

Rapport de stage Ingénieur

-

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

-

BELPOIS Vincent

2023



Table des matières

Présentation du stage	3
0.1 Le L.I.A.S.	3
0.2 Le sujet du stage	3
1 Systèmes d'exploitation compatibles avec la carte ROCK960	4
1.1 Présentation de la carte de développement	4
1.2 Installation d'un système d'exploitation	4
1.2.1 Installation d'une image précompilée	4
1.2.2 Compilation de Linux depuis le code source	4
1.2.3 Compilation croisée	4
1.3 Etude des versions de Linux compatibles	5
1.3.1 Comment Linux gère le support d'un processeur	5
1.3.2 Essais de différentes versions	5
2 LITMUS^{RT}	6
2.1 Présentation de LITMUS ^{RT}	6
2.2 Présentation de <i>feather-trace</i>	6
2.3 Implémentation d'un ordonnanceur EDF partitionné	6
2.3.1 Algorithme considéré	6
2.3.2 Implémentation	6
Annexe	7

Présentation du stage

0.1 Le L.I.A.S.

0.2 Le sujet du stage

Mon stage s'intéresse à l'implémentation d'un ordonnanceur sur plateforme hétérogène[1]

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'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.

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.

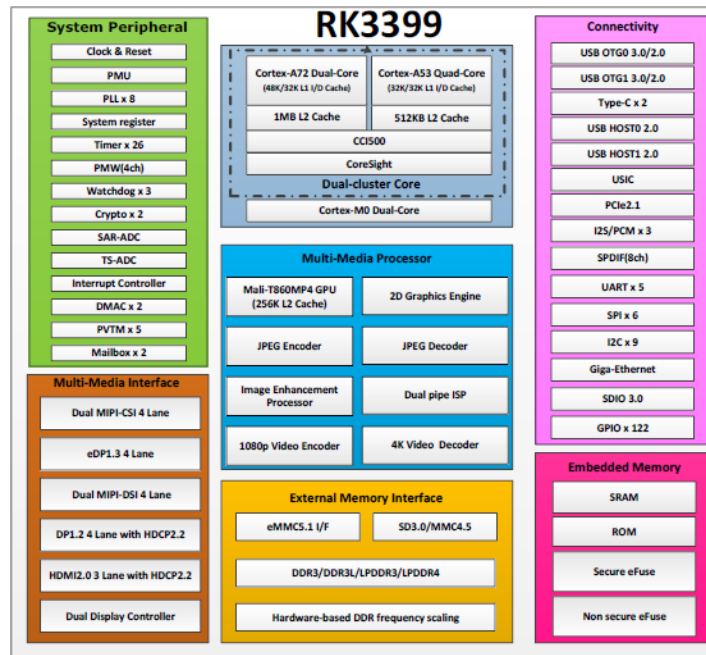


FIGURE 1 – Architecture du processeur RK3399

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.

1.2 Installation d'un système d'exploitation

1.2.1 Installation d'une image précompilée

Image Linux ubuntu fournie par 96boards

1.2.2 Compilation de Linux depuis le code source

Où est le code source linux ?

Comment changer de version du code cloné.

comment modifier quels modules sont chargés lors de la compilation de linux, interface graphique
<-> fichier de config

1.2.3 Compilation croisée

Toolchain : qu'est ce que c'est

Variables d'environnement

Réalisation de scripts linux pour accélérer le développement

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

1.3.2 Essais de différentes versions

2 LITMUS^{RT}

2.1 Présentation de LITMUS^{RT}

2.2 Présentation de *feather-trace*

2.3 Implémentation d'un ordonnanceur EDF partitionné

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 fournis 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 choisit 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 tâches périodiques :

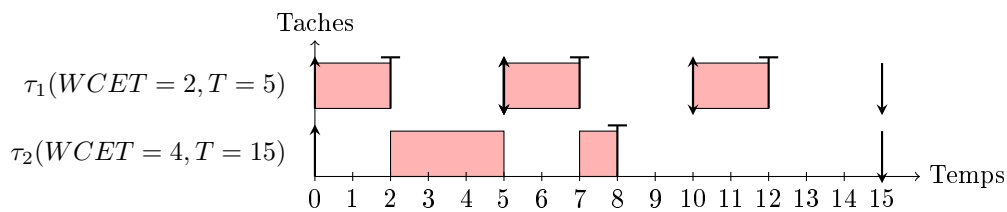


FIGURE 2 – Exemple de EDF à 2 tâches

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.

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 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 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 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 discutée dans la partie 1.2.2.

Annexe

Listing 1 – linux/litmus/Makefile

```

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 2 – linux/litmus/sched_p_edf.c

```

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 *

```



```

    cpu_state)
{
78     if (is_released(tsk, litmus_clock())) {
79         /* Uses __add_ready() instead of add_ready() because we already
80          * hold the ready lock. */
81         __add_ready(&cpu_state->local_queues, tsk);
82         TRACE_TASK(tsk, "added to ready queue on reschedule\n");
83     } else {
84         /* Uses add_release() because we DON'T have the release lock. */
85         add_release(&cpu_state->local_queues, tsk);
86         TRACE_TASK(tsk, "added to release queue on reschedule\n");
87     }
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, struct
94         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 static struct task_struct* p_edf_schedule(struct task_struct * prev)
133 {
134     struct p_edf_cpu_state *local_state = local_cpu_state();
135
136     /* next == NULL means "schedule background work". */

```

```

137     struct task_struct *next = NULL;
138
139     /* prev's task state */
140     int exists, out_of_time, job_completed, self_suspends, preempt, resched;
141
142     raw_spin_lock(&local_state->local_queues.ready_lock);
143
144     BUG_ON(local_state->scheduled && local_state->scheduled != prev);
145     BUG_ON(local_state->scheduled && !is_realtime(prev));
146
147     exists = local_state->scheduled != NULL;
148     self_suspends = exists && !is_current_running();
149     out_of_time = exists && budget_enforced(prev) && budget_exhausted(prev);
150     job_completed = exists && is_completed(prev);
151
152     /* preempt is true if task 'prev' has lower priority than something on
153      * the ready queue. */
154     preempt = edf_preemption_needed(&local_state->local_queues, prev);
155
156     /* check all conditions that make us reschedule */
157     resched = preempt;
158
159     /* if 'prev' suspends, it CANNOT be scheduled anymore => reschedule */
160     if (self_suspends) {
161         resched = 1;
162     }
163
164     /* also check for (in-)voluntary job completions */
165     if (out_of_time || job_completed) {
166         p_edf_job_completion(prev, out_of_time);
167         resched = 1;
168     }
169
170     if (resched) {
171         /* First check if the previous task goes back onto the ready
172          * queue, which it does if it did not self_suspend.
173          */
174         if (exists && !self_suspends) {
175             p_edf_requeue(prev, local_state);
176         }
177         next = __take_ready(&local_state->local_queues);
178     } else {
179         /* No preemption is required. */
180         next = local_state->scheduled;
181     }
182
183     local_state->scheduled = next;
184     if (exists && prev != next) {
185         TRACE_TASK(prev, "descheduled.\n");
186     }
187     if (next) {
188         TRACE_TASK(next, "scheduled.\n");
189     }
190
191     /* This mandatory. It triggers a transition in the LITMUSRT remote
192      * preemption state machine. Call this AFTER the plugin has made a
193      * local
194      * scheduling decision.
195      */
196     sched_state_task_picked();

```

```

197     raw_spin_unlock(&local_state->local_queues.ready_lock);
198     return next;
199 }
200
201 static long p_edf_admit_task(struct task_struct *tsk)
202 {
203     if (task_cpu(tsk) == get_partition(tsk)) {
204         TRACE_TASK(tsk, "accepted by p_edf plugin.\n");
205         return 0;
206     }
207     return -EINVAL;
208 }
209
210 static void p_edf_task_new(struct task_struct *tsk, int on_runqueue,
211                          int is_running)
212 {
213     /* We'll use this to store IRQ flags. */
214     unsigned long flags;
215     struct p_edf_cpu_state *state = cpu_state_for(get_partition(tsk));
216     lt_t now;
217
218     TRACE_TASK(tsk, "is a new RT task %llu (on runqueue:%d, running:%d)\n",
219               litmus_clock(), on_runqueue, is_running);
220
221     /* Acquire the lock protecting the state and disable interrupts. */
222     raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);
223
224     now = litmus_clock();
225
226     /* Release the first job now. */
227     release_at(tsk, now);
228
229     if (is_running) {
230         /* If tsk is running, then no other task can be running
231          * on the local CPU. */
232         BUG_ON(state->scheduled != NULL);
233         state->scheduled = tsk;
234     } else if (on_runqueue) {
235         p_edf_requeue(tsk, state);
236     }
237
238     if (edf_preemption_needed(&state->local_queues, state->scheduled))
239         preempt_if_preemptable(state->scheduled, state->cpu);
240
241     raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
242 }
243
244 static void p_edf_task_exit(struct task_struct *tsk)
245 {
246     unsigned long flags;
247     struct p_edf_cpu_state *state = cpu_state_for(get_partition(tsk));
248     raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);
249     rt_domain_t*      edf;
250
251     /* For simplicity, we assume here that the task is no longer queued
252      * anywhere else. This
253      * is the case when tasks exit by themselves; additional queue
254      * management is
255      * is required if tasks are forced out of real-time mode by other tasks
256      . */

```

```

255     if (is_queued(tsk)){
256         edf = task_edf(tsk);
257         remove(edf,tsk);
258     }
259
260     if (state->scheduled == tsk) {
261         state->scheduled = NULL;
262     }
263
264     preempt_if_preemptable(state->scheduled, state->cpu);
265     raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
266 }
267
268 /* Called when the state of tsk changes back to TASK_RUNNING.
269  * We need to requeue the task.
270  *
271  * NOTE: If a sporadic task is suspended for a long time,
272  * this might actually be an event-driven release of a new job.
273  */
274 static void p_edf_task_resume(struct task_struct *tsk)
275 {
276     unsigned long flags;
277     struct p_edf_cpu_state *state = cpu_state_for(get_partition(tsk));
278     lt_t now;
279     TRACE_TASK(tsk, "wake_up at %llu\n", litmus_clock());
280     raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);
281
282     now = litmus_clock();
283
284     if (is_sporadic(tsk) && is_tardy(tsk, now)) {
285         /* This sporadic task was gone for a "long" time and woke up past
286          * its deadline. Give it a new budget by triggering a job
287          * release. */
288         inferred_sporadic_job_release_at(tsk, now);
289         TRACE_TASK(tsk, "woke up too late.\n");
290     }
291
292     /* This check is required to avoid races with tasks that resume before
293      * the scheduler "noticed" that it resumed. That is, the wake up may
294      * race with the call to schedule(). */
295     if (state->scheduled != tsk) {
296         TRACE_TASK(tsk, "is being requeued\n");
297         p_edf_requeue(tsk, state);
298         if (edf_preemption_needed(&state->local_queues, state->scheduled))
299             {
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 static struct sched_plugin p_edf_plugin = {
308     .plugin_name      = "P-EDF",
309     .schedule         = p_edf_schedule,
310     .task_wake_up     = p_edf_task_resume,
311     .admit_task       = p_edf_admit_task,
312     .task_new         = p_edf_task_new,
313     .task_exit        = p_edf_task_exit,
314     .get_domain_proc_info = p_edf_get_domain_proc_info,

```

```
315     .activate_plugin      = p_edf_activate_plugin,  
316     .deactivate_plugin    = p_edf_deactivate_plugin,  
317     .complete_job         = complete_job,  
318 };  
319  
320 static int __init init_p_edf(void)  
321 {  
322     return register_sched_plugin(&p_edf_plugin);  
323 }  
324  
325 module_init(init_p_edf);
```

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.

Table des figures

1	Architecture du processeur RK3399	4
2	Exemple de EDF à 2 tâches	6

Glossaire

cluster ensemble interconnecté de plusieurs processeurs. 4

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

processeur Ca c'est la définition. 6

SOC ou *Système On a Chip* est . 4