FROM RESEARCH TO INDUSTRY

cea tech

Ordonnancement temps réel multiprocesseurs : théorie et pratique

Mathieu.Jan@cea.fr

Ecole d'été Temps Réel (ETR)

30 août 2017

Telecom ParisTech

Objectif de l'exposé

- Vous donner des bases pour …
  - … comprendre la problématique
  - … les stratégies existantes pour la résoudre
  - … et comment implémenter ces stratégies
- A travers des exemples concrets
- → Ne vise pas à être exhaustif sur la thématique

Plan

- Contexte & hypothèses
  - Architecture matérielle & modèle de tâche
  - Approches d'ordonnancement
- Stratégies d'ordonnancement et implémentations au sein de LITMUS^{RT}
  - Approche partitionnée : P-EDF
  - Approche globale : G-EDF
  - Illustration de propriétés & d'anomalies
  - Aperçu P-Fair

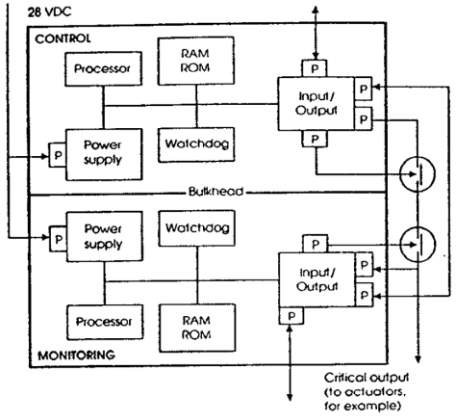
Evolution des architectures matérielles (1/3)

Apollo Guidance System (1960)

- 1 processeur 16 bits, fréquence 1 MHz
- 4 ko RAM et 72 Ko ROM
- Multi-tâches : jusqu'à 9 tâches courtes
  + 4 tâches longues
  - Waitlist : contrainte sur la durée des tâches (4ms max)
  - Executive : ordonnancement préemptif (~20ms) à priorité fixe

Commandes de vol

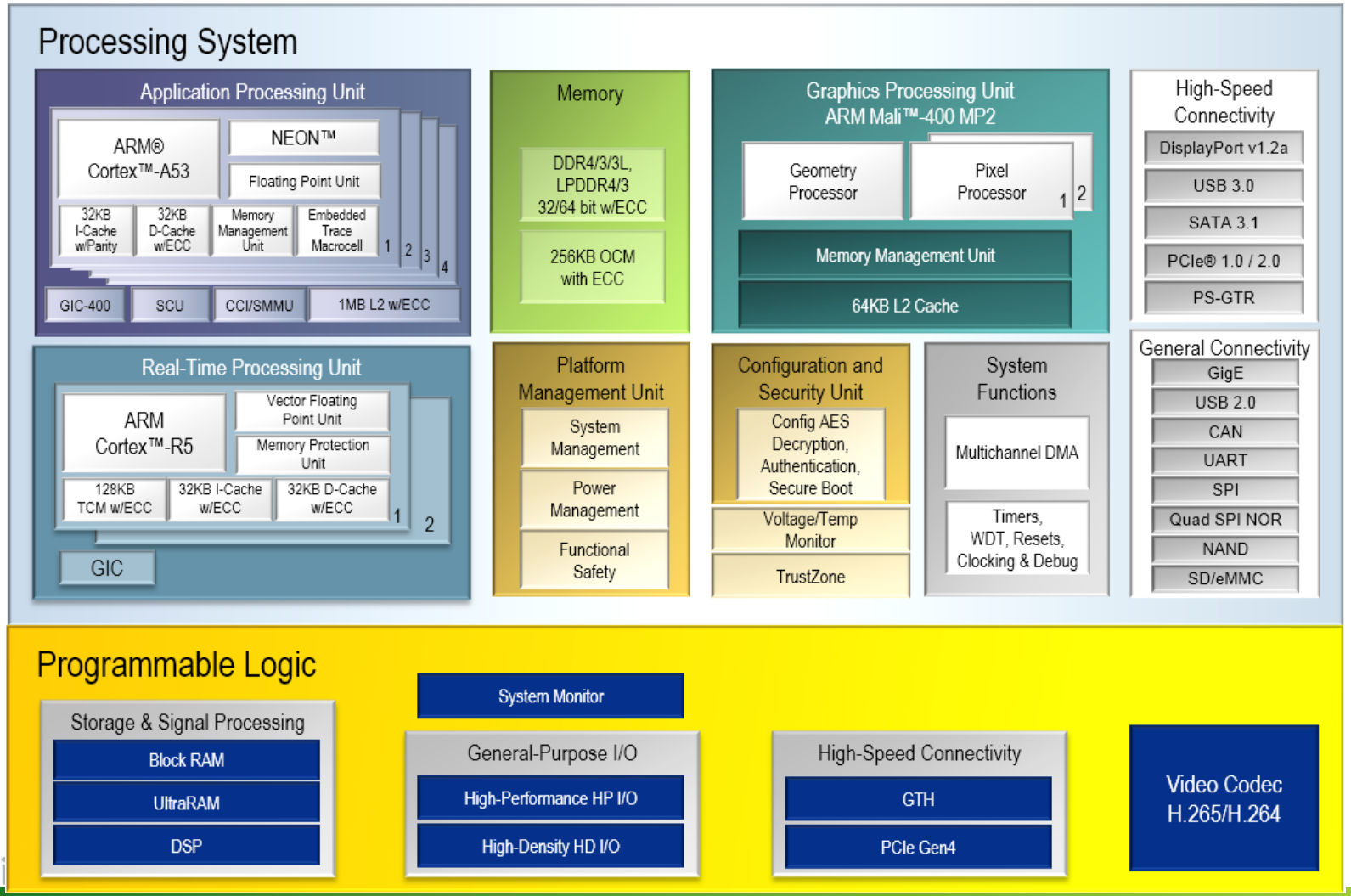
- Commercial : Concorde (électronique analogique) / A320 (entièrement numérique)
- Ensemble de Motorola 68010 et Intel 80186
  - 68010 : 2 à 10 MHz, 16 Mo et cache d'instruction basique
  - 80186 : 6 MHz, 1Mo



Evolution des architectures matérielles (2/3)

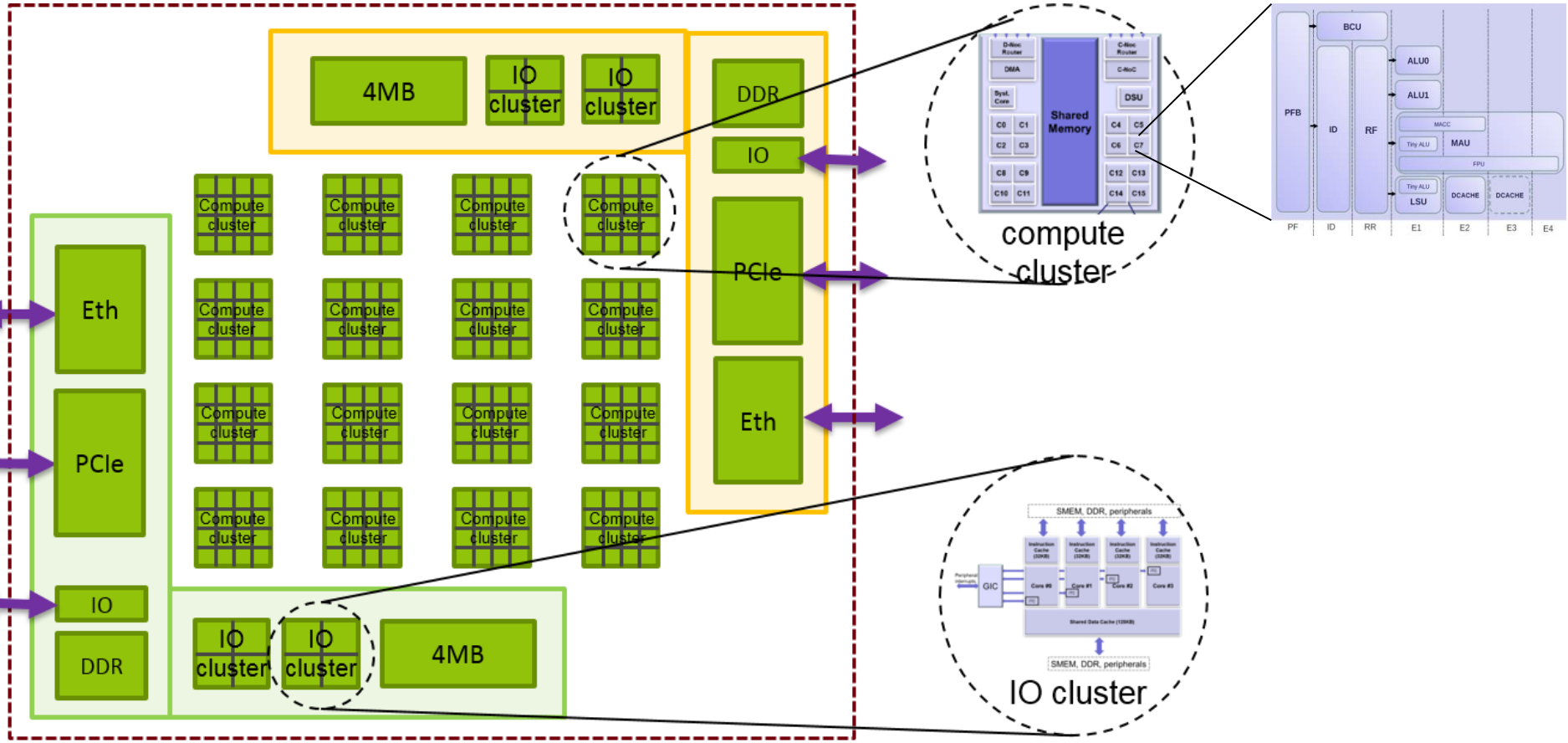
Xilinx Zynq Ultrascale+ MPSoc

- 1-4 Cortex A53 avec caches d'instruction et données 32 Ko + 1 Mo L2
- 2 Cortex R5 avec caches d'instruction et données 32 Ko + TCM de 128 Ko



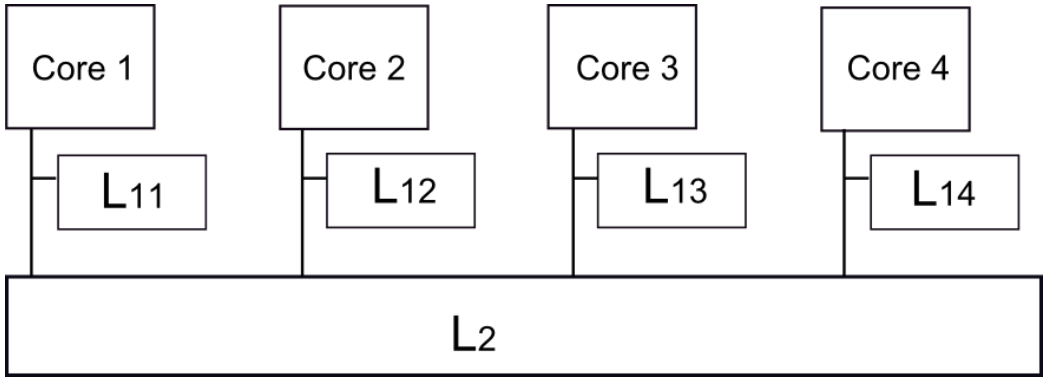
Evolution des architectures matérielles (3/3)

- **MPPA Bostan Kalray (2016)**
  - 256 cœurs de calculs
  - 16 x 2 Mo SRAM + 2 x 4 Mo SRAM + caches instruction (8Ko) et données (8ko)



Architecture matérielle considérée

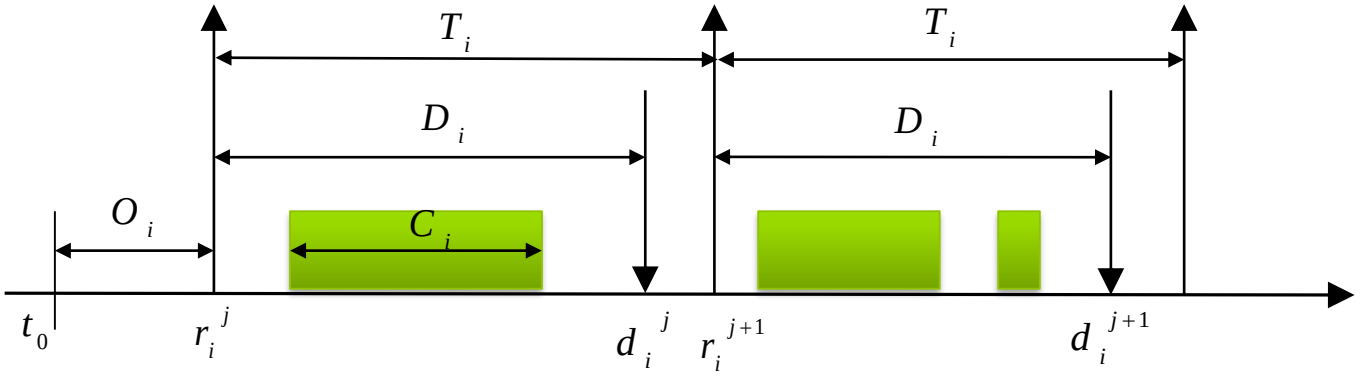
Un exemple d'architecture



- Hiérarchie mémoire : caches
  - L1 privé à chaque cœur
  - L2 partagé : noté « Last Level Cache » (LLC)
- Ensemble de m processeurs
  - **Identiques/homogènes** : les processeurs ont la même capacité de calcul
  - Uniformes : chaque processeur a une capacité de calcul différente
  - Indépendants/hétérogènes : la capacité de calcul est fonction du processeur et de la tâche exécutée
- Base de temps commune

Modèles de tâche

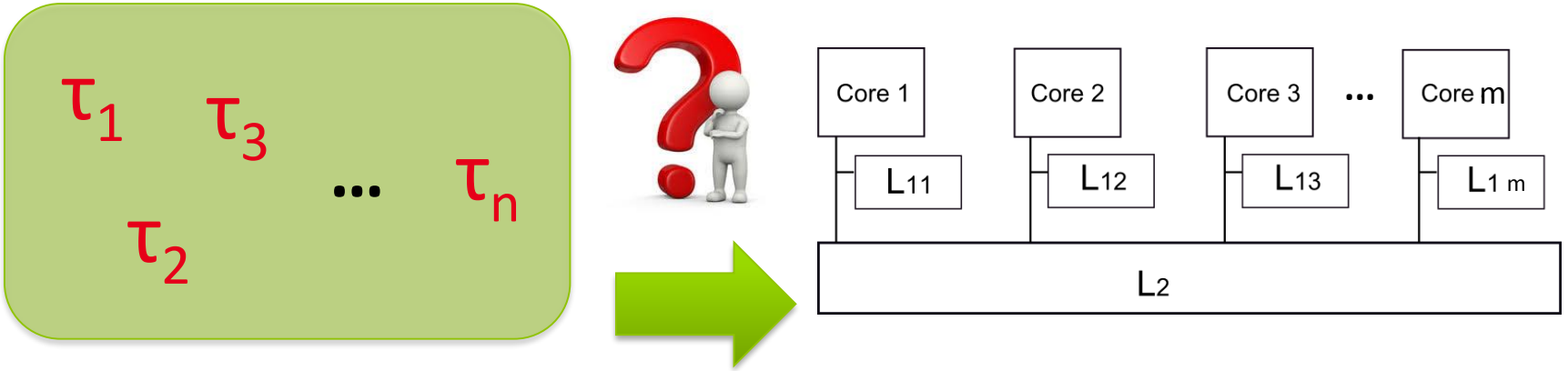
- Ensemble de n tâches indépendantes $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$
  - Tâches **périodiques** générant une collection infinie de travaux (non parallèles)
  - Paramètres statiques : (O_i, C_i, D_i, T_i)
  - A échéance implicites $(D_i = T_i)$, contraintes $(D_i < T_i)$, arbitraire $(D_i ? T_i)$
  - **Date de réveil : synchrones**
  - Utilisation : $u_i = C_i / T_i$



- Rappel : algorithmes d'ordonnancement à …
  - … priorité fixe au niveau des tâches (exemple : RM)
  - … priorité fixe au niveau des travaux (exemple : EDF)
  - … priorité dynamique au niveaux des travaux (exemple : LLF)

Conséquences sur l'ordonnancement

- Multiprocesseurs : problème à 2 dimensions [Liu 1969]
  - « bringing in additional processors adds a new dimension to the scheduling problem »
  - « Few of the results obtained for a single processor generalize directly to the multiple processor case »
- Problème d'affectation des priorités
  - Organisation temporelle : quand démarrer, suspendre, reprendre l'exécution des tâches
- Problème d'allocation des tâches
  - Organisation spatiale : sur quels processeurs ?
  - Pas de migration, migration au niveau de la tâche, au niveau du travail



Approche partitionnée (1/2)

- Organisation spatiale : répartition de n tâches en m sous-ensembles
  - Classement des tâches selon un paramètre : croissant ou décroissant
    - Période, charge, priorité
  - Affectation aux processeurs : heuristiques lors du parcours séquentiel des tâches
    - First Fit, Best Fit, Worst Fit, Next Fit
  - Problème NP-difficile : analogie avec le problème du sac à dos
- Organisation temporelle : ordonnanceurs monoprocesseur pour chacun des m sous-ensembles
- Avantages/inconvénients
  - Pas coûts de migration mais surcharge limité à un processeur
  - Facilité mise en œuvre
  - Borne d'utilisation : 50%
    - $n = m + 1$ tâches avec $T_i = 1, C_i = 0,5 + \varepsilon$ → au moins un processeur avec $1 + 2\varepsilon$

Approche partitionnée (2/2)

- Next Fit (NF)
  - Ajout de la tâche τ_i au processeur j (si condition de faisabilité respectée)
  - Sinon, au processeur $j + 1$
- First Fit (FF)
  - Ajout de la tâche τ_i au processeur j en partant du processeur $j = 0$ (….)
- Best Fit (BF)
  - Ajout de la tâche τ_i au processeur j en partant du processeur $j = 0$ et pour lequel le taux d'utilisation généré est maximisé
- Worst Fit (WF)
  - L'inverse de Best Fit

Approche globale et autres

- Organisation spatiale et temporelle conjointe
  - Une seule liste des tâches à traiter par l'ordonnanceur
  - Allocation de au plus de m tâches par décision d'ordonnancement
- Migrations possibles et ordonnançabilité théorique optimale mais problème de passage à l'échelle en pratique
- Autres
  - Semi-partitionnée : partitionnée + quelques tâches migrantes (définis statiquement)
  - Clustérisée : processeurs organisés en $n_{cluster}$ cluster ($\leq m$) et ordonnancement global au sein de chaque cluster

Plan

- Contexte & hypothèses
  - Architecture matérielle & modèle de tâche
  - Approches d'ordonnancement
- Stratégies d'ordonnancement et implémentations au sein de LITMUS^{RT}
  - Approche partitionnée : P-EDF
  - Approche globale : G-EDF
  - Illustration de propriétés & d'anomalies
  - Aperçu P-Fair

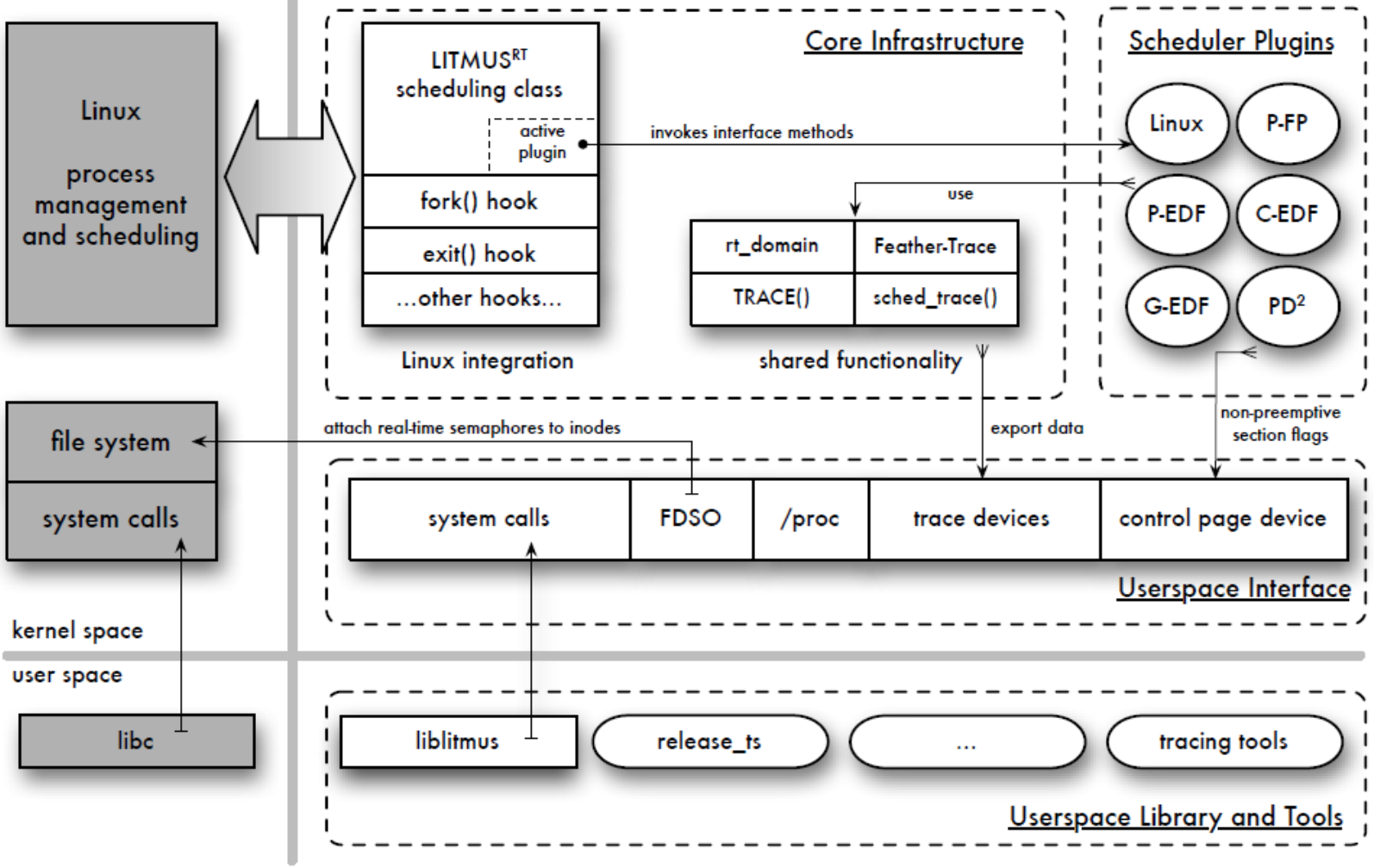
LITMUS^{RT} : c'est quoi ? (1/2)

- Patch kernel + interface en espace utilisateur + infrastructure de traces
  - `litmus-rt` : ensemble d'ordonnanceurs sous la forme de plugins
  - `liblitmus` : API en C (modèle de tâches et appels systèmes) + outils
    - `showsched, setsched, rt_spin, release_ts`
  - `feather-trace-tools` : acquisition de traces, visualisation de diagrammes de Gantt et calcul de statistiques (temps réponse, retard, taux utilisation, etc.)
- Ordonnanceurs : P-EDF, G-EDF, C-EDF, P-FP, P-RES et PD^2 (PFAIR)
- Algorithmes de synchronisation : SRP, PCP, MPCP, DPCP, etc.

LITMUS^{RT} : c'est quoi ? (2/2)

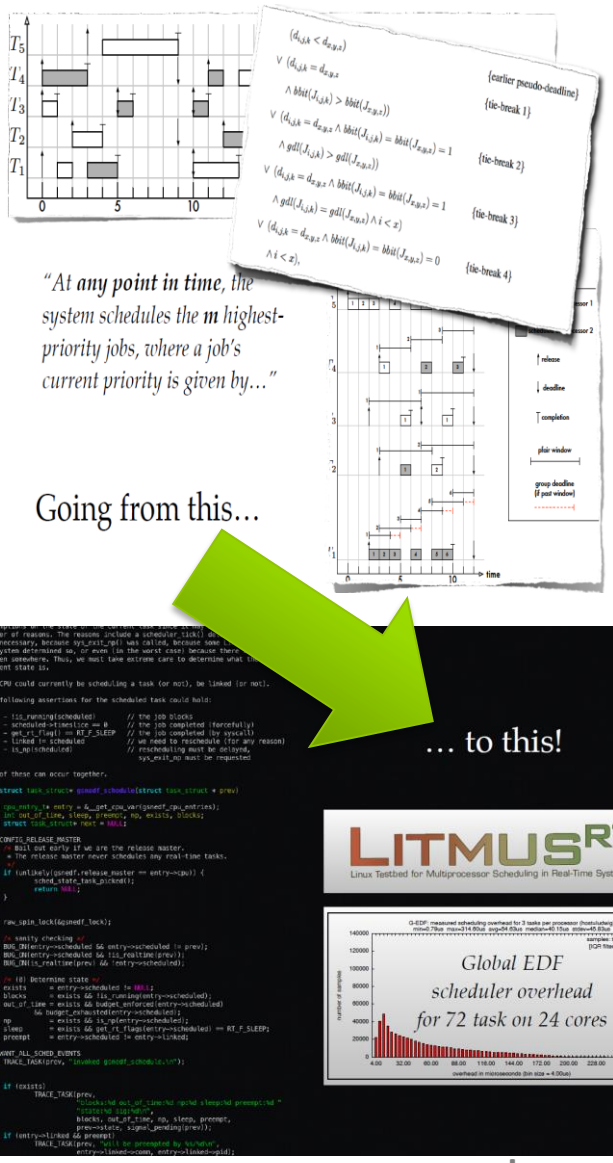
LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems



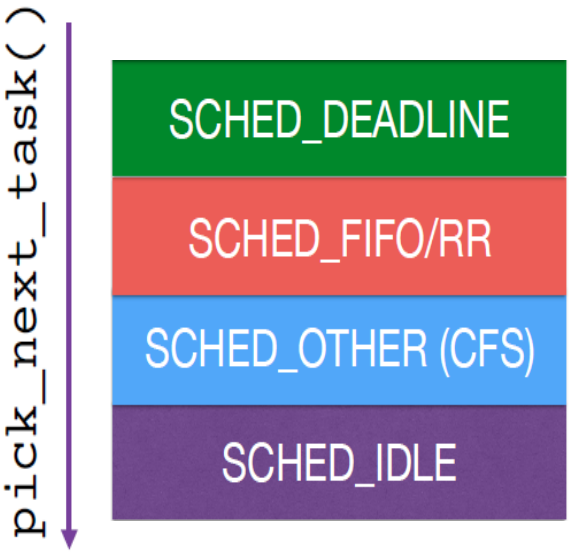
LITMUS^{RT} : ca sert à quoi ?

- Recherche expérimentale sur l'ordonnancement temps réel multiprocesseur dans des conditions réalistes
  - Pour évaluer les performances d'implémentations d'un algorithme
    - En pratique certaines hypothèses sont délicates à implémenter : base de temps commune, partage de structure de données, concurrence de décisions, etc.
  - Pour comparer des algorithmes dans le même environnement
    - Supporte l'exécution d'applicatifs
- Ne cherche pas …
  - … à maintenir la compatibilité POSIX
  - … la stabilité des APIs fournies
  - … et à être intégré au sein du kernel Linux standard
  - … être un RTOS pour tâche temps réel strict



Ordonnancement Linux (1/2)

- Hiérarchie extensible de classes d'ordonnanceurs, classée par priorité
  - Temps réel :
    - SCHED_DEADLINE : EDF global
    - SCHED_FIFO / RR : P-FP ou G-FP (affinitées)
  - Généraliste & tâche de fond : CFS et IDLE
  - Ensemble de fonctions à implementer

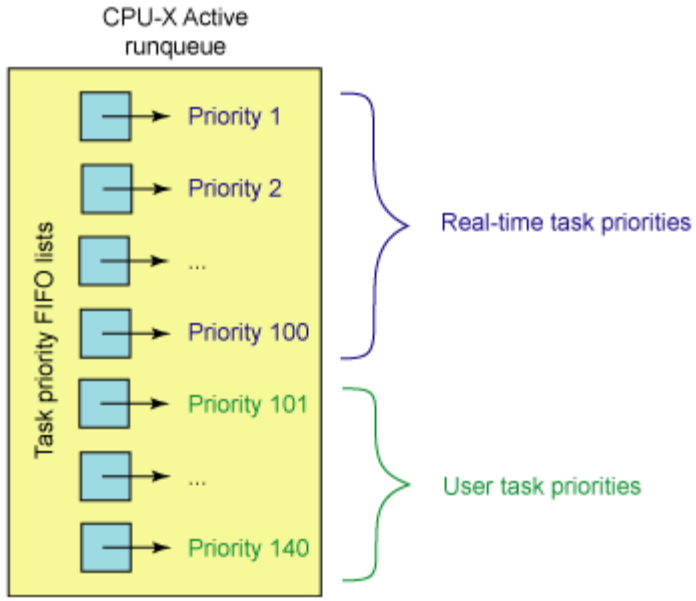


- Rôle d'un ordonnanceur
  - Déterminer le placement des processus au sein de chaque liste associée à un niveau de priorité
  - SCHED_FIFO :
    - Une fois sélectionné le processus est remis à la fin de la liste
    - Peut-être preempté par un processus de priorité supérieure
  - SCHED_RR : SCHED_FIFO + quantum de temps maximum (paramétrable)

Ordonnancement Linux (2/2)

- Changement d'implémentation : Linux 2.6
  - Passage d'une seule « runqueue » globale à m « runqueue » (1 par processeur)
    - 1 processus affecté à une et une seule « runqueue » protégée par un spinlock
  - Complexité : $O(n) \rightarrow O(1)$ et performances accrues
  - Stratégies d'équilibrage de charge
    - Modèle push / pull

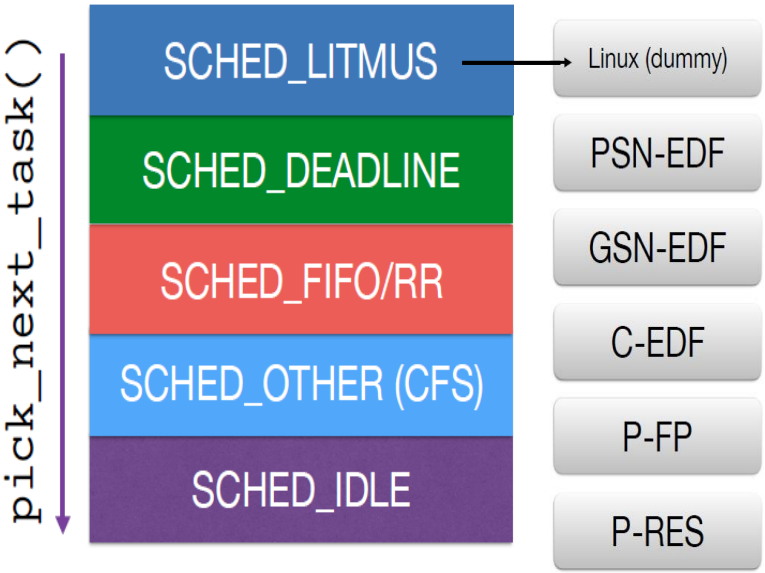
- Fonctionnement (pour certaines classes d'ordonnanceur)
  - Tâches temps réel : 100 niveaux de priorité
  - Instructions pour identifier le niveau de priorité le plus élevé actif
  - Tête de liste sélectionnée de ce niveau de priorité



LITMUS^{RT} : fonctionnement

- Ajout d'une classe d'ordonnancement
  - Tâches LITMUS^{RT} : priorité maximale
  - Changement dynamique d'ordonnanceur
    - Commande `setsched`
- API d'un plugin d'ordonanncement

Method	Purpose
schedule()	Select next process to be scheduled.
finish_switch()	Bookkeeping after context switch.
release_at()	Prepare future timed job release.
task_block()	Remove process from ready queue.
task_wake_up()	Add process to ready queue.
complete_job()	Prepare next periodic job release.
admit_task()	Check if process is correctly configured.
task_new()	Allocate and initialize per-process scheduler state.
task_exit()	Free per-process scheduler state.
activate_plugin()	Allocate and initialize plugin state.
deactivate_plugin()	Free plugin state.
allocate_lock()	Allocate real-time semaphore for process.



Attention !
« runqueue » locale Linux acquise

LITMUS^{RT} : « real-time domain »

- Ensemble de tâches ordonnancées sur un ensemble de processeurs → 1 cluster
  - Ready-queue : liste des tâches prêtes
    - Protégée par un spinlock
  - Release-queue : liste des tâches à activer
    - Protégée par un spinlock
  - Code pour traiter l'activation de travaux

- Operations
  - `add_release` / `add_ready`
  - `peek_ready` : pour vérifier si une préemption est nécessaire
  - `take_ready` : pour réaliser une préemption

- P-EDF : m clusters → m « real-time domains »

IMPLEMENTATION DE P-EDF AU SEIN DE LITMUS^{RT}

L'allocation des tâches aux processeurs a été faite au préalable

Etapes :

1) Ajout & squelette d'un plugin
2) Structures de données & initialisation
3) Fonctions auxiliaires pour l'implémentation d'EDF
4) Logique d'ordonnancement
5) Support de la préemption et utilisation
6) Transitions d'états : admission, démarrage et arrêt

LITMUS^{RT} : ajout d'un plugin

- Point de départ
  - http://www.litmus-rt.org/tutorial/litmus–2016.1.qcow.tar.gz
  - Compte utilisateur
    - Login : litmus
    - Mot de passe : litmus
  - Compte root
    - Mot de passe : litmus

- Ajout dans les sources du kernel
  - Localisée dans `/opt`
  - `cd litmus-rt/litmus`
  - `touch sched_pedf_demo.c`
  - Ajout de `sched_pedf_demo.o` dans la liste `obj-y` du `Makefile`

LITMUS^{RT} : squelette d'un plugin

```
#include <linux/module.h>
#include <litmus/preempt.h>
#include <litmus/sched_plugin.h>

static struct task_struct* demo_schedule(struct task_struct * prev)
{
        /* This mandatory. It triggers a transition in the LITMUS^RT remote
         * preemption state machine. Call this AFTER the plugin has made a local
         * scheduling decision. */
        sched_state_task_picked();
        /* We don't schedule anything for now. NULL means "schedule background work".*/
        return NULL;
}
static long demo_admit_task(struct task_struct *tsk)
{
        /* Reject every task. */
        return -EINVAL;
}
static struct sched_plugin demo_plugin = {
        .plugin_name            = "P-EDF-DEMO",
        .schedule               = demo_schedule,
        .admit_task             = demo_admit_task,
};

static int __init init_demo(void)
{
        return register_sched_plugin(&demo_plugin);
}
module_init(init_demo);
```

Liste de call-back à surcharger pour spécialiser le comportement du plugin

Liste complète : `include/litmus/sched_plugin.h`, Par défaut, les fonctions ne font rien

Enregistrement plugin Module Linux

P-EDF : structures

1 rt-domain

Tâche actuellement exécutée

```
#include <linux/percpu.h>
#include <linux/sched.h>
#include <litmus/rt_domain.h>

struct demo_cpu_state {
        rt_domain_t local_queues;
        int cpu;
        struct task_struct* scheduled;
};

static DEFINE_PER_CPU(struct demo_cpu_state, demo_cpu_state);
```

1 structure par processeur

```
#define cpu_state_for(cpu_id)   (&per_cpu(demo_cpu_state, cpu_id))
#define local_cpu_state()       (this_cpu_ptr(&demo_cpu_state))
```

Macros d'accès

P-EDF : initialisation

```
#include <litmus/litmus.h>
#include <litmus/edf_common.h>
#include <litmus/debug_trace.h>

static long demo_activate_plugin(void)
{
        int cpu;
        struct demo_cpu_state *state;

        // Use our macros to access the per-CPU state for all CPUs
        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,
                                check_for_preemption, NULL);
        }
        return 0;
}
```

Nouvelle fonction à surcharger dans la structure `sched_plugin`

Fonction à appeler lors du passage de travaux dans la file « ready »

P-EDF : fonctions auxiliaires EDF

```
#include <litmus/jobs.h>
#include <litmus/budget.h>

/* This helper is called when task `prev` exhausted its budget or when
 * it signaled a job completion. */
static void demo_job_completion(struct task_struct *prev, int budget_exhausted)
{
         /* set flags */
         tsk_rt(t)->completed = 0;
        /* Call common helper code to compute the next release time, deadline, etc. */
        prepare_for_next_period(prev);
}


/* Add the task 'tsk' in the appropriate queue. */
static void demo_requeue(struct task_struct *tsk, struct demo_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);
        } else {
                /* Uses add_release() because we DON'T have the release lock. */
                add_release(&cpu_state->local_queues, tsk);
        }
}
```

P-EDF : étapes d'ordonnancement

1. Déterminer l'état de la tâche en cours d'exécution

2. Déterminer s'il est nécessaire de prendre une nouvelle décision d'ordonnancement

3. Si oui, déterminer la prochaine tâche à exécuter
   - Sinon, continuer avec la tâche existante au besoin
   - Si pas de tâche, Linux va appeler les autres ordonnanceurs

- Nouvelle fonction à surcharger dans la structure `sched_plugin`

```
static struct task_struct* demo_schedule(struct task_struct * prev) {
        1.
        2.
        3.
}
```

P-EDF : ordonnancement (1/3)

1. Obtenir l'état de la tâche en cours d'exécution

```
static struct task_struct* demo_schedule(struct task_struct * prev) {

        struct demo_cpu_state *local_state = local_cpu_state();

        /* next == NULL means "schedule background work". */
        struct task_struct *next = NULL;

        /* prev's task state */
        int exists, out_of_time, job_completed;

        raw_spin_lock(&local_state->local_queues.ready_lock);

        exists = local_state->scheduled != NULL;
        out_of_time = exists && budget_enforced(prev) && budget_exhausted(prev);
        job_completed = exists && is_completed(prev);
```

...

P-EDF : ordonnancement (2/3)

2. Déterminer s'il est nécessaire de prendre une nouvelle décision d'ordonnancement

```
static struct task_struct* demo_schedule(struct task_struct * prev)
{
                                        •••

        int preempt, resched;

        /* preempt is true if task `prev` has lower priority than something on
         * the ready queue. */
        preempt = edf_preemption_needed(&local_state->local_queues, prev);
        /* check all conditions that make us reschedule */
        resched = preempt;

        /* also check for (in-)voluntary job completions */
        if (out_of_time || job_completed) {
                demo_job_completion(prev, out_of_time);
                resched = 1;
        }

                                        •••
```

P-EDF : ordonnancement (3/3)

3. Si oui, déterminer la prochaine tâche à exécuter
   - Sinon, continuer avec la tâche existante au besoin
   - Si pas de tâche, Linux va appeler les autres ordonnanceurs

```
static struct task_struct* demo_schedule(struct task_struct * prev)
{
                                        •••

        if (resched) {
                /* First check if the previous task goes back onto the ready
                 * queue, which it does if it did not self_suspend.*/
                if (exists && !self_suspends) {
                        demo_requeue(prev, local_state);
                }
                next = __take_ready(&local_state->local_queues);
        } else {
                next = local_state->scheduled; /* No preemption is required. */
        }
        local_state->scheduled = next;
        /* This mandatory. It triggers a transition in the LITMUS^RT remote
         * preemption state machine. */
        sched_state_task_picked();
        raw_spin_unlock(&local_state->local_queues.ready_lock);
        return next;
}
```

P-EDF : support de la préemption

Callback lors du transfert d'au moins un travail de la file « release » à « ready »

```
static int check_for_preemption (rt_domain_t *local_queues)
{
        struct demo_cpu_state *state = container_of(local_queues,
                                                    struct demo_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(state->scheduled, state->cpu);
                return 1;
        }

        return 0;
}
```

Appel à `litmus_reschedule(state->cpu)`

Appel à Linux pour réaliser le changement de contexte

P-EDF : support de la préemption

```
/* need_to_preempt - check whether the task t needs to be preempted */
int edf_preemption_needed(rt_domain_t* rt, struct task_struct *t)
{

        /* no need to preempt if there is nothing pending */
        if (!__jobs_pending(rt))
                return 0;

        /* we need to reschedule if t doesn't exist */
        If (!t)
                return 1;

        /* edf_higher_prio -  returns true if first has a higher EDF priority
         *                    than second. Deadline ties are broken by PID. */
        return edf_higher_prio(__next_ready(rt), t);
}
```

P-EDF : admission des tâches

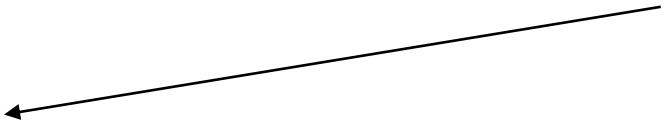
```
static long demo_admit_task(struct task_struct *tsk)
{
        if (task_cpu(tsk) == get_partition(tsk)) {
                TRACE_TASK(tsk, "accepted by demo plugin.\n");
                return 0;
        }
        return -EINVAL;
}
```

P-EDF : initialisation des tâches

```
static void demo_task_new(struct task_struct *tsk, int on_runqueue, int is_running)
{
        unsigned long flags;
        struct demo_cpu_state *state = cpu_state_for(get_partition(tsk));

        /* Acquire the lock protecting the state and disable interrupts. */
        raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);

        /* Release the first job now. */
        release_at(tsk, litmus_clock());

        if (is_running) {
                state->scheduled = tsk;
        } else if (on_runqueue) {
                demo_requeue(tsk, state);
        }
        if (edf_preemption_needed(local_queues, state->scheduled)) {
                preempt(state->scheduled, state->cpu);
        }

        raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
}
```

P-EDF : reprise de l'exécution

Lorsque la tâche `tsk` est ajoutée dans la « ready » queue



```
static void demo_task_resume(struct task_struct  *tsk)
{
        unsigned long flags;
        struct demo_cpu_state *state = cpu_state_for(get_partition(tsk));

        raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);


        if (edf_preemption_needed(&state->local_queues, state->scheduled)) {
                preempt(state->scheduled, state->cpu);
        }

        raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
}
```

P-EDF : arrêt des tâches

```
static void demo_task_exit(struct task_struct *tsk)
{
        unsigned long flags;
        struct demo_cpu_state *state = cpu_state_for(get_partition(tsk));
        raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);

        /* For simplicity, we assume here that the task is
         * no longer queued anywhere else.
         */
        if (state->scheduled == tsk) {
                state->scheduled = NULL;
        }

        raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
}
```

IMPLEMENTATION DE G-EDF AU SEIN DE LITMUS^{RT}

Différences par rapport à P-EDF

Etapes :

1) Structures de données & initialisation
2) Code de base pour P-EDF
3) Fonctions auxiliaires pour l'implémentation d'EDF
4) Logique d'ordonnancement

G-EDF : structures

```
typedef struct  {
          int cpu;
          struct task_struct* linked;
} cpu_entry_t;
DEFINE_PER_CPU(cpu_entry_t, gsnedf_cpu_entries);

cpu_entry_t* gsnedf_cpus[NR_CPUS];
```

Mapping processeur

```
static rt_domain_t gsnedf;
#define gsnedf_lock (gsnedf.ready_lock)
```

Un seul « rt-domain »

G-EDF : fonctions auxiliaires EDF

```
/* This helper is called when task `current` exhausted its budget or when
 * it signaled a job completion. */
static noinline void curr_job_completion(int forced)
{
        struct task_struct *t = current;

        /* set flags */
        tsk_rt(t)->completed = 0;
        /* prepare for next period */
        prepare_for_next_period(t);
        /* unlink */
        unlink(t);
}


/* Add the task 'task' in the appropriate queue. */
static noinline void requeue(struct task_struct* task)
{
        if (is_released(task, litmus_clock()))
                __add_ready(&gedf, task);
        else {
                /* it has got to wait */
                add_release(&gedf, task);
        }
}
```

G-EDF : étapes d'ordonnancement

1. Déterminer l'état de la tâche en cours d'exécution
2. Déterminer s'il est nécessaire de prendre une nouvelle décision d'ordonnancement
3. Si oui, déterminer la prochaine tâche à exécuter
   - Sinon, continuer avec la tâche existante au besoin
   - Si pas de tâche, Linux va appeler les autres ordonnanceurs

G-EDF : ordonnancement (1/3)

1. Déterminer l'état de la tâche en cours d'exécution

```
static struct task_struct* gedf_schedule(struct task_struct * prev)
{

        cpu_entry_t* entry = this_cpu_ptr(&gedf_cpu_entries);
        int out_of_time, job_completed, preempt, exists;
        struct task_struct* next = NULL;

        raw_spin_lock(&gsnedf_lock);

        exists = entry->linked != NULL;

        out_of_time = exists && budget_enforced(entry->linked) &&
                                budget_exhausted(entry->linked);

        job_completed= exists && is_completed(entry->linked);
```

...

G-EDF : ordonnancement (2/3)

2. Déterminer s'il est nécessaire de prendre une nouvelle décision d'ordonnancement

```
...

if (out_of_time || sleep) curr_job_completion(!sleep);

/* Link pending task if we became unlinked. */
if (!entry->linked) link_task_to_cpu(__take_ready(&gedf), entry);

/* The final scheduling decision. Do we need to switch for some reason?
preempt = prev != entry->linked;

...
```

G-EDF : ordonnancement (3/3)

3. Si oui, déterminer la prochaine tâche à exécuter
   - Sinon, continuer avec la tâche existante au besoin
   - Si pas de tâche, Linux va appeler les autres ordonnanceurs

...

```
        if (preempt) {
                /* Schedule a linked job? */
                if (entry->linked) {
                        entry->linked->rt_param.scheduled_on = entry->cpu;
                        next = entry->linked;
                }
        } else
                /* Only override Linux scheduler if we have a real-time task
                * scheduled that needs to continue. */
                if (exists)
                        next = prev;

        sched_state_task_picked();
        raw_spin_unlock(&gedf_lock);
        return next;
}
```

G-EDF : support de la préemption

```
static void check_for_preemptions(void)
{
        struct task_struct *task;
        cpu_entry_t *last;

        for (last = lowest_prio_cpu();
             edf_preemption_needed(&gedf, last->linked);
             last = lowest_prio_cpu()) {

                /* preemption necessary */
                task = __take_ready(&gedf);
                requeue(last->linked);
                link_task_to_cpu(task, last);

                /* force a CPU to reschedule*/
                preempt(last);
        }
}
```

Identifier le travail le moins prioritaire

Note : les temps d'inactivité ont une priorité la plus faible

G-EDF : admission des tâches

```
static long gedf_admit_task(struct task_struct* tsk)
{

        return 0;

}
```

G-EDF : initialisation des tâches

```
static void gedf_task_new(struct task_struct * t, int on_runqueue, int is_running)
{
        unsigned long flags;
        cpu_entry_t* entry;

        raw_spin_lock_irqsave(&gedf_lock, flags);

        /* setup job params */
        release_at(t, litmus_clock());

        if (is_running) {
               entry = &per_cpu(gedf_cpu_entries, task_cpu(t));
               entry->linked = t;
        }

        if (on_runqueue || is_running) { requeue(t); check_for_preemptions(); }

        raw_spin_unlock_irqrestore(&gedf_lock, flags);
}
```

G-EDF : arrivées de travaux

```
static void gsdf_release_jobs(rt_domain_t* rt, struct bheap* tasks)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&gedf_lock, flags);

        __merge_ready(rt, tasks);
        check_for_preemptions();

        raw_spin_unlock_irqrestore(&gedf_lock, flags);
}
```

G-EDF : arrêt de tâches

```
static void gedf_task_exit(struct task_struct * t)
{
        unsigned long flags;


        raw_spin_lock_irqsave(&gedf_lock, flags);

        unlink(t);

        if (tsk_rt(t)->scheduled_on != NO_CPU) {
                gedf_cpus[tsk_rt(t)->scheduled_on]->scheduled = NULL;
                tsk_rt(t)->scheduled_on = NO_CPU;
        }

        raw_spin_unlock_irqrestore(&gedf_lock, flags);
}
```

ILLUSTRATION DE PROPRIETES AVEC LITMUS^{RT}

Points :

1) Utilisation de LITMUS^{RT}
2) Effet Dhall
3) Incomparabilité
4) Robustesse temporelle
5) Instant critique
6) Priorité fixe/dynamique pour les travaux

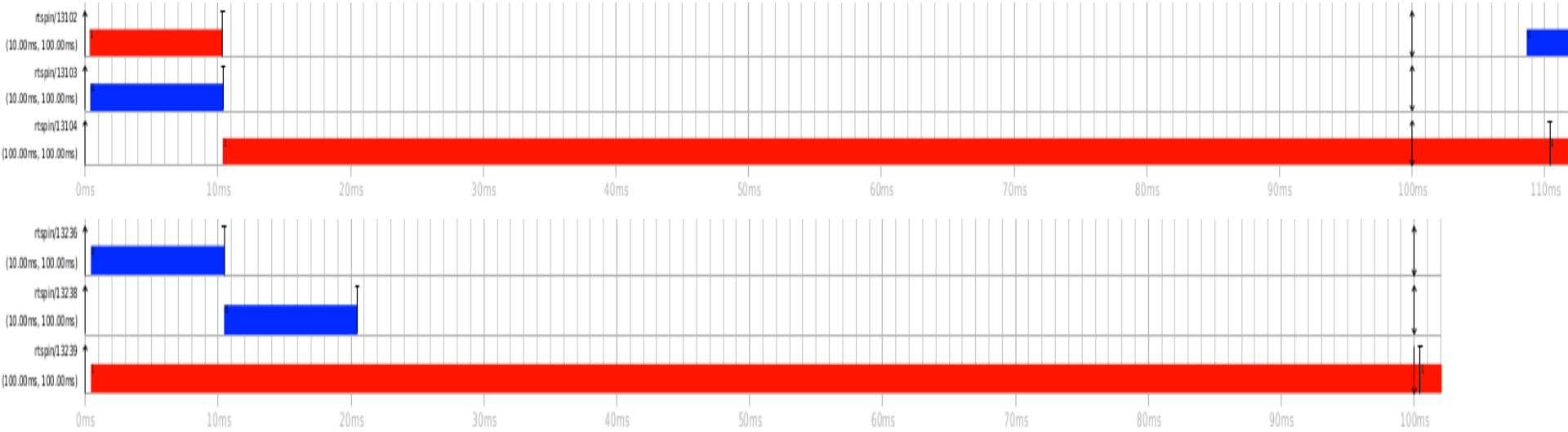
LITMUS^{RT} : outils disponibles

- Ordonnanceurs
  - Lister les ordonnanceurs disponibles : `cat /proc/litmus/plugins/loaded`
  - Afficher l'ordonnanceur utilisé : `showsched`
  - Changer l'ordonnanceur choisi : `setsched PLUGIN_NAME`
- Exécution …
  - … d'un programme : `rt_launch WCET PERIOD -- PROGRAM ARGS`
  - … d'une tâche qui boucle : `rtspin OPTIONS WCET PERIOD DURATION`
    - Spécifier une échéance : `-d DEADLINE`
    - Spécifier un offset : `-o OFFSET`
    - Spécifier une partition : `-p NCPU`
    - Spécifier une priorité : `-q PRIORITY` (highest : 1, lowest : 511)
    - Tâches synchrones : `-w` puis lancement via la commande `release_ts`
- Visualisation de l'ordonnancement
  - Lancement des traces : `st-trace-schedule my-trace`
  - Production diagramme de Gantt : `st-draw *.bin`
  - Statistiques sur les travaux exécutés : `st-job-stats *my-trace*.bin | head`

Effet Dhall

- Jeu de $n = m + 1$ tâches
  - Les m premières tâches : $(C_i = 2\varepsilon, T_i = 1) \Rightarrow u_i = 2\varepsilon$
  - La $m + 1$ tâche : $(C_{m+1} = 1, T_{m+1} = 1 + \varepsilon) \Rightarrow u_i = 1/(1 + \varepsilon)$

- Ordonnanceur utilisé : global rate monotonic (G-RM)
  - Les m premières tâches sont prioritaires et donc la $m + 1$ tâche va rater son échéance
  - Si $\varepsilon \rightarrow 0, U = 1$

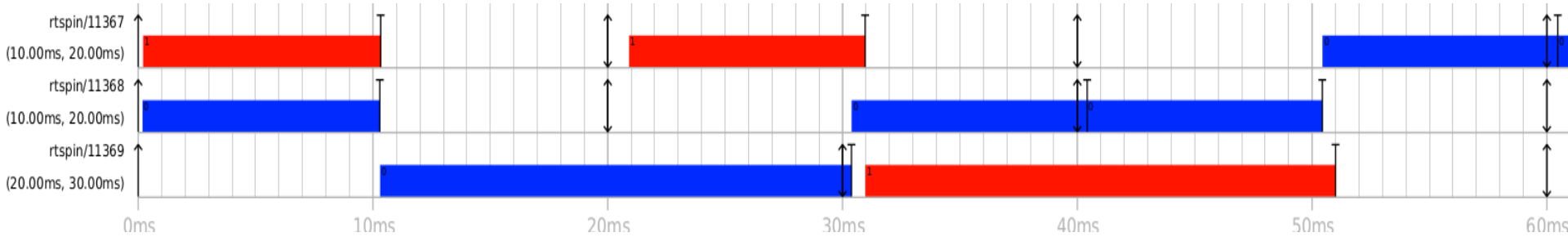


Incomparabilité (priorité fixe)

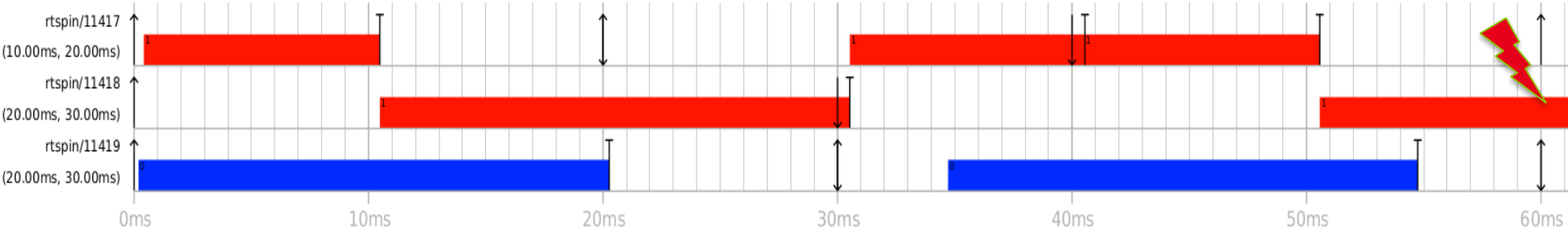
	C_i	$T_i = D_i$
τ_1	10	20
τ_2	20	30
τ_3	20	30

$m = 2, n = 3$

Ordonnançable avec une **approche globale**
(priorité fixe tâche ou travail)



Non-ordonnçable avec une **approche partitionnée**
(priorité fixe tâche ou travail)



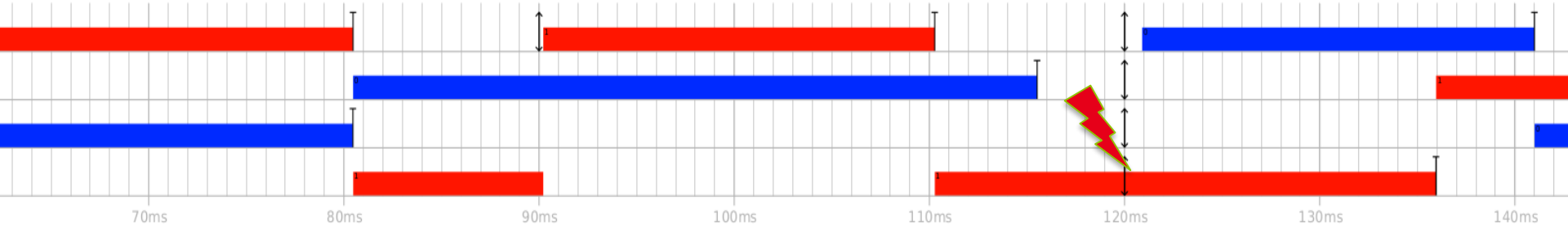
Incomparabilité (priorité fixe)

	C_i	$T_i = D_i$
τ_1	20	30
τ_2	35	60
τ_3	20	60
τ_4	50	120

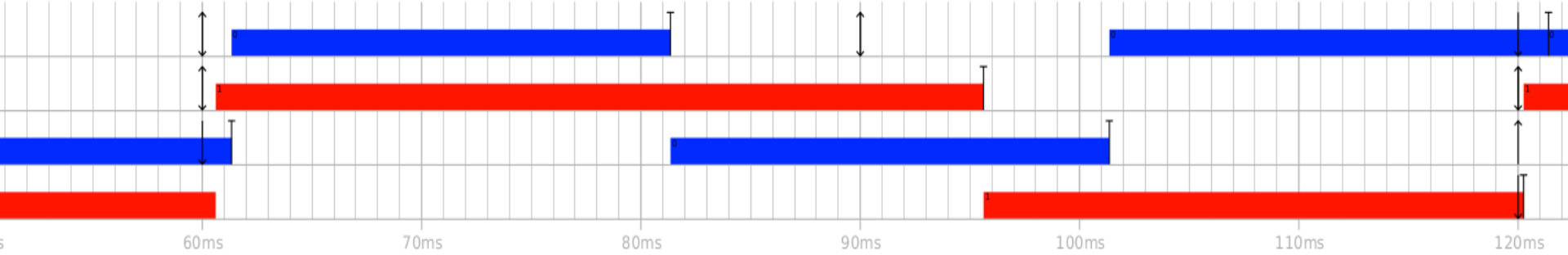
$m = 2, n = 4$

L'inverse !

Non-ordonnançable avec une **approche globale**
(priorité fixe tâche ou travail)



Ordonnçable avec une **approche partitionnée**
(priorité fixe tâche ou travail)

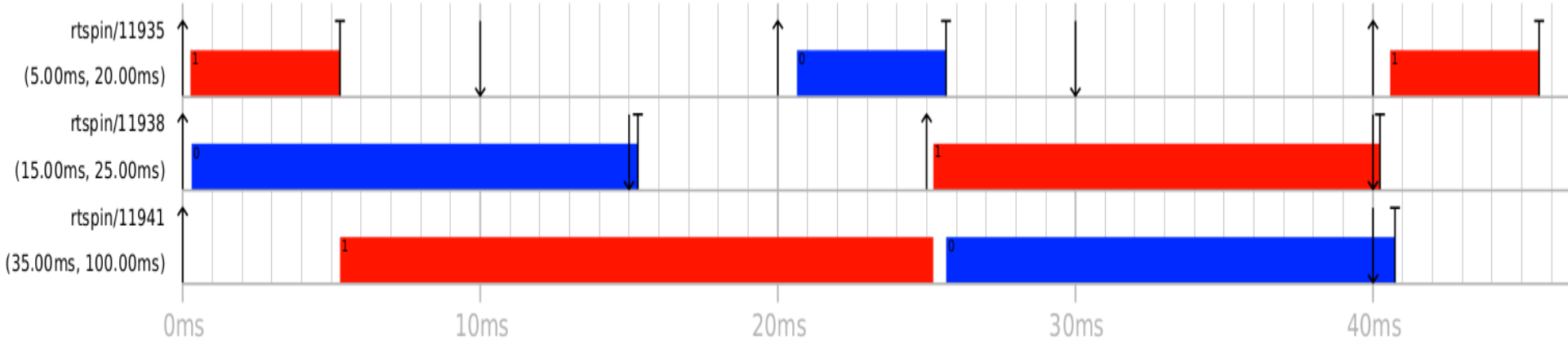


Robustesse temporelle

$m = 2, n = 3$

	C_i	D_i	T_i
τ_1	5	10	20
τ_2	15	15	25
τ_3	35	40	100

Système **ordonnançable**
(ordonnancement global à priorité fixe
au niveau des tâches)

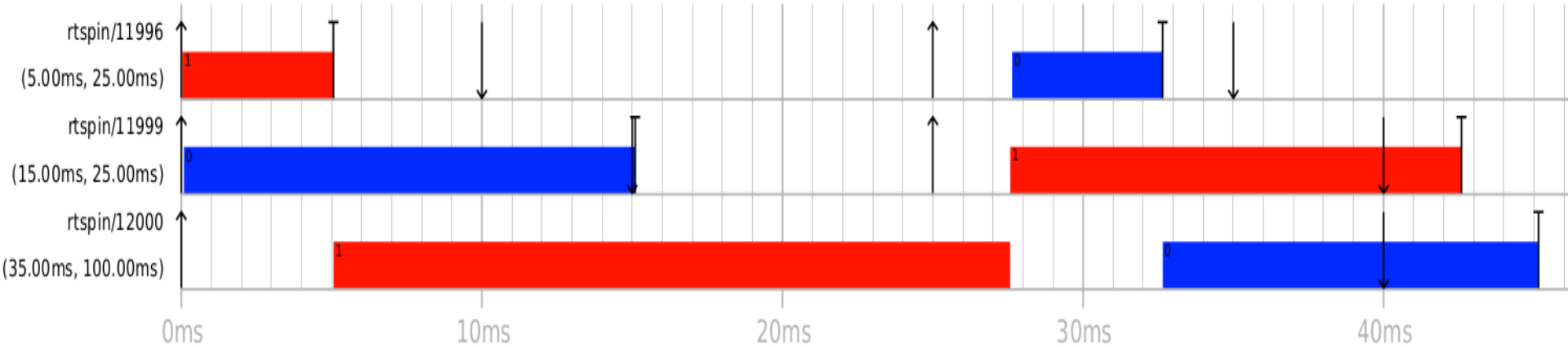


Robustesse temporelle

$m = 2, n = 3$

	C_i	D_i	T_i
τ_1	5	10	**20 → 25**
τ_2	15	15	25
τ_3	35	40	100

Système **non-ordonnançable**
(ordonnancement global à priorité fixe au niveau des tâches)



→ Une **augmentation** de la **période** d'une tâche peut rendre le système **non-ordonnançable**

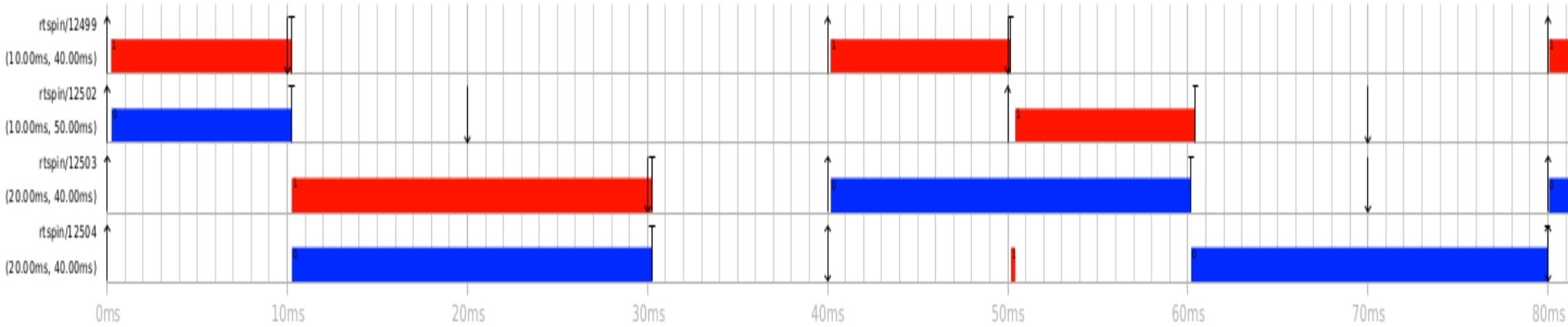
Instant critique

$m = 2, n = 4$

	C_i	D_i	T_i
τ_1	2	2	8
τ_2	2	4	10
τ_3	4	6	8
τ_4	4	7	8

Instant critique d'un ordonnancement : instant où toutes les tâches seraient prêtes à être exécutées de manière simultanée

Hypothèse utilisée pour le calcul du temps de réponse en mono-processeur pour des tâches périodiques (pire cas)



$R_4^1 = 30$

$R_4^2 = 40$

→ En multi-processeur ce n'est plus le cas

Priorité fixe/dynamique pour les travaux

- Distinction …
  - … peut faite en monoprocesseur compte tenu de l'optimalité d'EDF
  - … faite en multiprocesseur : découle de l'analyse de l'effet de Dhall
- Laxité $L_i(t)$: caractéristique dynamique d'une tâche
  - Délai maximum que peut subir une tâche sans engendrer un dépassement de son échéance
  - $L_i(t) = D_i(t) - C_i(t) = d_i - t - C_i(t)$ avec $C_i(t)$ durée d'exécution restante
- Algorithmes « xZL » : priorité max à un travail dès que $L_i(t) = 0$

$m = 2, n = 3$

	C_i	D_i	T_i
τ_1	2	3	3
τ_2	2	3	3
τ_3	2	3	3

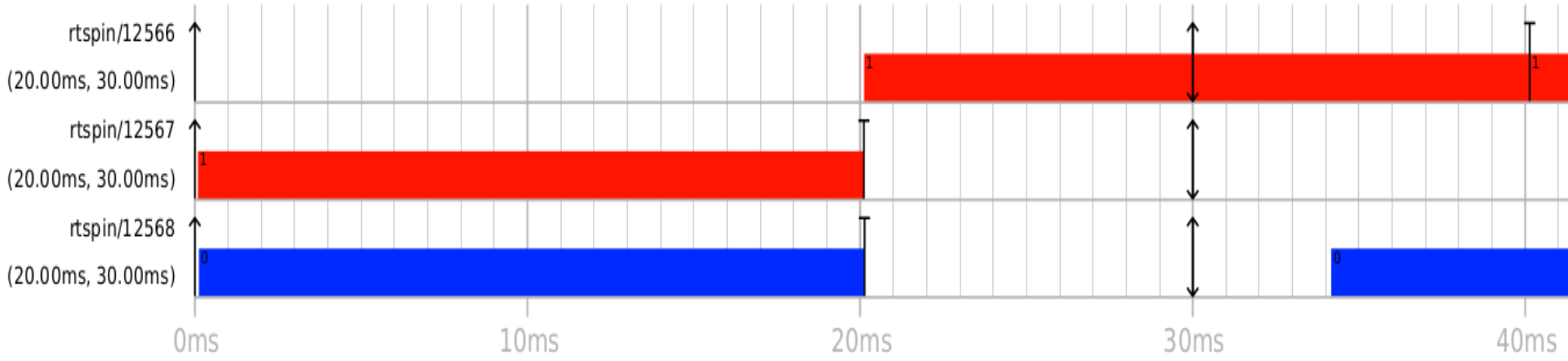
Jeu de tâche …

… **non-ordonnançable** G-EDF

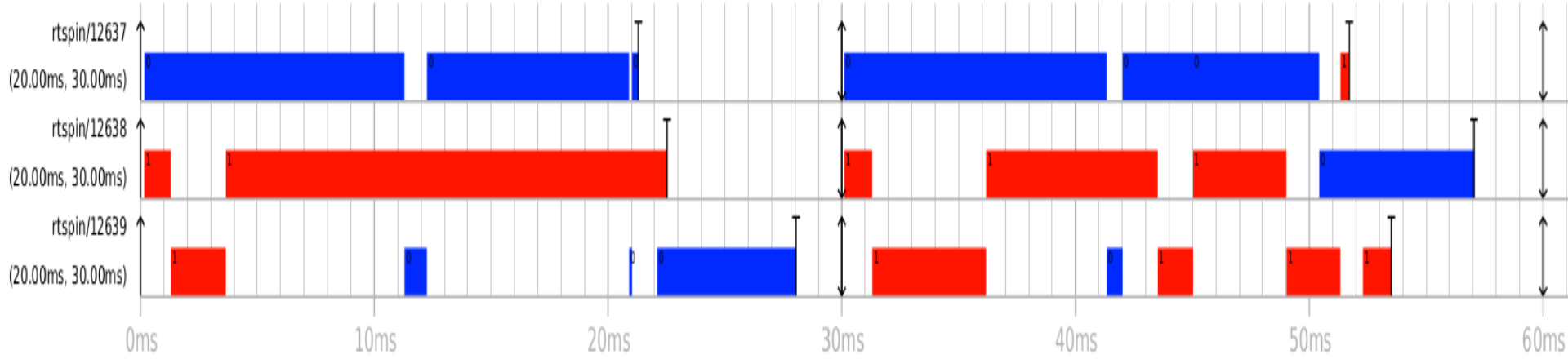
… **ordonnançable** avec EDZL

Visualisation ordonnancement PFAIR

Non-ordonnançable G-EDF



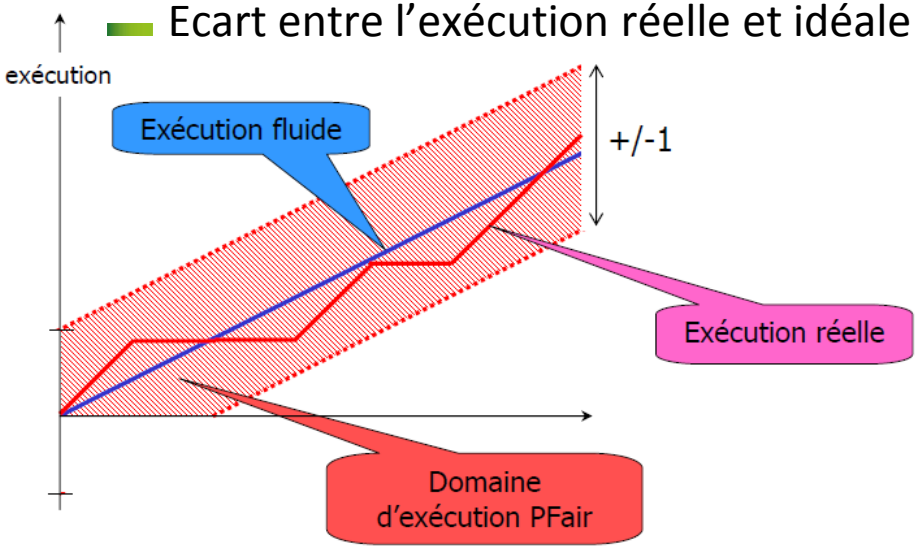
Ordonnançable avec PFAIR



APERCU P-FAIR & LITMUS^{RT}

Algorithme P-fair : idée & construction

- « Proportionate Fair » : stratégie multiprocesseur optimale
  (périodiques échéances sur requêtes, synchrones)
  - Allocation des processeurs aux tâches avec un taux d'exécution constant proportionnel à leurs facteurs d'utilisation u_i
  - Temps discret : slots successifs $[t; t+1[$
  - Ecart entre l'exécution réelle et idéale : $lag\ (\tau_i, t) = u_i \times t - \sum_{k=0}^{k-1} S(\tau_i, k)$



$S(\tau_i, k) = 1$ si τ_i exécutée dans le slot v

$S(\tau_i, k) = 0$ sinon

PFair ssi

$\forall\ \tau_i, \forall\ t, -1 < lag\ (\tau_i, t) < 1$

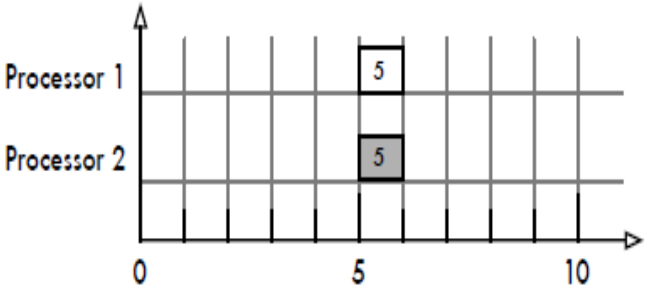
- Chaque tâche est découpée en plusieurs sous-tâches ayant chacune une **capacité d'une unité de temps** (quantum)

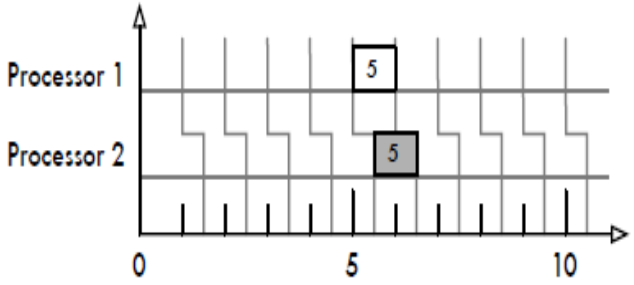
Pseudo-réveil : $r(\tau_i^j) = \left\lfloor \frac{j-1}{u_i} \right\rfloor$ Pseudo-échéance : $d(\tau_i^j) = \left\lceil \frac{j}{u_i} \right\rceil$

Plugin P-Fair & LITMUS^{RT}

- Admission des tâches : calcul des pseudo-reveils/échéances des sous-tâches

- Ordonnancement
  - Un seul « rt-domain »
  - Timer par processeur pour implémenter les quantum de temps (1ms)
    - on_quantum_boundary → `pfair_tick` → `schedule_next_quantum`
    - `schedule_next_quantum : advance_subtasks` (calcul du lag pour suivre la progression) & `schedule_subtasks`
    - Alignement/échelonnement de la progression des quantums





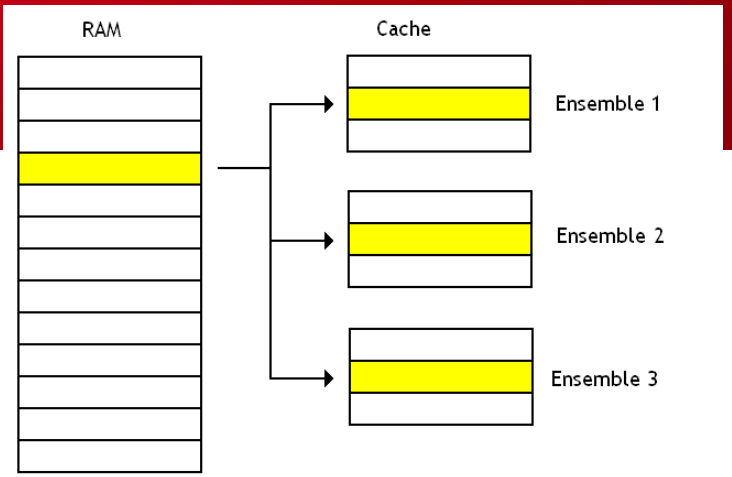
  - La fonction schedule met juste à jour l'état du travail, ne prends pas de décision d'ordonnancement

Si vous voulez aller plus loin …

- … avec LITMUS^{RT}
  - http://litmus-rt.org/tutorial
    - Tutoriels en tant qu'utilisateur et développeur d'un ordonnanceur
  - Image fournie exploitable sous Virtualbox (LITMUS 2016.1)
  - Attention, de fortes latence d'activation sont observable
    - Effets de la virtualisation

- … sur l'ordonnancement multiprocesseurs
  - Les résultats importants : exposé Joël Goossens à ETR 2013
  - L'approche globale : exposé/Article Anne-Marie Déplanche à ETR 2011
  - "A Survey of Hard Real-Time Scheduling Algorithms and Schedulability Analysis Techniques for Multiprocessor Systems", University of York, Journal ACM Computing Surveys (CSUR) Volume 43 Issue 4, October 2011

Influence du LLC



- Accès concurrents au LLC
  - Défauts de cache conflictuels
  - Gestion de la cohérence

- Problème : comment déterminer l'état des caches ?
  - Caractéristiques des caches
    - Correspondance / mapping : pleinement associatif, N-voies associatifs et direct
    - Unifié ou séparé
    - Politique de mise à jour : écrite immédiate ou différée
    - Politique de remplacement des lignes de cache : FIFO, LRU, random, etc.
  - Possibilités : ignorer (!), partitionner, verrouiller des lignes de cache, modèles de tâches spécifiques