Scheduling

AISO

Manuel Prieto-Matías
High Performance Computing Division
ArTeCS Group
Universidad Complutense de Madrid

Scheduling en Linux

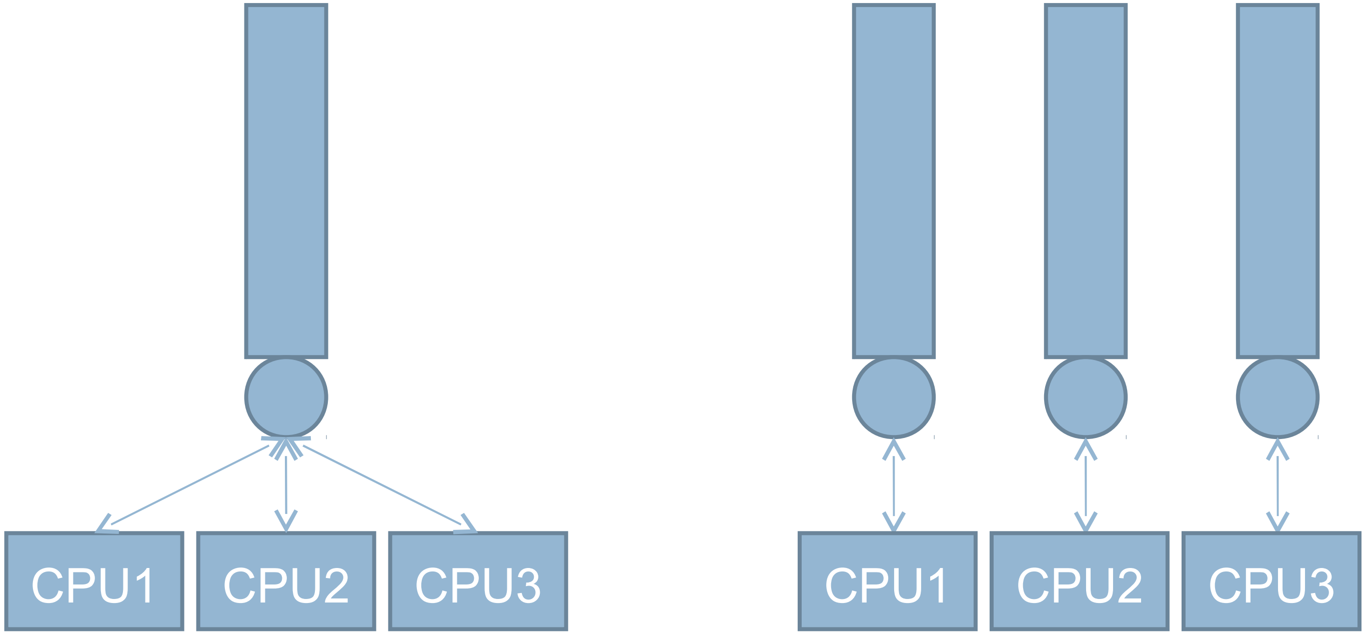
- `linux/sched.c`

Linux version	Scheduler
Linux pre 2.5	Multilevel Feedback Queue
Linux 2.5-2.6.23	O(1) scheduler
Linux post 2.6.23	Completely Fair Scheduler

Linux O(1) Scheduler

- Planificación basadas en épocas + prioridades
  - Respecto a 2.4 reducción de la complejidad: de O(n) a O(1)
- Soporte para sistemas SMP
  - Runqueues individuales (locks individuales)
  - Afinidad de procesos a CPUs (Localidad Cache)
- Expropiación / Preemptive:
  - Un proceso mayor prioridad puede expropiar a un proceso en ejecución con menor prioridad

Linux O(1) Scheduler



Linux O(1) Scheduler – runqueue

- **struct runqueue {**

```
spinlock_t lock;  spin lock which protects this runqueue
unsigned long nr_running;  number of runnable tasks
unsigned long nr_switches; number of contextswitches
unsigned long expired_timestamp; time of last array swap
unsigned long nr_uninterruptible; number of tasks in uinterruptible sleep
struct task_struct *curr; this processor's currently running task
struct task_struct *idle; this processor's idle task
struct mm_struct *prev_mm; mm_struct of last running task
struct prio_array *active; pointer to the active priority array
struct prio_array *expired; pointer to the expired priority array
struct prio_array arrays[2]; the actual priority arrays
int prev_cpu_load[NR_CPUS]; load on each processor
struct task_struct *migration_thread; the migration thread on this processor
struct list_head migration_queue; the migration queue for this processor
atomic_t nr_iowait; number of tasks waiting on I/O
}
```

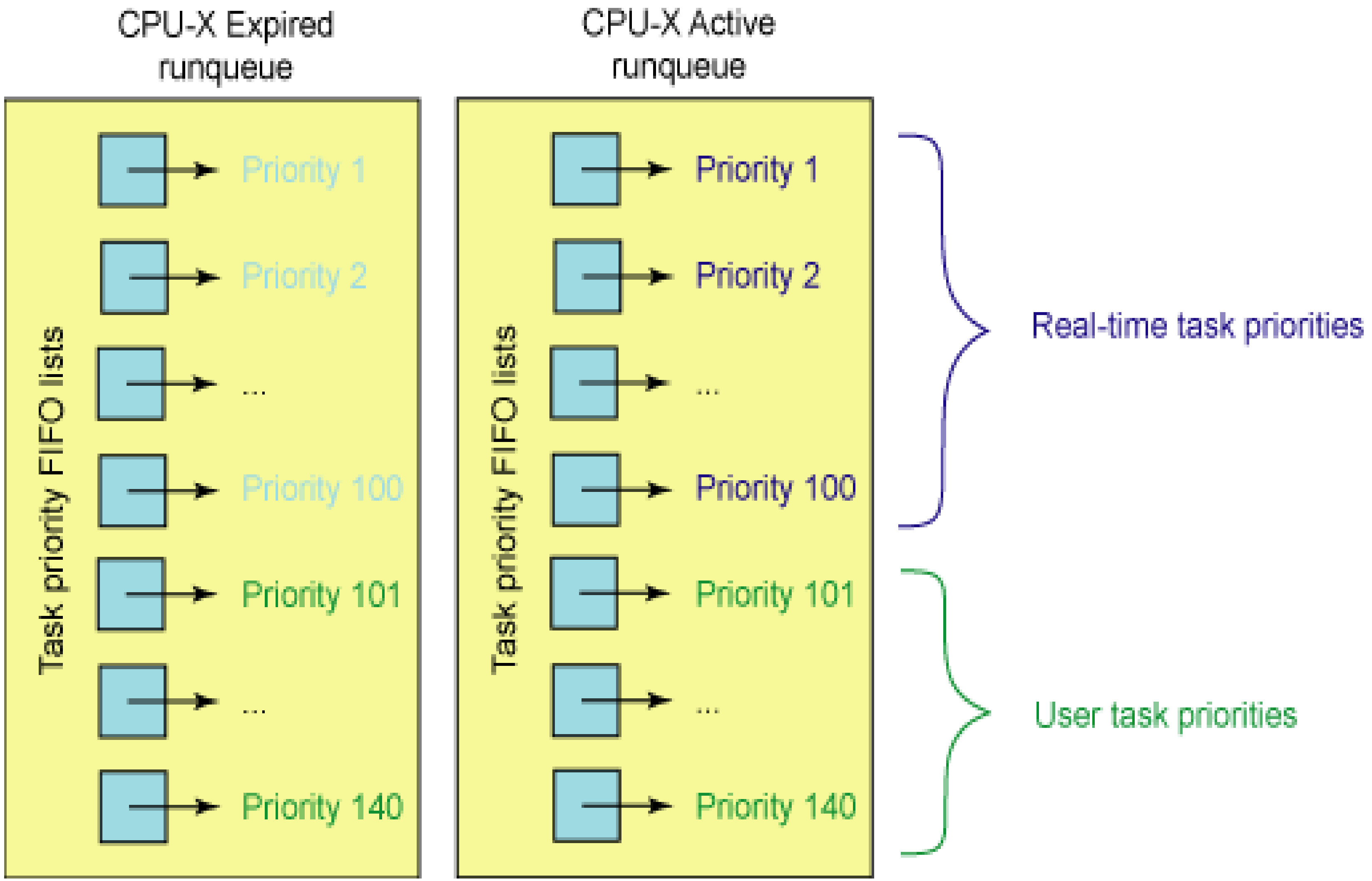
Manuel Prieto-Matías
Universidad Complutense de Madrid

Linux O(1) Scheduler – array de prioridad

- **struct prio_array {**

```
int nr_active;  number of task in the queue
unsigned long bitmap[BITMAP_SIZE];  priority bitmap
Struct list_head queue[MAX_PRIO]; priority queues
}
```

Linux O(1) Scheduler – experidos y activos –

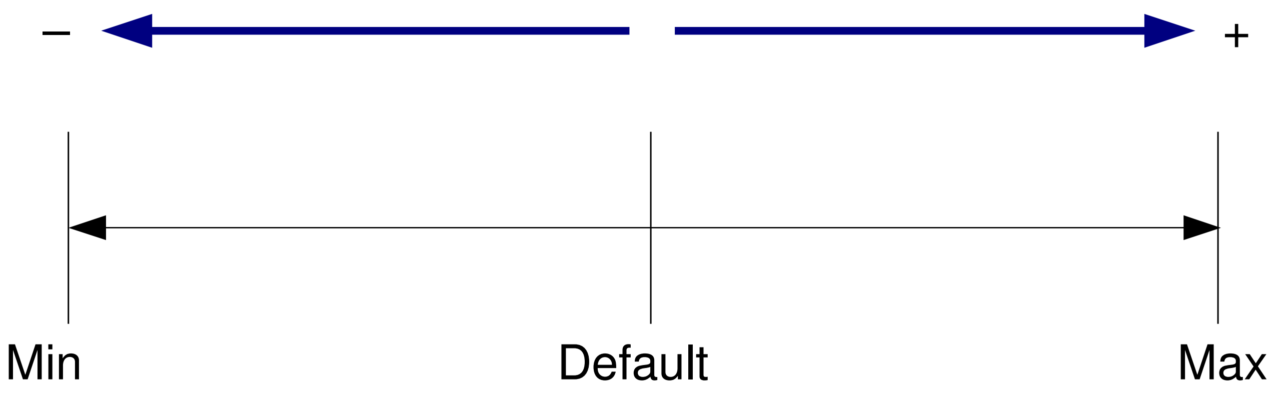


Linux O(1) Scheduler – Prioridades –

- 140 Niveles de Prioridad
  - 1-100 : Prio RT (MAX_RT_PRIO = 100)
  - 101-140 : Prio Usuario (MAX_PRIO = 140)
- Tres tipos diferentes de Políticas de Scheduling
  - Una política para los procesos usuario
    - Objetivos: Fairness, Buena Respuesta Procesos Interactivos
  - Dos políticas para los procesos de TR (RR o FIFO)
    - De acuerdo al estándar POSIX RT
- Los Procesos "normales" tienen asignado
  - Nice (lo hereda del padre, aunque puede cambiarse)
  - PRIO = MAX_RT_PRIO + NICE + 20
  - Timeslice

Linux O(1) Scheduler – Timeslice –

- Timeslice
  - Prioridad Estática
  - Inicial (la mitad del timeslice del padre)



Linux O(1) Scheduler – Timeslice –

	Static Priority	NICE	Quantum
High Priority	100	-20	800 ms
	110	-10	600 ms
	120	0	100 ms
	120	+10	50 ms
Low Priority	139	+19	5 ms

Linux O(1) Scheduler – Prio Dinámica –

- Prioridad Dinámica `effective_prio()`:
  - DP = max (100,min(SP – bonus + 5, 139))
    - SP: Prioridad Estática
    - Bonus [0,10]
      - Depende de la **interactividad del proceso**
      - Heuristica basada en comparar tiempo del proceso en estado running vs. tiempo de espera promedio (**sleep_avg [0..MAX_SLEEP_AVG]**)
      - 5 Neutral, 10 aumenta prioridad, 0 disminuye prioridad
- Los procesos muy interactivos *no expiran*

CFS – Completely Fair Scheduler –

- Linux 2.6.23 (Ingo Molnar)
  - Se definen **clases de planificación**
    - Estructura más modular
  - Clase Fair (CFS)
    - Desaparece de forma explicita los conceptos de época y timeslice
      - Aunque existe el concepto de latencia de planificación
    - No se utilizan heurísticas para calcular la interactividad
    - Los procesos se mantienen ordenados en un árbol RB según su “tiempo de espera”
      - Se elige al proceso que lleva más tiempo “esperando”

CFS sched_class

- **struct sched_class {**

```
struct sched_class *next;
void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
void (*yield_task) (struct rq *rq, struct task_struct *p);
void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);
struct task_struct * (*pick_next_task) (struct rq *rq);
void (*put_prev_task) (struct rq *rq, struct task_struct *p);
unsigned long (*load_balance) (struct rq *this_rq, int this_cpu,
               struct rq *busiest,
               unsigned long max_nr_move, unsigned long max_load_move,
               struct sched_domain *sd, enum cpu_idle_type idle,
               int *all_pinned, int *this_best_prio);
…
void (*set_curr_task) (struct rq *rq);
void (*task_tick) (struct rq *rq, struct task_struct *p);
void (*task_new) (struct rq *rq, struct task_struct *p);
};
```

Manuel Prieto-Matías
Universidad Complutense de Madrid

CFS `sched_class` – Jerarquía –

- `#define sched_class_highest(&rt_sched_class)`
- ```
  #define for_each_class(class) \
  for (class=sched_class_highest;class;class=class->next)
  ```
- ```
  static const struct sched_class rt_sched_class = {
  .next                = &fair_sched_class,
  .enqueue_task        = enqueue_task_rt,
  .dequeue_task        = dequeue_task_rt
  .yield_task          = yield_task_rt,

  .check_preempt_curr = check_preempt_curr_rt,

  .pick_next_task              = pick_next_task_rt,
  .put_prev_task       = put_prev_task_rt,
  ...
  ```

Manuel Prieto-Matías
Universidad Complutense de Madrid

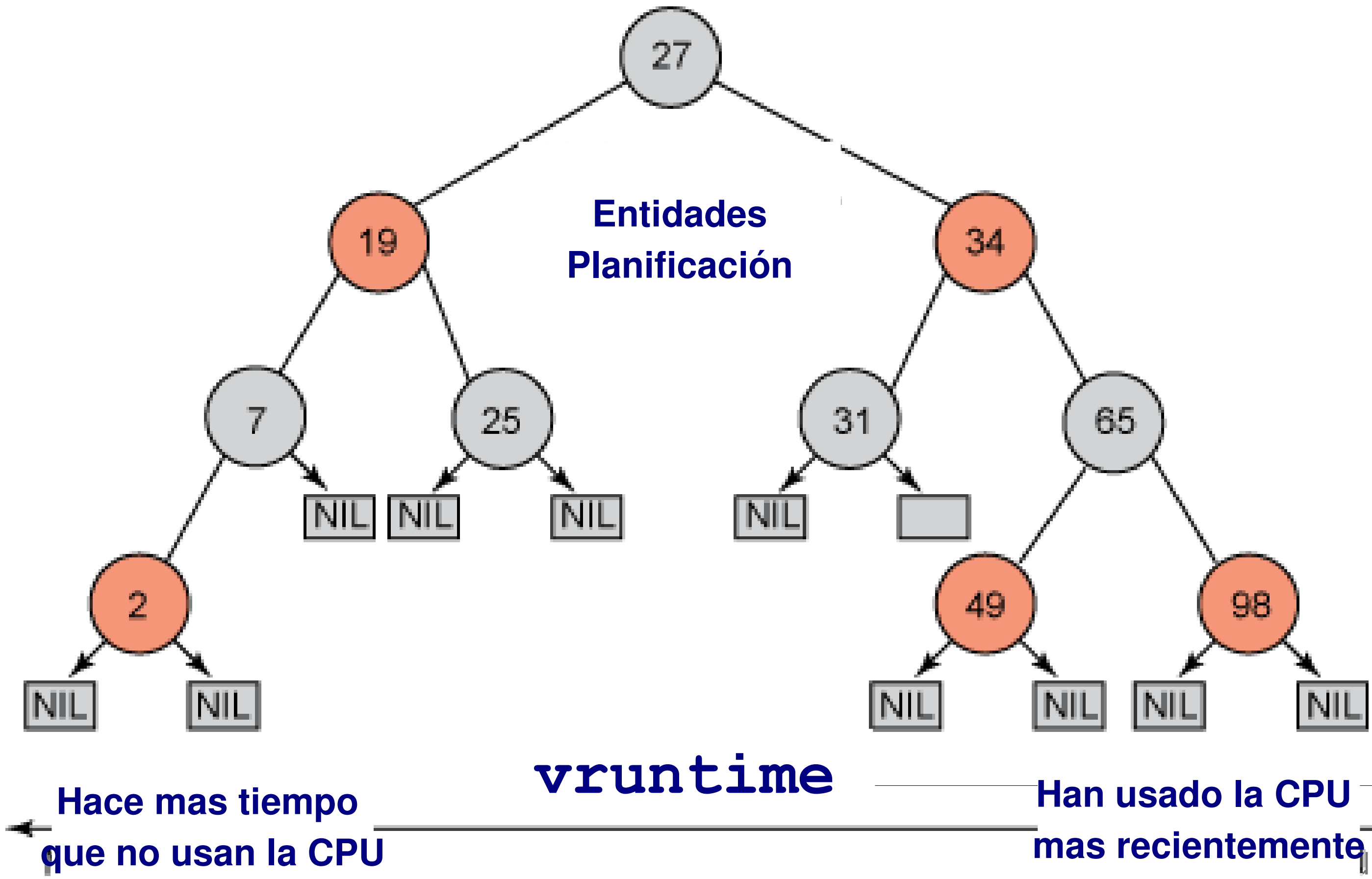
runqueue

- struct rq {

```
spinlock_t lock;
unsigned long nr_running;
unsigned long cpu_load[CPU_LOAD_IDX_MAX];
struct load_weight load; capture load from *all* tasks on this cpu:
unsigned long nr_load_updates;
u64 nr_switches;
u64 nr_migrations_in;
struct cfs_rq cfs;
struct rt_rq rt;
unsigned long nr_uninterruptible;
struct task_struct *curr, *idle;
unsigned long next_balance;
struct mm_struct *prev_mm;
u64 clock;
atomic_t nr_iowait;
}
```

CFS – Clase Fair –

- Árbol RB
  - Complejidad O(log N) – Búsquedas, Inserciones, Eliminaciones –
  - Nodos: entidades de planificación (habitualmente procesos)



CFS `cfs_rq` – subrunqueu –

- **struct cfs_rq {**

```
struct load_weight load;
unsigned long nr_running;

u64 exec_clock;
u64 min_vruntime;

struct rb_root tasks_timeline;
struct rb_node *rb_leftmost;

struct list_head tasks;
struct list_head *balance_iterator;

/*
 * 'curr' points to currently running entity on this cfs_rq.
 * It is set to NULL otherwise (i.e when none are currently running).
 */

struct sched_entity *curr, *next, *last;
```

CFS sched_entity

- struct sched_entity {

```
struct load_weight  load;          /* Peso entidad */
struct rb_node      run_node;
struct list_head    group_node;
unsigned int        on_rq;

u64                 exec_start;
u64                 sum_exec_runtime;
u64                 vruntime;
u64                 prev_sum_exec_runtime;

u64                 last_wakeup;
u64                 avg_overlap;

u64                 nr_migrations;

u64                 start_runtime;
u64                 avg_wakeup;
```

CFS sched_entity

- **struct sched_rt_entity {**

```
   struct list_head run_list;
   unsigned long timeout;
   unsigned int time_slice;
   int nr_cpus_allowed;
   struct sched_rt_entity *back;
}
```

CFS `task_struct` – Prioridades,... –

- `struct task_struct {`

```
  …
  int prio, static_prio, normal_prio;
  unsigned int rt_priority;

  const struct sched_class *sched_class;

  struct sched_entity se;
  struct sched_rt_entity rt;
  …
  unsigned int policy;
  cpumask_t cpus_allowed;
  …
}
```

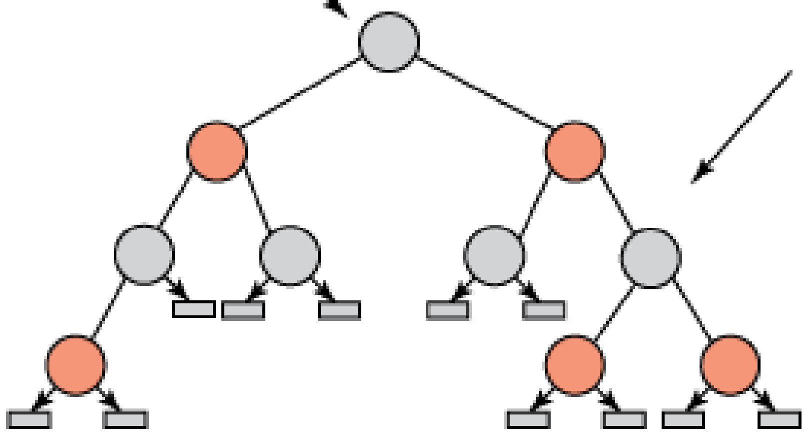
CFS – Clase Fair –

```
struct task_struct {
  volatile long state;
  void *stack;
  unsigned int flags;
  int prio, static_prio normal_prio;
  const struct sched_class *sched_class;
  struct sched_entity se;
  ...
};
```

```
struct sched_entity {
  struct load_weight load;
  struct rb_node run_node;
  struct list_head group_node;
  ...
};
```

```
struct ofs_rq {
   ...
   struct rb_root tasks_timeline;
   ...
};
```

```
struct rb_node {
  unsigned long rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;
};
```



CFS `vruntime`

- El RBTree registra en orden cronológico como será la ejecución de procesos futura
  - No hay *array switch artifacts*
- Los procesos se ordenan por la clave
  - **`p->se.vruntime`** (substrayendo **`rq->cfs.min_vruntime`**)
- Mientras un proceso se ejecuta se va incrementando su
  - **`p->se.vruntime`** (cada tick)
- Eventualmente el siguiente proceso en el *timeline* tendrá un **`vruntime`** inferior (mas una cierta distancia –granularidad– para evitar thrashing) que será el nuevo seleccionado...

Periodic scheduler `scheduler_tick`

- `void scheduler_tick(void){`

```
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;

    sched_clock_tick();

    spin_lock(&rq->lock);
    update_rq_clock(rq);
    update_cpu_load(rq);
    curr->sched_class->task_tick(rq, curr, 0);
    spin_unlock(&rq->lock);

    perf_counter_task_tick(curr, cpu);

#ifdef CONFIG_SMP
    rq->idle_at_tick = idle_cpu(cpu);
    trigger_load_balance(rq, cpu);
#endif
```

Periodic Scheduler – CFS –

- `task_tick_fair`
  - `entity_tick`
    - `update_curr(cfs_rq);`
      - `__update_curr()`
    - <u>Hay mas de un proceso activo?</u>
      - `check_preempt_tick`
        - <u>Se ha excedido el limite de latencia?</u>

          `resched_task`

CFS update_curr

- static void update_curr(struct cfs_rq *cfs_rq){

```
struct sched_entity *curr = cfs_rq->curr;
u64 now = rq_of(cfs_rq)->clock;
unsigned long delta_exec;

if (unlikely(!curr))
    return;

delta_exec = (unsigned long)(now - curr->exec_start);

if (!delta_exec)
    return;

__update_curr(cfs_rq, curr, delta_exec);

curr->exec_start = now;
```

CFS update_curr

- `__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr, unsigned long delta_exec){`

```
unsigned long delta_exec_weighted;
schedstat_set(curr->exec_max, max((u64)delta_exec, curr->exec_max));
curr->sum_exec_runtime += delta_exec;
schedstat_add(cfs_rq, exec_clock, delta_exec);

delta_exec_weighted = calc_delta_fair(delta_exec, curr);
curr->vruntime += delta_exec_weighted;

update_min_vruntime(cfs_rq);
```

Manuel Prieto-Matías
Universidad Complutense de Madrid

Main scheduler `schedule`

- `asmlinkage void __sched schedule(void){`

```
    struct task_struct *prev, *next;
    struct rq *rq;
    int cpu;
need_resched:
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;
need_resched_nonpreemptible:
    spin_lock_irq(&rq->lock);
    update_rq_clock(rq);
    clear_tsk_need_resched(prev);
    if (unlikely(signal_pending_state(prev->state, prev)))
     prev->state = TASK_RUNNING;
    else
     deactivate_task(rq, prev, 1);
    put_prev_task(rq, prev);
    next = pick_next_task(rq);
```

Main scheduler `schedule`

- `asmlinkage void __sched schedule(void){`

```
  if (likely(prev != next)) {
    rq->nr_switches++;
    rq->curr = next;
    ++*switch_count;
    context_switch(rq, prev, next); /* unlocks the rq */
    ...
  } else
    spin_unlock_irq(&rq->lock);
   if (unlikely(reacquire_kernel_lock(current) < 0))
     goto need_resched_nonpreemptible;
   preempt_enable_no_resched();
   if (need_resched())
     goto need_resched;
   }

EXPORT_SYMBOL(schedule);
```

Manuel Prieto-Matías
Universidad Complutense de Madrid

AISO Introducción

Versión 0.1