# Task Scheduling

Real-Time Operating Systems Programming (RTOSP)
Master in Critical Computing Systems Engineering (MCCSE)

2022/23

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

Instituto Superior de Engenharia do Porto

P.PORTO

## Disclaimer

**Material and Slides**

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

## Outline

## Linux scheduling framework

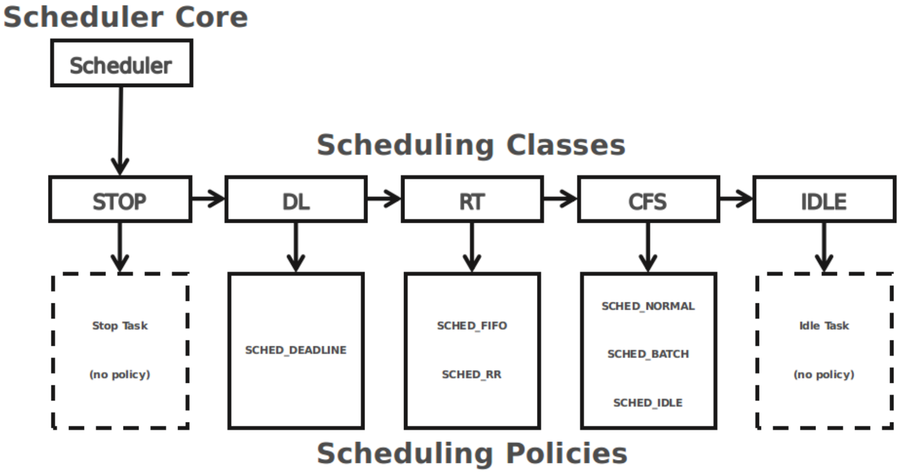
## Overview (I)

- The **scheduler** (or dispatcher) is the part of the kernel responsible, at run-time, **for allocating CPUs to tasks for execution** and the **scheduling classes are responsible for selecting those tasks**.
- The scheduling classes encapsulate scheduling policies.
- **These scheduling classes are hierarchically organized by priority** and the scheduler inquires each scheduling class in a decreasing priority order for a ready task.
- Linux has five main scheduling classes:
  - Stop
  - Deadline (DL)
  - Real-Time (RT)
  - Completely Fair Scheduling (CFS)
  - Idle

## Overview (II)

1. The scheduler first inquires the STOP scheduling class for a ready task.
2. If STOP scheduling class does not have any ready task, then it inquires the DL scheduling class.
3. If DL does not have any ready task, it proceeds by inquiring RT.
4. If RT does not have any ready task it proceeds inquiring CFS.
5. If CFS does not have ready task then it reaches the Idle scheduling class, used for the idle task.
6. Every processor has an idle task in its ready-queue that is executed whenever there is no other ready task.

## Overview (III)



## Overview (IV)

- STOP
  - Only available for SMP (`stop_machine()` is not used in UP).
  - Can **preempt everything and is preempted by nothing**.
  - **Mechanism to stop running everything else and run a specific function on CPU(s)**.
  - No scheduling policies.
  - One kernel thread per CPU: `migration/`*N*.
  - Used by task migration, CPU hotplug, RCU, ftrace, clockevents, etc.
- DL
  - **Highest priority tasks in the system**.
  - Scheduling policy: `SCHED_DEADLINE`.
  - Implemented with red-black tree (self balancing).
  - Used for periodic real time tasks, eg. Video encoding/decoding.

## Overview (V)

- RT
  - POSIX "real-time" tasks.
  - Task priorities: 0 - 99.
  - Scheduling policies for tasks at same priority:
    - `SCHED_FIFO`
    - `SCHED_RR`, 100ms timeslice by default.
  - Implemented with Linked lists.
  - Used for short latency sensitive tasks, eg. IRQ threads.
- CFS
  - Scheduling policies:
    - `SCHED_NORMAL`: Normal tasks
    - `SCHED_BATCH`: Batch (non-interactive) tasks
    - `SCHED_IDLE`: Low priority tasks
  - Implemented with red-black tree (self balancing).
  - Tracks Virtual runtime (`vruntime`) of tasks
  - Tasks with shortest `vruntime` runs first.
  - Priority is used to set task's weight, that affects `vruntime`.
    - Higher the weight, slower will `vruntime` increase.
    - Task's priority is calculated by 120 + nice (-20 to +19).

## Overview (VI)

- IDLE
  - **Lowest priority scheduling class**.
  - No scheduling policies.
  - One kernel thread (idle) per CPU: `swapper/`*N*.
  - Idle thread runs only when nothing else is runnable on a CPU.
  - Idle thread may take the CPU to low power state.

## Data structures (I)

- For **every active processes in the system**, Linux kernel create an instance of `struct task_struct` to manage them.
  - The kernel must know, for instance, the process's priority, whether it is running on a CPU or blocked on an event, what address space has been assigned to it, which files it is allowed to address, and so on.
    - `struct task_struct` is composed by data fields for all this issues
- Each **CPU has its own runqueue**, as an instance of `struct rq`
  - A `struct rq` is a container for all processes in a `TASK_RUNNING` state.
  - Further, it contains many other fields for managing the CPU.

## Data structures (II)

- The **Linux scheduler is modular, enabling different algorithms/policies to schedule different types of tasks**.
- An algorithm's implementation is wrapped in a so called **scheduling class**.
- A scheduling class offers an interface to the main scheduler skeleton which it can use to handle tasks according to the implemented algorithm.
- A scheduling class is an instance of `struct sched_class` data structure.

## `struct task_struct` data structure

- `/include/linux/sched.h`

```
struct task_struct {
  ...
  const struct sched_class *sched_class;
  ...
  struct sched_entity  se;
  struct sched_rt_entity  rt;
    ...
  struct sched_dl_entity  dl;
  ...
};
```

```
struct sched_entity {
  ...
  struct rb_node   run_node;
  ...
#endif
```

```
struct sched_rt_entity {
  struct list_head  run_list;
  ...
};
```

```
struct sched_dl_entity {
  struct rb_node   rb_node;
  ...
}
```

- This data structure incorporates data structures per scheduling classes: `struct sched_entity`, `struct sched_rt_entity` and `struct sched_dl_entity` for CFS, RT and DL scheduling classes, respectively.

## `struct rq` data structure

- It keeps track of all runnable tasks assigned to a particular CPU:
  - a lock to synchronize scheduling operations for this CPU
  - Pointers to the currently running (`curr`), stop (`stop`) and the idle (`idle`) tasks.
  - Actually, runqueue incorporates sub-runqueues per scheduling classes: `struct dl_rq`, `struct cfs_rq` and `struct rt_rq`, for DL, CFS and RT scheduling classes, respectively.
- `/kernel/sched/sched.h`

```
struct rq {
  raw_spinlock_t lock;
  ...
  unsigned int nr_running;
  ...
  struct cfs_rq cfs;
  struct rt_rq rt;
  struct dl_rq dl;
  ...
  struct task_struct *curr;
  struct task_struct *idle;
  struct task_struct *stop;
  ...
};
```

## struct sched_class data structure (I)

- /kernel/sched/sched.h

```
struct sched_class {
  ...
 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);
...
 void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);
  ...
 struct task_struct * (*pick_next_task)(struct rq *rq,
             struct task_struct *prev,
             struct rq_flags *rf);
 void (*put_prev_task)(struct rq *rq, struct task_struct *p);

#ifdef CONFIG_SMP
 int  (*select_task_rq)(struct task_struct *p, int task_cpu, int sd_flag, int flags);
 void (*migrate_task_rq)(struct task_struct *p, int new_cpu);
  ...
#endif
 void (*set_curr_task)(struct rq *rq);
 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);
...
};
```

## `struct sched_class` data structure (II)

- `enqueue_task`: Called when a task enters a runnable state.
- `dequeue_task`: Called when a task is no longer runnable.
- `check_preempt_curr`: This function checks if a task that entered the runnable state should preempt the currently running task.
- `pick_next_task`: This function chooses the most appropriate task eligible to run next.
- `task_tick`: This function is mostly called from time tick functions;

## `struct sched_class` data structure (III)

- All existing scheduling classes in the kernel are in an array which is ordered by the priority of the scheduling class. The highest priority is in the first position and the lowest in the last position.

```
#define SCHED_DATA    \
STRUCT_ALIGN();    \
__sched_class_highest = .;  \
*(__stop_sched_class)   \
*(__dl_sched_class)   \
*(__rt_sched_class)   \
*(__fair_sched_class)   \
*(__idle_sched_class)   \
__sched_class_lowest = .;
```

- The scheduling class array iteration begins always in the first position of the array.

```
#define for_class_range(class, _from, _to) \
 for (class = (_from); class < (_to); class++)

#define for_each_class(class) \
 for_class_range(class, __sched_class_highest, __sched_class_lowest)
```

## The Scheduler Entry Point

- The main entry point into the process **scheduler is the function `__schedule`**.
- This is the function that the rest of the kernel uses to invoke the process scheduler, deciding which process to run and then running it.
- Its main goal is to find the next task to be run.
- It has two `struct task_struct` pointers, `prev` and `next` that are set with the currently executing task (which will relinquish CPU) and the next task to be executed (which will be assigned to CPU), respectively.

## __schedule function

```
static void __sched notrace __schedule(bool preempt)
{
 struct task_struct *prev, *next;
 struct rq *rq;
 int cpu;
 cpu = smp_processor_id();
 rq = cpu_rq(cpu);
 prev = rq->curr;
 local_irq_disable();
 rq_lock(rq, &rf);
 update_rq_clock(rq);
 if (!preempt && prev->state) {
  if (signal_pending_state(prev->state, prev)) {
   prev->state = TASK_RUNNING;
  } else {
   deactivate_task(rq, prev, DEQUEUE_SLEEP | DEQUEUE_NOCLOCK);
  }
 }
 next = pick_next_task(rq, prev, &rf);
 clear_tsk_need_resched(prev);
 clear_preempt_need_resched();

 if (likely(prev != next)) {
  rq->curr = next;
  rq = context_switch(rq, prev, next, &rf);
 }
```

## `__schedule` function: main points (I)

- Since the Linux kernel is pre-emptive, it can happen that a task executing code in kernel space is involuntarily pre-empted by a higher priority task.

1. It assigns the current executing task to `prev` pointer (`prev = rq->curr`)
2. It disables the interrupts by calling `local_irq_disable`.
3. It locks the current CPU's runqueue (`rq_lock`)
4. It examines the state of the currently executing task, `prev`.
    - If it is not runnable and has not been pre-empted in kernel mode, then it should be removed from the runqueue.
    - To remove a task from the runqueue, `deactivate_task` is called which internally calls the `dequeue_task` hook of the task's scheduling class.
    - However, if it has nonblocked pending signals, its state is set to `TASK_RUNNING` and it is left in the runqueue.
    - This means `prev` gets another chance to be selected for execution.

## `__schedule` function: main points (II)

5. Next, it is time to pick the next task to be assigned to the CPU calling `pick_next_task` function.
6. After that, it checks if `pick_next_task` found a new task or if it picked the same task again that was running before.
   - If the latter is the case, no task switch is performed and the current task just keeps running.
7. If a new task is found, which is the more likely case, the actual task switch is executed by calling `context_switch`.
   - Internally, `context_switch` switches to the new task's memory map and swaps register state and stack.
8. To finish up, the runqueue is unlocked and pre-emption is re-enabled.

## Calling the Scheduler

- `__schedule` function is invoked when:
  - Whenever a task is mark to be preempted (**preemption is required**).
  - At regular times, at **timer tick expiration**.
  - Currently running task goes to **sleep** or **finishes**.
  - **Sleeping task wakes up**
  - Newly **forked tasks**.

## Requiring preemption

- `resched_curr` function **marks the currently executing task to be preempted.**
  - This sets the `TIF_NEED_RESCHED` flag in the task structure, and the scheduler core will initiate a rescheduling at the next opportune moment.
- Example:
  - An interrupt occurred.
  - Interrupt handler is invoked to manage the interrupt request.
  - If the `resched_curr` is invoked in the interrupt handle function.
  - When it proceeds to the IRQ Exit path, it checks:
    - If `TIF_NEED_RESCHED` flag is set , it calls `__schedule` function.

## `scheduler_tick` function (I)

- The function `scheduler_tick` is called regularly by a timer interrupt, called **tick**.
  - This function **gets called by the timer code**, with `HZ` frequency.
- Its purpose is to update runqueue clock, CPU load and runtime counters of the currently running task.
- It calls the scheduling class hook `task_tick` of the currently executing task task update for the corresponding class.
  - At this point it can mark the current executing task to be preempted, by calling `resched_curr` function.
- Load balancing is invoked if SMP is configured.

## scheduler_tick function (II)

- Defined in kernel/sched/core.c

```
void scheduler_tick(void){
  int cpu = smp_processor_id();
  struct rq *rq = cpu_rq(cpu);
  struct task_struct *curr = rq->curr;
  struct rq_flags rf;

  sched_clock_tick();

  rq_lock(rq, &rf);

  update_rq_clock(rq);
  curr->sched_class->task_tick(rq, curr, 0);
  calc_global_load_tick(rq);
  psi_task_tick(rq);

  rq_unlock(rq, &rf);

  perf_event_task_tick();

  ...
}
```

## `wake_up_process` function

- Defined in `kernel/sched/core.c`

```
...
/**
 * wake_up_process - Wake up a specific process
 * @p: The process to be woken up.
 *
 * Attempt to wake up the nominated process and move it to the set of runnable
 * processes.
 *
 * Return: 1 if the process was woken up, 0 if it was already running.
 *
 * This function executes a full memory barrier before accessing the task state.
 */
int wake_up_process(struct task_struct *p)
{
 return try_to_wake_up(p, TASK_NORMAL, 0);
}
EXPORT_SYMBOL(wake_up_process);
...
```