

Rapport de stage Ingénieur

Implémentation d'un ordonnanceur temps réel sur plateforme multi-cœur hétérogène

BELPOIS Vincent

2023



Table des matières

Table des figures	2
1 Introduction	3
2 OS compatibles avec la carte	4
2.1 Présentation de la carte de développement	4
2.1.1 Le processeur RK3399	4
2.1.2 Interfacer avec la carte	4
2.2 Installation d'un système d'exploitation	5
2.2.1 Installation d'une image pré-compilée	5
2.2.2 Compilation de Linux depuis le code source	5
2.2.3 Compilation croisée	7
2.3 Etude des versions de Linux compatibles	8
2.3.1 Comment Linux gère le support d'un processeur	8
2.3.2 Essais de différentes versions	8
3 LITMUS^{RT}	10
3.1 Présentation de LITMUS ^{RT}	10
3.2 Présentation de <i>feather-trace</i>	10
3.3 Implémentation d'un ordonnanceur EDF partitionné	11
3.3.1 Algorithme considéré	11
3.3.2 Implémentation	12
3.3.3 Résultats et essais	12
3.4 Implémentation d'un ordonnanceur RM partitionné	13
3.4.1 Implémentation	13
3.4.2 Résultats et essais	13
4 Génération de tâches	15
4.1 Mesure de temps d'exécution	15
4.2 Génération de tâches répétables	15
4.2.1 Première idée : <i>checksum</i> d'un fichier	15
Conclusion	16
Annexe	17
Bibliographie	29
Glossaire	29

1 Introduction

J'ai pu réaliser mon stage ingénieur, durant ma deuxième année d'étude à l'ISAE-ENSMA, au laboratoire du LIAS de Chasseneuil-du-Poitou. Le LIAS, ou Laboratoire d'Informatique et d'Automatique pour les Systèmes, regroupe plusieurs dizaines d'enseignants chercheurs dans les domaines de l'automatique, le génie électrique et l'informatique. Le site de Chasseneuil-du-Poitou regroupe deux équipes, l'équipe Ingénierie des Données et des moDèles (IDD) et l'équipe Systèmes Embarqués Temps Réel (SETR).

J'ai été accueilli au sein de cette dernière afin de travailler avec **Antoine BERTOUT** et **Thomas GASPARD** sur le projet SHRIMP. Mon stage s'intéresse à l'implémentation d'un ordonnanceur sur plateforme hétérogène[1], tandis que le reste du projet s'intéresse entre autre à la conception d'un ordonnanceur temps réel global et dynamique pour des plateformes *unrelated*.

Mon stage aura donc pour objectif d'identifier une solution permettant de programmer un ordonnanceur temps réel sur une carte de développement particulière. Il me faudra ensuite étudier les mécanismes les plus adaptés pour la migration tâches sur cette plateforme. L'objectif final sera alors d'implémenter une politique d'ordonnancement temps réel globale hétérogène naïve, voir plus évoluée en fonction de l'avancement des travaux de thèse.

2 Systèmes d'exploitation compatibles avec la carte ROCK960

Afin d'étudier les systèmes d'exploitations temps réel compatibles avec la carte de développement qui m'a été fourni, je me suis d'abord familiariser avec celle-ci en installant des OS fournis par le fabriquant avant d'étudier la compatibilité avec des systèmes plus complexes.

2.1 Présentation de la carte de développement

Le stage s'intéressant à l'implémentation d'un algorithme d'ordonnancement sur une plateforme hétérogène, une carte possédant un tel processeur est mis à ma disposition. Cette carte se nomme ROCK960 et est fabriquée par l'entreprise *96Boards*. Cette carte de développement contient de nombreuses interfaces mais nous nous contenteront d'utiliser l'interface Série TTL à laquelle nous nous connecterons via un convertisseur USB vers TTL. Cela me permettra d'interfacer via un terminal qui fonctionnera avec une liaison série.

2.1.1 Le processeur RK3399

Au centre de la carte est un SOC Rockchip RK3399. Ce processeur contient deux type de cœurs, ou processeurs. Deux d'entre eux sont des processeurs Cortex-A72 et les quatre autres sont des processeurs Cortex-A53. Ces 6 processeurs utilisent le même jeu d'instruction : ARMv8-A 64-bit. Cela sera important par la suite afin de faciliter la migration de tâche entre les processeurs, en effet si les jeux d'instructions des processeurs étaient différents, plusieurs copies du code compilé devrait exister tout en maintenant un lien d'équivalence entre les deux codes. Cela est bien au delà de la portée de mon stage mais sera un point intéressant à explorer.

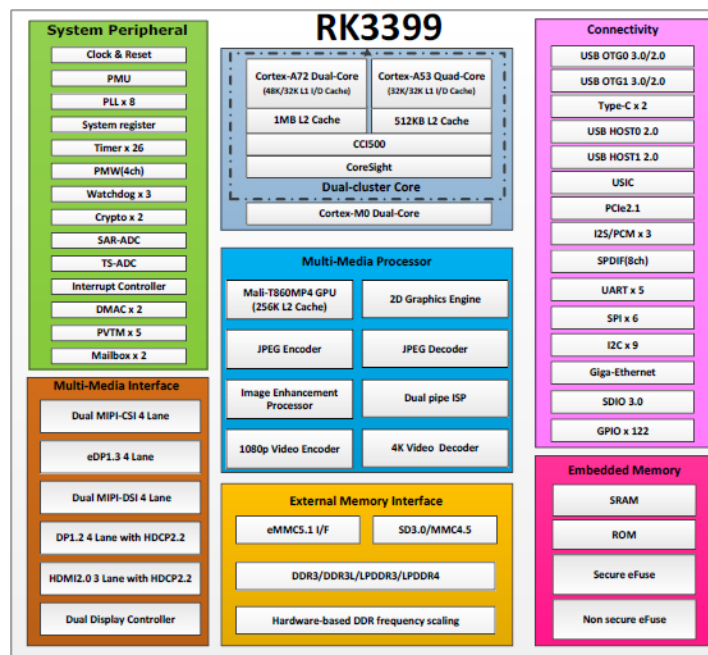


FIGURE 1 – Architecture du processeur RK3399

Le SOC RK3399 contient bien d'autres composants et peut interfacer avec de nombreux périphériques (écran HDMI, USB, caméra MPI-CSI, SPI, UART, I2C, etc...) comme le montre la figure 1. Ce diagramme nous montre aussi que les deux cluster de processeurs ne partagent pas les cache L1 ni L2 mais sont interconnectés par une interface CCI-500 qui, selon le site des développeurs ARM, permet la cohérence des caches des deux clusters.

2.1.2 Interfacer avec la carte

Expliquer comment on interface, les différentes options, pk est ce que l'on reste avec du Serial

2.2 Installation d'un système d'exploitation

Le code que l'on veut exécuter sur la carte de développement doit être compilé pour celle-ci puis placé sur un support de stockage. Ce code peut être un simple programme ou bien un système d'exploitation complet. J'ai premièrement installé des images pré-compilées de Linux, puis j'ai compilé moi-même le noyau Linux depuis son code source en réalisant une compilation croisée.

2.2.1 Installation d'une image pré-compilée

Pour premier tester l'Installation de linux sur la carte de développement, j'ai utilisé une image de la distribution Ubuntu fournie par le fabricant 96Boards disponible sur leur site. Cette image se présente sous la forme d'une archive au format `.tar.gz`. Elle contient à la fois le bootloader, le noyau Linux, et le système de fichier. Cette image (`system.img`) peut alors être gravée (ou *flashée*) sur une carte micro SD.

Depuis un terminal, en se déplaçant dans le dossier de l'archive extraite, on exécute la commande suivante :

```
1 $ sudo dd if=system.img of=/dev/XXX bs=4M oflag=sync status=noxfer
```

Listing 1 – Linux Command

EXPLIQUER CE QUE FAIT CETTE COMMANDE

Aussi dire en quoi on s'en servira dans des scripts afin d'accélérer le développement.

2.2.2 Compilation de Linux depuis le code source

Afin d'utiliser une version de Linux différente de la version précompilé par le fabricant de la carte de développement, il faut se premièrement se procurer le code source du noyau Linux. Celui-ci est disponible sur un dépôt de code git hébergé par GitHub. Il est disponible à l'adresse <https://github.com/torvalds/linux>, sous le profile du créateur de Linux : Linus Torvalds. Durant mon stage j'étais libre d'utiliser le logiciel de gestion de version de mon choix, j'ai donc principalement utilisé *git* en ligne de commande et j'ai parfois utilisé un client git nommé *GitKraken* afin plus facilement explorer les anciens commits de certains projets comme LITMUS^{RT}.

Une fois le code source du noyau téléchargé, et en se déplaçant dans le dossier `linux` depuis un terminal, on peut alors procéder à la compilation. Pour mes premiers essais j'ai premièrement décidé de compiler Linux pour une machine virtuelle que je ferai tourner sur ma machine de travail.

Le noyau Linux est un programme ayant une compilation basée sur la configuration : cela signifie que certaines parties du code peuvent rajouter ou omises par le simple biais d'un fichier de configuration. Ce configuration se présente sous la forme d'un fichier `.config` qui doit être créé à la racine du noyau. Pour créer ce fichier, des utilitaires sont mise à notre disposition dans le noyau :

- `make defconfig` : cet outil est utilisé pour générer une configuration par défaut. Ici l'architecture de la machine qui réalise la compilation sera sélectionnée. Dans mon cas, exécuter cette commande créer un fichier de configuration basé sur la config '`x86_64_defconfig`'.
- `make menuconfig` : cet outil permet d'éditer la configuration actuelle du fichier `.config` via une interface graphique. Ce menu permet aussi de rechercher des paramètres, de voir leur description et d'enregistrer différentes configurations.

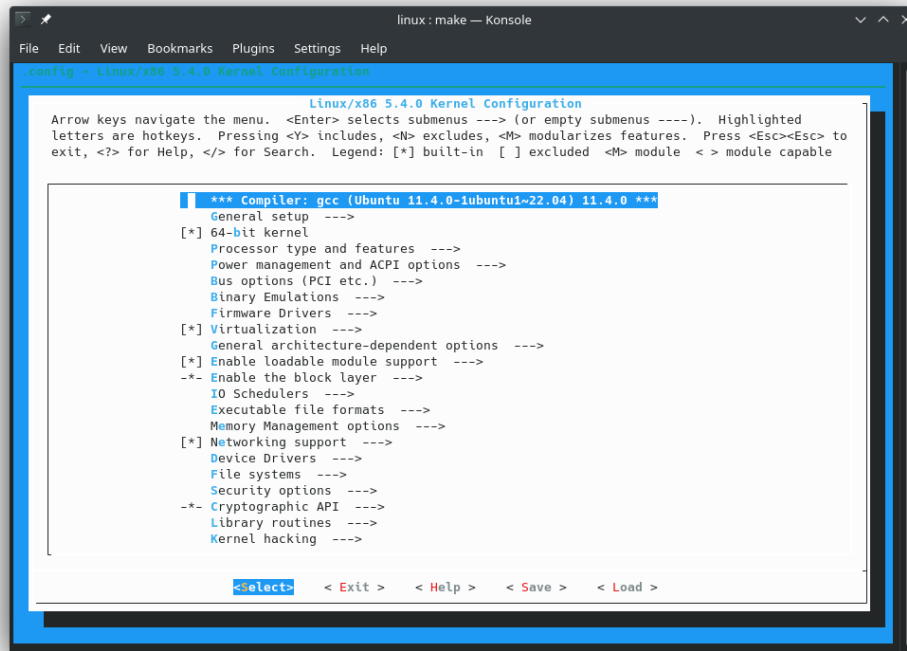


FIGURE 2 – Interface de configuration du noyau

EXPLIQUER ou est exactement stockée cette config
expliquer que l'on peut stocker ce fichier manuellement.

Une fois la configuration créée, nous pouvons passer à la compilation du noyau. Linux utilise l'utilitaire de compilation *GNU Make* : il permet l'automatisation de la compilation, la gestion des dépendances et gère la personnalisation de la compilation de chaque dossier. Ces règles sont alors dictés par des fichiers *MakeFile* présent dans chaque dossier contenant des fichiers à compiler du projet.

Note : chaque distribution Linux possède un ensemble différent de programmes préinstallés, il faudra alors peut-être installer des programmes nécessaires à la compilation. Par exemple, installer *libelf-dev* "une bibliothèque partagée qui permet de lire et écrire des fichiers ELF à un niveau élevé" ¹.

```
1 $ make -j16
```

Listing 2 – Compilation sur plusieurs processeurs

Le paramètre `-j 16` signifie que l'on veut exécuter la compilation avec 16 tâches en parallèles. Il est recommandé sur internet par beaucoup d'utiliser comme nombre de tâches, le double du nombre de processeurs dans l'ordinateur qui réalise la compilation.

Par la suite il me sera parfois nécessaires de changer la version de Linux que je compile afin de tester si celle-ci fonctionne. Sur le dépôt de code de linux, les différentes versions sont stockées sous forme d'un certain commit qui a été "tagé" afin de le retrouver. On peut alors changer de version en revenant à ce commit grâce à la commande `checkout` de git :

```
1 $ git checkout v5.4
```

Listing 3 – Retour sur un commit tagé

On peut avoir la liste de ces commits tagés de la manière suivante :

```
1 $ git tag -l
```

Listing 4 – Comment lister les tags

1. d'après la description sur packages.debian.org/fr/sid/libelf-dev

On aura alors l'ensemble des tags de tout le dépôt de code, et on peut filtrer ces résultats avec `grep` par exemple si l'on veut retrouver une version particulière.

Bien que j'étais déjà familier avec git, cela m'a pris un certain temps de comprendre comment ce changement de version s'effectuait. La nuance que l'on ne changeait pas de branche dans le dépôt, mais que l'on revenait simplement au commit correspondant à la version était la plus compliquée à comprendre. Tout au long de ce stage j'ai pu utiliser git afin d'explorer comment certains projet ont été construits en remontant l'historique de leurs commits, mais j'ai pu aussi utilisé git pour gérer le stockage du code que j'ai développé, que ce soit des outils ou des modifications du noyau.

2.2.3 Compilation croisée

Nous ne pouvons malheureusement pas compiler directement le noyau linux pour la carte de développement dû à la différence de jeu d'instruction. Il faut donc réaliser une *cross compilation* ou compilation croisée : c'est le fait de compiler un programme sur une architecture qui n'est pas celle de l'architecture cible.

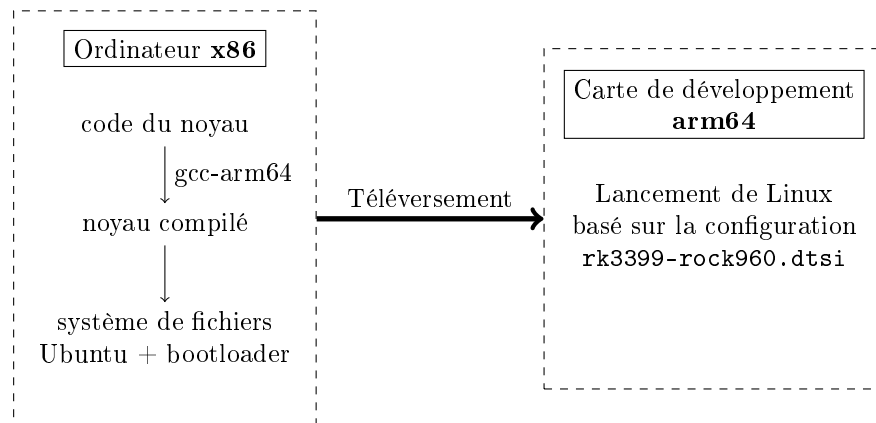


FIGURE 3 – Compilation croisée du noyau Linux

Dans notre cas, l'ordinateur qui réalisera la compilation a un jeu d'instruction `x86` tandis que la carte de développement a un jeu d'instruction `arm64`. Lors de la compilation du noyau Linux, on doit alors fixer des variables d'environnement comme l'architecture cible et le chemin vers la *toolchain* à utiliser :

```

1 ARCH="arm64"
2 CROSS_COMPILE="../toolchain/bin/aarch64-linux-gnu-"
  
```

Listing 5 – Variables pour la compilation croisée du noyau Linux

Ces variables seront lues par le fichier `Makefile` principal du noyau. Par exemple, le compilateur qui sera utilisé est `aarch64-linux-gnu-gcc` et fait partie de la *toolchain* que j'ai pu me procurer sur le site de Linaro². Cette *toolchain* contient entre autre un compilateur qui s'exécutera sur une machine `x86` et compilera vers une architecture `arm64`. Cette *toolchain* contient aussi des bibliothèques propres à l'architecture cible, comme `arm_neon.h` par exemple (bibliothèque qui permet d'utiliser les unités de calcul NEON sous les architecture ARM compatibles qui permettent le calcul en parallèle grâce aux instructions SIMD).

Le noyau linux est ainsi compilé par la commande :

```

1 make Image dtbs -j16
  
```

Listing 6 – Linux Command

Cela produit entre autre un fichier `Image` dans le dossier `litmus-rt/arch/arm64/boot`. Ce fichier est le noyau Linux compilé, non compressé, contenant les modules compilés en tant que `built-in-modules` mais ne contenant pas les modules externes. Ces derniers peuvent être compilés séparément et installés par une simple copie sur le système de notre choix.

². releases.linaro.org/components/toolchain/binaries/latest-7/aarch64-linux-gnu/gcc-*-x86_64_aarch64-linux-gnu.tar.xz

Une fois le noyau étant compilé, on peut le flasher sur la carte microSD que lira la carte de développement à son démarrage. Cependant, il faut joindre ce noyau a plusieurs autres programmes afin d’avoir un système d’exploitation utilisable :

- un *bootloader*, dans notre cas **u-boot**
- un système de fichiers et des programmes utilitaires, dans mon cas j’ai choisi une image minimale de Ubuntu³

Le *bootloader* est un court programme qui est chargé d’amorcer le système d’exploitation principal. Il est stocké dans une mémoire non volatile et est exécuté au démarrage de la carte de développement. Il est alors chargé de charger le noyau Linux et de lui passer la main. Dans notre cas, le fabricant de la carte de développement fourni un *bootloader* nommé **u-boot** qui est déjà compilé et qui est disponible sur leur dépôt de code. Il est alors possible de le télécharger et de le flasher sur la carte microSD.

Toolchain : qu’est ce que c’est, de quoi elle est constituée? Expliquer que l’on compile sur du x86 mais qu’on veut compiler pour du ARMv8xxx.

Variables d’environnement? Qu’est ce que c’est sous linux, comparer a des

Réalisation de scripts linux pour accélérer le développement. Que doit on charger pour charger le nouveau code compilé?

Copie de l’image du noyau pour faire encore plus rapide.

2.3 Etude des versions de Linux compatibles

2.3.1 Comment Linux gère le support d’un processeur

Pour gérer la compatibilité avec un processeur, le *bootloader* charge au démarrage de Linux le fichier *Device Tree* qui contient les informations sur le matériel présent sur la carte. Ce fichier est ensuite utilisé par le noyau Linux pour initialiser le matériel. Dans notre cas, le fichier **rock960-rk3399.dts** charge le fichier **rk3399.dtsi** qui contient les informations sur le processeur. On peut y trouver les informations sur la connectique, les périphériques, les contrôleurs, les bus, etc. La partie nous intéressant est celle sur la structure des processeurs qui se trouve dans le fichier **rk3399.dtsi**. On y trouve les informations sur les différents cœurs du processeur, leur fréquence, leur cache. C’est vers la fin de mon stage que j’ai pu me rendre compte d’un oubli dans le fichier décrivant ce processeur : les caches ne sont pas décrits. Cela a des conséquences sur les algorithmes d’ordonnancement utilisant cette information pour concevoir les clusters de processeurs les plus adaptés. En effet, les caches sont utilisés pour déterminer les coûts de migration d’une tâche d’un cœur à un autre. Sans cette information, les algorithmes d’ordonnancement ne peuvent pas déterminer les coûts de migration et ne peuvent donc pas déterminer les clusters les plus adaptés.

Dans le listing 11 de l’annexe, on peut voir les modifications que j’ai apporté au fichier **rk3399.dtsi** pour ajouter les informations sur les caches. Je me suis appuyés sur les informations du *datasheet* du processeur pour ajouter ces informations. La manière d’ajouter ces informations n’était pas documentée mais j’ai pu trouver des exemples d’autres processeurs pour m’aider à ajouter ces informations.

Ces modifications ne sont toujours pas présentes dans la version actuelle du noyau Linux, il y a alors ici une possibilité de soumettre une *pull request* pour ajouter ces informations au noyau Linux. Je me suis renseigné sur la procédure à suivre pour soumettre une *pull request* au noyau Linux et j’ai pu trouver un guide[4] expliquant la procédure à suivre. Cependant je n’ai pas eu de réponses au patch que j’ai soumis⁴.

De plus, ces informations ne sont pas indispensables pour le fonctionnement de Linux sur ce processeur et ne sont donc pas une priorité pour les développeurs du noyau Linux. Il est donc possible que cette *pull request* ne soit pas acceptée : j’ai pu trouver une pull request similaire datant de plusieurs années qui a mis du temps à être acceptée⁵ alors qu’elle s’intéresse à un processeur plus répandu que le RK3399.

2.3.2 Essais de différentes versions

Le problème principal que j’ai rencontré est que le noyau LITMUS^{RT}, dont plus de détails sont donné dans la partie 3, n’est pas compatible avec les versions récentes du noyau Linux. En effet, la dernière version officielle du noyau LITMUS^{RT} est basé sur la version 4.9.30 du noyau Linux et

3. ubuntu_server_16.04_arm64_rootfs_20171108.ext4

4. Patch envoyé à : <linux-rockchip@lists.infradead.org> et <linux-arm-kernel@lists.infradead.org>

5. <https://github.com/torvalds/linux/commit/618682b350990f8f1bee718949c4b3858711eb58>

date de Mai 2017. Cependant, la carte de développement ROCK960 n'est supportée que depuis la version 6.6 du noyau Linux comme en témoigne un commit de septembre 2018⁶. Il est donc nécessaire de trouver une version du noyau Linux compatible avec la carte de développement et avec le noyau LITMUS^{RT}.

Comme recommandé par Antoine Bertout, les *mailing lists* de LITMUS^{RT} m'ont permis de trouver d'autres chercheurs ayant patché des versions plus récentes de Linux. Après plusieurs essais, je me suis arrêté sur la version 5.4 du noyau Linux pour laquelle Joshua Bakita a fait le travail de patcher linux pour le rendre compatible avec LITMUS^{RT}. J'ai pu trouver son travail sur son *github*⁷.

J'ai moi même essayé de réaliser ce travail pour une autre version de Linux, celle fournie par le fabricant de la carte de développement. En effet, je n'ai jamais pu faire fonctionner le WiFi de la carte avec une autre version de Linux que celle faite par 96Boards, c'est pourquoi j'ai tenté de patcher cette version de Linux avec les commits nécessaires pour faire fonctionner LITMUS^{RT}. Cependant j'ai rencontré de nombreux problèmes, à la fois lors de la compilation, et lors du fonctionnement de ma version patchée de Linux. C'est donc pourquoi j'ai arrêté ce développement pour me concentrer sur la version de Joshua Bakita. Ma compréhension du noyau Linux n'était pas suffisante pour comprendre les problèmes que je rencontrais et j'ai donc préféré me concentrer sur la version du noyau patché qui fonctionnait.

Je n'ai pas testé le bon fonctionnement de toutes les fonctionnalités de la carte sous cette version de Linux (HDMI, USB, GPU, etc.) car je n'en avais pas besoin pour mon développement et que je n'avais pas le temps de tester toutes les fonctionnalités de la carte. Dans cette version le WiFi ne fonctionne pas, cela aurait été utile afin d'accélérer le développement lors de mon stage : téléverser les fichiers sur la carte de développement aurait été plus rapide. Cependant, le WiFi n'est pas indispensable pour le fonctionnement de la carte et je n'ai donc continué ainsi.

Cette étape d'essais de versions et de drivers pour le WiFi était fastidieuse et malgré l'existence d'outils pour accélérer cela comme *Yocto* ou *Buildroot*, je n'ai pas eu le temps de les utiliser. En effet, ces outils permettent de compiler un système d'exploitation Linux complet pour une carte de développement donnée. Cependant, il faut alors configurer ces outils pour qu'ils utilisent les bons drivers et les bonnes versions de Linux. Cela aurait donc nécessité de comprendre comment ces outils fonctionnent et comment les configurer pour qu'ils utilisent les bons drivers et les bonnes versions de Linux. Cela aurait été utile pour accélérer le développement mais j'ai préféré me concentrer sur le développement de l'ordonnanceur plutôt que sur la configuration de ces outils.

6. <https://github.com/torvalds/linux/commit/ffb7b25e8ac3c94f61576ca9cbfd0f16ada1be6d>

7. Dépôt de code de linux-5.4-litmus

3 LITMUS^{RT}

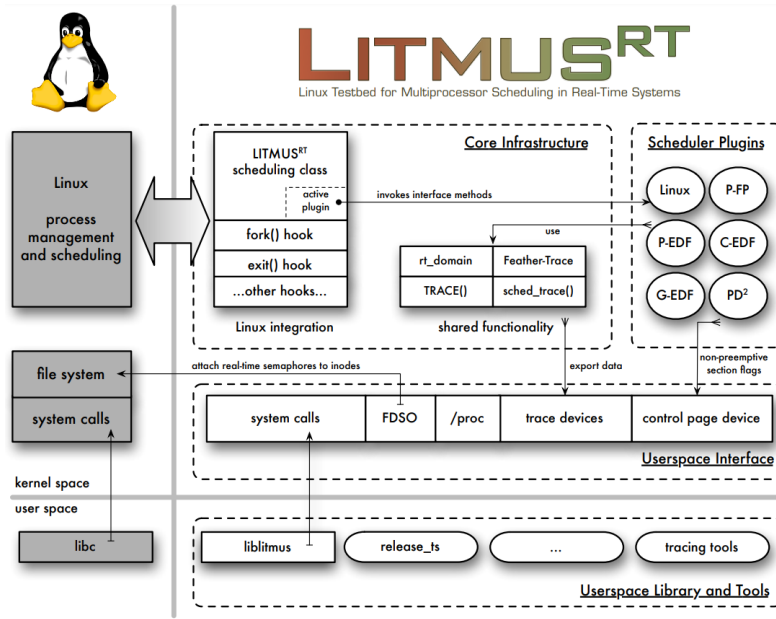


FIGURE 4 – Architecture de LITMUS^{RT}

LITMUS^{RT} est un patch au noyau Linux constitué de quatre parties :

- **LITMUS^{RT} Core** : le patch au noyau Linux qui permet d'ajouter les fonctionnalités temps réel à Linux
- plusieurs ordonnanceur
- une interface dans l'espace utilisateur
- des outils dans l'espace utilisateur tel que *feather-trace* et *liblrmus*

3.1 Présentation de LITMUS^{RT}

LITMUS^{RT}, qui signifie *Linux Testbed for Multiprocessor Scheduling in Real-Time Systems* est un moyen de développer des applications temps réel sur le noyau Linux. Il contient des modifications au noyau habituel de Linux, des interfaces utilisateurs permettant d'interagir à bas niveau avec l'ordonnancement des tâches sous Linux, ainsi qu'une infrastructure de traçage de l'exécution de l'ordonnanceur. LITMUS^{RT} a été développé par Björn B Brandenburg [3] afin de faciliter la recherche et la comparaison des algorithmes d'ordonnancement. Actuellement, beaucoup de publications utilisent LITMUS^{RT} afin de comparer différents protocoles de gestion de ressources partagées par plusieurs processeurs. Mais LITMUS^{RT} est aussi utilisé pour sa facilité à être implémenté sur des plateformes récente dû au fait qu'il est construit par dessus le noyau Linux et que ce dernier est le système d'exploitation qui supporte le plus de plateformes.

Ce dernier point est principalement pourquoi nous avons choisis LITMUS^{RT} comme système d'exploitation sur lequel nous implémenterons des algorithmes d'ordonnancement pour la carte de développement ROCK960. Les autres candidats, comme FreeRTOS, étaient souvent dirigés vers les microcontrôleurs ou bien n'étaient simplement pas compatibles avec la carte de développement.

3.2 Présentation de *feather-trace*

Feather-trace [2] est outil de suivi d'événements léger conçu pour être intégré dans des applications, systèmes d'exploitation ou systèmes embarqués. Il est dans notre cas, à la fois intégré dans le noyau modifié LITMUS^{RT}, mais aussi dans les algorithmes d'ordonnancement que nous implémenterons. Il a été choisi pour sa simplicité et sa légèreté. Il permet d'enregistrer sous forme de fichier de log de multiples données de l'ordonnancement, par exemple l'arrivée d'une nouvelle tâche, le début d'un nouveau job de cette tâche, la date de la fin d'exécution, et bien d'autres événements. De multiples *wrapper* des fonctions de base de *feather-trace* sont fournies dans LITMUS^{RT} afin de pouvoir log des informations supplémentaires, comme le processeur depuis lequel l'exécution du log est effectuée ou encore depuis quelle fonction l'appel est fait.

Cela a été très utile lorsque j'ai développé des nouveaux ordonnanceur sous LITMUS^{RT} afin de corriger des erreurs. Mais cet outil m'a aussi été essentiel afin de comprendre comment fonctionnait les algorithmes d'ordonnancement fournis avec LITMUS^{RT}. J'ai aussi pu comprendre comment le noyau Linux communiquait avec les ordonnanceur en activant des sorties de debug additionnels dans la configuration du noyau.

Enfin, des outils permettant d'extraire, de synthétiser ou de tracer des graphique de certaines données de ces fichiers de logs sont mis à notre disposition sur un dépôt de code présent sur github nommé *feather-trace-tools*. Voici un exemple du tracé de l'ordonnancement réel de C-EDF :

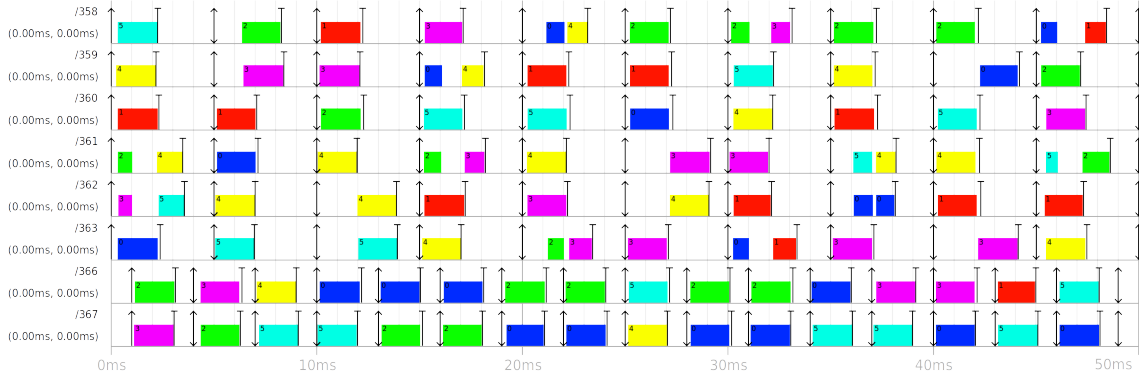


FIGURE 5 – Tracé de l'ordonnancement réel de C-EDF avec 8 tâches sur les 6 processeurs

Pour cela, les temps d'exécution, les débuts et les fins de chaque jobs s'exécutant sur chaque processeur ont été enregistrés sur la carte de développement avec l'outil *st-trace-schedule* du dépôt de code mentionnée précédemment. Cet outil générer autant de fichiers que de processeurs sont présents. On peut alors tracer l'exécution réel avec cette fois ci l'outil *st-draw* en lui fournissant les fichiers générés au préalable. Ici une durée de 50ms a aussi été donnée en argument afin de limiter la durée du tracé.

3.3 Implémentation d'un ordonnanceur EDF partitionné

Le but du stage étant l'implémentation d'algorithmes d'ordonnancement sur plateforme hétérogène avec migration de tâches et de jobs entre les différent processeur, il faudra être capable de réaliser des préemptions de jobs (une exécution de tâche), les migrer, assurer le traitement d'égalités et bien d'autre problèmes.

3.3.1 Algorithme considéré

On cherche alors, pour commencer, à implémenter un algorithme d'ordonnancement simple afin de se familiariser avec les méthodes et fonctions fourni par LITMUS^{RT}. J'ai donc choisi un algorithme partitionné pour la simplicité d'ordonnancement par processeur que cela offre. Un algorithme EDF (*Earliest Deadline First*) est alors choisi pour la simplicité du choix de la tâche à exécuter. Comme son nom l'indique, on choisi à chaque instant la tâche ayant l'échéance la plus proche. On nommera par la suite cet algorithme P-EDF (*Partitionned Earliest Deadline First*).

Pour montrer le fonctionnement de cet algorithme, si l'on se place sur un même processeur, on peut visualiser l'exécution de deux tâche periodiques :

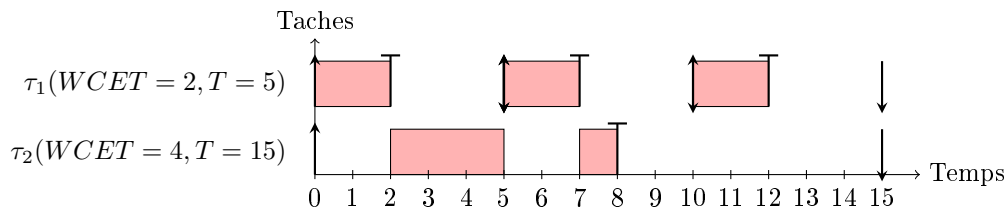


FIGURE 6 – Exemple de EDF à 2 tâches

On a ici une première tâche τ_1 avec un pire temps d'exécution (*Worst Case Execution Time*) de 2 et une période de 5, et une seconde tâche τ_2 avec un pire temps d'exécution de 4 et une période de 15. On a alors préemption de la τ_2 à $t = 5$ afin d'exécuter τ_1 . Cela est dû au réveil de la tâche τ_1 (représenté par la flèche montante) et à la date d'échéance plus proche de cette dernière.

3.3.2 Implémentation

La construction d'un plugin d'ordonnancement nécessite la déclaration d'un module au sens de Linux. Pour Linux un module est un élément de code qui peut être chargé dynamiquement lors de l'exécution du système d'exploitation. Un module permet alors d'étendre les fonctionnalités du noyau, il a donc ont accès aux fonctions du noyau, à ses ressources et peut aussi réaliser des appels systèmes.

Pour que notre nouvel ordonnanceur soit reconnu par le noyau Linux modifié (LITMUS^{RT}), il faut déclarer une fonction d'initialisation :

```

1 #include <linux/module.h> // used for calling module_init()
2
3 static int __init init_p_edf(void)
4 {
5     return 0; // indicates a successful initialisation
6 }
7
8 module_init(init_p_edf); // specify the entry point of the module

```

On peut alors enregistrer ce fichier sous le nom `sched_p_edf.c` pour suivre la nomenclature des autres ordonnanceurs fournis avec avec LITMUS^{RT}. Ce fichier est enregistré dans le dossier `linux/litmus`. On peut alors modifier le fichier `Makefile` de ce dossier afin de l'ajouter au fichier à compiler :

```

1 obj-y = sched_p_edf.o

```

On place notre fichier à compiler sous le mot clé `obj-y` pour signifier que l'on veut ce module compilé et inclus lors de la compilation du noyau Linux.

Une fois le `makefile` modifié, la compilation de notre module sera exécutée lors de la compilation du noyau Linux à l'aide de `make`. La compilation du noyau est discuté dans la partie 2.2.2.

AU MOINS MENTIONNER LES AUTRES FONCTIONS NECESSAIRES

3.3.3 Résultats et essais

Un algorithme partitionné nécessite le démarrage des tâches sur un processeur en particulier. Par exemple, j'ai ici démarré deux tâches `rtspin` avec `liblitmus` : l'une à un pire temps d'exécution de 2ms et une période de 5ms tandis que l'autre a un pire temps d'exécution de 4ms et une période de 7ms.

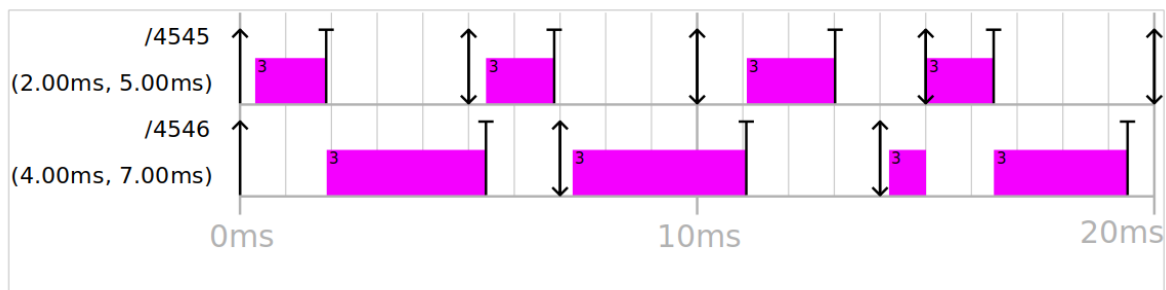


FIGURE 7 – Ordonnancement de deux tâches avec P-EDF

On peut voir que à $t = 5ms$, il n'y a pas préemption de la première tâche et la seconde termine son exécution. En effet, selon EDF, la deuxième tâche a une *deadline* dans 2ms, tandis que la première a une *deadline* dans 5ms : la seconde est donc à cet instant plus prioritaire que la

première. Cependant, à $t = 15ms$, la seconde tâche est préemptée par la première car cette dernière se réveille et a une *deadline* dans 5ms tandis que la seconde a une *deadline* dans 6ms. On peut alors voir que la seconde tâche est préemptée à $t = 15ms$ et reprend son exécution à $t = 17ms$.

3.4 Implémentation d'un ordonnanceur RM partitionné

Comme le montre le listing de code 9, je me suis appuyé sur la librairie `litmus/edf_common.h` fournie dans LITMUS^{RT}. J'ai donc ensuite décidé d'implémenter un algorithme d'ordonnancement qui ne l'utilisait pas. J'ai donc choisi d'implémenter un algorithme RM (Rate Monotonic) partitionné. Cet algorithme est plus simple que P-EDF car il ne prend pas en compte les échéances des tâches. Il suffit alors de trier les tâches par période croissante et de les ordonner en fonction de leur période. Cependant, cet algorithme ne permet pas de garantir l'ordonnancabilité des tâches. En effet, il existe des ensembles de tâches qui ne sont pas ordonnancables par cet algorithme alors qu'ils le sont par P-EDF. Cependant, cet algorithme est plus simple à implémenter et permet de se familiariser avec les fonctions de LITMUS^{RT}.

3.4.1 Implémentation

Pour implémenter un algorithme RM partitionné, j'ai dû réimplémenter les fonctions de `litmus/edf_common.h` pour suivre l'ordonnancement RM. Notre algorithme P-EDF faisait appel aux fonctions :

- `edf_domain_init`, qui initialise le domaine temps réel avec l'ordre qui régit la priorité des tâches
- `edf_preemption_needed`, qui vérifie si la tâche en cours d'exécution doit être préemptée

J'ai donc implémenté les fonctions :

- `rm_domain_init`
- `rm_preemption_needed`

```

1 void rm_domain_init(rt_domain_t* rt, check_resched_needed_t resched,
2                     release_jobs_t release)
3 {
4     rt_domain_init(rt, rm_ready_order, resched, release);
5 }
```

Listing 7 – Fonction `rm_domain_init`

La complexité de cette fonction se cache derrière la nouvelle fonction d'ordre implémenté sous le nom de `rm_ready_order`. Cette fonction est passée en paramètre à `rt_domain_init` et permet d'initialiser le domaine temps réel avec l'ordre qui régit la priorité des tâches sous RM. Comme le montre le listing de code 12, cette fonction est simple et permet de trier les tâches par période croissante. En cas, d'égalité, la priorité est départagée par PID croissant (la tâche avec le PID le plus petit est la plus prioritaire). On effectue aussi d'autres vérifications sur les tâches comme leur nature (tâche temps réel ou non), si deux fois la même tâche est passée en argument, ou encore si une des tâches est NULL (cas où qu'une seule tâche n'est présente).

3.4.2 Résultats et essais

En raison de la grande quantité de nouveau code, faire fonctionner cet algorithme à nécessité une plus grande phase de débogage. J'ai donc utilisé l'outil *feather-trace* pour tracer l'ordonnancement réel de cet algorithme. De cela, j'ai pu déterminer les étapes qui ne fonctionnaient pas. Par exemple, voici un exemple d'un essai avec plusieurs problèmes :

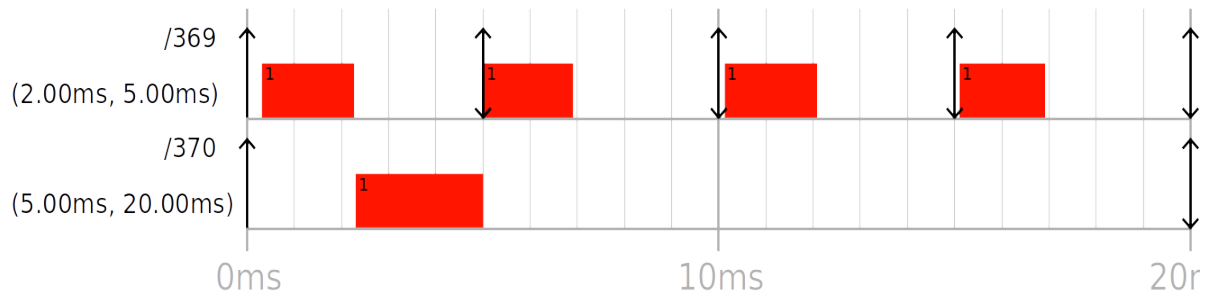


FIGURE 8 – Exemple d'ordonnancement avec défauts

Premièrement, l'ordonnanceur ne stipule pas a *feather-trace* la fin d'exécution d'une tâches. Cela peut être corrigé en appelant `sched_trace_task_completion(prev, budget_exhausted)` lors de la fin d'un job. Secondement, on peut voir que la deuxième tâche se fait préempter en $t = 5ms$ par la première tâche, plus prioritaire. Cependant, l'exécution de cette deuxième tâche ne continue pas lorsque le processeur est libre. Cet erreur était alors du à une erreur de logique dans la fonctions principale d'ordonnancement dans laquelle je ne remplaçait pas les taches préemptés dans la `ready_queue`.

Après ces erreurs corrigées voici la résultat de l'ordonnancement de **XXX** tâches par RM toutes deux lancées sur le même processeur :

AJOUTER IMAGE DE L'ORDONNEMENT DE RM

On peut aussi lancer les taches avec un *offset* afin de voir si l'ordonnancement est correcte. Ce décalage est donné en tant que paramètre a la commande `rt-spin` de *liblrmus* (paramètre -o). Voici un exemple d'ordonnancement avec un *offset* de XXms pour la deuxième tâche.

AJOUTER IMAGE DE L'ORDONNEMENT DE RM AVEC OFFSET

On peut aussi lancer un plus grand nombre de tâches sur une multitude de processeurs (la carte de développement en ayant 6), et on obtient le tracé de tâches suivantes :

AJOUTER IMAGE DE L'ORDONNEMENT DE RM PLEIN DE TACHES

Enfin, on peut aussi voir ce qu'il se passe lorsque l'on cherche a ordonnancer les même tâches que celles que l'on à ordonnancé avec P-EDF et que l'on peut voir dans la figure 7.

IL FAUT QUE JE FASSE L'ESSAI, MAIS NORMALEMENT C'EST PAS ORDONNANCABLE

Cela montre bien que RM est moins performant que P-EDF car il ne permet pas d'ordonnancer toutes les tâches ordonnancables par P-EDF. Cependant, il est plus simple à implémenter et m'a permis de me familiariser avec les fonctions de LITMUS^{RT}.

4 Génération et étude de tâches sur plateforme hétérogène

Lors de tout mes tests sur la carte de développement, j'ai utilisé l'outil *rtspin* de *liblitmus* afin de générer des tâches temps réel. Cet outil permet de générer des tâches avec des paramètres spécifiques, comme le pire temps d'exécution, la période, le processeur sur lequel la tâche doit s'exécuter, etc. Cependant, il ne permet pas de générer des tâches ayant des temps d'exécution différents sur différents processeurs. Cela est un problème car cela ne permet pas de mettre en valeur la nature hétérogène de la plateforme sur laquelle nous travaillons. C'est pourquoi je me suis intéressé durant une partie de mon stage à créer de telles tâches.

4.1 Mesure de temps d'exécution

Il est important de pouvoir mesurer de manière suffisamment précise le temps d'exécution d'une tâche. Pour cela, ma première idée était d'utiliser le module `time` de Linux. Cependant, ce module ne permet pas de mesurer des temps d'exécution inférieur à la milliseconde. Cela est dû au fait que le module `time` utilise le timer du noyau Linux qui a une précision de 1ms. Cela est bien trop imprécis pour mesurer des temps d'exécution de tâches temps réel. J'ai donc d'abord créé un script `bash` utilisant un autre temps Linux. Ce script est présent dans le listing 13 et fut utilisé pour tout mes essais préliminaires. Cependant, ce script, malgré sa plus grande précision, mesurait toujours un temps minimum : environ 6ms. Je n'ai pas pu le montrer, mais ce temps semble venir du démarrage du script, puis du démarrage du programme appelé. Il a donc été utile pour comparer des tâches entre elles, mais n'était pas assez précis pour connaître le temps d'exécution d'une tâche.

4.2 Génération de tâches répétables

Mon objectif était alors de générer des tâches qui s'exécutent à des vitesses différentes sur les différents processeurs, tout en ayant un temps d'exécution qui ne varie qu'un minimum entre deux exécutions sur un même processeur.

4.2.1 Première idée : *checksum* d'un fichier

Ma première idée était de calculer la checksum d'un fichier de petite taille. La taille du fichier permettrait alors de faire varier le temps d'exécution. Cette opération était principalement calculatoire, j'avais espoir que le temps d'exécution ne varie pas trop entre deux exécutions sur un même processeur. Cependant, cette opération semblait posséder trop d'accès à la mémoire et au stockage : deux choses que je ne voulais pas prendre en compte dans mon temps d'exécution. Comme on peut le voir sur l'exécution d'une telle tâche, sur laquelle je réalise la checksum md5 sur un fichier de code de 95KiB, J'ai donc abandonné cette idée.

Conclusion

Il faut une petite conclusion qui fait le taf

Annexe

```

1 #
2 # Makefile for LITMUS-RT
3 #
4
5 obj-y = sched_plugin.o litmus.o \
6         preempt.o \
7         litmus_proc.o \
8         budget.o \
9         clustered.o \
10        jobs.o \
11        sync.o \
12        rt_domain.o \
13        edf_common.o \
14        fp_common.o \
15        fdso.o \
16        locking.o \
17        srp.o \
18        bheap.o \
19        binheap.o \
20        ctrldev.o \
21        uncachedev.o \
22        sched_gsn_edf.o \
23        sched_psn_edf.o \
24        sched_pfp.o \
25        sched_p_edf.o
26
27 obj-$(CONFIG_PLUGIN_CEDF) += sched_cedf.o
28 obj-$(CONFIG_PLUGIN_PFAIR) += sched_pfair.o
29
30 obj-$(CONFIG_FEATHER_TRACE) += ft_event.o ftdev.o
31 obj-$(CONFIG_SCHED_TASK_TRACE) += sched_task_trace.o
32 obj-$(CONFIG_SCHED_DEBUG_TRACE) += sched_trace.o
33 obj-$(CONFIG_SCHED_OVERHEAD_TRACE) += trace.o
34
35 obj-y += sched_pres.o
36
37 obj-y += reservations/

```

Listing 8 – linux/litmus/Makefile

```

1 #include <linux/module.h>
2 #include <linux/percpu.h>
3 #include <linux/sched.h>
4 #include <litmus/litmus.h>
5 #include <litmus/budget.h>
6 #include <litmus/edf_common.h>
7 #include <litmus/jobs.h>
8 #include <litmus/litmus_proc.h>
9 #include <litmus/debug_trace.h>
10 #include <litmus/preempt.h>
11 #include <litmus/rt_domain.h>
12 #include <litmus/sched_plugin.h>
13 #include <litmus/sched_trace.h>
14
15 struct p_edf_cpu_state {
16     rt_domain_t local_queues;
17     int cpu;
18     struct task_struct* scheduled;
19 };

```

```

20
21 static DEFINE_PER_CPU(struct p_edf_cpu_state, p_edf_cpu_state);
22
23 #define cpu_state_for(cpu_id) (&per_cpu(p_edf_cpu_state, cpu_id))
24 #define local_cpu_state() (this_cpu_ptr(&p_edf_cpu_state))
25 #define remote_edf(cpu) (&per_cpu(p_edf_cpu_state, cpu).local_queues)
26 #define remote_pedf(cpu) (&per_cpu(p_edf_cpu_state, cpu))
27 #define task_edf(task) remote_edf(get_partition(task))
28
29 static struct domain_proc_info p_edf_domain_proc_info;
30
31 static long p_edf_get_domain_proc_info(struct domain_proc_info **ret)
32 {
33     *ret = &p_edf_domain_proc_info;
34     return 0;
35 }
36
37 static void p_edf_setup_domain_proc(void)
38 {
39     int i, cpu;
40     int num_rt_cpus = num_online_cpus();
41
42     struct cd_mapping *cpu_map, *domain_map;
43
44     memset(&p_edf_domain_proc_info, 0, sizeof(p_edf_domain_proc_info));
45     init_domain_proc_info(&p_edf_domain_proc_info, num_rt_cpus, num_rt_cpus);
46     p_edf_domain_proc_info.num_cpus = num_rt_cpus;
47     p_edf_domain_proc_info.num_domains = num_rt_cpus;
48
49     i = 0;
50     for_each_online_cpu(cpu) {
51         cpu_map = &p_edf_domain_proc_info.cpu_to_domains[i];
52         domain_map = &p_edf_domain_proc_info.domain_to_cpus[i];
53
54         cpu_map->id = cpu;
55         domain_map->id = i;
56         cpumask_set_cpu(i, cpu_map->mask);
57         cpumask_set_cpu(cpu, domain_map->mask);
58         ++i;
59     }
60 }
61
62 /* This helper is called when task 'prev' exhausted its budget or when
63 * it signaled a job completion. */
64 static void p_edf_job_completion(struct task_struct *prev, int budget_exhausted)
65 {
66     sched_trace_task_completion(prev, budget_exhausted);
67     TRACE_TASK(prev, "job_completion(forced=%d).\n", budget_exhausted);
68
69     tsk_rt(prev)->completed = 0;
70     /* Call common helper code to compute the next release time, deadline,
71     * etc. */
72     prepare_for_next_period(prev);
73 }
74
75 /* Add the task 'tsk' to the appropriate queue. Assumes the caller holds the
76    ready lock.
77 */
78 static void p_edf_requeue(struct task_struct *tsk, struct p_edf_cpu_state *
79     cpu_state)
80 {

```

```

79     if (is_released(tsk, litmus_clock())) {
80         /* Uses __add_ready() instead of add_ready() because we already
81          * hold the ready lock. */
82         __add_ready(&cpu_state->local_queues, tsk);
83         TRACE_TASK(tsk, "added to ready queue on reschedule\n");
84     } else {
85         /* Uses add_release() because we DON'T have the release lock. */
86         add_release(&cpu_state->local_queues, tsk);
87         TRACE_TASK(tsk, "added to release queue on reschedule\n");
88     }
89 }
90
91 static int p_edf_check_for_preemption_on_release(rt_domain_t *local_queues)
92 {
93     struct p_edf_cpu_state *state = container_of(local_queues,
94                                                  struct p_edf_cpu_state,
95                                                  local_queues);
96
97     /* Because this is a callback from rt_domain_t we already hold
98      * the necessary lock for the ready queue. */
99
100    if (edf_preemption_needed(local_queues, state->scheduled)) {
101        preempt_if_preemptable(state->scheduled, state->cpu);
102        return 1;
103    }
104    return 0;
105 }
106
107 static long p_edf_activate_plugin(void)
108 {
109     int cpu;
110     struct p_edf_cpu_state *state;
111     for_each_online_cpu(cpu) {
112         TRACE("Initializing CPU%d...\n", cpu);
113         state = cpu_state_for(cpu);
114         state->cpu = cpu;
115         state->scheduled = NULL;
116         edf_domain_init(&state->local_queues,
117                        p_edf_check_for_preemption_on_release,
118                        NULL);
119     }
120
121     p_edf_setup_domain_proc();
122     return 0;
123 }
124
125 static long p_edf_deactivate_plugin(void)
126 {
127     destroy_domain_proc_info(&p_edf_domain_proc_info);
128     return 0;
129 }
130
131
132
133 static struct task_struct* p_edf_schedule(struct task_struct * prev)
134 {
135     struct p_edf_cpu_state *local_state = local_cpu_state();
136
137     /* next == NULL means "schedule background work". */
138     struct task_struct *next = NULL;
139

```

```

140     /* prev's task state */
141     int exists, out_of_time, job_completed, self_suspends, preempt, resched;
142
143     raw_spin_lock(&local_state->local_queues.ready_lock);
144
145     BUG_ON(local_state->scheduled && local_state->scheduled != prev);
146     BUG_ON(local_state->scheduled && !is_realtime(prev));
147
148     exists = local_state->scheduled != NULL;
149     self_suspends = exists && !is_current_running();
150     out_of_time = exists && budget_enforced(prev) && budget_exhausted(prev);
151     job_completed = exists && is_completed(prev);
152
153     /* preempt is true if task 'prev' has lower priority than something on
154     * the ready queue. */
155     preempt = edf_preemption_needed(&local_state->local_queues, prev);
156
157     /* check all conditions that make us reschedule */
158     resched = preempt;
159
160     /* if 'prev' suspends, it CANNOT be scheduled anymore => reschedule */
161     if (self_suspends) {
162         resched = 1;
163     }
164
165     /* also check for (in-)voluntary job completions */
166     if (out_of_time || job_completed) {
167         p_edf_job_completion(prev, out_of_time);
168         resched = 1;
169     }
170
171     if (resched) {
172         /* First check if the previous task goes back onto the ready
173         * queue, which it does if it did not self_suspend.
174         */
175         if (exists && !self_suspends) {
176             p_edf_requeue(prev, local_state);
177         }
178         next = __take_ready(&local_state->local_queues);
179     } else {
180         /* No preemption is required. */
181         next = local_state->scheduled;
182     }
183
184     local_state->scheduled = next;
185     if (exists && prev != next) {
186         TRACE_TASK(prev, "descheduled.\n");
187     }
188     if (next) {
189         TRACE_TASK(next, "scheduled.\n");
190     }
191
192     /* This mandatory. It triggers a transition in the LITMUS-RT remote
193     * preemption state machine. Call this AFTER the plugin has made a
194     * local scheduling decision.
195     */
196     sched_state_task_picked();
197
198     raw_spin_unlock(&local_state->local_queues.ready_lock);
199     return next;
200 }

```

```

201
202 static long p_edf_admit_task(struct task_struct *tsk)
203 {
204     if (task_cpu(tsk) == get_partition(tsk)) {
205         TRACE_TASK(tsk, "accepted by p_edf plugin.\n");
206         return 0;
207     }
208     return -EINVAL;
209 }
210
211 static void p_edf_task_new(struct task_struct *tsk, int on_runqueue,
212                           int is_running)
213 {
214     /* We'll use this to store IRQ flags. */
215     unsigned long flags;
216     struct p_edf_cpu_state *state = cpu_state_for(get_partition(tsk));
217     lt_t now;
218
219     TRACE_TASK(tsk, "is a new RT task %llu (on runqueue:%d, running:%d)\n",
220               litmus_clock(), on_runqueue, is_running);
221
222     /* Acquire the lock protecting the state and disable interrupts. */
223     raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);
224
225     now = litmus_clock();
226
227     /* Release the first job now. */
228     release_at(tsk, now);
229
230     if (is_running) {
231         /* If tsk is running, then no other task can be running
232          * on the local CPU. */
233         BUG_ON(state->scheduled != NULL);
234         state->scheduled = tsk;
235     } else if (on_runqueue) {
236         p_edf_requeue(tsk, state);
237     }
238
239     if (edf_preemption_needed(&state->local_queues, state->scheduled))
240         preempt_if_preemptable(state->scheduled, state->cpu);
241
242     raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
243 }
244
245 static void p_edf_task_exit(struct task_struct *tsk)
246 {
247     unsigned long flags;
248     struct p_edf_cpu_state *state = cpu_state_for(get_partition(tsk));
249     raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);
250     rt_domain_t*      edf;
251
252     /* For simplicity, we assume here that the task is no longer queued
253      * anywhere else. This
254      * is the case when tasks exit by themselves; additional queue
255      * management is
256      * is required if tasks are forced out of real-time mode by other tasks
257      * . */
258
259     if (is_queued(tsk)){
260         edf = task_edf(tsk);
261         remove(edf, tsk);
262     }

```

```

259     }
260
261     if (state->scheduled == tsk) {
262         state->scheduled = NULL;
263     }
264
265     preempt_if_preemptable(state->scheduled, state->cpu);
266     raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
267 }
268
269 /* Called when the state of tsk changes back to TASK_RUNNING.
270  * We need to requeue the task.
271  *
272  * NOTE: If a sporadic task is suspended for a long time,
273  * this might actually be an event-driven release of a new job.
274  */
275 static void p_edf_task_resume(struct task_struct *tsk)
276 {
277     unsigned long flags;
278     struct p_edf_cpu_state *state = cpu_state_for(get_partition(tsk));
279     lt_t now;
280     TRACE_TASK(tsk, "wake_up at %llu\n", litmus_clock());
281     raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);
282
283     now = litmus_clock();
284
285     if (is_sporadic(tsk) && is_tardy(tsk, now)) {
286         /* This sporadic task was gone for a "long" time and woke up past
287          * its deadline. Give it a new budget by triggering a job
288          * release. */
289         inferred_sporadic_job_release_at(tsk, now);
290         TRACE_TASK(tsk, "woke up too late.\n");
291     }
292
293     /* This check is required to avoid races with tasks that resume before
294      * the scheduler "noticed" that it resumed. That is, the wake up may
295      * race with the call to schedule(). */
296     if (state->scheduled != tsk) {
297         TRACE_TASK(tsk, "is being requeued\n");
298         p_edf_requeue(tsk, state);
299         if (edf_preemption_needed(&state->local_queues, state->scheduled))
300             {
301                 preempt_if_preemptable(state->scheduled, state->cpu);
302             }
303     }
304
305     raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
306 }
307
308 static struct sched_plugin p_edf_plugin = {
309     .plugin_name      = "P-EDF",
310     .schedule         = p_edf_schedule,
311     .task_wake_up     = p_edf_task_resume,
312     .admit_task       = p_edf_admit_task,
313     .task_new         = p_edf_task_new,
314     .task_exit        = p_edf_task_exit,
315     .get_domain_proc_info = p_edf_get_domain_proc_info,
316     .activate_plugin   = p_edf_activate_plugin,
317     .deactivate_plugin = p_edf_deactivate_plugin,
318     .complete_job     = complete_job,

```

```

319 };
320
321 static int __init init_p_edf(void)
322 {
323     return register_sched_plugin(&p_edf_plugin);
324 }
325
326 module_init(init_p_edf);

```

Listing 9 – linux/litmus/sched_p_edf.c

```

1  # LITMUS^RT
2  #
3
4  #
5  # Scheduling
6  #
7  CONFIG_PLUGIN_PFAIR=y
8  # CONFIG_RELEASE_MASTER is not set
9  CONFIG_PREFER_LOCAL_LINKING=y
10 CONFIG_LITMUS_QUANTUM_LENGTH_US=1000
11 CONFIG_BUG_ON_MIGRATION_DEADLOCK=y
12 # end of Scheduling
13
14 #
15 # Real-Time Synchronization
16 #
17 CONFIG_NP_SECTION=y
18 CONFIG_LITMUS_LOCKING=y
19 # end of Real-Time Synchronization
20
21 #
22 # Performance Enhancements
23 #
24 CONFIG_ALLOW_EARLY_RELEASE=y
25 # CONFIG_EDF_TIE_BREAK_LATENESS is not set
26 CONFIG_EDF_TIE_BREAK_LATENESS_NORM=y
27 # CONFIG_EDF_TIE_BREAK_HASH is not set
28 # CONFIG_EDF_PID_TIE_BREAK is not set
29 # end of Performance Enhancements
30
31 #
32 # Tracing
33 #
34 CONFIG_FEATHER_TRACE=y
35 CONFIG_SCHED_TASK_TRACE=y
36 CONFIG_SCHED_TASK_TRACE_SHIFT=9
37 CONFIG_SCHED_OVERHEAD_TRACE=y
38 CONFIG_SCHED_OVERHEAD_TRACE_SHIFT=22
39 CONFIG_SCHED_DEBUG_TRACE=y
40 CONFIG_SCHED_DEBUG_TRACE_SHIFT=18
41 CONFIG_SCHED_DEBUG_TRACE_CALLER=y
42 # CONFIG_PREEMPT_STATE_TRACE is not set
43 # CONFIG_REPORT_TIMER_LATENCY is not set
44 # end of Tracing
45 # end of LITMUS^RT

```

Listing 10 – Partie du fichier .config liée à LITMUS^{RT}

```

1 cpus {
2     #address-cells = <2>;

```

```

3      #size-cells = <0>;
4
5      cpu-map {
6          cluster0 {
7              core0 {
8                  cpu = <&cpu_10>;
9              };
10             core1 {
11                 cpu = <&cpu_11>;
12             };
13             core2 {
14                 cpu = <&cpu_12>;
15             };
16             core3 {
17                 cpu = <&cpu_13>;
18             };
19         };
20
21         cluster1 {
22             core0 {
23                 cpu = <&cpu_b0>;
24             };
25             core1 {
26                 cpu = <&cpu_b1>;
27             };
28         };
29     };
30
31     cpu_10: cpu@0 {
32         device_type = "cpu";
33         compatible = "arm,cortex-a53";
34         reg = <0x0 0x0>;
35         enable-method = "psci";
36         next-level-cache = <&l2_0>;
37         capacity-dmips-mhz = <485>;
38         clocks = <&cru ARMCLKL>;
39         #cooling-cells = <2>; /* min followed by max */
40         dynamic-power-coefficient = <100>;
41         cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
42
43         l2_0: l2-cache {
44             compatible = "cache,arm,arch-cache";
45         };
46     };
47
48     cpu_11: cpu@1 {
49         device_type = "cpu";
50         compatible = "arm,cortex-a53";
51         reg = <0x0 0x1>;
52         enable-method = "psci";
53         next-level-cache = <&l2_0>;
54         capacity-dmips-mhz = <485>;
55         clocks = <&cru ARMCLKL>;
56         #cooling-cells = <2>; /* min followed by max */
57         dynamic-power-coefficient = <100>;
58         cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
59     };
60
61     cpu_12: cpu@2 {
62         device_type = "cpu";
63         compatible = "arm,cortex-a53";

```



```

64         reg = <0x0 0x2>;
65         enable-method = "psci";
66         next-level-cache = <&l2_0>;
67         capacity-dmips-mhz = <485>;
68         clocks = <&cru ARMCLKL>;
69         #cooling-cells = <2>; /* min followed by max */
70         dynamic-power-coefficient = <100>;
71         cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
72     };
73
74     cpu_l3: cpu@3 {
75         device_type = "cpu";
76         compatible = "arm,cortex-a53";
77         reg = <0x0 0x3>;
78         enable-method = "psci";
79         next-level-cache = <&l2_0>;
80         capacity-dmips-mhz = <485>;
81         clocks = <&cru ARMCLKL>;
82         #cooling-cells = <2>; /* min followed by max */
83         dynamic-power-coefficient = <100>;
84         cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
85     };
86
87     cpu_b0: cpu@100 {
88         device_type = "cpu";
89         compatible = "arm,cortex-a72";
90         reg = <0x0 0x100>;
91         enable-method = "psci";
92         next-level-cache = <&l2_1>;
93         capacity-dmips-mhz = <1024>;
94         clocks = <&cru ARMCLKB>;
95         #cooling-cells = <2>; /* min followed by max */
96         dynamic-power-coefficient = <436>;
97         cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
98
99         l2_1: l2-cache {
100             compatible = "cache,arm,arch-cache";
101         };
102     };
103
104     cpu_b1: cpu@101 {
105         device_type = "cpu";
106         compatible = "arm,cortex-a72";
107         reg = <0x0 0x101>;
108         enable-method = "psci";
109         next-level-cache = <&l2_1>;
110         capacity-dmips-mhz = <1024>;
111         clocks = <&cru ARMCLKB>;
112         #cooling-cells = <2>; /* min followed by max */
113         dynamic-power-coefficient = <436>;
114         cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
115     };
116     ...
117 }

```

Listing 11 – Modifications apportées au fichier rk3399.dtsi

```

1  /*
2  * litmus/rm_common.c
3  */
4  #include <linux/percpu.h>

```

```

5 #include <linux/sched.h>
6 #include <linux/list.h>
7
8 #include <litmus/litmus.h>
9 #include <litmus/sched_plugin.h>
10 #include <litmus/sched_trace.h>
11 #include <litmus/debug_trace.h>
12
13 #include <litmus/rm_common.h>
14
15
16 /* rm_higher_prio - returns true if first has a higher RM priority
17 *                  than second. Deadline ties are broken by PID.
18 *
19 * both first and second may be NULL
20 */
21 int rm_higher_prio(struct task_struct* first,
22                   struct task_struct* second)
23 {
24     struct task_struct *first_task = first;
25     struct task_struct *second_task = second;
26
27     /* There is no point in comparing a task to itself. */
28     if (first && first == second) {
29         TRACE_TASK(first,
30                    "WARNING: pointless edf priority comparison.\n");
31         return 0;
32     }
33
34
35     /* check for NULL tasks */
36     if (!first || !second)
37         return first && !second;
38
39
40     if (shorter_exec_time(first_task, second_task)) {
41         return 1;
42     }
43     else if (get_rt_period(first_task) == get_rt_period(second_task)) {
44         /* Need to tie break. All methods must set pid_break to 0/1 if
45          * first_task does not have priority over second_task.
46          */
47         int pid_break;
48
49         /* CONFIG_EDF_PID_TIE_BREAK */
50         pid_break = 1; // fall through to tie-break by pid;
51
52
53         /* Tie break by pid */
54         if(pid_break) {
55             if (first_task->pid < second_task->pid) {
56                 return 1;
57             }
58             else if (first_task->pid == second_task->pid) {
59                 /* If the PIDs are the same then the task with the
60                  * inherited priority wins.
61                  */
62                 if (!second->rt_param.inh_task) {
63                     return 1;
64                 }
65             }

```

```

66         }
67     }
68     return 0; /* fall-through. prio(second_task) > prio(first_task) */
69 }
70
71 int rm_ready_order(struct bheap_node* a, struct bheap_node* b)
72 {
73     return rm_higher_prio(bheap2task(a), bheap2task(b));
74 }
75
76 void rm_domain_init(rt_domain_t* rt, check_resched_needed_t resched,
77                     release_jobs_t release)
78 {
79     rt_domain_init(rt, rm_ready_order, resched, release);
80 }
81
82
83 /* need_to_preempt - check whether the task t needs to be preempted
84 *                     call only with irqs disabled and with ready_lock acquired
85 *                     THIS DOES NOT TAKE NON-PREEMPTIVE SECTIONS INTO ACCOUNT!
86 */
87 int rm_preemption_needed(rt_domain_t* rt, struct task_struct *t)
88 {
89     /* we need the read lock for edf_ready_queue */
90     /* no need to preempt if there is nothing pending */
91     if (!__jobs_pending(rt))
92         return 0;
93     /* we need to reschedule if t doesn't exist */
94     if (!t)
95         return 1;
96
97     /* NOTE: We cannot check for non-preemptibility since we
98     *         don't know what address space we're currently in.
99     */
100
101     /* make sure to get non-rt stuff out of the way */
102     return !is_realtime(t) || rm_higher_prio(__next_ready(rt), t);
103 }

```

Listing 12 – litmus/rm_common.c

```

1  #!/bin/bash
2  start="$(date +%s.%N)"
3  $@
4  end="$(date +%s.%N)"
5  diff=$(awk "BEGIN { print $end - $start }")
6  echo "$diff" >&2

```

Listing 13 – Script bash

Table des figures

1	Architecture du processeur RK3399	4
2	Interface de configuration du noyau	6
3	Compilation croisée du noyau Linux	7
4	Architecture de LITMUS ^{RT}	10
5	Tracé de l'ordonnancement réel de C-EDF avec 8 taches sur les 6 processeurs . . .	11
6	Exemple de EDF à 2 taches	11
7	Ordonnancement de deux taches avec P-EDF	12
8	Exemple d'ordonnancement avec défauts	14

Références

- [1] Antoine Bertout, Joël Goossens, Emmanuel Grolleau, and Xavier Poczekajlo. Workload assignment for global real-time scheduling on unrelated multicore platforms. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, pages 139–148, 2020.
- [2] B Brandenburg and J Anderson. Feather-trace : A lightweight event tracing toolkit. In *Proceedings of the third international workshop on operating systems platforms for embedded real-time applications*, pages 19–28. Citeseer, 2007.
- [3] Bjorn B Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.
- [4] Nick Desaulniers. Submitting your first patch to the linux kernel and responding to feedback. <https://nickdesaulniers.github.io/blog/2017/05/16/submitting-your-first-patch-to-the-linux-kernel-and-responding-to-feedback/>, 2021. Accessed : 2021-05-01.

Glossaire

bootloader court programme chargé au démarrage de l'ordinateur initialisant le système d'exploitation. 5

checksum md5 algorithme de hachage cryptographique de 128 bits permettant de produire un résultat (appelé aussi empreinte) à partir d'un fichier. 15

cluster ensemble interconnecté de plusieurs processeurs. 4

git système de gestion de versions décentralisé, utilisé pour suivre les modifications apportées à des fichiers sources dans un projet de développement logiciel. 5, 6, 7

ordonnancement ou *scheduling* est l'activité qui consiste à affecter des ressources à des tâches. 3

ordonnanceur ou *scheduler* est un composant d'un système d'exploitation chargé de gérer l'ordonnancement des processus. 3, 10

plateforme hétérogène système formé d'un ensemble de processeurs différents. 3

préemption processus par lequel un système d'exploitation interrompt temporairement l'exécution d'une tâche en cours pour donner la priorité à une autre tâche de plus haute priorité. 11

processeur ou *CPU* est un composant présent dans tout ordinateur. Il est chargé d'effectuer les calculs et de gérer les flux de données. 11

SHRIMP *Scheduling of Real-Time Heterogeneous Multiprocessor Platform* ou Ordonnancement Temps réel de Plateforme Multiprocesseur Hétérogène. 3

SOC ou *Système On a Chip* est . 4