui est lui de base, est plus prioritaire.

## 4.2 Multicœur

Considérons un processeur avec deux cœurs où les tâches A et B sont des lots d'exécutions contenant chacun 5 threads, avec la même priorité. Le taux CPU attribué aux deux tâches est le même. Aucun des threads de A ne prend de verrou. Au bout d'un certains temps un thread de B prend un verrou, bloquant tous les autres threads de B. Il faudra 100 `timeslice` pour le thread propriétaire du verrou pour exécuter sa section critique et ainsi relâcher le verrou, débloquant tous les autres threads de B. Si les 5 threads de A s'exécutent chacun en 10 `timeslice` alors le thread A se terminera avant la libération du verrou de B. Ainsi, la tâche A va libérer son cœur et ce dernier n'exécutera aucune instruction avant que les threads de B ne soit débloqués. Notre solutions doit donc éviter le gâchis de cœur. En favorisant B ce dernier va exécuter sa section critique plus rapidement, permettant ainsi de finir avant, ou en même temps, que le thread A. Lorsque A finira son exécution les autres threads de B, débloqués, pourront utiliser le cœur libéré.

## 4.3 Charge

Notre solution pour influencer la priorité d'une tâche, tout en restant équitable, est d'introduire un système de charge à chaque verrou.

### 4.3.1 Augmentation de la priorité

Pour reprendre le scénario du monoprocesseur nous pouvons envisager de favoriser la tâche A, détenant un verrou, en augmentant sa priorité. Cette augmentation ne doit cependant pas être un inconvénient pour B, plus prioritaire. Une augmentation d'une valeur de 10 est un bon compromis. La tâche A devient alors plus prioritaire que C sans dépasser B.

Cependant, si C détient lui aussi un verrou, et que A et C bloquent respectivement 10 et 100 tâches, alors augmenter la priorité de 10 au deux tâches semble non équitable étant donné que C bloque plus de tâches que A. Ainsi, une augmentation dans un intervalle de 0 à 10 est plus cohérent. La tâche A aura une priorité augmentée de 2 contre 10 pour C. Cette augmentation de priorité sera déterminé avec un système de **charge**.

Une autre solution aurait pu être de modifier directement la valeur `vruntime` pour forcer la tâche à être à gauche de l'arbre `rbtree` et ainsi assurer sa réélection. Cependant, cela implique que la tâche passera devant tous les autres tâches, y compris celles plus prioritaires. Cette solution écrase toute la logique du système de priorité mis en place par le scheduler et donc n'a pas été retenue.

### 4.3.2 Calcul de charge

Chaque tâche propriétaire d'un verrou se verra attribuer une charge, celle-ci évoluera en fonction des tâches bloquées sur le verrou. Plus une tâche a une charge élevée, plus sa priorité augmente.

L'implémentation du calcul de charge est assez simple. La charge correspondra au nombre total de tâches couramment bloquées sur le verrou. Ainsi, la priorité ajoutée sera dans l'intervalle de 0 à 10 : une tâche bloquée augmentera la priorité de 1, bornant l'augmentation à 10 tâches bloquées.

### 4.3.3 Héritage de charge

Pour rendre le système de charge plus efficace nous introduisons un héritage de charge entre les tâches. Considérons que B détient un verrou et se bloque en essayant de prendre un deuxième verrou détenu par A. Alors A verra sa charge augmenter par la valeur celle de B. Ainsi, la tâche A sera plus favorisée même si A et B ont la même priorité et la même charge au départ. Si une troisième tâche C, ne détenant pas de verrou, se bloque sur le verrou détenu par A, alors l'héritage fera incrémenter de 1 la charge de A et de B.

Dans le même principe, si C meurt avant d'avoir récupéré le verrou, alors un deshéritage est appliqué : A et B voient leur charge décrémenter de 1.

L'héritage de charge est très utile mais demande aussi d'être très précis. En effet, si la charge d'une tâche est incorrecte par rapport à la réalité (i.e. la valeur ne correspond pas au nombre de tâches bloquées) alors l'héritage ne peut qu'aggraver l'erreur. Il est très important de s'assurer que la charge est valide en la décrémentant lorsqu'une tâche n'attend plus sur le verrou, notamment lorsque la tâche meurt avant de l'avoir pris.

# 5 Implémentation

Notre choix d'implémentation a donc été de ne pas modifier directement le code du scheduler (`<kernel/sched/fair.c>`), mais d'ajouter notre mécanisme lors de la prise de verrou dans le code du futex (`<kernel/futex.c>`) et d'agir directement sur les priorités des tâches.

L'ensemble de nos modifications dans le kernel on été préfixés par un commentaire `MAS code`.

## 5.1 Structure

La première étape à donc été de créer un mécanisme permettant au kernel de pouvoir facilement suivre l'évolution du propriétaire d'un futex. Nous avons donc ajouté une structure `futex_state` qui représente l'état d'un futex :

```
struct futex_state {
    struct list_head list;
    struct task_struct *owner;
    raw_spinlock_t spin_lock;
    struct kref refcount;
    int load;
    union futex_key *key;
};
```

Les champs sont les suivants :

- `list` permet de créer un chaînage entre les futex d'un même propriétaire.
- `owner` est une référence sur la `task_struct` du propriétaire du futex.
- `spin_lock` permet d'éviter les accès concurrents lors de la manipulation de la structure.
- `kref` est un compteur de référence pour protéger la suppression des structures, et ainsi éviter de libérer la structure utilisée ailleurs.
- `load` est le poids associé au futex.
- `key` est la clé du futex, permettant d'identifier la structure pour un futex donné.

Une modification sur la structure `task_struct` a été nécessaire :

```
struct task_struct {
    ...
    struct list_head futex_state_list;
    raw_spinlock_t futex_state_lock;
    struct futex_state *waiting_futex_state;
    int user_nice;
    int futex_state_prio;
    ...
}
```

Les champs ajoutés sont les suivants :

- `futex_state_list` est l'ensemble des `futex_state` que la tâche détient et sur lesquels d'autres tâches attendent.
- `futex_state_lock` est un verrou pour la manipulation de la liste.
- `waiting_futex_state` est un pointeur vers le `futex_state` sur lequel la tâche attend.
- `user_nice` est le valeur courante du nice utilisateur appliquée à la tâche.
- `futex_state_prio` est l'augmentation de priorité appliquée à la tâche en fonction des autres tâches qu'elles bloquent.

## 5.2 Fonctionnement

Lorsqu'une tâche souhaite prendre un verrou déjà détenu elle passe en mode kernel. C'est là que notre mécanisme rentre en jeu.

L'appel système place dans `uaddr` l'adresse du verrou côté user. La valeur du verrou, contenant le pid du propriétaire, peut être récupérée avec `get_user(uval, uaddr)`. Ainsi, la tâche qui va se bloquer en attente du verrou connaît le pid du propriétaire avec une simple opération bit à bit sur `uval` et peut accéder sa structure `task_struct` avec `find_task_by_vpid(vpid)`.

### 5.2.1 Prise de verrou

Lors de la prise de verrou notre première étape est de récupérer la `futex_key` associé au futex sur lequel la tâche va se bloquer :

```
get_futex_key(uaddr, 0, key, VERIFY_READ);
```

Ensuite il faut récupérer le `futex_state` associé à cette `futex_key` :

```
get_futex_state(owner, key, &state);
```

Cette fonction va regarder si le `futex_state` associé à la clé `key` existe dans la liste du `owner`, si la structure n'existe pas alors elle sera allouée, initialisée et ajoutée à la liste du `owner`, sinon elle sera retournée. Dans les deux cas le compteur de référence du `state` retourné sera incrémenté.

Pour indiquer que la tâche courante va s'endormir en attendant sur le futex, on passe la structure du `futex_state` précédemment récupérée à la `task_struct` :

```
current->waiting_futex_state = state;
```

### 5.2.2 Héritage de charge

Après l'ajout de la tâche courante comme tâche en attente du futex il faut appliquer le changement de charge par héritage :

```
futex_state_inherit(current, state,
                    FUTEX_STATE_LOAD);
```

La tâche courante `current` peut elle aussi bloquer plusieurs tâches sur plusieurs verrous, ainsi il est important de ne pas simplement incrémenter de 1 le poids du futex sur lequel on va attendre, mais de la somme des poids des `futex_state` que la tâche courante détient. Cette somme est calculée avec un simple parcourt sur la liste des `futex_state` de la tâche :

```
get_futex_state_sumload(task);
```

Le propriétaire de `state` peut lui aussi attendre sur un `futex_state`, détecté grâce au champs `waiting_futex_state`. Le principe d'héritage demande donc de procéder par récurrence sur l'arbre et de mettre à jour le poids de chaque `futex_state` et ainsi remonter à la tâche qui n'attend pas sur un verrou, le *master futex owner*. C'est ce dernier qui verra sa priorité changer à partir de son poids.

```
int futex_state_inherit(struct task_struct *task,
```

```
                        struct futex_state *state,
                        int op)
{
    struct futex_state *m_state;
    int sumload;

    if (op != FUTEX_STATE_LOAD &&
        op != FUTEX_STATE_UNLOAD)
        return -1;

    /* Recupere la somme des poids des eventuelles
       futex_state que l'on detient */
    sumload = get_futex_state_sumload(task);

    /* Remonte l'arbre jusqu'au master futex
       owner et applique le poids a chaque futex_state */
    do {
        m_state = state;
        state->load += (sumload + 1) * op;
    } while ((state = state->owner->waiting_futex_state)
        != NULL);

    /* Applique la priorite sur le master futex owner */
    futex_state_prio(m_state->owner);

    return 0;
}
```

### 5.2.3 Application de la priorité

La fonction `futex_state_prio(task)` permet de définir la priorité d'une tâche en se basant sur son `sumload`. L'implémentation finale est très simple et peut être la cible d'amélioration.

```
void futex_state_prio(struct task_struct *task)
{
    int load = get_futex_state_sumload(task);
```

```
    if (load < 0)
        load = 0;

    if (load > FUTEX_STATE_MAX_PRIO)
        load = FUTEX_STATE_MAX_PRIO;

    task->futex_state_prio = load;
    set_static_prio(task);
}
```

L'augmentation attribuée à la tâche correspond au nombre de tâche qu'elle bloque, borné sur `FUTEX_STATE_MAX_PRIO` valant 20.

La fonction `set_static_prio(task)` est ensuite appelé. Cette fonction se situe dans `<kernel/sched/core.c>` et permet de modifier la priorité statique d'une tâche en combinant le nice utilisateur (`user_nice`) et l'augmentation de priorité lié au futex state (`futex_state_prio`). Voici son code simplifié :

```
void set_static_prio(struct task_struct *p)
{
    int prio;

    prio = NICE_TO_PRIO(p->user_nice) - p->futex_state_prio;

    ...

    p->static_prio = prio;
    set_load_weight(p, true);
    old_prio = p->prio;
    p->prio = effective_prio(p);

    ...
}
```

La fonction appelée lors de la modification du nice par l'utilisateur a aussi subi des changements :

```
void set_user_nice(struct task_struct *p, long nice)
{
    if (task_nice(p) == nice || nice < MIN_NICE
```

```
        || nice > MAX_NICE)
    return;

    p->user_nice = nice;
    set_static_prio(p);
}
```

Ce mécanisme de changement de priorité permet de faire cohabiter le nice utilisateur et l'augmentation de priorité des `futex_state`. Quand l'une des deux valeurs change l'autre est toujours prise en compte. On peut donc voir la variable `futex_state_prio` comme un offset qui décale la priorité de base (`DEFAULT_PRIO` = 120) sur lequel le nice base son calcul.

### 5.2.4 Changement de propriétaire

Lorsqu'une tâche détenant un verrou le relâche, celui-ci réveille toutes les autres tâches en attente du verrou. La tâche qui sera le nouveau propriétaire le déclare avec la fonction :

```
fixup_state_owner_current(state);
```

Cette fonction va notamment modifier le `owner` de `state`, décrémenter son poids, ajouter le `state` à sa liste, et ainsi pouvoir appliquer sa priorité basé sur ces changements avec `futex_state_prio(task)`. Le champs `waiting_futex_state` de la tâche courante est marqué à `NULL` pour indiquer que celle-ci n'attend plus sur un futex. Voici le code de la fonction :

```
int fixup_state_owner_current(struct futex_state *state)
{
    int sumload = get_futex_state_sumload(current);
    state->owner = current;
    add_futex_state(state);

    state->load -= (sumload + 1);

    futex_state_prio(state->owner);

    kref_put(&state->refcount, free_futex_state);
    current->waiting_futex_state = NULL;
```

```
    return 0;
}
```

Il faut noter que le poids est décrémenté de 1 plus la somme des poids des `futex_state` que le nouveau propriétaire détient avant l'ajout du `state`. En effet, en se bloquant on a hérité notre `sumload` au `state`, en devenant son propriétaire il faut donc déshériter ce même `sumload`.

### 5.2.5 Relâchement d'un verrou

Lors du relâchement d'un verrou la tâche courante, le propriétaire, doit récupérer le `futex_state` associé :

```
fetch_futex_state(current, key, &state);
```

Si le `state` existe alors il est retiré de la liste :

```
del_futex_state(state);
```

Après ces changements la priorité de la tâche courante doit être mis à jour :

```
futex_state_prio(current);
```

Ainsi, si la tâche relâche son dernier verrou alors sa liste devient vide et l'appel à `futex_state_prio(task)` annulera les modifications de priorité apportées.

### 5.2.6 Mort subite

Il se peut qu'un processus meurt alors qu'il attend sur un verrou. De la même manière que lors de l'attente sur un verrou, il faut cette fois-ci décrémenter par récurrence sur l'arbre le poids des `futex_state`. Une modification de la fonction `do_exit` dans `<kernel/exit.c>` a donc été nécessaire :

```
if (tsk->waiting_futex_state != NULL) {
    futex_state_inherit(tsk, tsk->waiting_futex_state,
                        FUTEX_STATE_UNLOAD);
    kref_put(&tsk->waiting_futex_state->refcount,
             free_futex_state);
}
```

Si la tâche courante attend sur un futex on applique donc un deshéritage et on relâche la référence sur le `futex_state`.

## 5.3 Shrinker

Les shrinker permettent une optimisation mémoire. Dans notre cas leur utilisation n'est pas possible. En effet, on ne peut pas se permettre de supprimer ou bien de libérer une référence d'une structure `futex_state` en cours d'utilisation, cela peut compromettre tout le mécanisme de verrouillage des futex.

# 6 Benchmark

Pour évaluer notre implémentation nous avons réalisé un petit benchmark qui se charge de simuler un scénario basique.

**Scénario :** Nous voulons tester un scénario comparable à celui présenté dans la partie **4.1 Monocœur**. Un thread parent prend un mutex et lance l'exécution de N threads enfant se bloquant sur son mutex. Une fois les enfants lancés, le parent exécute son calcul. X autres threads s'exécutent en parallèle du thread parent et effectue leur calcul. Le programme s'arrête brutalement quand le parent a fini l'exécution de son calcul.

**Calcul des threads :** Calcul du déterminent d'une matrice 11x11.

**Résultat attendu :** Les N threads enfant doivent augmenter la priorité du thread parent de N. Le thread parent doit donc voir son temps d'exécution significativement réduit en fonction de N et finir avant les X threads parallèles.

**Environnement de test :** Machine virtuelle avec 1 cœur.

**Résultat :** Temps d'exécution **sans** notre implémentation des futex state :

```
real    0m43.359s
user    0m43.326s
sys     0m0.011s
```

Le thread parent ne fini pas avant les threads parallèles.

Temps d'exécution **avec** notre implémentation des futex state :

```
real    0m4.406s
user    0m4.396s
sys     0m0.003s
```

Le thread parent fini bien avant les threads parallèles.

Par manque de temps nous n'avons pas pu évaluer notre implémentation sur des scénarios plus complexes, notamment en jouant avec le multicœur.

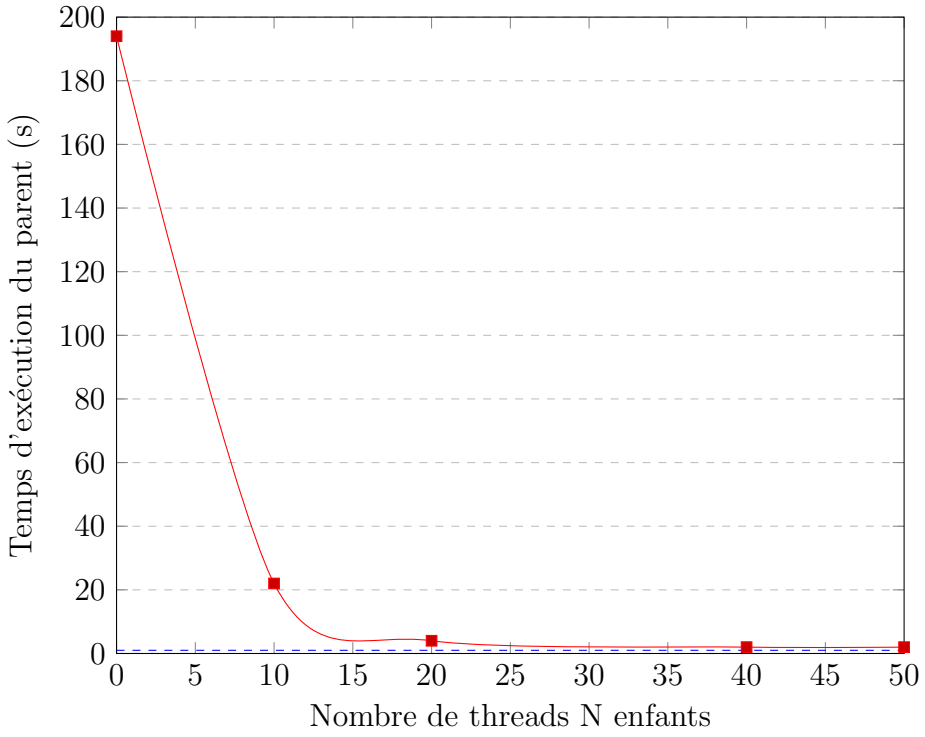


FIGURE 2 – Évolution du temps d'exécution en fonction des N threads enfants

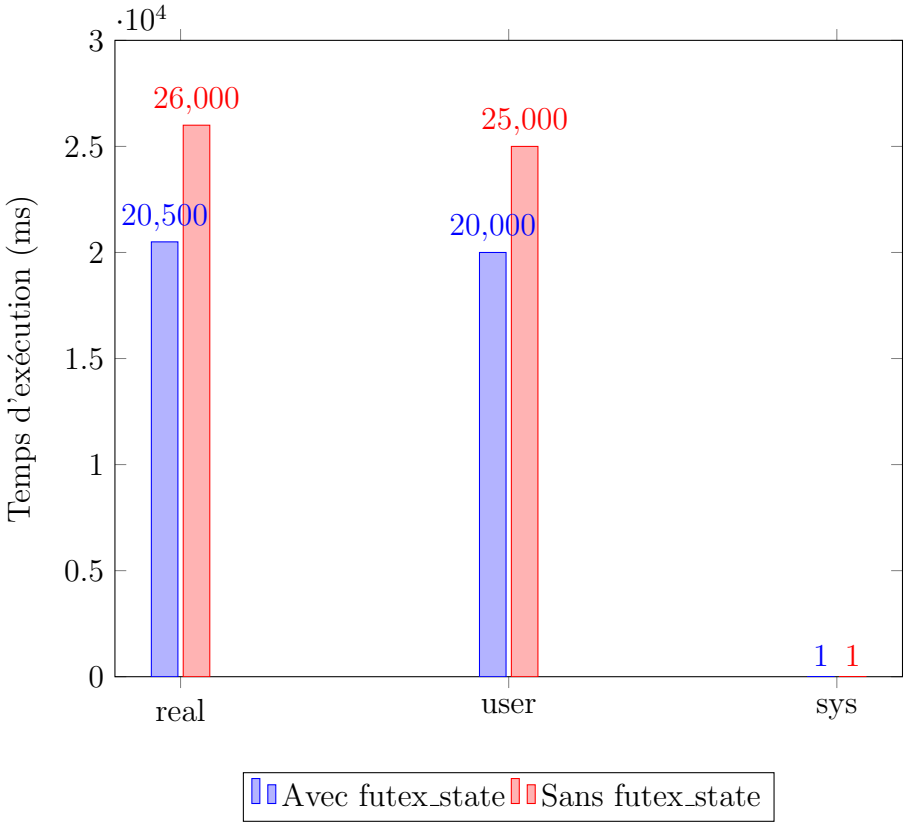


FIGURE 3 – Overhead

# 7 Discussion

Nous concluons ce projet par un bilan sur sa réalisation et ses résultats ainsi qu'une discussion sur ses limites et les améliorations envisageables.

## 7.1 Réalisation du projet et limite

La réalisation du projet s'est bien déroulé, toutes les fonctionnalités attendues on été implémentées. Les résultats sur l'évaluation de notre mécanisme sont globalement concluant, bien que par manque de temps nous aurions préféré affiner nos benchmarks.

Cependant le projet a montré des limites, notamment sur la manipulation de la liste `futex_state_list` d'une tâche. Son utilisation requière la prise du verrou `futex_state_lock`. Ainsi, lors de la recherche, d'un ajout ou d'une suppression d'un `futex_state` sur la liste mais aussi lors du calcul de la somme, le verrou est pris. Cela empêche d'autres tâches bloquées sur la même tâche propriétaire d'y accéder. Créant ainsi des groupes de tâches à cette prise de verrou commun. Il y a donc un risque de contention entre ces tâches.

Dans les premières implémentations nous avions créé un liste globale, recensant l'ensemble des `futex_state` quelque soit leur propriétaire. Sa manipulation était donc commune à toutes les tâches, en plus du manque de performance de cette méthode, le problème de contention était d'autant plus présent. L'utilisation des listes, dites locales, à chaque propriétaire permet de réduire ce problème de contention. Cependant le risque persiste et aucune solution viable nous semble contourner ce problème pour le moment.

## 7.2 Travaux à court terme

Par manque de temps certaines tâches n'ont pas pu être réalisées dans le temps imparti. Leur implémentation n'ont pas d'impact majeur et elles ont donc été laissé pour les travaux à court terme.

### 7.2.1 Poids

La clé de voûte de notre implémentation est son système de poids. Lors de l'héritage nous avons vu que la tâche qui va se bloquer doit calculer la somme des poids des `futex_state` qu'elle détient, représenté ensuite par une variable `sumload`. Ce calcul est effectué plusieurs fois, et demande de parcourir la liste des `futex_state` de la tâche. Or, nous pouvons réduire ces calculs par un champs dans la structure `task_struct` qui représenterai la valeur `sumload` à tout instant. Cette variable serait mis à jour lors de l'héritage, deshéritage, ajout et suppression de `futex_state`.

### 7.2.2 Généralisation

Par manque de temps l'implémentation n'a été qu'effectué que pour l'utilisation des mutex de type PI (héritage de priorité). Un travail à court terme serait de le rendre fonctionnel pour l'utilisation des mutex de type ROBUST, soit le protocole par défaut.

Cette décision est une question de débogage, nous permettant de tester notre implémentation volontairement avec des programmes tests en utilisant des mutex PI. En effet, en implémentant le mécanisme, en cours de développement, pour le type ROBUST le kernel l'aurait utilisé, ce qui aurait compliqué le débogage lors d'éventuelles erreurs.

## 7.3 Travaux à long terme : les priorités

L'implémentation actuelle du changement de priorité d'une tâche par la fonction `futex_state_prio(task)` est très simpliste. Elle réside en une réduction de la `static_prio` par le `sumload` borné sur 20.

Pour rendre plus juste la modification de la priorité nous pouvons envisager l'utilisation d'une formule. Une approche peut consister à attribuer un coefficient à chaque tâche bloquée. Ce coefficient est calculé en fonction de la priorité de la tâche. Plus la tâche est prioritaire et plus son coefficient est élevé. Le poids qu'ajoutera la tâche bloquée au `futex_state` du propriétaire dépendra de son coefficient. Ainsi par exemple, une tâche de priorité 120 ajoutera 1 au poids, tandis qu'une tâche de priorité 100 ajoutera 2.

Ce système permet d'être encore plus équitable entre les tâches. Deux tâches de même priorité bloquant le même nombre de tâches auront un `sumload` différent en fonction des priorités des tâches qu'elles bloquent.

Cette nouvelle approche complexifie le mécanisme, notamment si une tâche bloquée voit sa priorité modifiée lorsqu'elle dort, son coefficient devra être mis à jour en conséquent.