Rapport de stage Ingénieur

-

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

-

BELPOIS Vincent

2023

Table des matières

Présentation du stage

0.1 Le L.I.A.S.

Parler des différentes équipes. Dans quelle équipe je suis ?

0.2 Le sujet du stage

Mon stage s'intéresse à l'implémentation d'un ordonnanceur sur plateforme hétérogène[1]
Parler du projet SHRIMP.

1 Systèmes d'exploitation compatibles avec la carte ROCK960

1.1 Présentation de la carte de développement

Le stage s'intéressant à l'implémentation d'un algorithme d'ordonnacement 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.

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 tache 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.

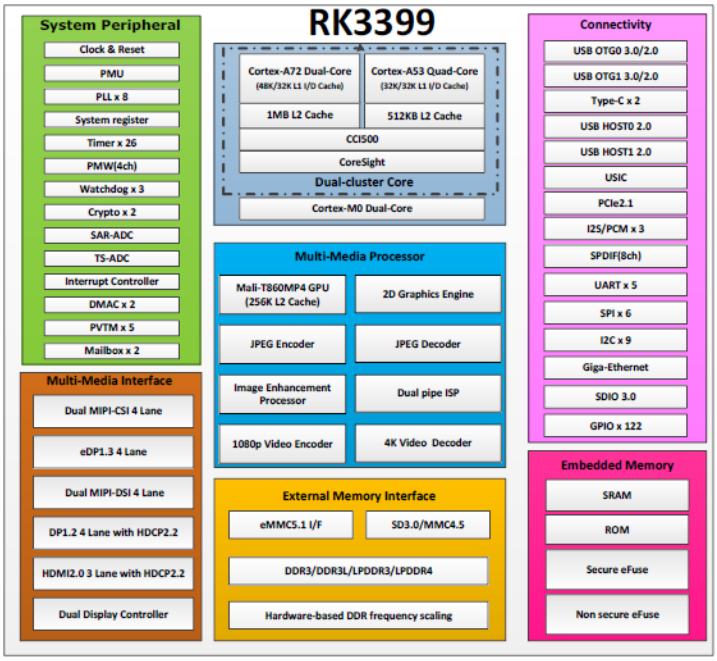

FIGURE 1 – Architecture du processeur RK3399

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.

1.2 Installation d'un système d'exploitation

1.2.1 Installation d'une image précompilée

Pour premier tester l'Installation de linux sur la carte de développement, j'ai utilisé une image de la distribution Ubuntu fournie par le fabriquant 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.

1.2.2 Compilation de Linux depuis le code source

Afin d'utiliser une version de Linux différente de la version précompilé par le fabriquant 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 a 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 peu alors procédé a 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'une 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 a notre disposition dans le noyau :

- `make defconfig` : cet outil est utilisé pour générer une configuration pas défaut. Ici l'architecture de la machine qui réalise la compilation sera sélectionnée. Dans mon cas, éxécuter 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.

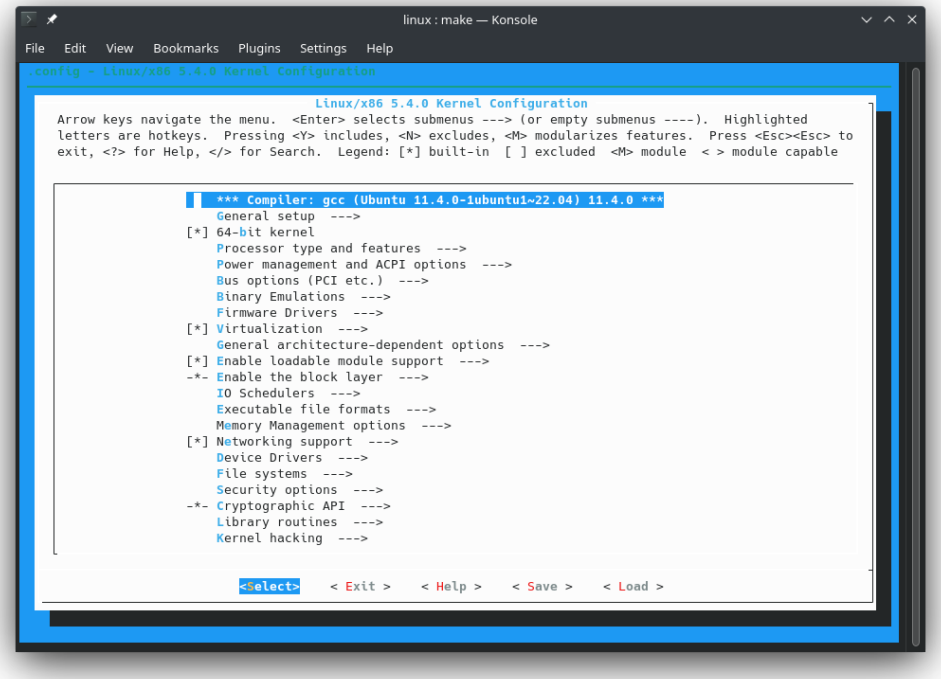

FIGURE 2 – Interface de configuration du noyau

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

Une fois la configuration crée, 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 a la compilation. Par exemple, installer `libelf-dev` "une bibliothèque partagée qui permet de lire et écrire des fichiers ELF à un niveau élevé" [1].

```
$ make -j 16
```

Listing 2 – Compilation sur plusieurs processeurs

Le paramètre `-j 16` signifie que l'on veux 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 si fonctionne. Sur le dépôt de code de linux, les différentes versions sont stockés sous forme d'un certain commit qui à été "tagé" afin de le retrouver. On peut alors changer de version en revenant à ce commit grâce à la commande `checkout` de git :

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

Listing 3 – Retour sur un commit tagé

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

```
$ git tag -l
```

Listing 4 – Comment lister les tags

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 étais était la plus compliquée a 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 géré le stockage du code que j'ai développé, que ce soit des outils ou des modifications du noyau.

1.2.3 Compilation croisée

Nous ne pouvons malheureusement pas compiler directement le noyau linux pour la carte de développement dû à la différence

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'environement ? 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 ?

Copy de l'image du noyau pour faire encore plus rapide.

1.3 Etude des versions de Linux compatibles

1.3.1 Comment Linux gère le suport d'un processeur

fichier de configuration de la carte ? du processeur ?

Problème rencontré avec le wifi

1.3.2 Essais de différentes versions

Expliquer quels versions de linux sont compatibles avec la carte. Quelles versions de LITMUS sont disponibles.

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

2 LITMUS^{RT}

2.1 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 a bas niveau avec l'ordonnancement des taches sous Linux, ainsi qu'une infrastructure de traçage de l'exécution de l'ordonnanceur. LITMUS^{RT}à 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 resources partagées par plusieurs processeurs. Mais LITMUS^{RT}est aussi utilisé pour sa facilité à être implémenté sur des plateformes récente dù 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.

2.2 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 tache, le début d'un nouveau job de cette tache, la date de la fin d'exécution, et bien d'autres événements. De multiples *wrapper* des fonctions de base de *feather-trace* sont fournie 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'ordonancement réel de INSERER ORDONANCEUR :

METTRE UNE IMAGE D'UN DES TRACES

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 XXXXX a aussi été donnée en argument afin de limiter la durée du tracé.

2.3 Implémentation d'un ordonnanceur EDF partitionné

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

2.3.1 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 tache

a exécuter. Comme son nom l'indique, on choisi à chaque instant la tache 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 tache periodiques :

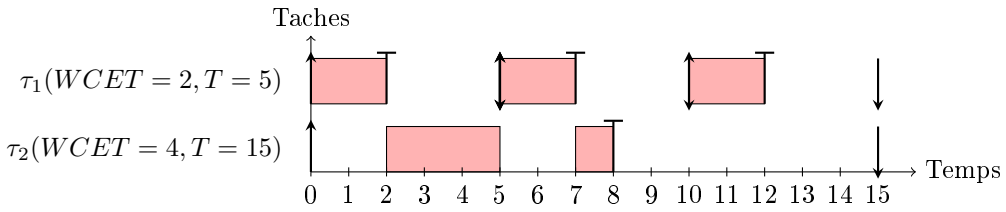

FIGURE 3 – Exemple de EDF à 2 taches

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

2.3.2 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 resources et peut aussi réaliser des appels systèmes.

Pour que notre nouvel ordonnanceur soit reconnu par le noyau Linux modifié (LITMUS^{RT}) ; il faut alors 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 llinux/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 1.2.2.

2.4 Implémentation d'un ordonnanceur RM partitionné

Conclusion

Annexe

Listing 5 – linux/litmus/Makefile

```
#
# Makefile for LITMUS^RT
#

obj-y = sched_plugin.o litmus.o \
        preempt.o \
        litmus_proc.o \
        budget.o \
        clustered.o \
        jobs.o \
        sync.o \
        rt_domain.o \
        edf_common.o \
        fp_common.o \
        fdso.o \
        locking.o \
        srp.o \
        bheap.o \
        binheap.o \
        ctrldev.o \
        uncachedev.o \
        sched_gsn_edf.o \
        sched_psn_edf.o \
        sched_pfp.o \
        sched_p_edf.o

obj-$(CONFIG_PLUGIN_CEDF) += sched_cedf.o
obj-$(CONFIG_PLUGIN_PFAIR) += sched_pfair.o

obj-$(CONFIG_FEATHER_TRACE) += ft_event.o ftdev.o
obj-$(CONFIG_SCHED_TASK_TRACE) += sched_task_trace.o
obj-$(CONFIG_SCHED_DEBUG_TRACE) += sched_trace.o
obj-$(CONFIG_SCHED_OVERHEAD_TRACE) += trace.o

obj-y += sched_pres.o

obj-y += reservations/
```

Listing 6 – linux/litmus/sched_p_edf.c

```
#include <linux/module.h>
#include <linux/percpu.h>
#include <linux/sched.h>
#include <litmus/litmus.h>
#include <litmus/budget.h>
#include <litmus/edf_common.h>
#include <litmus/jobs.h>
#include <litmus/litmus_proc.h>
#include <litmus/debug_trace.h>
#include <litmus/preempt.h>
#include <litmus/rt_domain.h>
#include <litmus/sched_plugin.h>
#include <litmus/sched_trace.h>

struct p_edf_cpu_state {
        rt_domain_t local_queues;
        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
        ready lock.
76  */
77  static void p_edf_requeue(struct task_struct *tsk, struct p_edf_cpu_state *
```

```
    cpu_state)
{
        if (is_released(tsk, litmus_clock())) {
                /* Uses __add_ready() instead of add_ready() because we already
                    * hold the ready lock. */
                __add_ready(&cpu_state->local_queues, tsk);
                TRACE_TASK(tsk, "added to ready queue on reschedule\n");
        } else {
                /* Uses add_release() because we DON'T have the release lock. */
                add_release(&cpu_state->local_queues, tsk);
                TRACE_TASK(tsk, "added to release queue on reschedule\n");
        }
}

static int p_edf_check_for_preemption_on_release(rt_domain_t *local_queues)
{
        struct p_edf_cpu_state *state = container_of(local_queues,
                                                struct p_edf_cpu_state,
                                                    local_queues);

        /* Because this is a callback from rt_domain_t we already hold
            * the necessary lock for the ready queue. */

        if (edf_preemption_needed(local_queues, state->scheduled)) {
                preempt_if_preemptable(state->scheduled, state->cpu);
                return 1;
        }
        return 0;
}

static long p_edf_activate_plugin(void)
{
        int cpu;
        struct p_edf_cpu_state *state;
        for_each_online_cpu(cpu) {
                TRACE("Initializing CPU%d...\n", cpu);
                state = cpu_state_for(cpu);
                state->cpu = cpu;
                state->scheduled = NULL;
                edf_domain_init(&state->local_queues,
                                p_edf_check_for_preemption_on_release,
                                NULL);
        }

        p_edf_setup_domain_proc();
        return 0;
}

static long p_edf_deactivate_plugin(void)
{
        destroy_domain_proc_info(&p_edf_domain_proc_info);
        return 0;
}



static struct task_struct* p_edf_schedule(struct task_struct * prev)
{
        struct p_edf_cpu_state *local_state = local_cpu_state();

        /* 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
                anywhere else. This
253             * is the case when tasks exit by themselves; additional queue
                management is
254             * is required if tasks are forced out of real-time mode by other tasks
                . */
255
256         if (is_queued(tsk)){
```

```
257                 edf = task_edf(tsk);
258                 remove(edf,tsk);
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 reqeued\n");
298                 p_edf_requeue(tsk, state);
299                 if (edf_preemption_needed(&state->local_queues, state->scheduled))
                        {
300                     preempt_if_preemptable(state->scheduled, state->cpu);
301                 }
302         }
303
304         raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
305 }
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 7 – Partie du fichier .config liée a LITMUS^{RT}

```
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
```

Table des figures

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.

Glossaire

bootloader court programme chargé au démarrage de l'ordinateur initialisant le système d'exploitation. 4

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

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. 7

processeur Ca c'est la définition. 7, 8

SOC ou *Système On a Chip* est . 4