Rapport de stage
Étude et écriture d'un ordonnanceur

LIP6

Raïssa BOURI - 28602820

2024

Contents

1 Introduction

Ce stage s'inscrit dans le cadre de la thèse de Baptiste Pires au sein de l'équipe Delys du LIP6 qui étudie principalement les systèmes répartis, sa thèse vise à analyser et modifider l'ordonnanceur de Linux et est dirigée par Julien Sopena.
L'objectif principal de mon stage était de porter un algorithme d'ordonnancement sous forme de module dans le noyau. En premier lieu, c'est CFS qui devait être porté, puis, après une longue étape de documentation nous avons décidé de porter un autre ordonnanceur : Rusty.

2 L'ordonnanceur

2.1 Son importance

Dans un contexte multi-coeur, les ordonnanceurs doivent non seulement décider quel processus élir mais également choisir sur quel coeur il doit s'éxecuter. De plus, les architectures de type NUMA ainsi que la hiérarchisation de la mémoire avec les caches font que cette tâche est très complexe, en plus d'être cruciale en terme de performance puisqu'elle permet de répartir les ressources.
Aujourd'hui, de nombreux algorithmes d'ordonnancement existent dont le but principal est d'être équitable et sans famine, cependant, certaines applications dépendent d'autres critères selon des besoins différents : certaines sont sensibles à la latence, d'autres doivent être le plus intéractives possible ou encore respecter au mieux des deadlines.
Il est possible de développer son propre ordonnanceur pour répondre à ces besoins, cependant, cela nécessite une grande compréhension du kernel ainsi que de l'architecture des processeurs notamment concernant la mémoire et les caches, ce qui représente déjà un travail complexe, à cela s'ajoutent des connaissances en debuggage. Il faut également garder en tête que même si l'ordonnanceur fonctionne sans crash il peut être à l'origine de bugs de performance qui passent souvent innaperçus et sont difficiles à trouver.
Avec toutes ces contraintes les développeurs se tournent souvent vers des ordonannceurs génériques comme CFS.

2.2 CFS et EEVDF

CFS, le **C**ompletly **F**air **S**cheduler, est l'ordonnanceur par défaut de Linux depuis 2007. Son objectif principal est d'émuler un CPU "*ideal et multi-tâche*", celui-ci doit répartir le CPU entre les différents processus de la manière la plus équitable possible. Pour cela, un *quantum* est assigné à chaque thread selon son *poids*: un poids important signifie un plus grand quantum, cette importante notion est déterminée selon la valeur du paramètre ***nice***. Ce paramètre représente un handicap pour un thread, une sorte de priorité : plus sa valeur est importante plus sa priorité est basse, ainsi, un thread avec une petite valeur *nice* possède un poids important et donc une grande priorité.

Au moment de l'élection, CFS cherche le thread qui a le plus **besoin** du CPU, la notion de ***vruntime*** est donc introduite: cette valeur représente le temps d'exécution d'un thread ajustée par le poids, c'est un indicateur qui permet de montrer quels threads ont reçu le plus de temps d'éxecution. Le thread possédant le vruntime le plus petit sera élu plus rapidement qu'un thread possédant un vruntime élevé. D'autres fonctionnalités sont

implémentées comme le ***load-balancing*** par exemple, qui permet de répartir les processus entre les CPUs pour qu'aucun CPU ne soit sous-utilisé.

À présent, prenons le cas de tâches courtes qui ont urgemment besoin du CPU face à des tâches plus longues qui n'ont pas nécessairement besoin d'être élues rapidement. Avec CFS, il n'y a aucun moyen de différencier ces tâches puisque leur temps d'exécution n'entre pas dans le calcul du vruntime. Il serait possible d'utiliser la valeur de nice pour baisser la priorité des tâches plus longues mais ça ne répond pas entièrement au problème: le temps pour obtenir le CPU et non la valeur du quantum.

Un ordonnanceur propose une solution: **E**arliest **E**ligible **V**irtual **D**eadline **F**irst. Comme CFS, cet ordonnanceur se veut le plus équitable possible, cependant, il prend également en compte le temps d'éxécution réel d'un processus, pour certaines raisons, les processus ne consomment pas toujours leur quantum, ils peuvent consommer plus ou moins de temps que prévu. EEVDF prend en compte cette différence, c'est le ***lag***: un lag positif indique qu'un processus n'a pas reçu la part qui lui était dûe et devrait être élu avant un processus possédant un lag négatif ayant consommé plus que son quantum. Cela est assuré car dans EEVDF un processus n'est éligible que si son lag est égal ou supérieur à zéro, les processus inéligibles se voient attribuer un ***eligible time***, un moment futur à partir duquel ils pourront redevenir éligibles puisque ce temps d'attente aura compensé leur surplus de temps d'exécution.

Cet algorithme introduit une seconde notion : la ***virtual deadline***, qui représente le moment à partir duquel un processus aurait dû être élu en additionnant son *eligible time* et son quantum. Par exemple, un processus avec un *eligible time* de 30ms et un quantum de 10ms aurait une *virtual deadline* de 40ms.
Lors de l'éléction, ce sont les processus avec la *virtual deadline* la plus petite qui sont élus en premier; les processus avec un petit temps d'exécution auront donc une petite virtual deadline et seront élu plus rapidement que les autres. De ce fait, le problème soulevé plus tôt est résolu: les processus sensibles au temps de latence auront un accès au CPU rapide.
Depuis Octobre 2023 EEVDF a été patché dans la version 6.6 du kernel, et ne remplace pas CFS mais le complète, permettant d'améliorer la performance de l'ordonnanceur en terme de latence.

Un autre ordonnanceur étudié pendant ce stage est EEVDF BORE: **B**urst-**O**riented **R**esponse **E**nhancer, un patch visant a améliorer CFS et EEVDF en introduisant la notion de ***burstiness***. Pour chaque processus, un ***burst time*** est calculé, c'est le temps d'exécution depuis la dernière fois où le processus a été à l'état sleep ou yield, les tâches qui ont été le moins dormantes sont élues en priorité, la réactivité du système est alors améliorée ainsi que l'expérience des utilisateurs.
Essayer de comprendre et d'isoler les parties principales de ces algorithmes a été long et complexe et m'a fait réaliser que beaucoup de temps était nécessaire afin de réaliser cette tâche. Néanmoins, j'ai pu réaliser un diagramme qui permet d'identifier les différentes structures, variables et fonctions qui permettent de calculer les paramètres principaux de CFS-EEVDF ainsi que les fonctions qui les modifient, sur la **Figure 1** en annexe on peut voir six fonctions appelées lors de la mise à jour du vruntime.

L'objectif final étant d'avoir un ordonnanceur fonctionnel à la fin de ce stage, j'ai dû

faire des choix après cette étape de documentation, le plus important étant d'obtenir un produit fini et à améliorier, après avoir discuté avec mon encadrant nous avons décidé d'étudier un autre ordonnanceur et c'est celui-ci que j'ai porté : Rusty.

3 Réalisation

Nous avons vu que les ordonnanceurs impactent fortement les performances du système et qu'ils nécessitent de bien comprendre des notions complexes, afin de simplifier leur écriture, nous utilisons SaaKM, développé par Redha Gouicem lors de sa thèse au Lip6.

3.1 Ipanema et SaaKM

Ipanema est un **D**omain **S**pecific **L**anguage permettant d'écrire des *politques d'ordonnancement*, de cette façon, les développeurs ayant peu de connaissances du kernel peuvent écrire ordonnanceurs adaptés à leurs applications. Cela est possible grâce aux abstractions fournies par Ipanema, de plus, les politiques écrites pas ces développeurs sont vérifiées car elles doivent respecter certaines propriétées telles que l'équité ou le fait qu'il n'y ait pas de famine.
Dans le cadre de mon stage je n'ai pas eu à utiliser Ipanema mais plutôt SaaKM : **S**cheduler **a**s **a** **K**ernel **M**odule, une fonctionnalité qui permet d'ajouter ou de retirer plusieurs politiques d'ordonnancement sous forme de modules dynamiquement, cela permet un gain de temps important puisqu'il n'est plus nécessaire de recompiler son noyau à chaque modification de l'ordonnanceur et il est possible de charger plusieurs politiques d'ordonnancement simultanément, ainsi, différentes applications peuvent être lancées en parallèle avec des ordonnanceurs différents.
Pour écrire notre ordonnanceur avec SaaKM, il suffit de définir les fonctions offertes par son API, comme **new_prepare** qui est appelée lorsqu'un processus est crée ou bien **new_place** pour placer un processus sur un coeur.

3.2 Approche

Pour pouvoir commencer à définir les comportements des fonctions de SaaKM il faut bien isoler les points principaux de l'algorithme à porter.
Un bon moyen d'y parvenir est de répondre à ces questions:

- Dans quelle structure sont placés les processus? FIFO, arbre ?
- Lors de l'élection, comment est choisi le processus à élir ?
- Sur quel CPU le processus doit-il être placé ?
- Les processus sont-ils ordonnés ? si oui, d'après quel paramètre ? vruntime, load ?

Concernant Rusty, c'est un ordonnanceur fonctionnant sur Sched_ext, une fonctionnalité du kernel permettant de charger dynamiquement des ordonannceurs en BPF.
Rusty est composé de deux parties:

- Une partie BPF qui effectue le calcul du vruntime et du load des processus, cette partie s'occupe également de la gestion des domaines.

- Une partie userspace qui traite deux tâches à différentes fréquences:
  - À haute fréquence : des décisions de tuning, les CPUs les moins utilisés sont identifés et ces données sont stockées dans des masques et dans des structures
  - À basse fréquence : si nécessaire, le **load-balancing**, lorsqu'il y a d'importantes différences de poids entre les CPUs. Ces différences sont calculées d'après les données récoltées lors du tuning.

Rusty vise à être flexible et à accomoder différents besoins et différentes architectures, il permet donc aux utilisateurs de définir les valeurs de divers paramètres et seuils, par exemple il est possible de désactiver le load-balancing, de choisir si les processus doivent être ordonnés selon leur vruntime ou placés dans une FIFO, entre-autres.

La notion de domaine a été mentionnée, elle fait référence à la topologie qui hierarchise les caches et coeurs de l'architecture. Dans le cas de Rusty, dont on peut voir la topologie suivie dans la **Figure 2** de l'annexe, si l'architecture n'est pas NUMA, alors on ne considère qu'un seul noeud comportant tous les CPUs.

Les données concernant ces domaines sont stockées dans des tableaux en userspace et mit à jour par le tuning, la partie BPF les récupère dans le noyau.
À chaque domaine est associée une structure **dom_ctx** qui contient trois pointeurs sur des masques CPU : *cpumask*, *direct_ greedy_ cpumask* et *node_ cpumask*.
En plus des masques contenus par ces structures, il existe également un masque partagé par tous les coeurs : *direct_ greedy_ cpumask*. Les masques *direct_ greedy_ cpumask* indiquent quels CPUs sont sous-utilisés, ils sont mis à jour par la partie tuning: à chaque appel de la fonction **step()** du *tuner*, les statistiques des différents CPUs contenues dans le répertoire ***proc*** du kernel sont récupérées comme le temps passé en mode *system*, *user* ou *io_ wait*. Ces valeurs sont additionnées et selon un seuil que l'utilisateur peut définir un CPU est jugé sous-utilisé ou sur-utilisé. Cette information est mise à jour dans *direct_ greedy_ cpumask* et sera importante lors du choix du CPU, elle est également mise à jour dans le masque *direct_ greedy* de chaque domaine.
En plus des tableaux contenant les informations sur les domaines, il existe des tableaux pour les noeuds et les tâches contenant respectivement des structures **node_ctx** et **task_ctx**.

Lorsqu'une tâche est créée, il faut choisir sur quel CPU elle va être exécutée, pour cela, Rusty suit un ordre de priorité:

- Sur le coeur courant si le processus a été réveillé de façon synchrone et qu'il existe des CPUs idle dans ce domaine
- Dans le domaine du CPU précèdent s'il est idle
- Si une migration a été demandée par la partie userpsace, dans le domaine du coeur spécifié
- Sur n'importe quel domaine courant si aucunes des conditions précèdentes n'est validé

Une fois que ces principales notions ont été comprises, il est possible de commencer à implémenter cela avec l'API de SaaKM et de réfléchir aux choix d'implémentation à faire et de la stratégie à suivre.

3.3 Fonctionnalités implémentées et fonctionnelles

Le plus important avec Rusty est le choix du CPU sur lequel on place un processus, c'est la fonction **select_cpu()** qui représente la base de cet algortihme et qui a été la plus longue a implémenter.
Ensuite il a fallu choisir quelle stratégie de Rusty implémenter: FIFO ou runqueue avec poids et vruntime ? load-balancing ou pas? J'ai commencé par une stratégie simple pour pouvoir tout implémenter dans le temps imparti:

- Un ordonnanceur avec FIFO
- Pas de load-balancing
- Le masque global *direct_greedy* pour les CPUs sous-utilisés
- Un thread qui effectue le tuning et met à jour ce masque
- Lors de la création d'un processus on lui attribue un domaine, pour cela, les domaines sont round-robined et l'indice du domaine à choisir est incrémenté à chaque attribution

Comme mentionné précèdemment, plusieurs tableaux en userspace sont utilisés pour stocker des données sur les coeurs et domaines, l'API de SaaKM permet d'obtenir ces informations donc ils n'ont pas été copiés.
Ensuite, il faut regarder le comportement de l'ordonnanceur dans le cas d'un processus qui *yield* ou *sleep*, ou encore copier leur gestion du quantum.

L'ordonnanceur que j'ai écrit avec SaaKM fonctionne sans crash dans le noyau, il ne représente qu'une stratégie de Rusty mais il m'a permit d'implémenter les mécanismes de base de cet algorithme.

3.4 Fonctionnalités à implémenter

En plus du load-balancing qui n'a pas été implémenté, Rusty possède une runqueue globale, qui est utilisée dans les cas où les runqueues des coeurs ne possèdent pas de processus prêts.

La flexibilité de Rusty a été mentionnée et avec elle le fait que l'utilisateur puisse définir des seuils et choisir des options, cela peut être implémenté avec des paramètres à entrer lors du chargement du module.
Même si l'ordonnanceur fonctionne, il peut être responsable de bugs de performance qu'il faut identifier et trouver, afin de le comparer à Rusty il faut le compléter et l'analyser.

4 Annexe

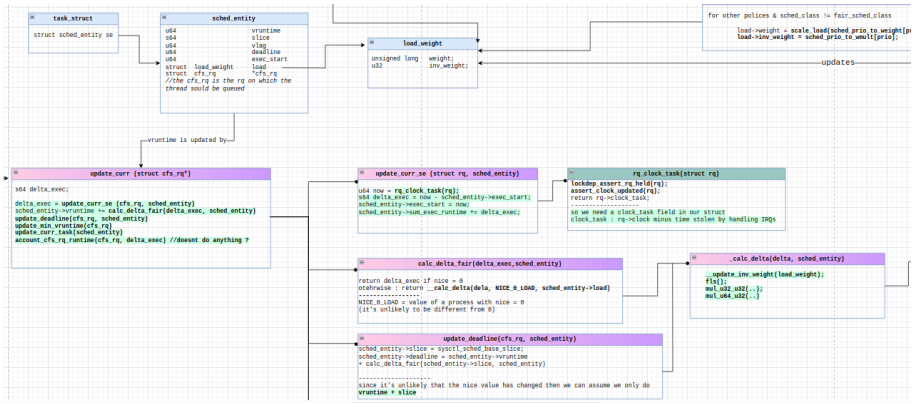

Figure 1: Extrait d'un diagramme illustrant les fonctions clées du calcul du vruntime

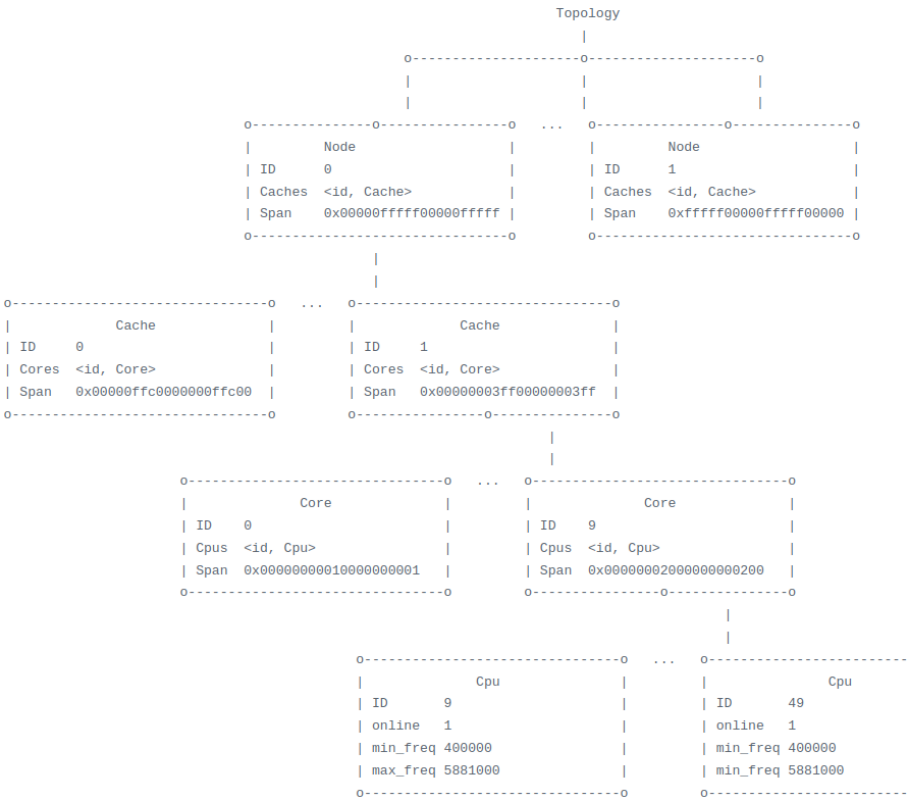

Figure 2: Enter Caption