Rapport LIFL # 2003-05

Éléments d'implantation d'un noyau ARTiS

Momtchil MOMTCHEV
Momtchil.Momtchev@lifl.fr

Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille
France

Septembre, 2003

Résumé

ARTiS est un ordonnancement asymétrique de processus temps réel [1]. ARTiS est une souche possible pour la production d'un système temps réel dans le cadre du projet ITEA HYADES.

Le présent rapport technique accompagne les premières modifications du noyau Linux 2.5.69 SMP dans le but d'implanter ARTiS.

Un modèle des processus temps réel ARTiS compatible avec les objectifs du projet HYADES est exposé. Les premiers éléments d'implantation sont commentés et des perspectives d'implantation sont fournies.

1 Contexte

Le but du modèle ARTiS est de fournir une approche de modification du noyau Linux qui évite certains des problèmes posés par les implémentations de systèmes temps réel existantes. Il existe aujourd'hui deux grands groupes de noyaux Linux modifiés pour le temps réel. Les noyaux à deux niveaux, tels que RT-Linux et RTAI, fournissent une couche basse temps réel implémentée dans un exécutif « greffé » au noyau. L'API temps réel est séparée de l'API traditionnelle Linux et en général il s'agit d'une API propriétaire et de bas niveau. Ces noyaux sont les seuls à proposer le support d'un temps réel « dur », à la QNX, dans lequel un temps de réponse du noyau (temps de latence) est garanti quand le système est en mode « opérationnel » (hors les phases de démarrage et de paramétrage). L'autre approche traditionnelle s'attache à baisser la durée des sections dites « critiques » au sein du noyau, dont la durée maximale est le facteur limitant du temps de latence. Ici il s'agit d'une autre notion de temps réel, le temps réel « mou » : seul un temps de latence moyen est garanti.

D'après les définitions adoptées par le projet HYADES, le temps réel visé est le temps réel « dur », ce que recouvre la définition classique du temps réel, dans laquelle la durée de toute attente d'un processus devra être bornée.

Le noyau Linux actuel (qu'il s'agit des versions 2.4.x ou de 2.5.x) échappe à cette définition à cause des nombreuses sections critiques, dont la durée d'exécution n'est pas bornée ou par des temps trop importants (de l'ordre de quelques secondes dans des situations particulières avec les noyaux 2.4.16+). Bien qu'en général ceci soit vrai aussi bien dans le cas de monoprocesseurs que de multiprocesseurs SMP, il existe une différence importante : les sections critiques en SMP sont rarement des sections critiques « globales », c'est-à-dire des sections critiques masquant les interruptions sur l'ensemble des processeurs. Nos études ont montré que sur une machine quadriprocesseur à 450 MHz la durée maximale des sections critiques globales peut être bornée par 125 μs — la période minimale de horloge RTC, sauf pour quelques exceptions. D'autres études placent cette limite dans les 50 μs pour certains noyaux spécifiques (les premières versions de 2.4.x, avant 2.4.15/16). Heureusement, la plupart des sections critiques globales de durée supérieure de 125 μs concernent la phase de démarrage et d'initialisation de matériel (par exemple le chargement d'un module ou l'appel de la commande `hdparm`, utilitaire de paramétrage des disques durs IDE/SCSI). Il existe cependant quelques sous-systèmes du noyau qui comportent des sections critiques globales de durées exceptionnelles – notamment la gestion des consoles virtuelles et les E/S sur bus AGP pour les cartes 3D. Le problème est connu de la communauté Linux et il est en train d'être résolu en 2.5.x, bien que l'inclusion du nouveau code de gestion de consoles virtuelles dans les versions 2.6 ne soit pas certain.

La durée d'exécution maximale des sections critiques globales étant limitée, on ordonnance les processus de façon asymétrique pour qu'un certain nombre de processeurs ne se trouvent jamais en section critique « locale », c'est-à-dire une section critique masquant uniquement les interruptions locales. Dans un tel système le temps de réponse sera limité par la durée maximale des sections critiques globales avec une seule exception : la routine de changement de contexte de l'ordonnanceur. Cette routine, dont le code se trouve surtout dans la fonction `schedule()` du noyau, devra être « exemptée » des restrictions d'exécution. Cette fonction était relativement pénalisante en 2.4.x, avec un algorithme en $O(N)$ par rapport au nombre de processus en exécution (donc non borné) et elle était le facteur principal limitant notre temps de réponse maximal. Le nouvel ordonnanceur implémenté en 2.5.x ne souffre plus de ces limitations, sa complexité vis-à-vis du nombre de processus étant en $O(1)$.

2 Le modèle des processus temps réel ARTiS

Le modèle ARTiS suppose une application composée

– d'un petit nombre de processus bas niveau effectuant des E/S sur du matériel spécialisé avec des exigences temps réel très strictes : les RT0 ;
– d'un plus grand nombre de processus de calcul scientifique, avec des exigences

temps réel moins strictes, mais toujours « durs » : les RT1+ ; et
- d'un certain nombre de processus de fond, comme la création de journaux ou la visualisation, avec des exigences temps réel « mou ».

Pour prendre en compte de telles applications on propose un modèle d'ordonnancement par préemption et par priorité statique pour les processus temps réel « dur » et le maintien du modèle d'ordonnancement Linux pour les autres processus. Dans ce modèle la priorité RT0 occupe une place un peu particulière : par définition ces processus seront les seuls à ne pas être préemptibles. Ceci va permettre les accès libres au matériel ou à un pilote temps réel dans le noyau Linux. Un processus RT0 peut également être un thread noyau (dans le cas d'un pilote en mode noyau), mais dans ce cas il faudra gérer son ordonnancement manuellement. Les processus RT1+ sont préemptibles par tout processus temps réel de priorité supérieure et ils sont soumis aux mêmes contraintes d'exécution que le reste du système.

Le but est de laisser la possibilité aux processus de bas niveau de priorité RT0 de tourner sur un processeur fixé qui pourra aussi accueillir les interruptions du matériel géré. Le reroutage des interruptions est une procédure assez lente à cause de limitations matérielles et il n'est pas envisageable de la faire au vol sans une perte non négligeable au niveau du temps de latence. De cette façon les processus temps réel de priorités inférieures seront ordonnancés dans les « trous » d'exécution des RT0. On suppose que ces processus de calcul scientifique vont effectuer moins de transitions mode utilisateur / mode noyau, afin de garantir l'efficacité du modèle.

Le nombre de processeurs dits temps réel, c'est-à-dire les processeurs avec restrictions, est déterminé par le nombre de processus RT0 et il est limité par le nombre total de processeurs moins un.

3 Implémentation

3.1 Entrée/sortie en section critique (spinlocks)

La restriction centrale d'ARTiS, restriction d'entrée en section critique sur certains processeurs, est implémentée dans le code des spinlocks.

```
#ifdef CONFIG_PREEMPT

 extern void preempt_schedule(void);
+extern void artis_barrier(void);

 #define preempt_disable() \
 do \{ \
+        artis_barrier(); \
         inc_preempt_count(); \
         barrier(); \
 \} while (0)
```

Deux cas sont exemptés de cette restriction : le code de l'ordonnanceur et le code des processus de priorité RT0. A fin de pouvoir distinguer les appels provenant de l'ordonnanceur, un drapeau, `schedule_in_progress` est utilisé pour marquer les phases d'ordonnancement. Le mode d'exécution des processeurs est stocké dans un tableau, le `artis_rt_cpus`, mais pour que la manipulation des spinlocks sur les processeurs non-temps réel ne soit pas pénalisée, il est envisageable de conserver les deux versions de la fonction.

```
@@ -2592,5 +2627,29 @@
                        cpu_relax();
                preempt_disable();
        } while (!_raw_write_trylock(lock));
+}
+
+void artis_barrier(void) {
+
+       if (in_atomic() || in_interrupt())
+               return;
+
+       if (!this_rq())
+               return;
+
+       task_t *p = this_rq()->curr;
+
+       if (!p)
+               return;
+
+       if (!artis_rt_cpus[smp_processor_id])
+               return;
+       if (p->policy == SCHED_FIFO)
+               return;
+       if (this_rq()->schedule_in_progress)
+               return;
+       if (p->pid == 0 || p->pid == 1)
+               return;
+       p->state = TASK_UNINTERRUPTIBLE;
+       schedule();
 }
 #endif
```

Malheureusement, la question des limites de l'ordonnanceur est plus difficile que cela le semble (le drapeau `schedule_in_progress`). On pourra étudier assez facilement le cas général à l'aide d'un traceur à l'intérieur du noyau mais il reste les cas

particuliers des fonctions `exit()` et `fork()`. Une partie de ces fonctions est imbriquée dans le code d'entrée/sortie de mode noyau et le code de changement de contexte. Cette partie est fortement dépendante de l'architecture.

3.2 Ordonnancement

Le modèle exige la création de nouvelles queues d'exécution temps réel, pour ne pas être restreint par le modèle de verrouillage de queues Linux. Ce modèle a beaucoup évolué lors du développement de 2.5.x, dans la situation actuelle on peut accéder à la queue de 5 endroits différents : l'ordonnanceur local, le thread de migration local, un thread de migration distant et les procédures `fork()` et `exit()`. Pour éviter tout mécanisme explicite de synchronisation, on accepte la restriction d'accéder à la queue temps réel uniquement à partir du processeur local en mode non-préemptible.

```
        spinlock_t lock;
        unsigned long nr_running, nr_switches, expired_timestamp,
                        nr_uninterruptible;
+       unsigned long nr_rt_running;
        task_t *curr, *idle;
        struct mm_struct *prev_mm;
-       prio_array_t *active, *expired, arrays[2];
+       prio_array_t *active, *expired, arrays[4];
+       prio_array_t *realtime, *blocked;
+       int schedule_in_progress;
        int prev_cpu_load[NR_CPUS];
 #ifdef CONFIG_NUMA
        atomic_t *node_nr_running;
@@ -331,6 +334,17 @@
 static inline void __activate_task(task_t *p, runqueue_t *rq)
 {
        enqueue_task(p, rq->active);
+       p->array = rq->active;
+       nr_running_inc(rq);
+}
+
+/*
+ * __activate_rt_task
+ */
+static inline void __activate_rt_task(task_t *p, runqueue_t *rq)
+{
+       enqueue_task(p, rq->realtime);
+       p->array = rq->realtime;
        nr_running_inc(rq);
```

```
 }
```

Les processus seront placés dans ces queues lors d'un appel système `set_schedule()`. La mémoire virtuelle qui leur appartient sera verrouillée en mémoire physique par `mlock()`.

```
@@ -1319,15 +1341,19 @@
                goto switch_tasks;
        }

-       array = rq->active;
-       if (unlikely(!array->nr_active)) {
-               /*
-                * Switch the active and expired arrays.
-                */
-               rq->active = rq->expired;
-               rq->expired = array;
+       if (likely(rq->realtime->nr_active))
+               array = rq->realtime;
+       else {
+               array = rq->active;
-               rq->expired_timestamp = 0;
+               if (unlikely(!array->nr_active)) {
+                       /*
+                        * Switch the active and expired arrays.
+                        */
+                       rq->active = rq->expired;
+                       rq->expired = array;
+                       array = rq->active;
+                       rq->expired_timestamp = 0;
+               }
        }

        idx = sched_find_first_bit(array->bitmap);

@@ -1778,12 +1805,16 @@
        retval = 0;
        p->policy = policy;
        p->rt_priority = lp.sched_priority;
-       if (policy != SCHED_NORMAL)
+       if (policy != SCHED_NORMAL) {
+               p->prio = MAX_USER_RT_PRIO-1 - p->rt_priority;
-       else
```

```
+               if (array)
+                       __activate_rt_task(p, task_rq(p));
+       }
+       else {
                p->prio = p->static_prio;
-       if (array)
-               __activate_task(p, task_rq(p));
+               if (array)
+                       __activate_task(p, task_rq(p));
+       }

out_unlock:
        task_rq_unlock(rq, &flags);
@@ -2501,12 +2532,15 @@
                rq = cpu_rq(i);
                rq->active = rq->arrays;
                rq->expired = rq->arrays + 1;
+               rq->realtime = rq->arrays + 2;
+               rq->blocked = rq->arrays + 3;
+               rq->schedule_in_progress = 1;
                spin_lock_init(&rq->lock);
                INIT_LIST_HEAD(&rq->migration_queue);
                atomic_set(&rq->nr_iowait, 0);
                nr_running_init(rq);

-               for (j = 0; j < 2; j++) {
+               for (j = 0; j < 4; j++) {
                        array = rq->arrays + j;
                        for (k = 0; k < MAX_PRIO; k++) {
                                INIT_LIST_HEAD(array->queue + k);
@@ -2533,6 +2567,7 @@
         */
        atomic_inc(&init_mm.mm_count);
        enter_lazy_tlb(&init_mm, current, smp_processor_id());
+       rq->schedule_in_progress = 0;
 }
```

Une modification de la fonction `schedule()` est également nécessaire pour s'assurer qu'elle ne reste jamais en attente d'un verrou partagé avec un autre processeur alors qu'un processus temps réel est prêt à l'exécution.

3.3 Migration « forcée »

Sur un processeur temps réel, toute tentative d'acquérir un spinlock provoque une migration forcée du processus. Les noyaux 2.5.x disposent déjà d'un mécanisme de migration qui implémente des queues de migration mais son utilisation directe n'est pas possible à cause du modèle de verrouillage utilisé (barrière). À moins que ce modèle évolue dans les prochaines versions du noyau, il faudra développer un mécanisme tampon utilisant une queue circulaire sans verrouillage avant le transfert vers la queue de migration.

3.4 Threads noyau

Les threads noyaux de bas niveau qui effectuent des tâches de fond (tel `ksoftirqd`) ne devront pas être exécutés sur les processeurs temps réel.

Il est assez facile de rediriger tous les appels vers les threads sur le reste des processeurs.

3.5 Appels système par les processus temps réel

Le code des appels système n'étant pas compatible temps réel, aucune garantie ne peut être fournie sur leur temps d'exécution. La possibilité d'utilisation directe du sous-système des appels système par les processus temps réel n'a pas d'intérêt particulier, mais elle est envisageable avec le modèle ARTiS. Pendant l'exécution des appels système les processus temps réel seront considérés des processus temps réel « mou » si ils tentent d'accéder à des verrous système. Cela laisse la possibilité de revoir le code de certains appels système (notamment les `read()` et `write()` non bloquants) pour permettre une communication entre les processus sans pénalisation du côté temps réel.

3.6 Communication entre les processus

Les moyens de communications classiques du noyau Linux n'étant pas adaptés aux besoin du temps réel, il sera nécessaire que les processus temps réel communiquent à l'aide de zones de mémoire partagées et verrouillées par `mlock()` en assurant eux-mêmes leur synchronisation. La modification de ces mécanismes n'est pas triviale parce qu'ils sont fortement imbriqués dans la couche VFS de Linux et toute modification importante sera difficile à maintenir avec l'évolution des différentes versions de Linux. Concernant les communications entre un processus temps réel et un processus non-temps réel, et comme nous le proposons pour les queues de migration, il est possible de créer un chemin d'exécution séparé temps réel pour les appels système `read()` et `write()` en mode non bloquant implanté au dessus d'un mécanisme tampon avec queue circulaire sans verrous.

4 Spécificités IA-64

Le modèle de fonctionnement d'ARTiS, tel que présenté ici dépend fortement de la possibilité de préempter un processus tournant en mode noyau quand il ne détient aucun verrou (spinlock). Cette fonctionnalité, déjà présente en 2.5.x, est toujours en développement dans la version IA-64.

Certains aspects du fonctionnement du noyau, tels que le changement de contexte, la transition entre mode noyau et mode utilisateur et la gestion des spinlocks (utilisation de barrières mémoire IA-64) présentent également des différences importantes dans la version IA-64 par rapport à la version IA-32.

Références

[1] Momtchil Momtchev and Philippe Marquet. An asymmetric real-time scheduling for Linux. In *Tenth International Workshop on Parallel and Distributed Real-Time Systems*, Fort Lauderdale, FL, April 2002.