Scheduling in Linux

Logistics

1. **HW4 deadline: 3/23** (right after spring break)
2. No TA support for HW4 during spring break
3. This Thursday 3/6 Midterm Review (solve together past midterms)
4. Next Tuesday 3/11 **NO CLASS**
5. Next Thursday 3/13 **In-class Midterm**

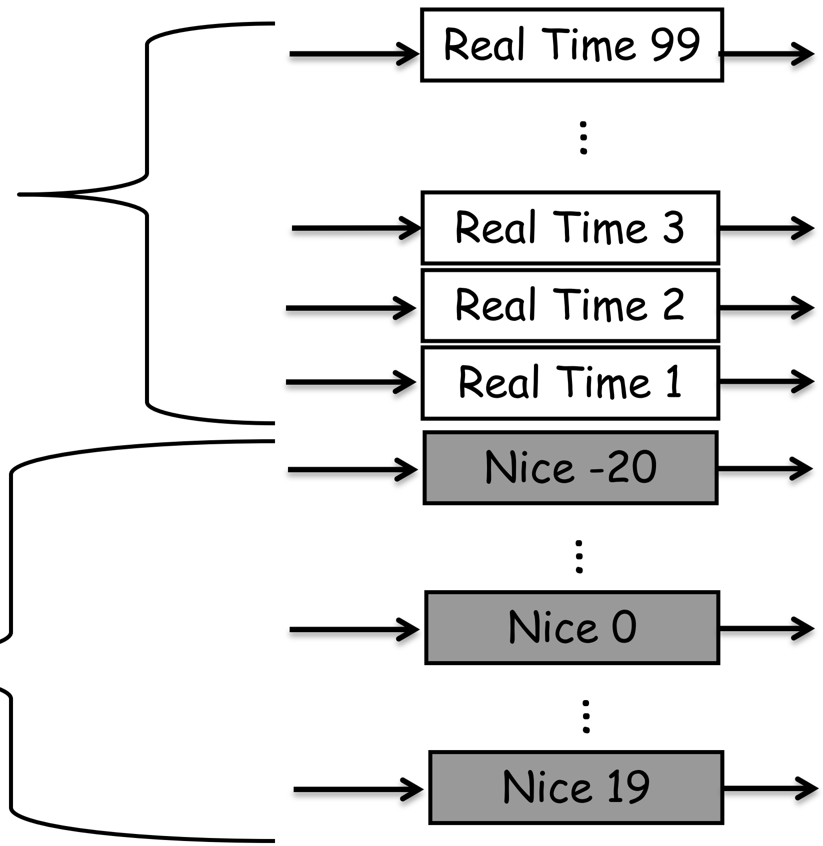
Real-time scheduling

- Hard real-time
  - complete critical task within guaranteed time period
- Soft real-time
  - critical processes have priority over others

- Linux supports soft real-time

Linux: multi-level queue with priorities

- Soft real-time scheduling policies
  - SCHED_FIFO (FCFS)
  - SCHED_RR (round robin)
  - Priority over normal tasks
  - 100 static priority levels (1..99)
- Normal scheduling policies
  - SCHED_NORMAL: standard
    - SCHED_OTHER in POSIX
  - SCHED_BATCH: CPU bound
  - SCHED_IDLE: lower priority
  - Static priority is 0
    - 40 dynamic priority
    - "Nice" values
- sched_setscheduler(), nice()
- See "man 7 sched" for detailed overview

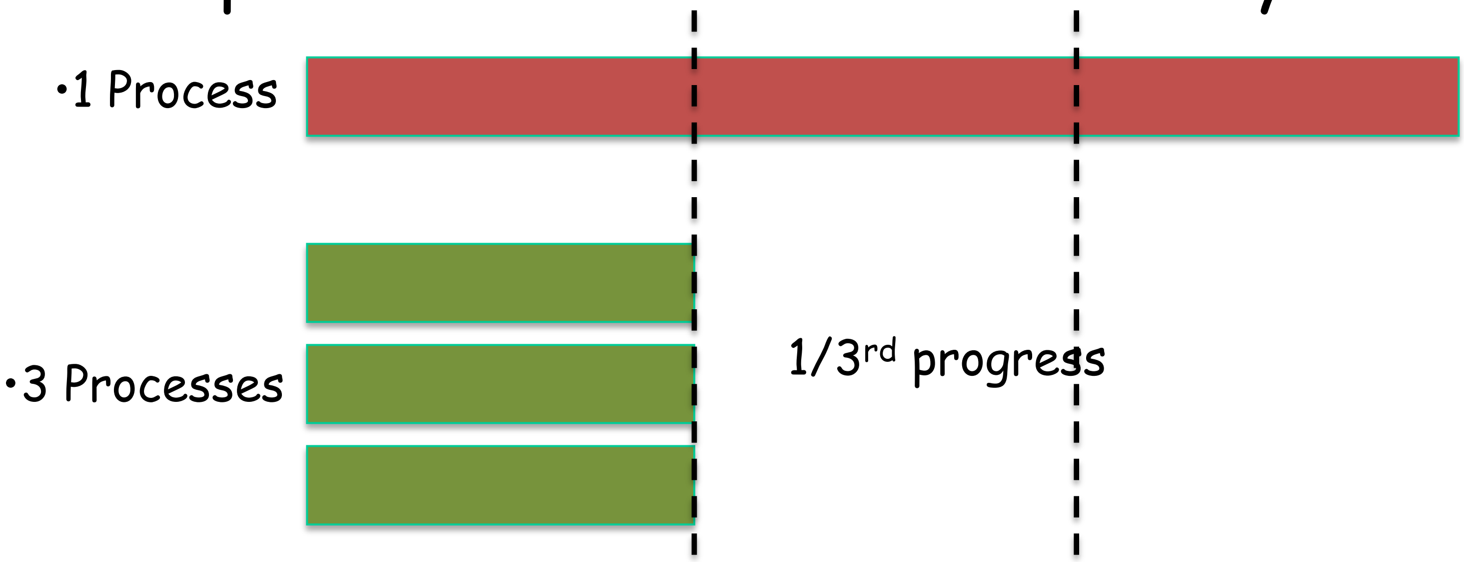


Linux scheduler history

- O(N) scheduler up to 2.4
  - Simple: global run queue
  - Poor performance on multiprocessor and large N
- O(1) scheduler in 2.5 & 2.6
  - Good performance: per-CPU run queue
  - Complex and error prone logic to boost interactivity
  - No fairness guarantee
- Completely Fair Scheduler (CFS) in 2.6 and later
  - Currently default scheduler for SCHED_NORMAL
  - Processes get fair share of CPU
  - Naturally boosts interactivity
- Alternative schedulers: BFS, MuQSS, PDS, BMQ, TT, etc.
  - https://wiki.archlinux.org/title/improving_performance#Alternative_CPU_schedulers

Ideal fair scheduling

- Infinitesimally small time slice
- n processes: each runs uniformly at 1/n^th rate



- Various approximations of the ideal
  - Lottery scheduling
  - Stride scheduling
  - Linux CFS

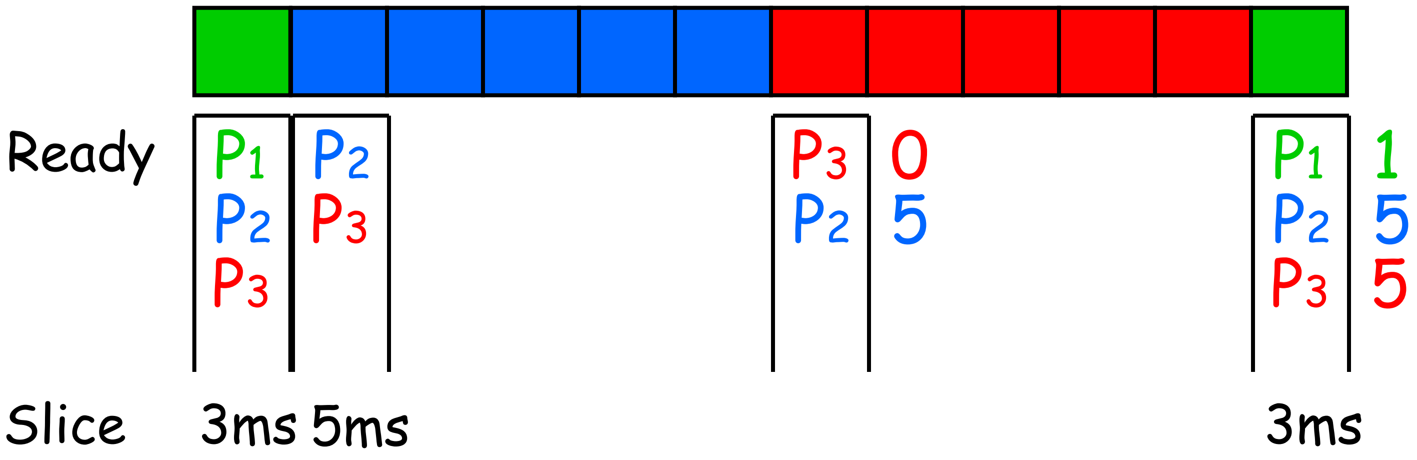
Completely Fair Scheduler (CFS)

- Approximate fair scheduling
  - Run each thread once per schedule latency (SL)
  - Weighted time slice: SL * Wi / (Sum of all Wi)

- Too many threads?
  - Lower bound on smallest time slice
  - Schedule latency = lower bound * (# threads)

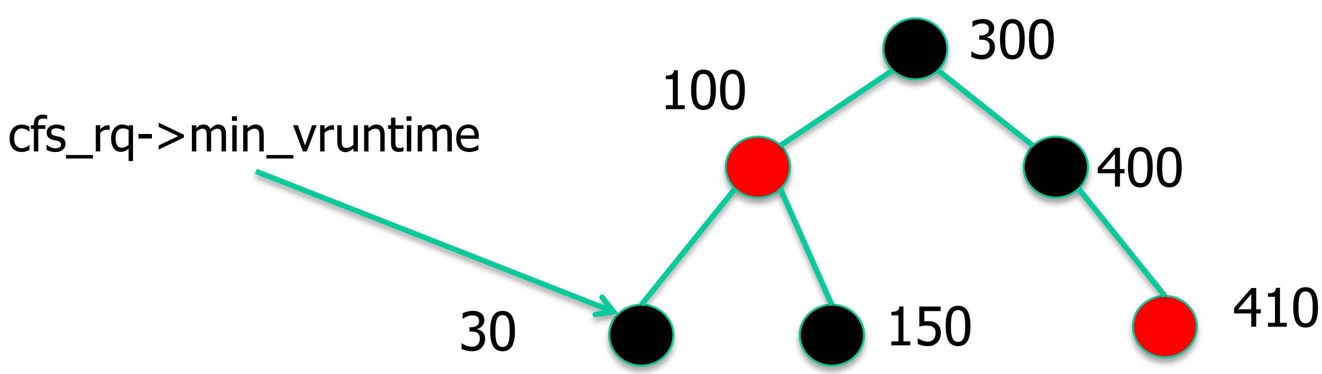
Picking the next process

- Pick proc with minimum virtual runtime so far
  - Virtual runtime: task->vruntime += executed time / Wi
- Example
  - P1: 1 ms burst per 10 ms (schedule latency)
  - P2 and P3 are CPU-bound
  - All processes have the same weight (1)



Finding proc with minimum runtime fast

- Red-black tree
  - Balanced binary search tree
  - Ordered by vruntime as key
  - O(lgN) insertion, deletion, update, O(1): find min



- Tasks move from left of tree to the right
- min_vruntime caches smallest value
- Update vruntime and min_vruntime
  - When task is added or removed
  - On every timer tick

Notable implementation details

- Integer table of nice-level to weight
  - static const int prio_to_weight[40] (kernel/sched/sched.h)
  - Nice level changes by 1 ➔ 10% weight
- cgroup
  - Fairness between users & apps, rather than threads
  - cgroup's vruntime == sum of its threads' vruntimes
- Upper bound on vruntime difference
  - New thread gets max vruntime in the RQ
  - When thread wakes up, its vruntime >= min_vruntime
- Load balancing based on many factors

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

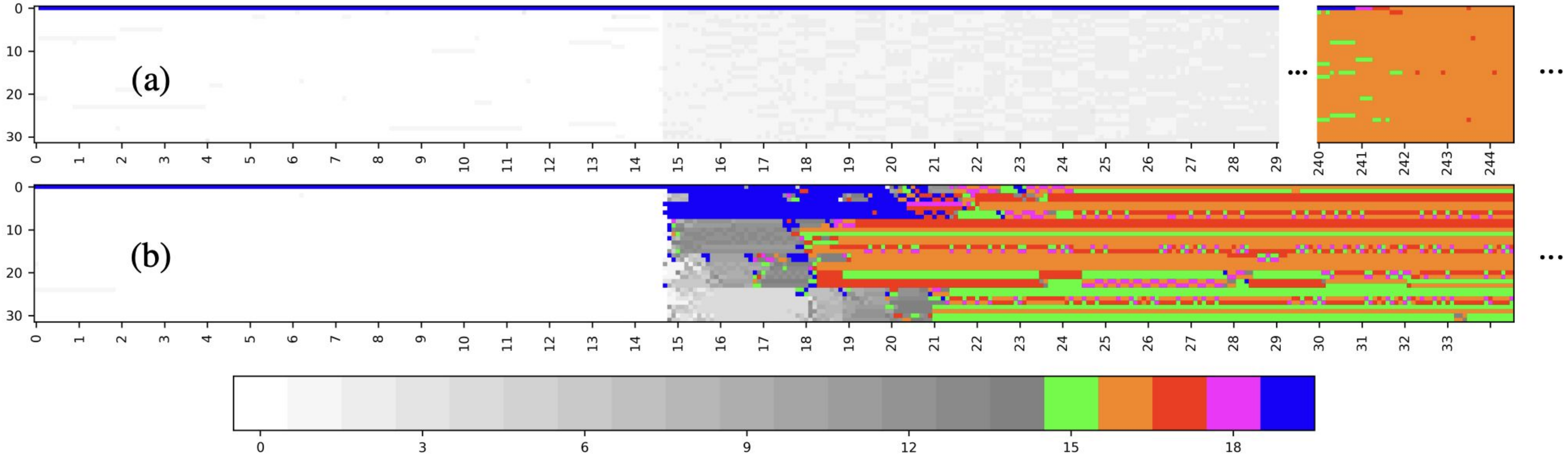
Wake-up/Creation:

- 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



(a) Slow "perfect" load balancing (b) CFS

CFS no more → EEVDF

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

Fairness

Process lag = weighted average of every task's vruntime - process current weighted vruntime

Weight based on nice value

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

CFS no more → EEVDF

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

Hierarchical scheduling class in Linux

`pick_next_task()` in kernel/sched/core.c essentially does this:

```
for (class = sched_class_highest;
     class != NULL;
     class = class->next;)
{
    p = class->pick_next_task(rq);
    if (p)
        return p;
}

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

struct sched_class (up to kernel 5.8)

```
const struct sched_class rt_sched_class = {
    .next               = &fair_sched_class,
    .enqueue_task       = enqueue_task_rt,
    .dequeue_task       = dequeue_task_rt,
    .yield_task         = yield_task_rt,
    .check_preempt_curr = check_preempt_curr_rt,
    .pick_next_task     = pick_next_task_rt,
    .put_prev_task      = put_prev_task_rt,
    .set_next_task      = set_next_task_rt,
```

Array of sched_class

- Sched classes are now arranged in an array by linker scripts
  - include/asm-generic/vmlinux.lds.h:

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

- for_class_range() macros in 5.8:

```
#define for_class_range(class, _from, _to) \
        for (class = (_from); class != (_to); class = class->next)
```

- for_class_range() macros in 5.10:

```
#define for_class_range(class, _from, _to) \
        for (class = (_from); class != (_to); class--)
```

Runqueue data structures

- **`struct rq`** (kernel/sched/sched.h)
  - Main per-CPU runqueue data structure
  - Contains per-sched_class runqueues: `cfs_rq`, `rt_rq`, etc.
- **`struct sched_entity`** (include/linux/sched.h)
  - sched_<class>_entity for each sched_class (except that sched_entity is for cfs)
  - Member of task_struct, one per each sched_class
- **`task_struct.sched_class`**
  - Pointer to the current sched_class for the task
  - `sched_setscheduler()` syscall changes process's sched_class