Logistics

- **HW4 deadline: 3/29**
- Mon 3/16 **NO CLASS**
- No TA support for HW4 during spring break

Linux Scheduler Evolution (30 years)

O(N) → O(1) → CFS -> EEVDF

O(N) Scheduling Algorithm [Linux 2.4]

- The scheduler used a linear time algorithm that scanned the entire list of runnable processes to select the next task utilizing a goodness function.
- Global Run Queue: All CPUs typically shared a single run queue, which caused contention as the number of processors increased.
- Preemptive Time-Slicing: It was a preemptive system, where a clock interrupt (measured in "jiffies") determined the next action and allowed for time-sliced multitasking.
- Epochs → slices: when blocked before the slice ends, half of the remaining slice is added in the next epoch (as an increase in priority).

- Simple, inefficient.
- Lacked scalability.
- Weak for real-time systems.

- Introducing scheduling classes (real-time, non-preemptive, non-real-time).

Scheduling - Policy and Priority

- Depending on the chosen task policy and associated rules the scheduler decides which task is swapped out and which task is processed next. The Linux kernel implements several scheduling policies. They are divided into non real-time and real-time policies. The scheduling policies are already implemented in mainline Linux.

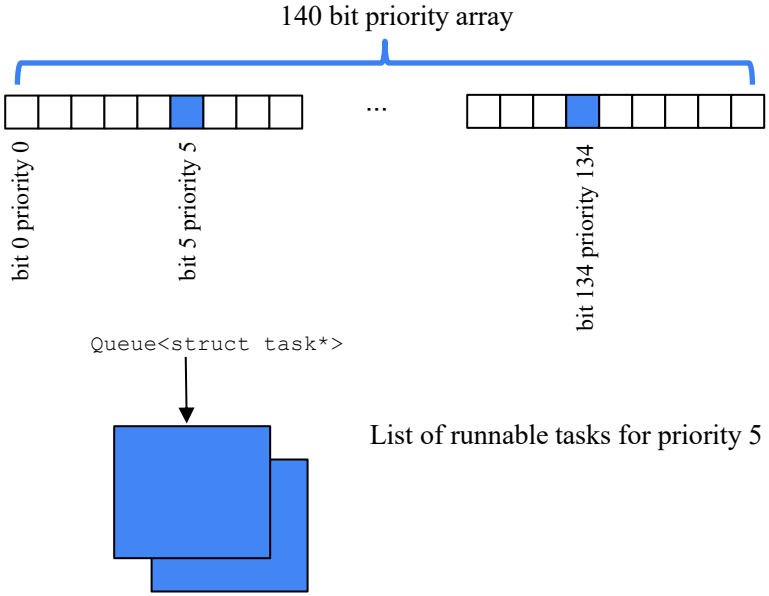
- ***Non real-time policies*:**
  - **SCHED_OTHER** – every task gets a so called 'nice value'. It is a value between -20 for the highest nice value and 19 for the lowest nice value. The average value of execution time of the task depends on the associated nice value.
  - **SCHED_BATCH** – is derived from SCHED_OTHER and is optimized for throughput.
  - **SCHED_IDLE** – is also derived from SCHED_OTHER, but it has nice values weaker than 19.

- ***Real-time policies:***
  - **SCHED_FIFO** – tasks have a priority between 1 (low) and 99 (high). A task running under this policy is scheduled until it finishes or a higher prioritized task preempts it.
  - **SCHED_RR** – is derived from SCHED_FIFO. The difference to SCHED_FIFO is that a task runs for the duration of a defined time slice (if it is not preempted by a higher prioritized task). It can be interrupted by a task with the same priority once the time slice is used up. The time slice definition is exported in procfs (/proc/sys/kernel/sched_rr_timeslice_ms).
  - **SCHED_DEADLINE** – implements the Global Earliest Deadline First (GEDF) algorithm. Tasks scheduled under this policy can preempt any task scheduled with SCHED_FIFO or SCHED_RR.

O(1) implementation (2.6.0)

- 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]
- 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 task's dynamic priority decays to (-1) are moved to the expired array.
- When the active array is empty, the arrays are exchanged.

- Main issue is the identification of interactive designation via tooooo many heuristics



- Maintain bitmap of state of the queues (in three 64-bit words)
- Identify highest prio queue that is not empty:
  - Use ffz() or ffnz() to find first bit that is set/notset
    supported by hardware instructions.
  - Only needs 3 64-bits to be examined to return correct priority (queue)
- Then dequeue_front from that queue.

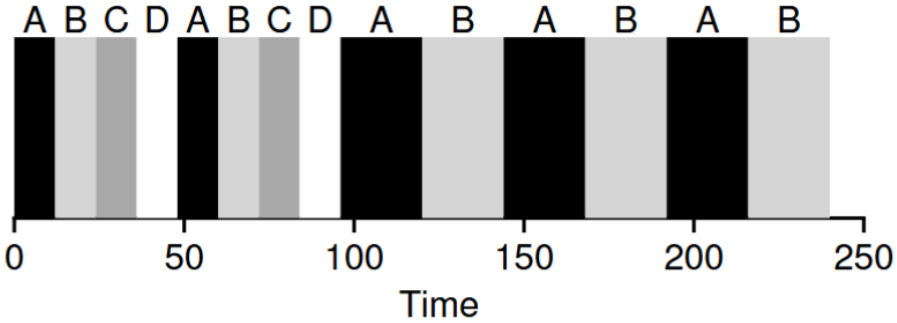
CFS Overview (2.6.23)

- 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).

- 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.

CFS Overview - cont

- CFS uses a dynamic time slice based on an EPOCH definition
- The EPOCH can be controlled via `sched_latency` config parameter (e.g. 48msec)
- Let N be size of RunQ: Each task in the RunQ gets 1/N of that EPOCH.
- If N is large per task time slice is very small and frequent context switch overhead can be a problem → config parameter `min_granularity` (e.g. 6msec)



Example of CFS scheduling with
- sched_latency = 48msec
- 4 tasks (A-D) till 96msec and C-D start sleeping after 95 msec

CFS and priorities and accounting

- vruntime is weighted based on a task's nice value

```
static const int 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,
};
```

- Calculation timeslice for task_k

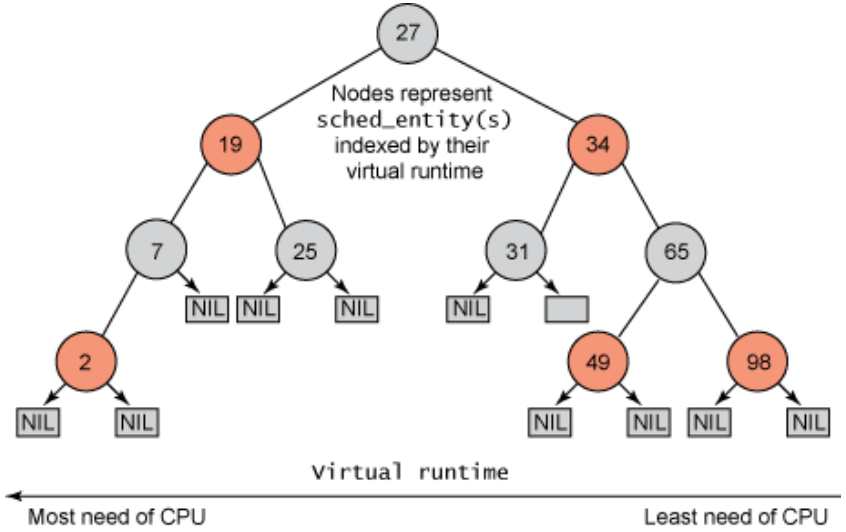
$$\text{time\_slice}_k = \frac{\text{weight}_k}{\sum_{i=0}^{n-1} \text{weight}_i} \cdot \text{sched\_latency}$$

- Accounting for vruntime scaled inversely to the weight

$$\text{vruntime}_i = \text{vruntime}_i + \frac{\text{weight}_0}{\text{weight}_i} \cdot \text{runtime}_i$$

CFS Structure

- Selecting the task with the minimum vruntime must be efficient
- 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.
- After preemption vruntime updated and reinserted
- Tasks that sleep are NOT in RB-tree

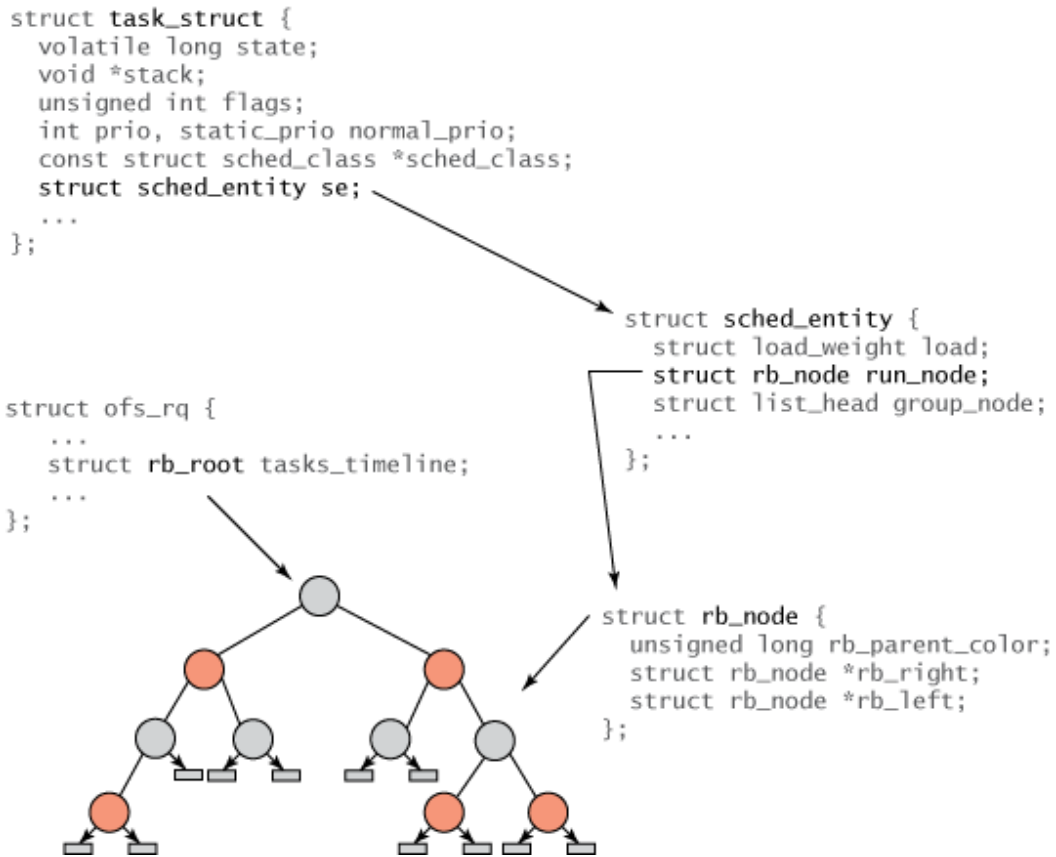


CFS Internals

- 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.



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 Domains

- Scheduling domains allow you to group one or more processors hierarchically for purposes load balancing and segregation.
- One or more processors can share scheduling policies (and load balance between them) or implement independent scheduling policies to intentionally segregate tasks.

Load Balancing in CFS

Goal: Equalize load across cores

What is load? The amount of work on all cores of the machine.
This is different from evening out the number of threads.

Example: if a user runs 1 CPU-intensive task and 10 tasks that mostly sleep, CFS might schedule the 10 mostly sleeping tasks on a single core.

How? Work stealing periodically from other cores (default every 4msec)

Can steal multiple tasks at a time to balance load quickly.

Load Balancing in CFS

Goal: Equalize load across cores

Goal 2: Maximize locality

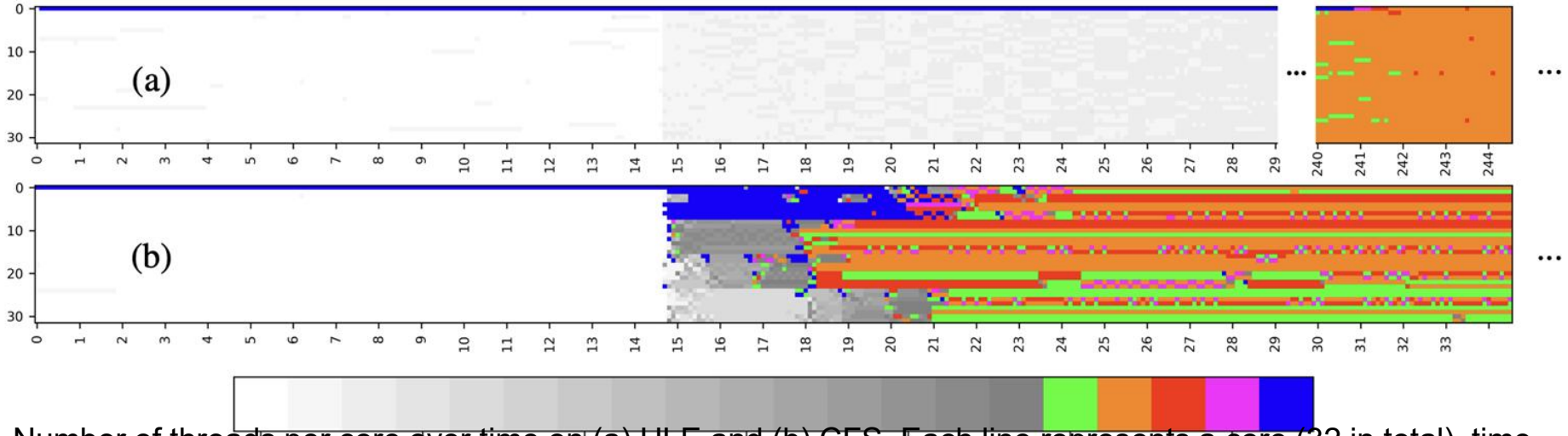
Wake-up/Creation:

- 1-to-1: Schedule the woken-up task nearby
- 1-to-many: Spread the tasks

Stealing:

- try to steal work more frequently from cores that are “close” to them than from cores that are “remote”
- hierarchical load balancing

Load Balancing in Practice



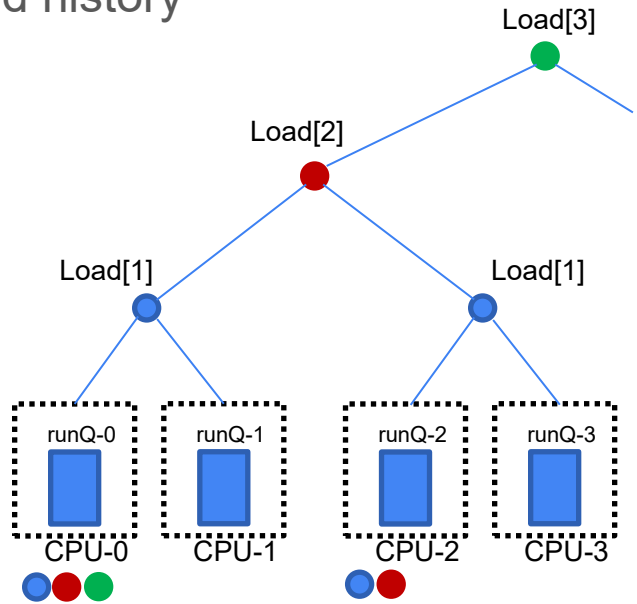
Number of threads per core over time on (a) ULE and (b) CFS. Each line represents a core (32 in total), time passes on the x-axis (in seconds), and colors represent the numbers of threads on the core. Thread counts below 15 are represented in shades of grey. Threads are pinned on core 0 for the first 14.5 seconds of the execution

(a) Slow "perfect" load balancing

(b) CFS

Load Balancing (LB)

- Occasionally or when no process is runnable, scheduler[i] looks to steal work elsewhere
- Each scheduler maintains a load average and history to determine stability of its load
- LB typically done in a hierarchy to scale
- Frequency of neighbor check:
  - Level in hierarchy ~ cost to migrate
  - Make “small” changes by pulling work from other cpu



CFS no more → EEVDF (Linux 6.6)

Address following CFS issues:
- Lack of Latency Guarantees, Sleeper Fairness,
  too many heuristics, too much tuning

Earliest Eligible Virtual Deadline First became the default scheduler in Linux 6.6

Fairness
Process lag = the difference between the CPU time a task *should* have received (ideal, fair share) and the time it *actually* received.

Weight based on nice value

A: vruntime=10 → lag = -10
B: vruntime=30 → lag = 10

CFS no more → EEVDF (Linux 6.6)

Interactivity
CFS uses a static minimum time slice

EEVDF time slice = base time slice / weight (weight depends on nice value)

Deadline = vruntime + time slice + lag

Assume Tasks A, B with same vruntime and lag and Wa > Wb

- Task A: Deadline = vruntime + lag + short time slice (due to high weight)
- Task B: Deadline = vruntime + lag + longer time slice (due to low weight)

EEVDF picks Task A to run