Process Scheduling

Traditionally we classified processes as I/O-bound or CPU-bound but there is an alternative more detailed classification distinguishes the processes,

- **Interactive processes:** these interact with users spending a lot of time waiting for key presses and mouse operations i.e. command shell, text editor, graphical applications….
- **Batch processes:** no user interaction, and often run in the background i.e. compilers, database search engines, scientific computations…
- **Real-time processes:** stringent in time, such processes should never be blocked by lower-priority processes i.e. sound applications, robot controllers…

Timeslice Concept

- Sometimes called quantum or processor slice, it refers to the numeric value that represents how long a task can run until it is preempted.
- Too long timeslice causes the system to have poor interactive performance, it will no longer feel as if applications are concurrently executed.
- Too short timeslice causes significant amounts of processor time to be wasted on the overhead of switching processes.

Timeslice Concept, *Cont'd*

- Most of the processor-bound processes needs long time-slices to keep their caches hot.
- While I/O-bound processes do not need longer time-slices.
- Because of these conflicting purposes, the CFS scheduler implemented such that not directly assign absolute timeslice to process instead, it assigns to the process a proportion of the processor according to the historic behavior and nice value as we discussed before.

Before CFS , Let's see traditional Unix systems,

Process priority and timeslice are the two main elements in scheduling such that Processes with a **higher priority**, receive a higher timeslice so running more frequently, so the **nice values** will be mapped in to an absolute timeslices and this leads to **suboptimal behavior**,

1) Suboptimal switching

- Let's assume this system scheme for absolute nice value mapping,

Nice	Time-slice assigned
Default Nice (0)	100ms
Highest Nice (19)	5ms

- If we have 2 low priority processes, nice(19) , so each will receive 50% of the processor 5 out of 10 ms, so the context switch twice every 10 ms.
- ***All the time I'm doing context switching than I process the tasks!!!***

2) Relative nice values reasonability

- If we have 2 processes, nice(0), nice(1), the timeslices will be 100 and 95 ms, so here for a 1 nice value difference the time slices difference is small..
- Another 2 processes, nice(18), nice(19), the timeslices will be 10 and 5 ms.
- *Nicing down a process by 1 has wildly different effects depending on the starting nice value!!!*

3) Nice value and tick multiplicity relation

- Because of performing a nice value to timeslice absolute mapping that leads to,
- *Restrict the minimum timeslice to have a floor of the period of the timer tick.*
- *The system timer limits the difference between two timeslices.*

4) Wake up a process, its timeslice was expired

- This happens mostly with the scheduler that optimize the performance against interactive tasks.
- ***Leads to unfair amount of processor time with processes.***

Nice	Time-slice assigned
Default Nice (0)	100ms
Highest Nice (19)	5ms

Completely Fair Scheduling “CFS”

- In CFS, each process runs for a “timeslice” proportional to its weight divided by the total weight of all ready threads.

- CFS sets a target for its approximation of the “infinitely small” scheduling duration in perfect multitasking called ***targeted latency*** which is the minimum amount of time required for every runnable task to get at least one turn on the processor.

- Smaller ***targeted latency*** yield better ***interactivity*** as each process will be serviced a lot but would lead to many context switching that costs a lot so that the throughput will be worse!!!

- i.e.1 if ***targeted latency*** is 20 ms and have 2 processes, so each one will run 20/2 = 10 ms.

- i.e.2 if ***targeted latency*** is 20 ms and have so many processes,so each one will run 20/∞ ~= 0 ms.

- So that CSF imposes ***minimum granularity***, each process will run at least with this time.

/kernel/sched/fair.c

```
/*
 * Targeted preemption latency for CPU-bound tasks:
 *
 * NOTE: this latency value is not the same as the concept of
 * 'timeslice length' - timeslices in CFS are of variable length
 * and have no persistent notion like in traditional, time-slice
 * based scheduling concepts.
 *
 * (to see the precise effective timeslice length of your workload,
 *  run vmstat and monitor the context-switches (cs) field)
 *
 * (default: 6ms * (1 + ilog(ncpus)), units: nanoseconds)
 */
unsigned int sysctl_sched_latency            = 6000000ULL;
static unsigned int normalized_sysctl_sched_latency = 6000000ULL;
```

```
/*
 * Minimal preemption granularity for CPU-bound tasks:
 *
 * (default: 0.75 msec * (1 + ilog(ncpus)), units: nanoseconds)
 */
unsigned int sysctl_sched_min_granularity            = 750000ULL;
static unsigned int normalized_sysctl_sched_min_granularity = 750000ULL;
```

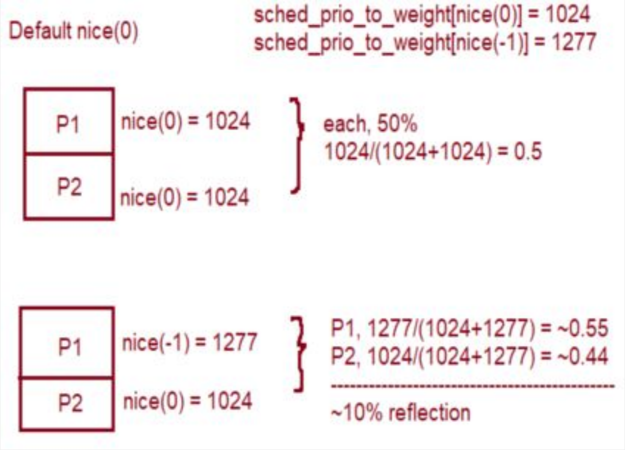
Completely Fair Scheduling “CFS”, *Cont’d*

- Relative nice/timeslice mapping “No absolute mapping anymore”

  Instead of using the nice value to calculate a timeslice, <u>CFS</u> uses the <u>nice value</u> to <u>weight the proportion</u> of processor a process is to receive such that,

/kernel/sched/core.c

```
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```



Completely Fair Scheduling "CFS", *Cont'd*

- Relative nice/timeslice mapping, *Cont'd*

i.e.1

process	Nice	Time Slice	Ideal Time	Targeted Latency
P1	nice(0)	~3x cpu	~15 ms	20 ms
P2	nice(5)	~1x cpu	~5 ms	20 ms

i.e.2

process	Nice	Time Slice	Ideal Time	Targeted Latency
P1	nice(10)	~3x cpu	~15 ms	20 ms
P2	nice(15)	~1x cpu	~5 ms	20 ms

- So, absolute nice values no longer affect scheduling decisions, only relative values affect the proportion of processor time allotted.

- Each sys tick, the timeslice is decremented by the tick period; when the timeslice reaches 0, the process is preempted in favor of another ready process with a nonzero timeslice.

/kernel/sched/core.c

```c
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

Completely Fair Scheduling “CFS”, *Cont’d*

- Task runtime statistics
  - After assign each process a time slice to run, we need to calculate some statistics about the process.

  - ***update_curr() :*** update the current task runtime statistics.
    The main job of ***update_curr()*** is to manage the ***vruntime***.

  - ***vruntime “Virtual Runtime” :*** The amount of time the process running weighted by number of running tasks at the time of enqueue of the task in rbtree.

  - CFS tries to run the task with the smallest ***vruntime***.

  Sum_exec_runtime : the consumed CPU time by the process.

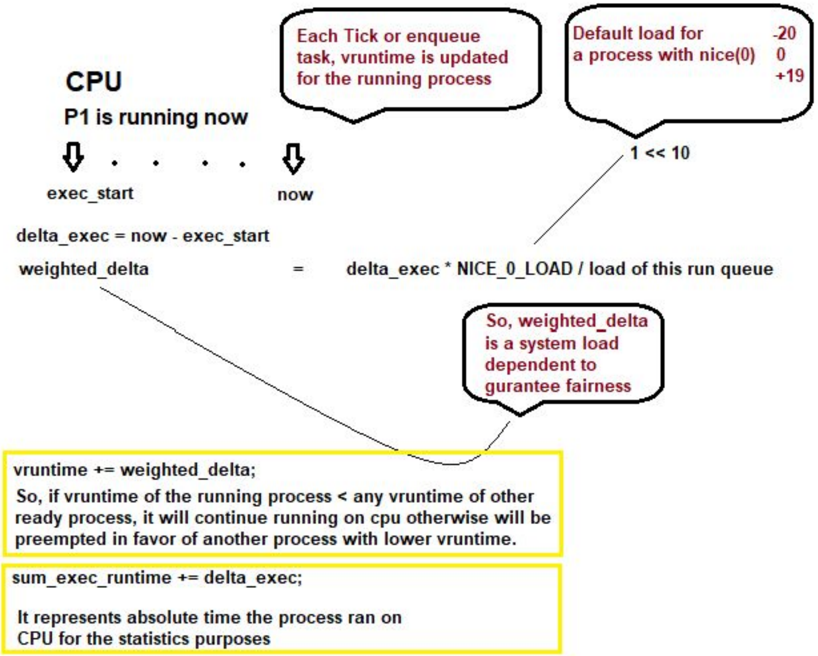
Note: Every task belongs to a scheduling entity (group of tasks), this group represented by the structure ***sched_entity***.

/include/linux/sched.h

```
struct sched_entity {
	/* For load-balancing: */
	struct load_weight		load;
	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				nr_migrations;

	struct sched_statistics		statistics;

#ifdef CONFIG_FAIR_GROUP_SCHED
	int				depth;
	struct sched_entity		*parent;
	/* rq on which this entity is (to be) queued: */
	struct cfs_rq			*cfs_rq;
	/* rq "owned" by this entity/group: */
	struct cfs_rq			*my_q;
	/* cached value of my_q->h_nr_running */
	unsigned long			runnable_weight;
#endif

#ifdef CONFIG_SMP
	/*
	 * Per entity load average tracking.
	 *
	 * Put into separate cache line so it does not
	 * collide with read-mostly values above.
	 */
	struct sched_avg		avg;
#endif
};
```

Completely Fair Scheduling "CFS", *Cont'd*

- Task runtime calculations

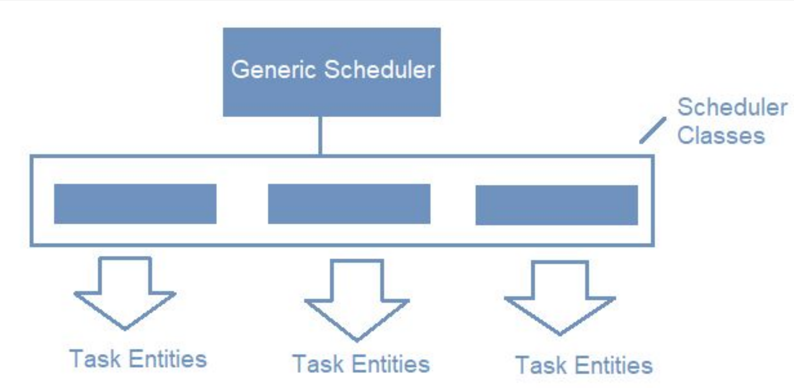


/include/linux/sched.h

```c
struct sched_entity {
	/* For load-balancing: */
	struct load_weight		load;
	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				nr_migrations;

	struct sched_statistics		statistics;

#ifdef CONFIG_FAIR_GROUP_SCHED
	int				depth;
	struct sched_entity		*parent;
	/* rq on which this entity is (to be) queued: */
	struct cfs_rq			*cfs_rq;
	/* rq "owned" by this entity/group: */
	struct cfs_rq			*my_q;
	/* cached value of my_q->h_nr_running */
	unsigned long			runnable_weight;
#endif

#ifdef CONFIG_SMP
	/*
	 * Per entity load average tracking.
	 *
	 * Put into separate cache line so it does not
	 * collide with read-mostly values above.
	 */
	struct sched_avg		avg;
#endif
};
```

Scheduler Classes

- As we know we have 3 scheduling classes in linux (**CFS, Real-time, Deadline**).
- Each scheduler class has a priority, the scheduler iterates over each class in order of priority, the highest priority class that has a runnable process wins the cpu.
- Real-Time or Deadline generally has preference over normal or CFS class.
- The generic scheduler defines abstract interfaces through a structure called ***sched_class***.
- Every scheduler class implements operations as defined in the ***sched_class*** structure.
- Each task_struct poins the class it belongs to, ***const struct sched_class *sched_class;***



Scheduler Classes, *Cont'd*

- **sched_class** Highlights
    - void ***enqueue_task***(struct rq *rq, struct task_struct *p, int flags);
      /*Enqueue task p on runqueue r*/.
    - struct task_struct ****pick_next_task*** (struct rq *rq, struct task_struct *prev, struct pin_cookie cookie);

      /*Pick the task that should be currently running*/
    - void ***yield_task***(struct rq *rq);

      /*When the process wants to relinquish CPU voluntarily*/
    - void ***task_tick***(struct rq *rq, struct task_struct *p, int queued);

      /*It is called whenever a timer interrupt happens to perform bookkeeping and set the ***need_resched*** flag if the running process needs to be preempted*/

/include/linux/sched.h

```
struct sched_class {

#ifdef CONFIG_UCLAMP_TASK
    int uclamp_enabled;
#endif

    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*yield_task)   (struct rq *rq);
    bool (*yield_to_task)(struct rq *rq, struct task_struct *p);

    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);

    struct task_struct *(*pick_next_task)(struct rq *rq);

    void (*put_prev_task)(struct rq *rq, struct task_struct *p);
    void (*set_next_task)(struct rq *rq, struct task_struct *p, bool first);

#ifdef CONFIG_SMP
    int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
    int  (*select_task_rq)(struct task_struct *p, int task_cpu, int flags);
    void (*migrate_task_rq)(struct task_struct *p, int new_cpu);

    void (*task_woken)(struct rq *this_rq, struct task_struct *task);

    void (*set_cpus_allowed)(struct task_struct *p,
                 const struct cpumask *newmask,
                 u32 flags);

    void (*rq_online)(struct rq *rq);
    void (*rq_offline)(struct rq *rq);

    struct rq *(*find_lock_rq)(struct task_struct *p, struct rq *rq);
#endif

    void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
    void (*task_fork)(struct task_struct *p);
    void (*task_dead)(struct task_struct *p);

    /*
     * The switched_from() call is allowed to drop rq->lock, therefore we
     * cannot assume the switched_from/switched_to pair is serliazed by
     * rq->lock. They are however serialized by p->pi_lock.
     */
    void (*switched_from)(struct rq *this_rq, struct task_struct *task);
    void (*switched_to)  (struct rq *this_rq, struct task_struct *task);
    void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                  int oldprio);

    unsigned int (*get_rr_interval)(struct rq *rq,
                   struct task_struct *task);

    void (*update_curr)(struct rq *rq);

#define TASK_SET_GROUP      0
#define TASK_MOVE_GROUP     1

#ifdef CONFIG_FAIR_GROUP_SCHED
    void (*task_change_group)(struct task_struct *p, int type);
#endif
};
```

Scheduler Classes, *Cont'd*

- **sched_class** Highlights, *Cont'd*
  - int ***select_task_rq***(struct task_struct *p, int task_cpu, int sd_flag, int flags);

    /*This is used for distributing processes across multiple CPU's as the core scheduler invokes this function to figure out which CPU to assign a task to*/

  - void ***migrate_task_rq***(struct task_struct *p,
    int new_cpu);
    /*Migrate task to specific CPU*/

/include/linux/sched.h

```c
struct sched_class {

#ifdef CONFIG_UCLAMP_TASK
    int uclamp_enabled;
#endif

    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*yield_task)   (struct rq *rq);
    bool (*yield_to_task)(struct rq *rq, struct task_struct *p);

    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);

    struct task_struct *(*pick_next_task)(struct rq *rq);

    void (*put_prev_task)(struct rq *rq, struct task_struct *p);
    void (*set_next_task)(struct rq *rq, struct task_struct *p, bool first);

#ifdef CONFIG_SMP
    int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
    int  (*select_task_rq)(struct task_struct *p, int task_cpu, int flags);
    void (*migrate_task_rq)(struct task_struct *p, int new_cpu);

    void (*task_woken)(struct rq *this_rq, struct task_struct *task);

    void (*set_cpus_allowed)(struct task_struct *p,
                 const struct cpumask *newmask,
                 u32 flags);

    void (*rq_online)(struct rq *rq);
    void (*rq_offline)(struct rq *rq);

    struct rq *(*find_lock_rq)(struct task_struct *p, struct rq *rq);
#endif

    void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
    void (*task_fork)(struct task_struct *p);
    void (*task_dead)(struct task_struct *p);

    /*
     * The switched_from() call is allowed to drop rq->lock, therefore we
     * cannot assume the switched_from/switched_to pair is serliazed by
     * rq->lock. They are however serialized by p->pi_lock.
     */
    void (*switched_from)(struct rq *this_rq, struct task_struct *task);
    void (*switched_to)  (struct rq *this_rq, struct task_struct *task);
    void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                  int oldprio);

    unsigned int (*get_rr_interval)(struct rq *rq,
                   struct task_struct *task);

    void (*update_curr)(struct rq *rq);

#define TASK_SET_GROUP      0
#define TASK_MOVE_GROUP     1

#ifdef CONFIG_FAIR_GROUP_SCHED
    void (*task_change_group)(struct task_struct *p, int type);
#endif
};
```

Scheduler Classes, *Cont'd*

Idle class ***/kernel/sched/idle.c***

```
/*
 * Simple, special scheduling class for the per-CPU idle tasks:
 */
DEFINE_SCHED_CLASS(idle) = {

	/* no enqueue/yield_task for idle tasks */

	/* dequeue is not valid, we print a debug message there: */
	.dequeue_task		= dequeue_task_idle,

	.check_preempt_curr = check_preempt_curr_idle,

	.pick_next_task		= pick_next_task_idle,
	.put_prev_task		= put_prev_task_idle,
	.set_next_task			= set_next_task_idle,

#ifdef CONFIG_SMP
	.balance		= balance_idle,
	.select_task_rq		= select_task_rq_idle,
	.set_cpus_allowed	= set_cpus_allowed_common,
#endif

	.task_tick		= task_tick_idle,

	.prio_changed		= prio_changed_idle,
	.switched_to		= switched_to_idle,
	.update_curr		= update_curr_idle,
};
```

Fair class ***/kernel/sched/fair.c***

```
DEFINE_SCHED_CLASS(fair) = {

	.enqueue_task		= enqueue_task_fair,
	.dequeue_task		= dequeue_task_fair,
	.yield_task		= yield_task_fair,
	.yield_to_task		= yield_to_task_fair,

	.check_preempt_curr = check_preempt_wakeup,

	.pick_next_task		= __pick_next_task_fair,
	.put_prev_task		= put_prev_task_fair,
	.set_next_task			= set_next_task_fair,

#ifdef CONFIG_SMP
	.balance		= balance_fair,
	.select_task_rq		= select_task_rq_fair,
	.migrate_task_rq	= migrate_task_rq_fair,

	.rq_online		= rq_online_fair,
	.rq_offline		= rq_offline_fair,

	.task_dead		= task_dead_fair,
	.set_cpus_allowed	= set_cpus_allowed_common,
#endif

	.task_tick		= task_tick_fair,
	.task_fork		= task_fork_fair,

	.prio_changed		= prio_changed_fair,
	.switched_from		= switched_from_fair,
	.switched_to		= switched_to_fair,

	.get_rr_interval	= get_rr_interval_fair,

	.update_curr		= update_curr_fair,

#ifdef CONFIG_FAIR_GROUP_SCHED
	.task_change_group	= task_change_group_fair,
#endif

#ifdef CONFIG_UCLAMP_TASK
	.uclamp_enabled		= 1,
#endif
};
```

Scheduler Classes, *Cont'd*

rt class

/kernel/sched/rt.c

```
DEFINE_SCHED_CLASS(rt) = {

    .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,
    .set_next_task          = set_next_task_rt,

#ifdef CONFIG_SMP
    .balance        = balance_rt,
    .select_task_rq     = select_task_rq_rt,
    .set_cpus_allowed       = set_cpus_allowed_common,
    .rq_online              = rq_online_rt,
    .rq_offline             = rq_offline_rt,
    .task_woken     = task_woken_rt,
    .switched_from      = switched_from_rt,
    .find_lock_rq       = find_lock_lowest_rq,
#endif

    .task_tick      = task_tick_rt,

    .get_rr_interval    = get_rr_interval_rt,

    .prio_changed       = prio_changed_rt,
    .switched_to        = switched_to_rt,

    .update_curr        = update_curr_rt,

#ifdef CONFIG_UCLAMP_TASK
    .uclamp_enabled     = 1,
#endif
};
```

dl class

/kernel/sched/deadline.c

```
DEFINE_SCHED_CLASS(dl) = {

    .enqueue_task       = enqueue_task_dl,
    .dequeue_task       = dequeue_task_dl,
    .yield_task     = yield_task_dl,

    .check_preempt_curr = check_preempt_curr_dl,

    .pick_next_task     = pick_next_task_dl,
    .put_prev_task      = put_prev_task_dl,
    .set_next_task      = set_next_task_dl,

#ifdef CONFIG_SMP
    .balance        = balance_dl,
    .select_task_rq     = select_task_rq_dl,
    .migrate_task_rq    = migrate_task_rq_dl,
    .set_cpus_allowed       = set_cpus_allowed_dl,
    .rq_online              = rq_online_dl,
    .rq_offline             = rq_offline_dl,
    .task_woken     = task_woken_dl,
    .find_lock_rq       = find_lock_later_rq,
#endif

    .task_tick      = task_tick_dl,
    .task_fork              = task_fork_dl,

    .prio_changed           = prio_changed_dl,
    .switched_from      = switched_from_dl,
    .switched_to        = switched_to_dl,

    .update_curr        = update_curr_dl,
};
```

Runqueues

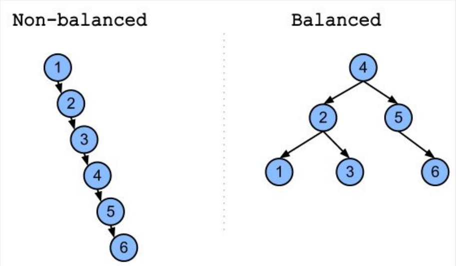
- Main runqueue
    - The runqueue contains all the processes that are ready to run on a given CPU core (**a run-queue is per-CPU**) in SMP (Symmetric Multi-processors) systems.
    - All scheduling classes embed their runqueues into the main runqueue structure (**cfs_rq, rt_rq, dl_rq**).
    - ***nr_running*** : denotes the number of processes in the runqueue.
    - ***curr*** : points to the task_struct of the current running task.
    - ***idle*** : points to the task_struct of the idle task, which is scheduled when there are no other tasks to run.

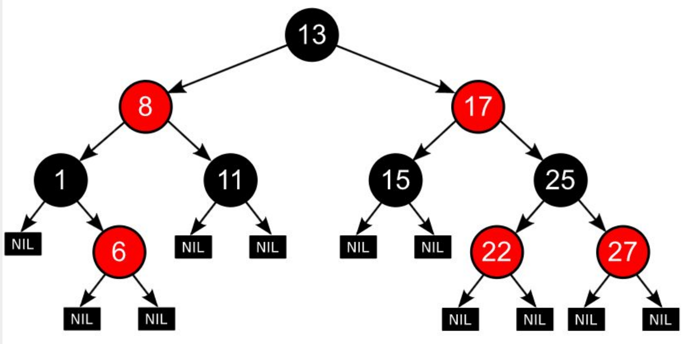
/kernel/sched/sched.h

```c
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {
    /* runqueue lock: */
    raw_spinlock_t      lock;

    /*
     * nr_running and cpu_load should be in the same cacheline because
     * remote CPUs use both these fields when doing load calculation.
     */
    unsigned int        nr_running;
    ...

    struct cfs_rq       cfs;
    struct rt_rq        rt;
    struct dl_rq        dl;
    ...
    struct task_struct __rcu    *curr;
    struct task_struct  *idle;
    ...
};
```

Runqueues, *Cont'd*

- cfs runqueue
  - It's a normal runqueue uses a ***self-balancing***, red-black tree instead to get to the next best process to run in the shortest possible time.
  - The RB tree holds all the ready processes and facilitates easy and quick ***insertion, deletion, and searching*** of processes.
  - The Completely Fair Scheduler was introduced in ***Kernel 2.6.23***, replacing the ***O(1)*** scheduler that causes some issues with interactive processes - "History".
  - Will talk about rbtree later in details.



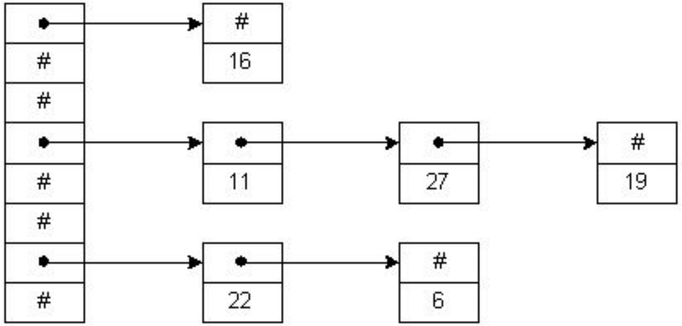



```c
/* CFS-related fields in a runqueue */
struct cfs_rq {
    struct load_weight  load;
    unsigned int        nr_running;
    unsigned int        h_nr_running;      /* SCHED_{NORMAL,BATCH,IDLE} */
    unsigned int        idle_h_nr_running; /* SCHED_IDLE */
    ...
    /*
     * 'curr' points to currently running entity on this cfs_rq.
     * It is set to NULL otherwise (i.e when none are currently running).
     */
    struct sched_entity *curr;
    struct sched_entity *next;
    struct sched_entity *last;
    struct sched_entity *skip;
    ...
};
```

/kernel/sched/sched.h

Runqueues, *Cont'd*

- rt runqueue
  - rt scheduling is based on the **static priority**.
  - The rt runqueue is sort of **normal list** linked with each static priority.
  - Talk about Real-Time later.

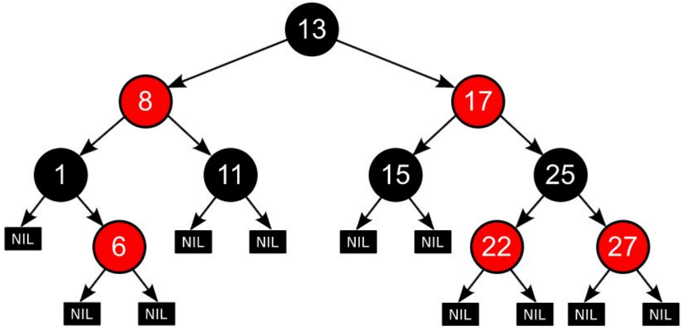


/kernel/sched/sched.h

```c
/* Real-Time classes' related field in a runqueue: */
struct rt_rq {
    struct rt_prio_array    active;
    unsigned int        rt_nr_running;
    unsigned int        rr_nr_running;
#if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
    struct {
        int     curr; /* highest queued rt task prio */
#ifdef CONFIG_SMP
        int     next; /* next highest */
#endif
    } highest_prio;
#endif
#ifdef CONFIG_SMP
    unsigned long       rt_nr_migratory;
    unsigned long       rt_nr_total;
    int         overloaded;
    struct plist_head   pushable_tasks;

#endif /* CONFIG_SMP */
    int         rt_queued;

    int         rt_throttled;
    u64         rt_time;
    u64         rt_runtime;
    /* Nests inside the rq lock: */
    raw_spinlock_t      rt_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
    unsigned long       rt_nr_boosted;

    struct rq       *rq;
    struct task_group   *tg;
#endif
};
```

Runqueues, *Cont'd*

- dl runqueue
  - The ***SCHED_DEADLINE*** policy is related to dl scheduling class.
  - Simply it's an implementation of the Earliest Deadline First ***(EDF)*** scheduling algorithm.
  - The dl runqueue is an rbtree, ordered by deadline.



/kernel/sched/sched.h

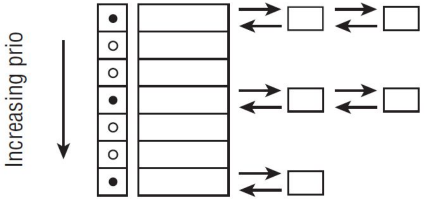
```
/* Deadline class' related fields in a runqueue */
struct dl_rq {
    /* runqueue is an rbtree, ordered by deadline */
    struct rb_root_cached   root;

    unsigned long       dl_nr_running;

#ifdef CONFIG_SMP
    /*
     * Deadline values of the currently executing and the
     * earliest ready task on this rq. Caching these facilitates
     * the decision whether or not a ready but not running task
     * should migrate somewhere else.
     */
    struct {
        u64     curr;
        u64     next;
    } earliest_dl;

    unsigned long       dl_nr_migratory;
    int         overloaded;

    /*
     * Tasks on this rq that can be pushed away. They are kept in
     * an rb-tree, ordered by tasks' deadlines, with caching
     * of the leftmost (earliest deadline) element.
     */
    struct rb_root_cached   pushable_dl_tasks_root;
#else
    struct dl_bw        dl_bw;
#endif
    ...
};
```

Real-time scheduling

- Linux supports **soft real-time** tasks and they are scheduled by the real-time scheduling class.
- **Soft real-time** refers to the notion that the kernel tries to schedule applications within timing deadlines, but the kernel does not promise to always achieve these goals.
- **Hard real-time** systems are guaranteed to meet any scheduling requirements within certain limits.
- Real-time Unlike CFS, which uses rbtree , the rt schedule uses a simple linked list.
- Linux applies two real-time policies, ***SCHED_FIFO*** (first in, first out), ***SCHED_RR*** (round-robin policy) indicated by the ***policy*** in struct ***task_struct***.
- FIFO Scheduling
    - A ready ***SCHED_FIFO*** task is always scheduled over any ***SCHED_NORMAL*** tasks.
    - When a ***SCHED_FIFO*** task becomes running, it continues to until it blocks or explicitly yields the processor; it has no timeslice and can run indefinitely.
    - Only a higher priority ***SCHED_FIFO***, ***SCHED_RR*** or ***SCHED_DEADLINE*** task can preempt a ***SCHED_FIFO*** task.
    - If two or more ***SCHED_FIFO*** tasks at the same priority, they will run **round-robin**, starting with the first process at the head of the list.
- SCHED_RR Scheduling
    - ***SCHED_RR*** (round-robin scheduling) is a ***SCHED_FIFO*** with timeslices.
    - When a SCHED_RR task exhausts its timeslice any other real-time processes at its priority are scheduled round-robin, as the timeslice is used to allow only rescheduling of same-priority processes.

Real-time scheduling, *Cont'd*

- Real-time core run queue
  - As we know the run queue of real time is straightforward normal "**linked list**".
  - All real-time tasks with the same priority are kept in a linked list headed by ***active.queue[prio]***.
  - The bitmap ***active.bitmap*** signals in which list tasks are present by a set bit.



/kernel/sched/sched.h

```
/*
 * This is the priority-queue data structure of the RT scheduling class:
 */
struct rt_prio_array {
	DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter */
	struct list_head queue[MAX_RT_PRIO];
};
```

/kernel/sched/sched.h

```
/* Real-Time classes' related field in a runqueue: */
struct rt_rq {
	struct rt_prio_array	active;
	unsigned int		rt_nr_running;
	unsigned int		rr_nr_running;
#if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
	struct {
		int		curr; /* highest queued rt task prio */
#ifdef CONFIG_SMP
		int		next; /* next highest */
#endif
	} highest_prio;
#endif
#ifdef CONFIG_SMP
	unsigned long		rt_nr_migratory;
	unsigned long		rt_nr_total;
	int		overloaded;
	struct plist_head	pushable_tasks;

#endif /* CONFIG_SMP */
	int		rt_queued;

	int		rt_throttled;
	u64		rt_time;
	u64		rt_runtime;
	/* Nests inside the rq lock: */
	raw_spinlock_t		rt_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
	unsigned long		rt_nr_boosted;

	struct rq		*rq;
	struct task_group	*tg;
#endif
};
```

Real-time scheduling, *Cont'd*

- Real-time core run queue, *Cont'd*
  - If no tasks are on the list, the bit is not set.
  - The analog of ***update_cur()***, for the real-time scheduler class is ***update_curr_rt()***.
  - ***update_curr_rt()*** keeps track of the time the current process spent executing on the CPU in ***sum_exec_runtime***.
  - For further rt related topics, https://www.linuxjournal.com/article/10165 Root Domain, CPU Priority Management,...

/kernel/sched/sched.h

```
/*
 * This is the priority-queue data structure of the RT scheduling class:
 */
struct rt_prio_array {
	DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter */
	struct list_head queue[MAX_RT_PRIO];
};
```

/kernel/sched/sched.h

```
/* Real-Time classes' related field in a runqueue: */
struct rt_rq {
	struct rt_prio_array	active;
	unsigned int		rt_nr_running;
	unsigned int		rr_nr_running;
#if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
	struct {
		int		curr; /* highest queued rt task prio */
#ifdef CONFIG_SMP
		int		next; /* next highest */
#endif
	} highest_prio;
#endif
#ifdef CONFIG_SMP
	unsigned long		rt_nr_migratory;
	unsigned long		rt_nr_total;
	int			overloaded;
	struct plist_head	pushable_tasks;

#endif /* CONFIG_SMP */
	int			rt_queued;

	int			rt_throttled;
	u64			rt_time;
	u64			rt_runtime;
	/* Nests inside the rq lock: */
	raw_spinlock_t		rt_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
	unsigned long		rt_nr_boosted;

	struct rq		*rq;
	struct task_group	*tg;
#endif
};
```

Deadline scheduling

- Before Jumping into dl scheduling, let's discuss one of the popular scheduling algorithm for RTOS, RMS (Rate Monotonic Scheduling) and what the dl scheduling can add!!!

- RMS, theoretically assumes,
  - No resource sharing such as hardware, a queue, or any kind of semaphore.
  - Static priorities (the task with the highest static priority that is runnable immediately preempts all other tasks).
  - Static priorities assigned according to the rate monotonic conventions (tasks with shorter periods are given higher priorities).
  - Context switch times has no impact on the model.
  - CPU utilization,
    n: number of periodic tasks, Ti: Release period, Ci: Computation time.

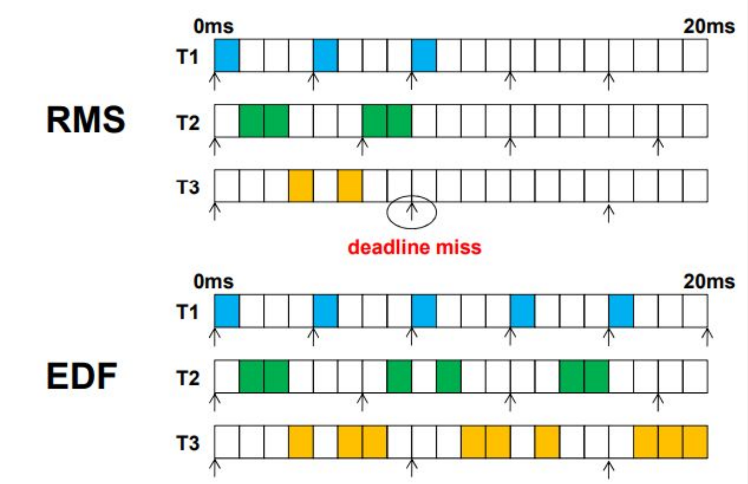
$$U = \sum_{i=0}^{n} C_i / T_i$$

Deadline scheduling, *Cont'd*

- RMS, *Cont'd*
  - Consider this example,

    Task1: budget 1ms period 4ms
    Task2: budget 2ms period 6ms
    Task3: budget 3ms period 8ms

  - **EDF**: Earlist Deadline First,selects the task with the earliest deadline as the one to be executed next. Assume dl = period.



Linux dl scheduling

- *SCHED_DEADLINE* uses three parameters, runtime, period and deadline.
  - runtime: *SCHED_DEADLINE* task should receive "runtime" of execution time every "period".
  - period and deadline: This "runtime" is available within certain "deadline" from the beginning of the "period".
- Deadline uses CBS (Constant Bandwidth Server) algorithm to assign deadlines to tasks so that each task runs for at most its runtime every "period", avoiding any interference between different tasks (bandwidth isolation).
- Bandwidth in this scope generally refers to the task "runtime and remaining runtime" and "the next deadline time".
- Then after assigning deadlines to tasks, the EDF (Earliest Deadline First) algorithm selects the task with the earliest deadline as the one to be executed next.

- CBS algorithm,
  - The state of the task is described by a "scheduling deadline", and a "remaining runtime".
  - When the task wakes up (becomes ready for execution), the scheduler checks if "scheduling deadline" time < the current time and that need to be adjusted so, the "scheduling deadline" time and the remaining runtime are re-initialized, otherwise it runs normally.
  - If the task executes for an amount of time t, so remaining runtime = remaining runtime - t;
  - When the remaining runtime <= 0, the task is said to be "throttled" and cannot be scheduled, and there a time called replenishment time is set for this task, in this case, the task will be able to run only after this replenishment time at the beginning of the next period.

Linux dl scheduling,*Cont'd*

- CBS algorithm,*Cont'd*
  - Therefore, CBS is used to both guarantee each task's CPU time based on its timing requirements and to prevent a misbehaving task from running for more than its run time and causing problems to other jobs.
  - In order to avoid overloading the system with deadline tasks, the deadline scheduler implements an **acceptance test**, which is done every time a task is configured to run with the deadline scheduler.
  - This **acceptance test** guarantees that deadline tasks will not use more than the maximum amount of the system's CPU time.
- Bandwidth Reclaiming
  - Bandwidth reclaiming for deadline tasks is based on the GRUB (Greedy Reclamation of Unused Bandwidth) algorithm.

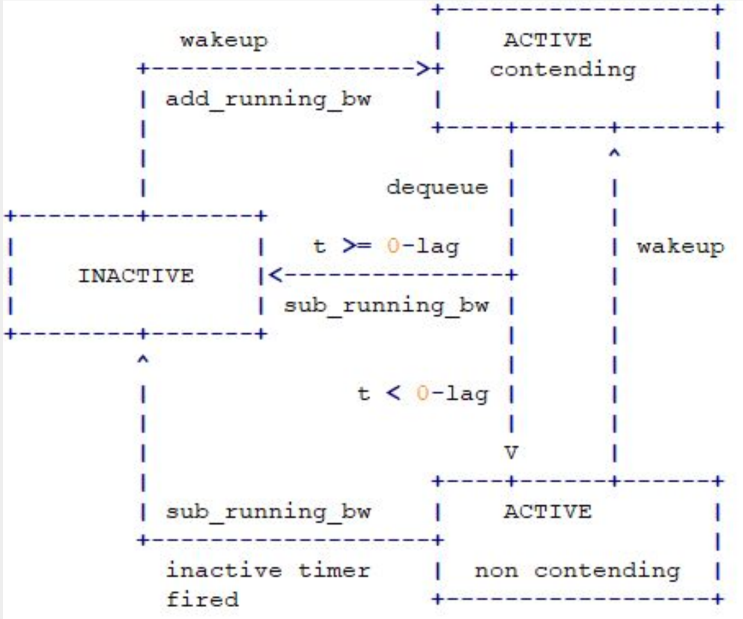
  - What's the problem? look at this example, tasks' bandwidth is fixed (can only be changed with ***sched_setattr()*** from the user).
  - What if the task blocked on resource!!! ***All the real-time guarantees will be broken.***
  - So, Bandwidth Reclaiming is needed.

```c
int main (int argc, char **argv)
{
        int ret;
        int flags = 0;
        struct sched_attr attr;

        memset(&attr, 0, sizeof(attr));
        attr.size = sizeof(attr);

        /* This creates a 200ms / 1s reservation */
        attr.sched_policy   = SCHED_DEADLINE;
        attr.sched_runtime  =  200000000;
        attr.sched_deadline = attr.sched_period = 1000000000;

        ret = sched_setattr(0, &attr, flags);
        if (ret < 0) {
            perror("sched_setattr failed to set the priorities");
            exit(-1);
        }

        do_the_computation_without_blocking();
        exit(0);
}
```

Fixed once attr created

Linux dl scheduling,*Cont'd*

- Bandwidth Reclaiming, *Cont'd*
    - Le's see the task state transitions,
    - When a task blocks, it does not become immediately *INACTIVE* since its bandwidth cannot be immediately reclaimed without breaking the real-time guarantees, It therefore enters a transitional state called *ACTIVENonContending*.
    - The algorithm calculates a time called **0-lag** time for the task in order to know if the task bandwidth can be adjusted directly and become *INACTIVE* and ready for any wakeup to be *ACTIVEContending* or I need sort of transition state *ACTIVENonContending*.
    - Also the algorithm tracks, something called Active bandwidth (*running_bw*), which is the sum of the bandwidths of all tasks in active state (i.e., *ACTIVEContending* or *ACTIVENonContending*).



Process grouping

- Group scheduling under CFS
  - As we know, every task belongs to a scheduling entity (group of tasks), this group represented by the structure ***sched_entity*** for CFS processes.
  - Imagine process A spawns 10 processes and B spawns 5 only.
  - Leading to process A and its spawned will get **most of the CPU time** because they are greater and CFS divides the timeslices across all the processes.
  - To address this issue and to keep up the fairness, **group scheduling** feature is introduced where **timeslices are allotted to groups of threads instead of individual threads**.
  - So, A and B get 50% of the time each, then process A shall divide its 50% time among its spawned 10 threads, with each thread getting 5% time internally.

- *CONFIG_FAIR_GROUP_SCHED* is to be set when configuring the kernel.

/kernel/sched/sched.h

```
struct sched_entity {
    ...
#ifdef CONFIG_FAIR_GROUP_SCHED
    int                     depth;
    struct sched_entity     *parent;
    /* rq on which this entity is (to be) queued: */
    struct cfs_rq           *cfs_rq;
    /* rq "owned" by this entity/group: */
    struct cfs_rq           *my_q;
    /* cached value of my_q->h_nr_running */
    unsigned long           runnable_weight;
#endif
    ...
};
```

Process grouping, *Cont'd*

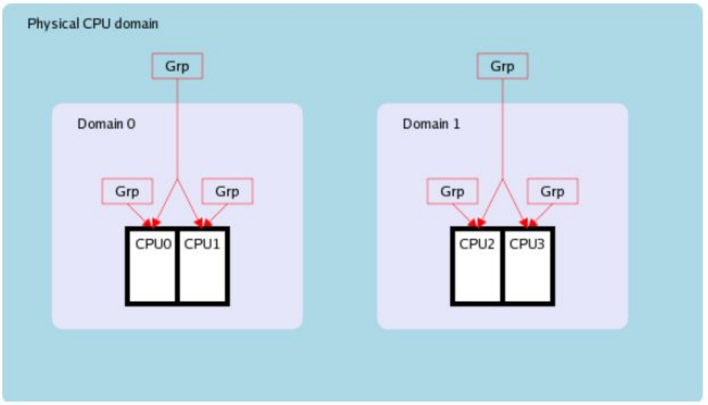
- Group scheduling under RT
  - real-time processes can also be grouped for scheduling with *CONFIG_RT_GROUP_SCHED* set.
  - For group scheduling to succeed, each group must be assigned a portion of CPU time, with a guarantee that the timeslice is enough to run the tasks under each entity, or it fails.
  - The kernel allocates for each group something called "run time" which is the execution time every period.
  - CPU time that is not allocated for real-time groups will be used by normal tasks.
  - Any time unused by the real-time entities will also be picked by the normal tasks.

/kernel/sched/sched.h

```
struct sched_rt_entity {
	struct list_head		run_list;
	unsigned long			timeout;
	unsigned long			watchdog_stamp;
	unsigned int			time_slice;
	unsigned short			on_rq;
	unsigned short			on_list;

	struct sched_rt_entity		*back;
#ifdef CONFIG_RT_GROUP_SCHED
	struct sched_rt_entity		*parent;
	/* rq on which this entity is (to be) queued: */
	struct rt_rq			*rt_rq;
	/* rq "owned" by this entity/group: */
	struct rt_rq			*my_q;
#endif
} __randomize_layout;
```

Process grouping, *Cont'd*

- Load Balancing on SMP (Symmetric Multi-processors) Systems
  - The main goal is to improve the performance of SMP systems by offloading tasks from busy CPUs to less busy or idle ones.
  - Scheduling domain: a set of CPUs which share properties and scheduling policies, and which can be balanced against each other.
  - Each domain can contain one or more scheduling groups (*sched_group*) which are treated as a single unit by the domain.
  - The scheduler tries to balance the load carried by each cpu group, such that the balancing within a sched domain occurs between groups.

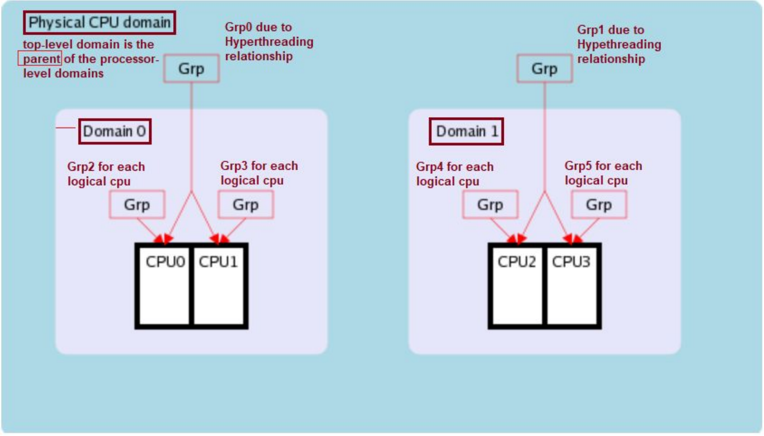


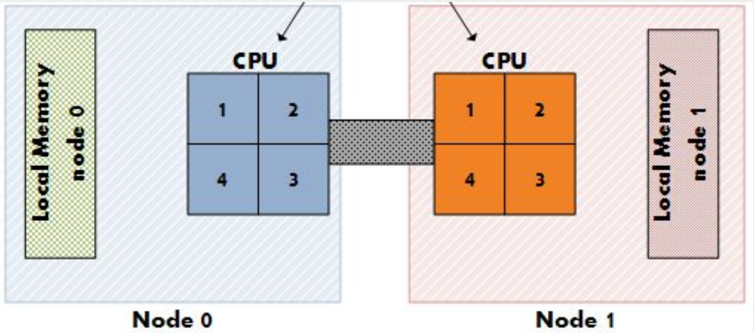
/kernel/sched/sched.h

```
struct sched_group {
    struct sched_group  *next;          /* Must be a circular list */
    atomic_t            ref;

    unsigned int        group_weight;
    struct sched_group_capacity *sgc;
    int             asym_prefer_cpu;    /* CPU of highest priority in group */

    /*
     * The CPUs this group covers.
     *
     * NOTE: this field is variable length. (Allocated dynamically
     * by attaching extra space to the end of the structure,
     * depending on how many CPUs the kernel has booted up with)
     */
    unsigned long       cpumask[];
};
```

Process grouping, *Cont'd*

- Load Balancing on SMP, *Cont'd*
  - Closer to this setup, we have 2 cpu cores but they are hyperthreaded by the hardware to be 4 cpu's, as each domain (*Domain0,1*) contains two CPU groups (*Grp2,3,4,5*), and each group contains exactly one CPU.
  - While each CPU appears to be a distinct processor, a pair of hyperthreaded processors has a different relationship internally so, *Grp0,1* created.
  - Physical CPU Domain is the parent of the processor-level domains, it contains two CPU groups (*Grp0,1*).
  - For **NUMA (Non-uniform memory access)** systems would have a little bit different hierarchy, will talk about it later.





Process grouping, *Cont'd*

- Load Balancing on SMP, *Cont'd*
    - Scheduling domain represented by struct ***sched_domain***.
    - ***parent*** : points to the base domain.
    - ***groups*** : points to the groups that the domain has.
    - ***span*** : each scheduling domain spans "balance process load among these CPU's" a number of cpu's stored in ***span*** field.
    - ***trigger_load_balance()*** is running periodically on each cpu through ***scheduler_tick()***, it raises a softirq to be deferred after the next regularly scheduled rebalancing event for the current runqueue has arrived.

/kernel/sched/sched.h

```
struct sched_domain {
    /* These fields must be setup */
    struct sched_domain __rcu *parent;  /* top domain must be null terminated */
    struct sched_domain __rcu *child;   /* bottom domain must be null terminated */
    struct sched_group *groups; /* the balancing groups of the domain */
    ...
    unsigned long span[];
};
```

/kernel/sched/core.c

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 */
void scheduler_tick(void)
{
    ...
    trigger_load_balance(rq);
    ...
};
```

```
void trigger_load_balance(struct rq *rq)
{
    ...
    if (time_after_eq(jiffies, rq->next_balance))
        raise_softirq(SCHED_SOFTIRQ);
    ...
}
```

/kernel/sched/fair.c

Process grouping, *Cont'd*

- Load Balancing on SMP, *Cont'd*
  - This *SCHED_SOFTIRQ* defined early at *init_sched_fair_class()* with *run_rebalance_domains()* that calls *rebalance_domains()*.
  - *rebalance_domains()* : iterates over all the different *sched_domain* 's that the running cpu is on, starting from its base domain and going up the parent chain and balance the process loads by *load_balance()* .
  - *load_balance()* finds the busiest group in the current sched domain then it looks for the busiest runqueue of that group (as each cpu has main rq), and starts moving tasks from busiest runqueue to the current running cpu.
  - Note: In the previous cpu cores setup, the group may running only on one cpu.

/kernel/sched/core.c

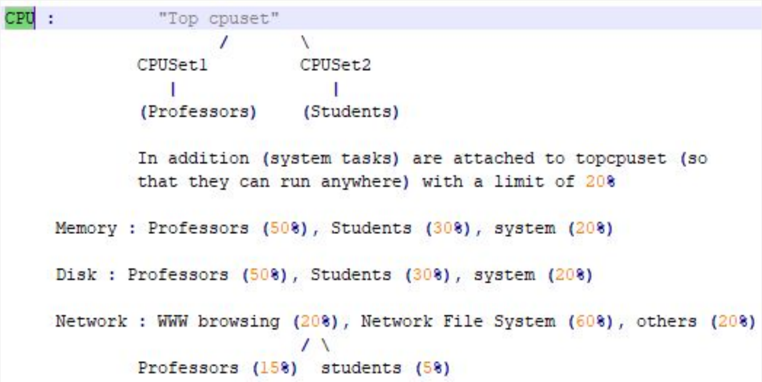
```c
__init void init_sched_fair_class(void)
{
#ifdef CONFIG_SMP
    open_softirq(SCHED_SOFTIRQ, run_rebalance_domains);

#ifdef CONFIG_NO_HZ_COMMON
    ...
}
```

/kernel/sched/fair.c

```c
static void rebalance_domains(struct rq *rq, enum cpu_idle_type idle)
{
        ...
        if (load_balance(cpu, rq, sd, idle, &continue_balancing)) {
            /*
             * The LBF_DST_PINNED logic could have changed
             * env->dst_cpu, so we can't know our idle
             * state even if we migrated tasks. Update it.
             */
            idle = idle_cpu(cpu) ? CPU_IDLE : CPU_NOT_IDLE;
            busy = idle != CPU_IDLE && !sched_idle_cpu(cpu);
        }
        ...
}
```

Process grouping, *Cont'd*

- Cgroup snippets "I hope will talk about it later in details ~~V1~~, V2"
    - Control Groups provide a mechanism for **aggregating/partitioning** sets of tasks, and all their future children, into **hierarchical** groups with specialized behaviour so, cgroup associates a set of tasks with a set of parameters for one or more **subsystems**.
    - Similar to the process model, where child cgroups inherit the attributes of the parent.
    - Why we need this feature?
      Consider an example, university server with various users - students, professors, system tasks and the resource planning as follows,



```
CPU :               "Top cpuset"
                       /       \
             CPUSet1         CPUSet2
                |               |
           (Professors)    (Students)

           In addition (system tasks) are attached to topcpuset (so
           that they can run anywhere) with a limit of 20%

    Memory : Professors (50%), Students (30%), system (20%)

    Disk : Professors (50%), Students (30%), system (20%)

    Network : WWW browsing (20%), Network File System (60%), others (20%)
                                / \
             Professors (15%)  students (5%)
```

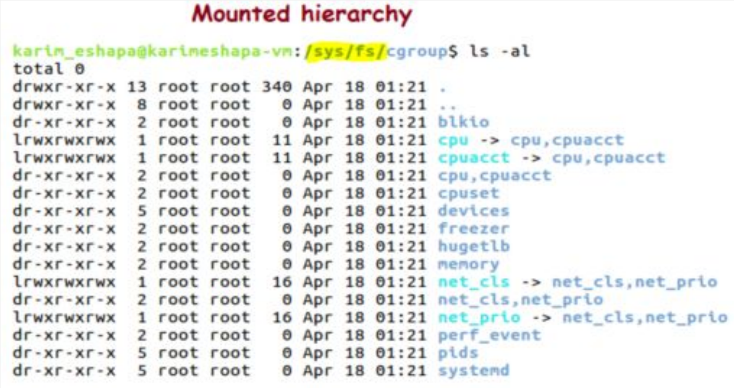
Process grouping, *Cont'd*

- Cgroup snippets, *Cont'd*
  - The kernel's **cgroup interface** is provided through a virtual filesystem, "like i.e. /proc filesystem that can extract and manipulate the kernel info", **cgroup fs** manipulating cgroups functionalities, this filesystem mounted by default at ***/sys/fs*** and can be moved whatever.

  - Why is this way of design!!!, because i.e. at multi-tenancy systems we need to mount cgroup interface in a separate disk areas for a company isolated from the others.

  - What **subsystem** means: a module that makes use of the task grouping facilities provided by cgroups to treat groups of tasks in particular ways.

  - What **hierarchy** means: is a set of cgroups arranged in a tree and each hierarchy has an instance of the cgroup virtual filesystem associated with it, as the administrator can create as many hierarchies as desired to control network bandwidth usage, memory usage,etc...



Process grouping, *Cont'd*

- Cgroup snippets, *Cont'd*
  - cgroup is composed of two parts, <u>cgroup-core</u> that responsible for hierarchically organizing processes and <u>cgroup-controller</u> that responsible for distributing a specific type of system resource along the hierarchy.

  - Every process in the system belongs to one and only one cgroup.

  - A process can be migrated to another cgroup, and migration of a process doesn't affect already existing descendant processes (Children).

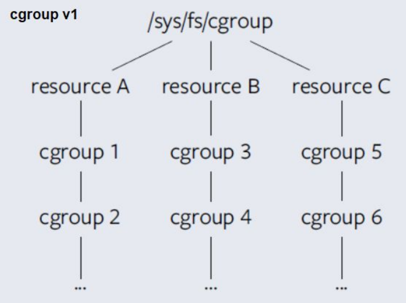
  - **cgroup~~V1~~** (It's replaced by **cgroupv2** but still supported for backward compatibility) has a hierarchy per-resource (blkio, cpuset, pids,...) such that, each resource hierarchy contains cgroups for this resource.

Mounted hierarchy

```
karim_eshapa@karimeshapa-vm:/sys/fs/cgroup$ ls -al
total 0
drwxr-xr-x 13 root root 340 Apr 18 01:21 .
drwxr-xr-x  8 root root   0 Apr 18 01:21 ..
dr-xr-xr-x  2 root root   0 Apr 18 01:21 blkio
lrwxrwxrwx  1 root root  11 Apr 18 01:21 cpu -> cpu,cpuacct
lrwxrwxrwx  1 root root  11 Apr 18 01:21 cpuacct -> cpu,cpuacct
dr-xr-xr-x  2 root root   0 Apr 18 01:21 cpu,cpuacct
dr-xr-xr-x  2 root root   0 Apr 18 01:21 cpuset
dr-xr-xr-x  5 root root   0 Apr 18 01:21 devices
dr-xr-xr-x  2 root root   0 Apr 18 01:21 freezer
dr-xr-xr-x  2 root root   0 Apr 18 01:21 hugetlb
dr-xr-xr-x  2 root root   0 Apr 18 01:21 memory
lrwxrwxrwx  1 root root  16 Apr 18 01:21 net_cls -> net_cls,net_prio
dr-xr-xr-x  2 root root   0 Apr 18 01:21 net_cls,net_prio
lrwxrwxrwx  1 root root  16 Apr 18 01:21 net_prio -> net_cls,net_prio
dr-xr-xr-x  2 root root   0 Apr 18 01:21 perf_event
dr-xr-xr-x  5 root root   0 Apr 18 01:21 pids
dr-xr-xr-x  5 root root   0 Apr 18 01:21 systemd
```

What can be controlled

```
karim_eshapa@karimeshapa-vm:/sys/fs/cgroup/cpuset$ ls
cgroup.clone_children   cpuset.memory_pressure
cgroup.procs            cpuset.memory_pressure_enabled
cgroup.sane_behavior    cpuset.memory_spread_page
cpuset.cpu_exclusive    cpuset.memory_spread_slab
cpuset.cpus             cpuset.mems
cpuset.effective_cpus   cpuset.sched_load_balance
cpuset.effective_mems   cpuset.sched_relax_domain_level
cpuset.mem_exclusive    notify_on_release
cpuset.mem_hardwall     release_agent
cpuset.memory_migrate   tasks
```



Process grouping, *Cont'd*

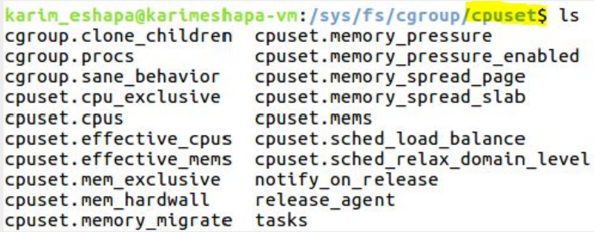
- Cgroup snippets, *Cont'd*
  - cgroup~~V1~~, *Cont'd*

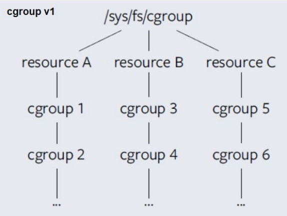
    any **cgroup**.*name* : is the **cgroup-core** interface files responsible for any hierarchy that could be created under this resource i.e. **cpusets** as,

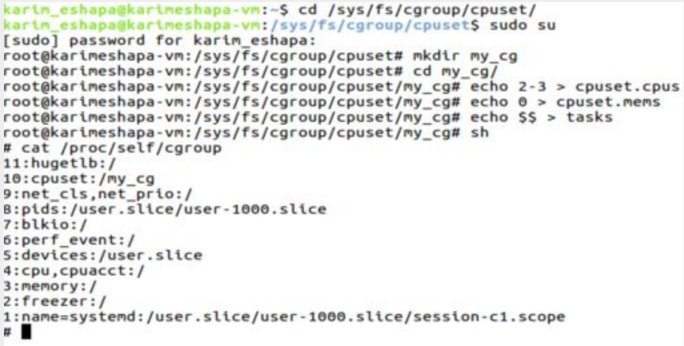
    cpusets controller: This cgroup can be used to bind the processes in a cgroup to a specified set of CPUs. and here any

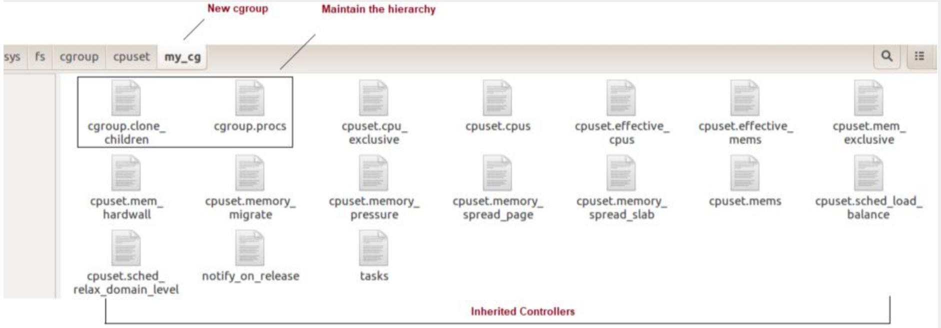
    cpuset.*name* : is a controller that is responsible for distributing a specific type of system resource along the hierarchy.

    Basic example:
    Create a **cgroup** containing CPUs 2 and 3, and Memory Node 0, and then start a **'sh'** in that cgroup.





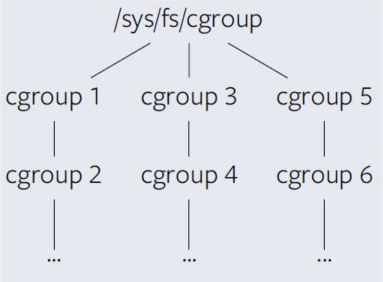




Process grouping, *Cont'd*

- Cgroup snippets, *Cont'd*
    - **cgroupv2**

      cgroupv2 has a **unified hierarchy**.
      Each cgroup can support multiple resource domains.
      By default controllers are disabled.



man7

Cgroups v2 subtree control

Each cgroup in the v2 hierarchy contains the following two files:

cgroup.controllers
This read-only file exposes a list of the controllers that are *available* in this cgroup. The contents of this file match the contents of the *cgroup.subtree_control* file in the parent cgroup.

cgroup.subtree_control
This is a list of controllers that are *active* (*enabled*) in the cgroup. The set of controllers in this file is a subset of the set in the *cgroup.controllers* of this cgroup. The set of active controllers is modified by writing strings to this file containing space-delimited controller names, each preceded by '+' (to enable a controller) or '-' (to disable a controller), as in the following example:

```
echo '+pids -memory' > x/y/cgroup.subtree_control
```

Process grouping, *Cont'd*

- Cgroup snippets, *Cont'd*
  - **cgroupv2**, *Cont'd*
    Basic example:
    Create **cgroup** and enable
    "io controller"

0-stage, we need first to un-mount **cgroupv1** that hold the resources due to backward compatibility.
Please check this link,
[unmount cgroup v1]()

```
karim@karim-Inspiron-5537:/sys/fs/cgroup$ ls
unified
karim@karim-Inspiron-5537:/sys/fs/cgroup$ cd unified/
karim@karim-Inspiron-5537:/sys/fs/cgroup/unified$ ls
cgroup.controllers      cgroup.procs             cgroup.threads         cpuset.mems.effective  io.cost.model  machine.slice    user.slice
cgroup.max.depth        cgroup.stat              cpu.pressure           cpu.stat               io.cost.qos    memory.pressure
cgroup.max.descendants  cgroup.subtree_control   cpuset.cpus.effective  init.scope             io.pressure    system.slice
karim@karim-Inspiron-5537:/sys/fs/cgroup/unified$ cat cgroup.controllers
cpuset cpu io hugetlb rdma
karim@karim-Inspiron-5537:/sys/fs/cgroup/unified$ sudo su
root@karim-Inspiron-5537:/sys/fs/cgroup/unified#
root@karim-Inspiron-5537:/sys/fs/cgroup/unified# mkdir my_cgv2
root@karim-Inspiron-5537:/sys/fs/cgroup/unified# ls my_cgv2/
cgroup.controllers  cgroup.max.depth        cgroup.stat             cgroup.type    cpuset.cpus.effective  cpuset.mems.effective  memory.pressure
cgroup.events       cgroup.max.descendants  cgroup.subtree_control  cpu.pressure   cpuset.cpus.partition  cpu.stat
cgroup.freeze       cgroup.procs            cgroup.threads          cpuset.cpus    cpuset.mems            io.pressure

root@karim-Inspiron-5537:/sys/fs/cgroup/unified# ls my_cgv2/
cgroup.controllers  cgroup.max.depth        cgroup.stat             cgroup.type    cpuset.cpus.effective  cpuset.mems.effective  memory.pressure
cgroup.events       cgroup.max.descendants  cgroup.subtree_control  cpu.pressure   cpuset.cpus.partition  cpu.stat
cgroup.freeze       cgroup.procs            cgroup.threads          cpuset.cpus    cpuset.mems            io.pressure
root@karim-Inspiron-5537:/sys/fs/cgroup/unified#
root@karim-Inspiron-5537:/sys/fs/cgroup/unified#
root@karim-Inspiron-5537:/sys/fs/cgroup/unified# echo '+io' > cgroup.subtree_control
root@karim-Inspiron-5537:/sys/fs/cgroup/unified# ls my_cgv2/
cgroup.controllers  cgroup.max.descendants  cgroup.threads  cpuset.cpus.effective  cpu.stat     io.weight
cgroup.events       cgroup.procs            cgroup.type     cpuset.cpus.partition  io.max       memory.pressure
cgroup.freeze       cgroup.stat             cpu.pressure    cpuset.mems            io.pressure
cgroup.max.depth    cgroup.subtree_control  cpuset.cpus     cpuset.mems.effective  io.stat
root@karim-Inspiron-5537:/sys/fs/cgroup/unified#
```

- Please check this link for more details
  [cgroup v2]()

Scheduler Entry point

- The main entry point into the process schedule is the function ***schedule()***, it finds the highest priority scheduler class with a runnable process and asks it what to run next.
- ***schedule()*** invokes ***pick_next_task()*** that goes through each scheduler class, starting with the highest priority, and selects the highest priority process in the highest priority class.
- This optimization chunk is a small hack to quickly select the next **CFS**-provided process because most systems run mostly normal processes.

```
/*
 * Optimization: we know that if all tasks are in the fair class we can
 * call that function directly, but only if the @prev task wasn't of a
 * higher scheduling class, because otherwise those lose the
 * opportunity to pull in more work from other CPUs.
 */
if (likely(prev->sched_class <= &fair_sched_class &&
           rq->nr_running == rq->cfs.h_nr_running)) {

    p = pick_next_task_fair(rq, prev, rf);
    if (unlikely(p == RETRY_TASK))
        goto restart;

    /* Assumes fair_sched_class->next == idle_sched_class */
    if (!p) {
        put_prev_task(rq, prev);
        p = pick_next_task_idle(rq);
    }

    return p;
}
```

/kernel/sched/core.c

```
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    ...
    next = pick_next_task(rq, prev, &rf);
    ...
}
```

```
/*
 * Pick up the highest-prio task:
 */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /*
     * Optimization: we know that if all tasks are in the fair class we can
     * call that function directly, but only if the @prev task wasn't of a
     * higher scheduling class, because otherwise those lose the
     * opportunity to pull in more work from other CPUs.
     */
    if (likely(prev->sched_class <= &fair_sched_class &&
               rq->nr_running == rq->cfs.h_nr_running)) {

        p = pick_next_task_fair(rq, prev, rf);
        if (unlikely(p == RETRY_TASK))
            goto restart;

        /* Assumes fair_sched_class->next == idle_sched_class */
        if (!p) {
            put_prev_task(rq, prev);
            p = pick_next_task_idle(rq);
        }

        return p;
    }

restart:
    put_prev_task_balance(rq, prev, rf);

    for_each_class(class) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
    }

    /* The idle class should always have a runnable task: */
    BUG();
}
```

/kernel/sched/core.c

Scheduler System Calls

- Most of the system calls are straightforward except the processor affinity system calls somehow!, so what's **affinity!!!**.
- Processor affinity, is the user may say, **"This task must remain on this *subset* of the available processors no matter what"**.
- This hard affinity is stored as a bitmask in the task's ***task_struct*** as ***cpus_mask.***
- Initially when a process is created, it inherits its parent's affinity mask, when a processor's affinity is changed, the kernel uses the **migration threads** to push the task onto a legal processor.

System Call	Description
nice()	Sets a process's nice value
sched_setscheduler()	Sets a process's scheduling policy
sched_getscheduler()	Gets a process's scheduling policy
sched_setparam()	Sets a process's real-time priority
sched_getparam()	Gets a process's real-time priority
sched_get_priority_max()	Gets the maximum real-time priority
sched_get_priority_min()	Gets the minimum real-time priority
sched_rr_get_interval()	Gets a process's timeslice value
sched_setaffinity()	Sets a process's processor affinity
sched_getaffinity()	Gets a process's processor affinity
sched_yield()	Temporarily yields the processor

/kernel/sched/core.c

```
/*
 * migration_cpu_stop - this will be executed by a highprio stopper thread
 * and performs thread migration by bumping thread off CPU then
 * 'pushing' onto another runqueue.
 */
static int migration_cpu_stop(void *data)
{
    ...
        if (task_on_rq_queued(p))
            rq = __migrate_task(rq, &rf, p, dest_cpu);
        else
            p->wake_cpu = dest_cpu;

    ...
}
```

We will keep this page as a template.

- A
- B
- C

Feel free to copy, pate and edit this page

- A
- B
- C