Operating Systems

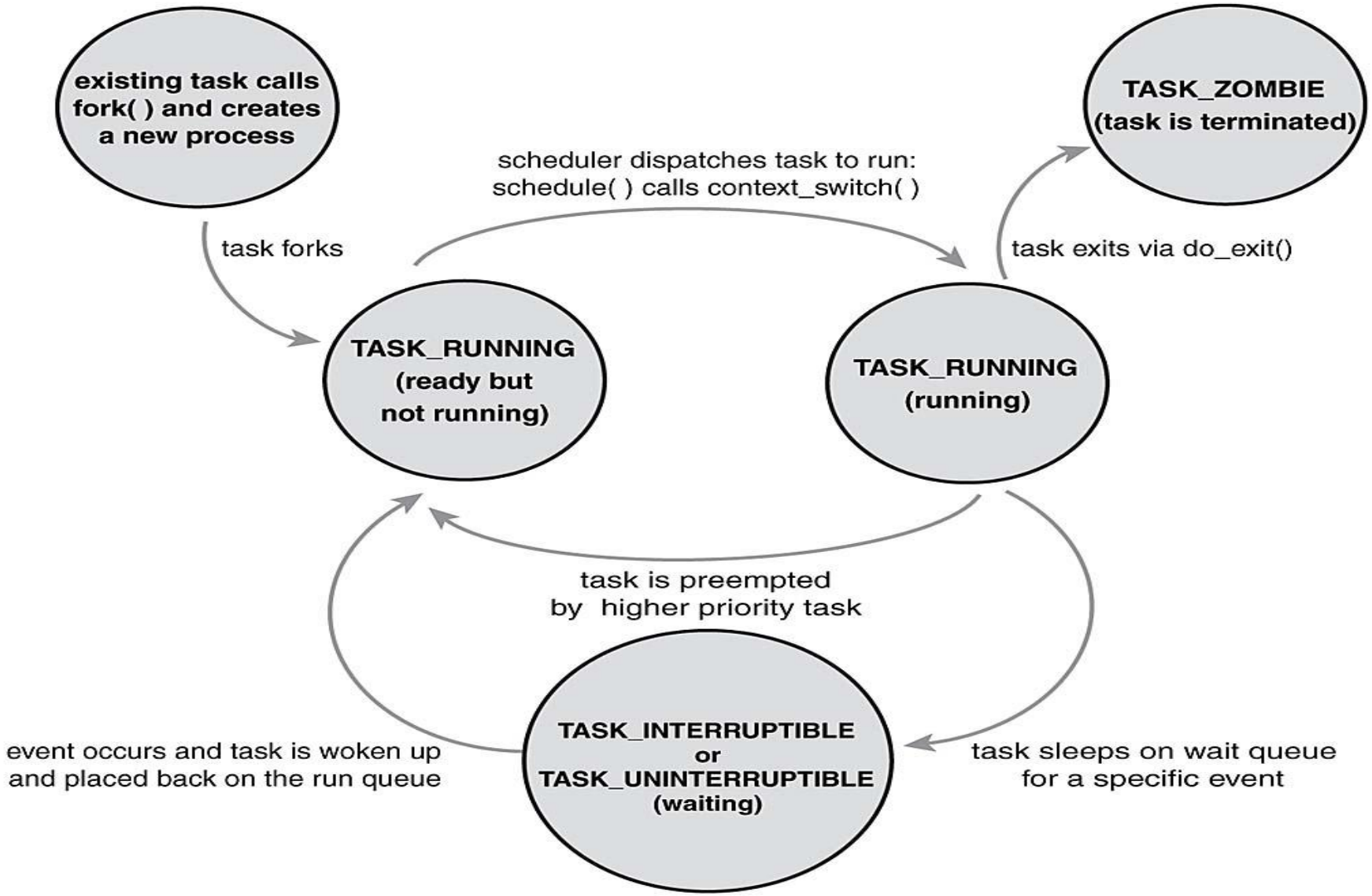
CSE 231

Instructor: Sambuddho Chakravarty

(Semester: Monsoon 2020)

Week 5: Oct 12 – Oct 15,2020

Linux Task States



Past Linux Schedulers: v1.2 & v2.2

v1.2: circular queue with round-robin policy.

- Simple and minimal.
- Not focused on massive architectures.

v2.2: introducing scheduling classes (real-time, non-preemptive, non-real-time).

- Support SMP (Symmetric Multi-Processing)

Past Schedulers : 2.4

- 2.4: O(N) scheduler.
  - Epochs → slices: when blocked before the slice ends, half of the remaining slice is added in the next epoch (as an increase in priority).
  - Iterate over tasks with a goodness function.
  - Simple, inefficient.
  - Lacked scalability.
  - Weak for real-time systems.

Past Schedulers: 2.4

- Static priority:
  - The maximum size of the time slice a process should be allowed before being forced to allow other processes to compete for the CPU.
- Dynamic priority:
  - The amount of time remaining in this time slice; declines with time as long as the process has the CPU.

Past Schedulers: O(1)

- An independent runqueue for *each* CPU
  - Active array
  - Expired array
- Tasks are indexed according to their priority [0,140]
  - Real-time [0, 99]
  - Nice value (others) [100, 140]
- When the active array is empty, the arrays are exchanged.

Past Schedulers: O(1)

- Real-time tasks are assigned static priorities.
- All others have dynamic priorities:
  - *nice* value ± 5
  - Depends on the tasks interactivity: more interactivity means longer blockage.
- Dynamic priorities are recalculated when tasks are moved to the expired array.

CFS Overview

- Since 2.6.23 (Circa 2007)
- Maintain balance (fairness) in providing CPU time to tasks.
- When the time for tasks is out of balance, then those out-of-balance tasks should be given time to execute.
- To determine the balance, the amount of time provided to a given task is maintained in the virtual runtime (amount of time a task has been permitted access to the CPU).

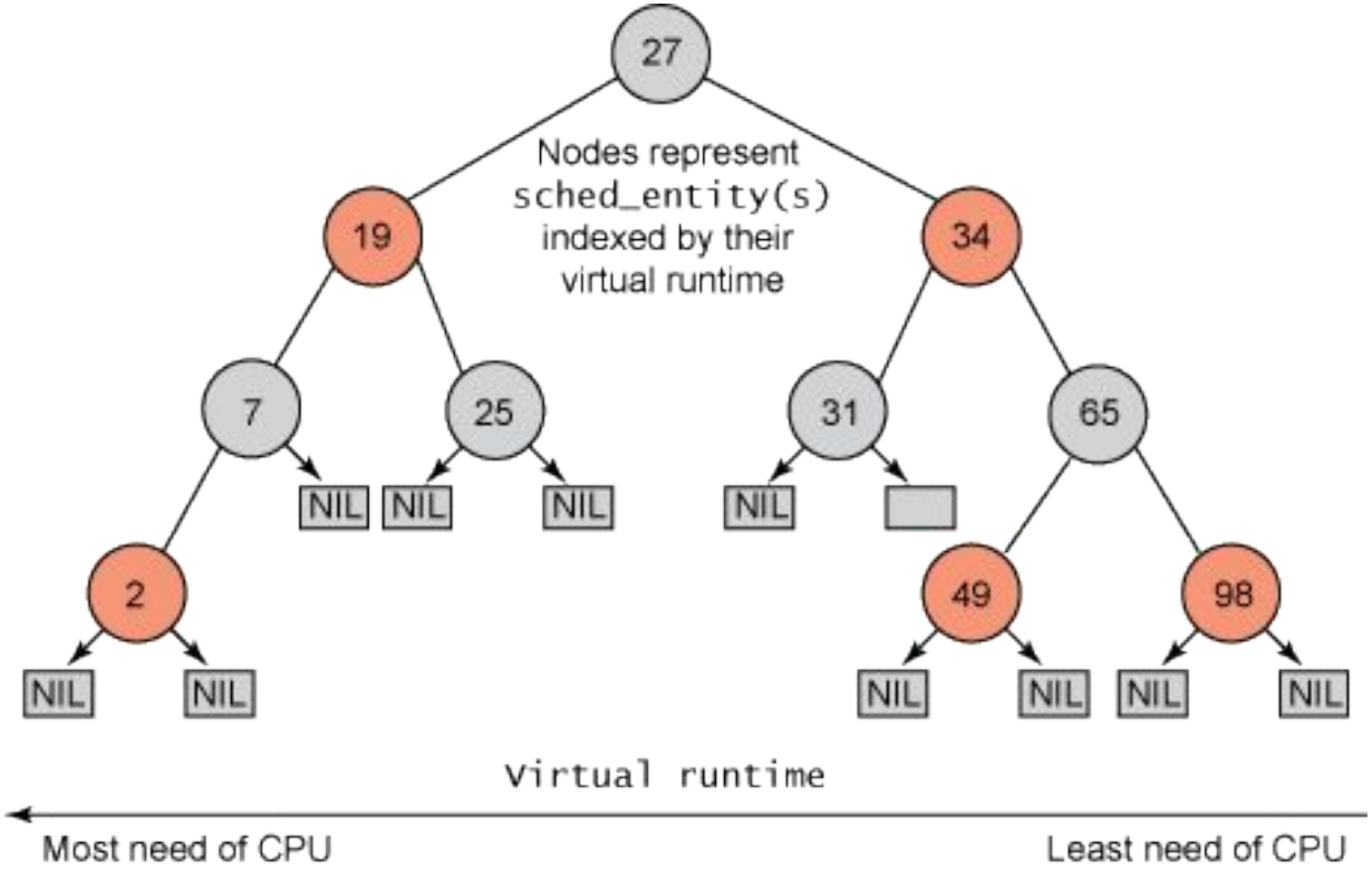
CFS Overview

- The smaller a task's virtual runtime, the higher its need for the processor.
- The CFS also includes the concept of sleeper fairness to ensure that tasks that are not currently runnable receive a comparable share of the processor when they eventually need it.
- Basic philosophy: CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.
  - If two processes are running each run at 50% of CPU resources.
  - A ``fairer'' form of weighted/prioritized RR, with average case faster than an linear runqueue (where the running time is O(N) complexity).

CFS Structure

- Tasks are maintained in a time-ordered red-black tree for *each* CPU, instead of a run queue.
  - self-balancing
  - operations on the tree occur in O(log n) time.
- Tasks with the gravest need for the processor (lowest virtual runtime) are stored toward the left side of the tree, and tasks with the least need of the processor (highest virtual runtimes) are stored toward the right side of the tree.
- The scheduler picks the left-most node of the red-black tree to schedule next to maintain fairness.

CFS Overview

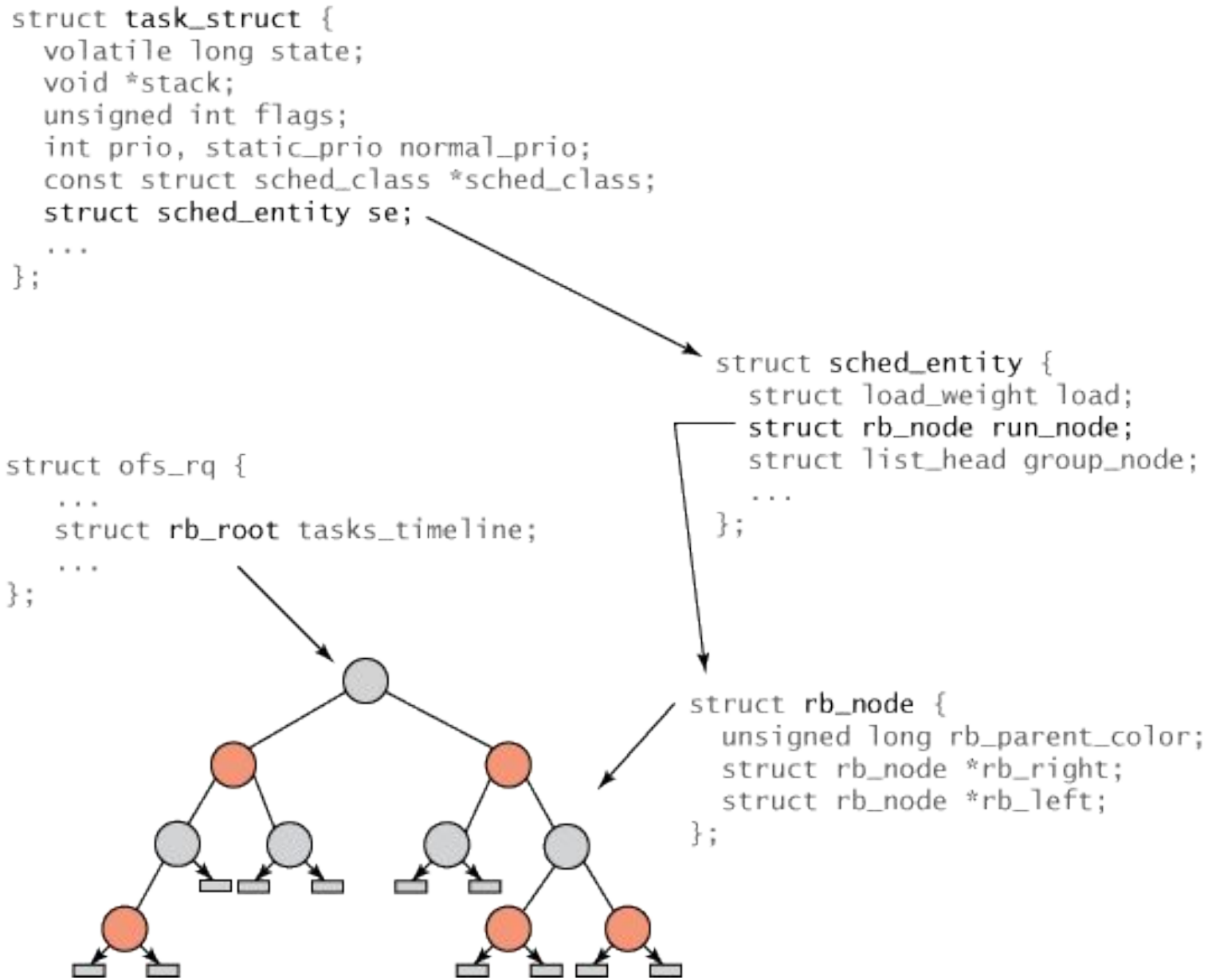


CFS Structure

- The task accounts for its time with the CPU by adding its execution time to the virtual runtime and is then inserted back into the tree if runnable.
- The contents of the tree migrate from the right to the left to maintain fairness.

CFS Internals

- All tasks are represented by a structure called `task_struct`.
- This structure fully describes the task: current state, stack, process flags, priority (static and dynamic)...
- ./linux/include/linux/sched.h.
- Not all tasks are runnable → No CFS-related fields in `task_struct`.
  - `sched_entity`
- Each node in the tree is represented by an rb_node, which contains the child references and the color of the parent.


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;
};

Scheduling

1. **`schedule()`** (`./kernel/sched.c`) preempts the currently running task – unless it preempts itself with **`yield()`**.
   - CFS has no real notion of time slices for preemption, because the preemption time is variable.
2. The currently running task (now preempted) is returned to the red-black tree through a call to **`put_prev_task`** (via the scheduling class).

Scheduling

3. When the schedule function comes to identifying the next task to schedule, it calls **`pick_next_task()`**, which calls the CFS scheduler through **`pick_next_task_fair()`** (`./kernel/sched_fair.c`).
4. It picks the left-most task from the red-black tree and returns the associated **`sched_entity`**.
   - With this reference, a simple call to **`task_of()`** identifies the **`task_struct`** reference returned.
5. The generic scheduler finally provides the processor to this task.

Priorities in CFS

- CFS doesn't use priorities directly.
- Decay factor for the time a task is permitted to execute.
  - Lower-priority tasks have higher factors of decay.
  - The time a task is permitted to execute dissipates more quickly for a lower-priority task than for a higher-priority task
  - Avoid maintaining run queues per priority.

Priorities in CFS

- Unlike previous schedulers CFS does not rely on actual timeslices or clock ticks.
- There is only one central tunable (you have to switch on CONFIG_SCHED_DEBUG):

  /proc/sys/kernel/sched_min_granularity_ns

  Can be changed between ``desktop'' (i.e. low latencies) to ``server''(i.e. batching).

  SCHED_NORMAL (high interactive desktop) and SCHED_BATCH (server batch process)

Is CFS ``nice'' ?

- Linux uses the ``nice'' factor to determine process priority – values:

  -20 (highest) □ +19 (lowest) – "More the value of nice, more nice is the process to other processes". Default: 0.

- Nice translated to weight. Weight determines what proportion of time the CPU time is granted to the process.

- Weight = 1024 / (1.25)^(nice).

  weight of Run Queue of all tasks = sum of weight of all tasks in the queue.

  proportion of CPU allotted = weight(process)/weight(RunQueue)

Is CFS ``nice'' ?

nice(A) = 0 => weight(A) = 1024
nice(B) = 0 => weight(B) = 1024.

Assuming single CPU and no other process running, run queue of CPU has two tasks.

weight(RunQueue) = weight(A) + weight(B) = 2048.

proportion of CPU allotted = wieght(process)/weight(RunQueue)

cpuPercentage(A) = weight(A)/weight(RunQueue) = 1024/2048 = 50%

cpuPercentage(B) = 50%

Is CFS ``nice'' ?

weight(A) with nice 1 is 820

weight(B) with nice 0 is 1024

weight(RunQueue) = weight(A) + weight(B) = 1844.

cpuPercentage(A) = weight(A)/weight(RunQueue) = 820/1844 = ~45%
cpuPercentage(B) = weight(B)/weight(RunQueue) = 1024/1844 = ~55%

Vruntime vs Nice – How nice value influences vruntimes

- By default the CFS scheduler selects the process with the least elapsed vruntime – viz. the leftmost child of the red-black tree.
- Addition to the weight there is a weight factor.

Weight factor (wf) = Weight of the process with nice value 0 / weight of current task.

If nice == 0 ⇒ wf == 1: vruntime is same as actual time spent by CPU executing the task.

If nice < 0 ⇒ wf < 1: vruntime is less than the real time spent; vruntime updated no as much (grows slowly)

If nice > 0 ⇒ wf > 1: vruntime is more than the real time spent; vruntime updated more often (grows more frequently)

CFS Group Scheduling

- Since 2.6.24
- Bring fairness to scheduling in the face of tasks that spawn many other tasks (e.g. HTTP server).
- Instead of all tasks being treated fairly, the spawned tasks with their parent share their virtual runtimes across the group (in a hierarchy).
  - Other single tasks maintain their own independent virtual runtimes.
  - Single tasks receive roughly the same scheduling time as the group.
- There's a `/proc` interface to manage the process hierarchies, giving full control over how groups are formed.
  - Fairness can be assigned across users, processes, or a variation of each.

CFS Scheduling Classes

- Each task belongs to a scheduling class, which determines how a task will be scheduled.
- A scheduling class defines a common set of functions (via **`sched_class`**) that define the behavior of the scheduler.
- For example, each scheduler provides a way to add a task to be scheduled, pull the next task to be run, yield to the scheduler, and so on.

CFS Scheduling Classes

- sched/fair.c implements the CFS scheduler.

- sched/rt.c implements SCHED_FIFO and SCHED_RR semantics.

Scheduling classes are implemented through the sched_class structure, which contains hooks to functions that must be called whenever scheduling event occurs.

CFS Scheduling Classes

- enqueue_task(...)

Called when a task enters a runnable state. It puts the scheduling entity (task) into the red-black tree and increments the nr_running variable.

- dequeue_task(...)

When a task is no longer runnable, this function is called to keep the corresponding scheduling entity out of the red-black tree. It decrements the nr_running variable.

- yield_task(...)

This function is basically just a dequeue followed by an enqueue, unless the compat_yield sysctl is turned *on; in that case, it places the scheduling entity* at the right-most end of the red-black tree.

- 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.