Project 3
Hello, Scheduler!

April 27, 2021
SNU Operating Systems

Project 3 Overview

- Implement WRR (Weighted Round-Robin) scheduler
  - The WRR scheduler handles *normal* tasks and completely replaces the CFS scheduler
  - Incorporate the new scheduler into the existing Linux scheduling framework
  - The WRR scheduler should also perform load balancing between CPUs
- Experiment on how the turnaround time of a program changes as its WRR weight increases from 1 to 20
- Clean your code and add comments before submission

WRR Scheduler

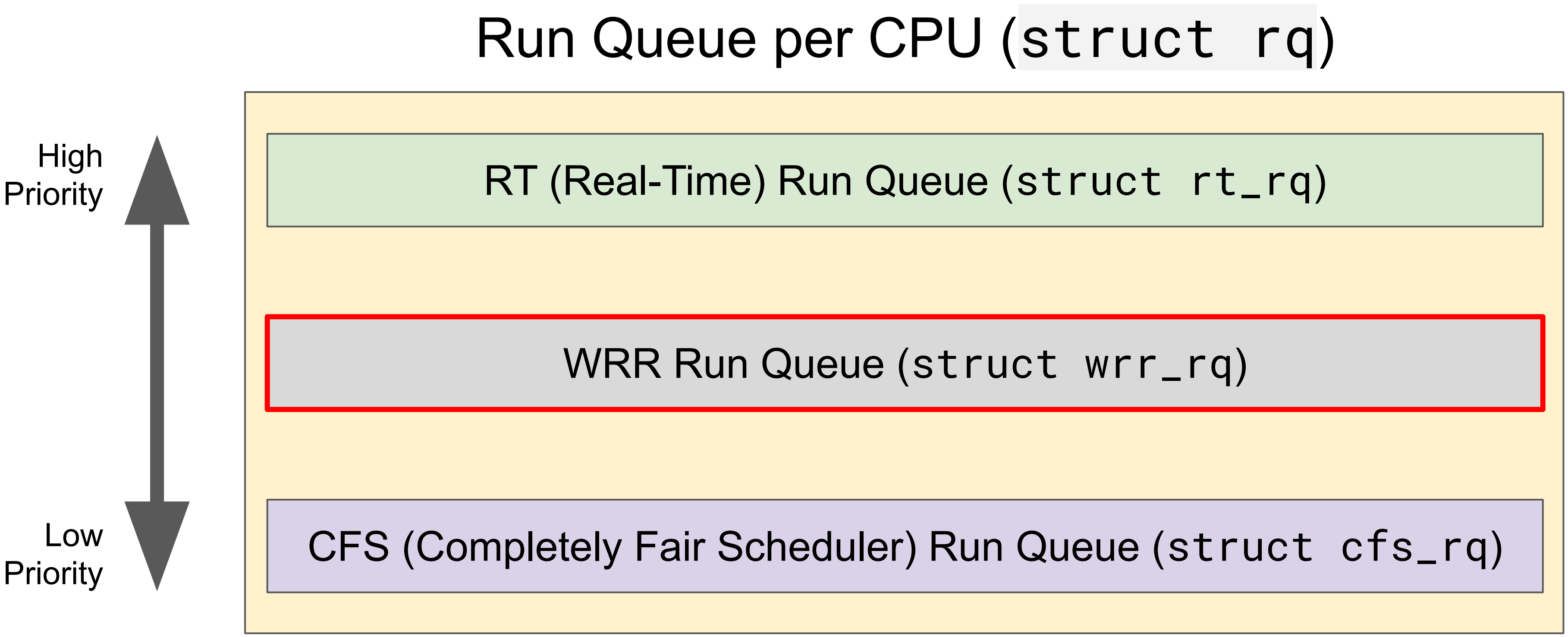
Linux Scheduler Basics

- Multi-level scheduling
  - Real-time tasks have higher priority than normal tasks
- Real-time tasks: FIFO, RR
- Normal tasks: CFS
- Each CPU maintains separate run queues for different types of tasks
  - To prevent contention while accessing run queue

WRR Scheduler

- Weighted Round-Robin Scheduler
- Tasks are executed in a round-robin fashion, but get different time slices according to their weights
  - Default weight is 10
  - Time slice = Weight * 10ms
- Priority: RT > WRR > CFS
- Load balancing

Multi-level Run Queue with WRR



Run Queue per CPU (`struct rq`)

High Priority

RT (Real-Time) Run Queue (`struct rt_rq`)

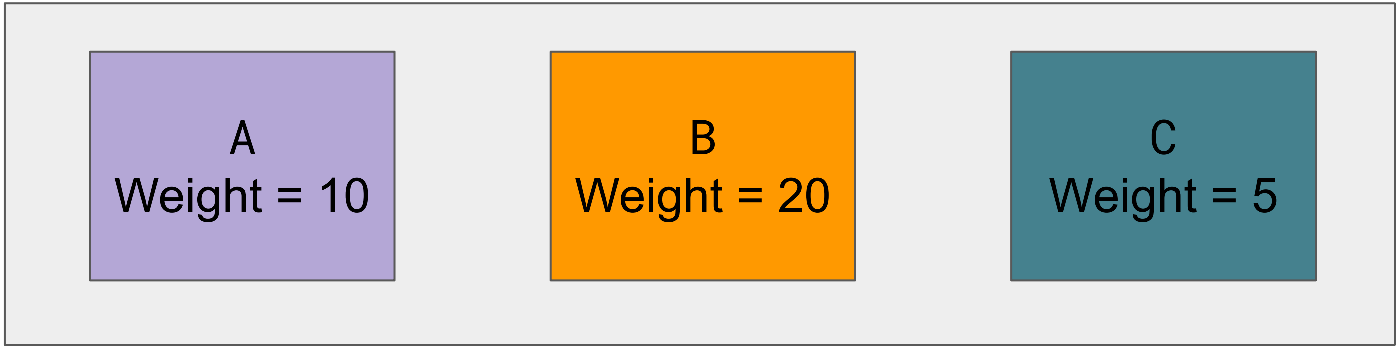
WRR Run Queue (`struct wrr_rq`)

CFS (Completely Fair Scheduler) Run Queue (`struct cfs_rq`)

Low Priority

WRR Scheduling Example

Three tasks currently in WRR run queue



A
Weight = 10

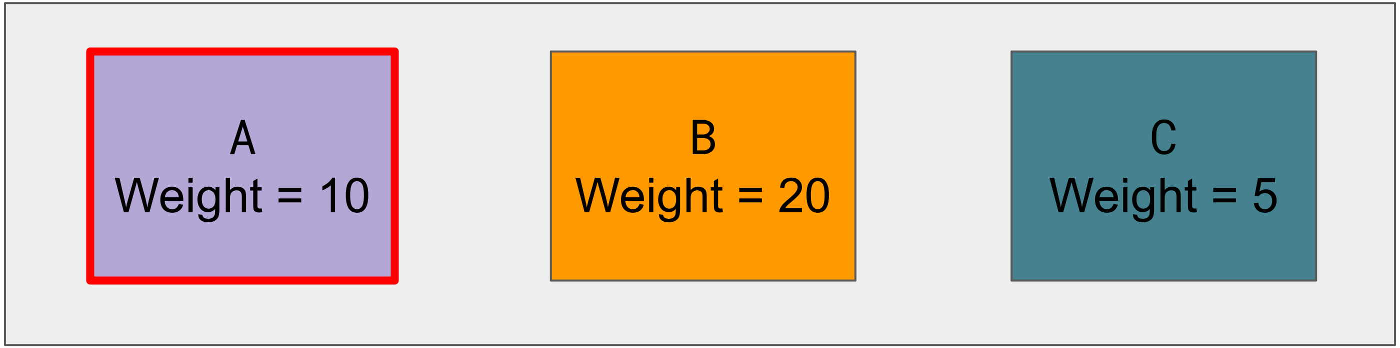
B
Weight = 20

C
Weight = 5

WRR Scheduling Example

t = 0ms

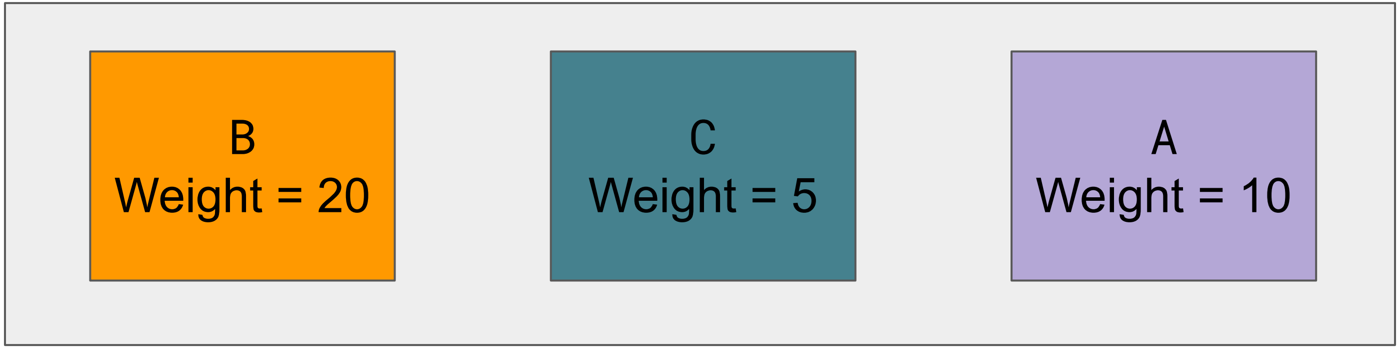
`A` starts running first.



WRR Scheduling Example

t = 100ms (Δt = 100ms)

A stops, and is moved to the tail of the run queue if the task is not finished.



B
Weight = 20

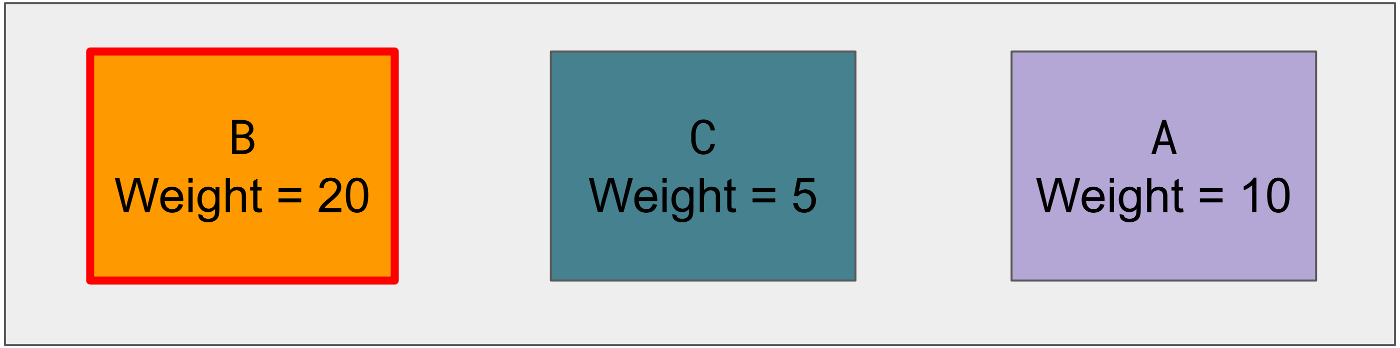
C
Weight = 5

A
Weight = 10

WRR Scheduling Example

t = 100ms

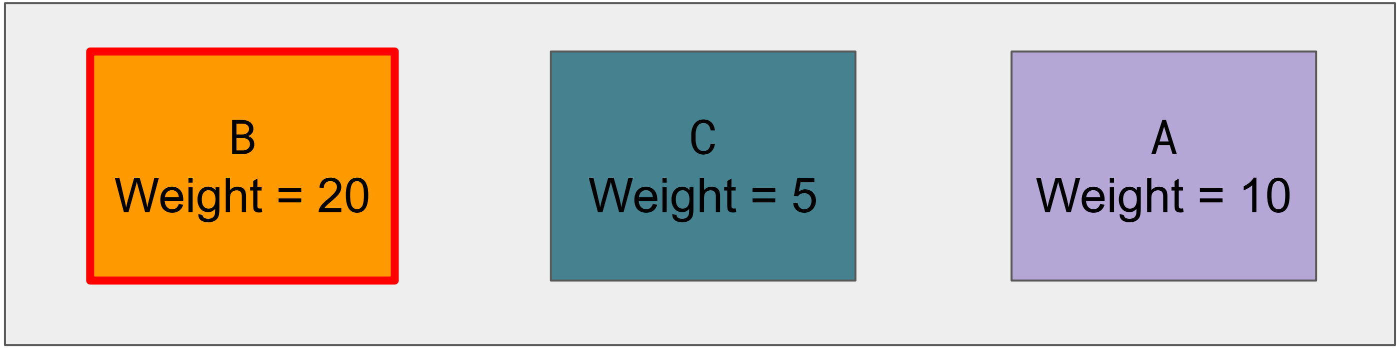
… and the next task (B) starts running.



WRR Scheduling Example

t = 200ms (Δt = 100ms)

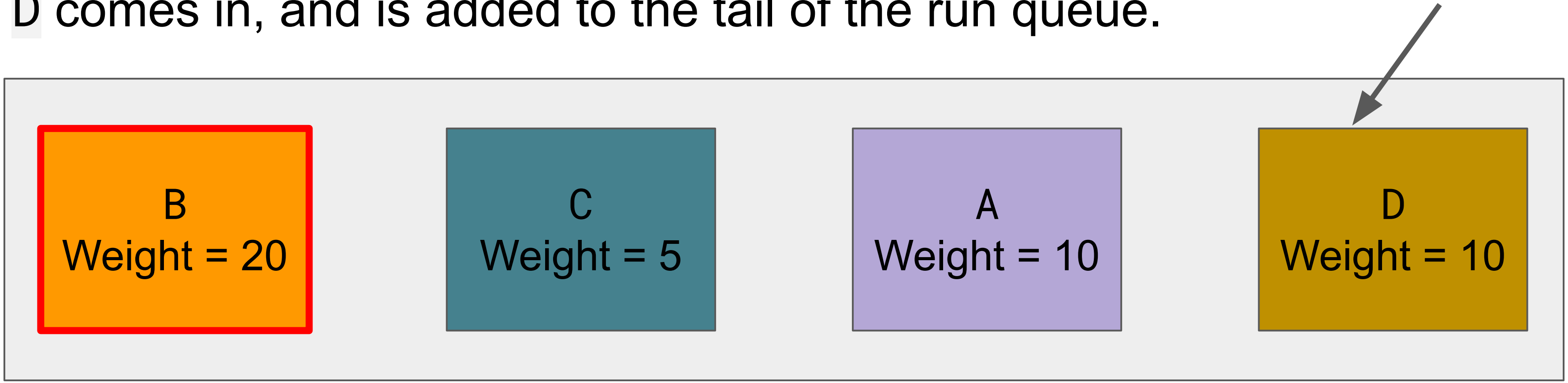
B is still running, because its time slice is 200ms.



WRR Scheduling Example

t = 250ms (Δt = 50ms)

D comes in, and is added to the tail of the run queue.



Newly added!

B
Weight = 20

C
Weight = 5

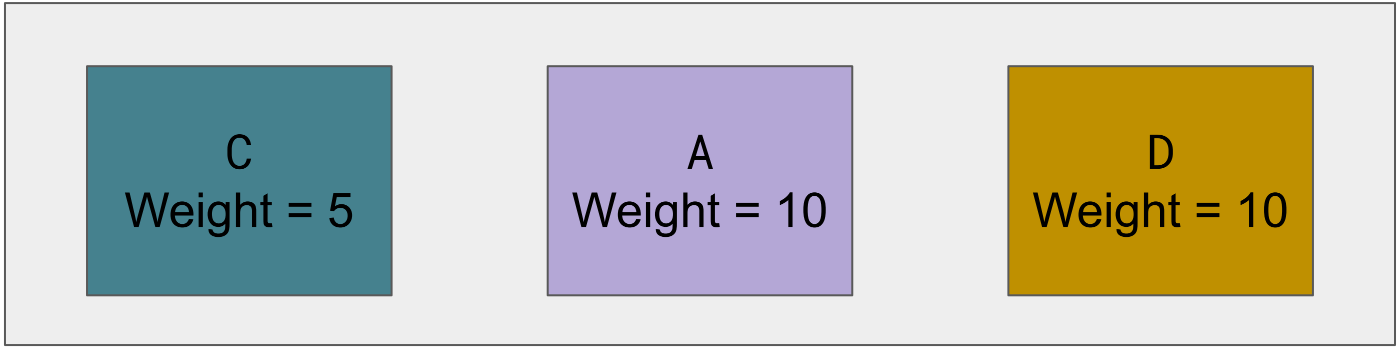
A
Weight = 10

D
Weight = 10

WRR Scheduling Example

t = 280ms (Δt = 30ms)

B has finished its work and is terminated; now removed from the run queue…



C
Weight = 5

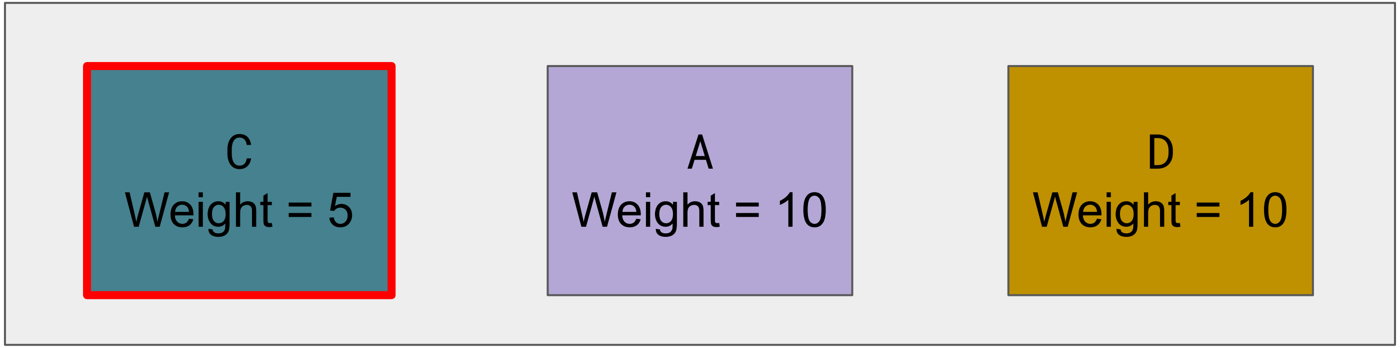
A
Weight = 10

D
Weight = 10

WRR Scheduling Example

t = 280ms

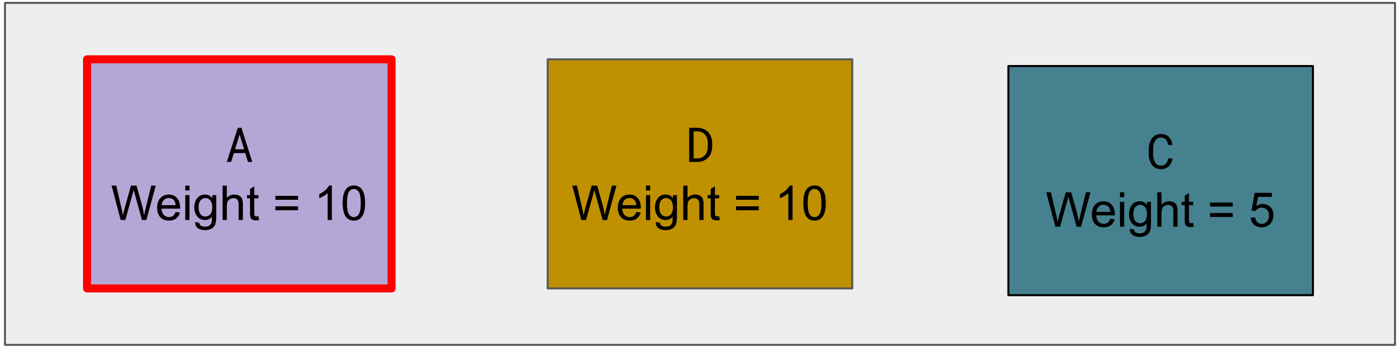
… and C starts running.



WRR Scheduling Example

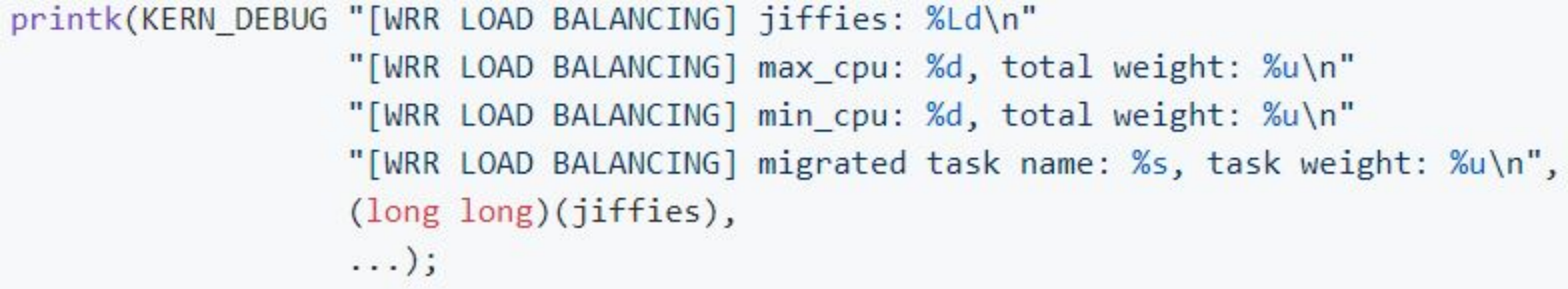
t = 330ms (Δt = 50ms)

`C` is stopped and is moved to the tail. `A` starts running again.



Load Balancing

- Balance the loads (*total weights*) of CPUs
  - Migrate a task from one CPU to another
- Print logs



```
printk(KERN_DEBUG "[WRR LOAD BALANCING] jiffies: %Ld\n"
                  "[WRR LOAD BALANCING] max_cpu: %d, total weight: %u\n"
                  "[WRR LOAD BALANCING] min_cpu: %d, total weight: %u\n"
                  "[WRR LOAD BALANCING] migrated task name: %s, task weight: %u\n",
                  (long long)(jiffies),
                  ...);
```

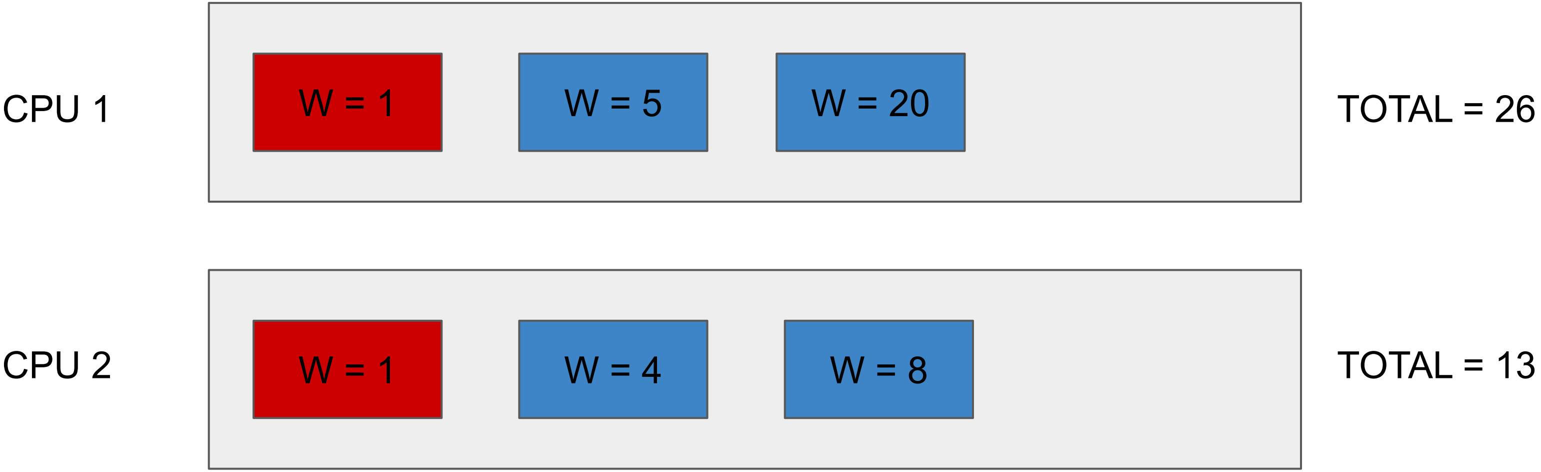
- Repeat every 2000 ms

Load Balancing Algorithm

- Select two CPUs with the largest/smallest total weights
    - Call them `max_cpu` and `max_cpu` respectively
- Pick a **single** task with the largest weight, which satisfies the following conditions:
    - The task's CPU affinity should allow migrating the task to `min_cpu`
    - Migration should not make the total weight of `min_cpu` **equal to or greater than** that of `max_cpu`
    - The task should not be running on a CPU
- Perform load balancing if a transferable task exists
    - There may be no such task

Load Balancing Example

Attempt to migrate a task from CPU 1 to CPU 2



CPU 1 — W = 1 — W = 5 — W = 20 — TOTAL = 26

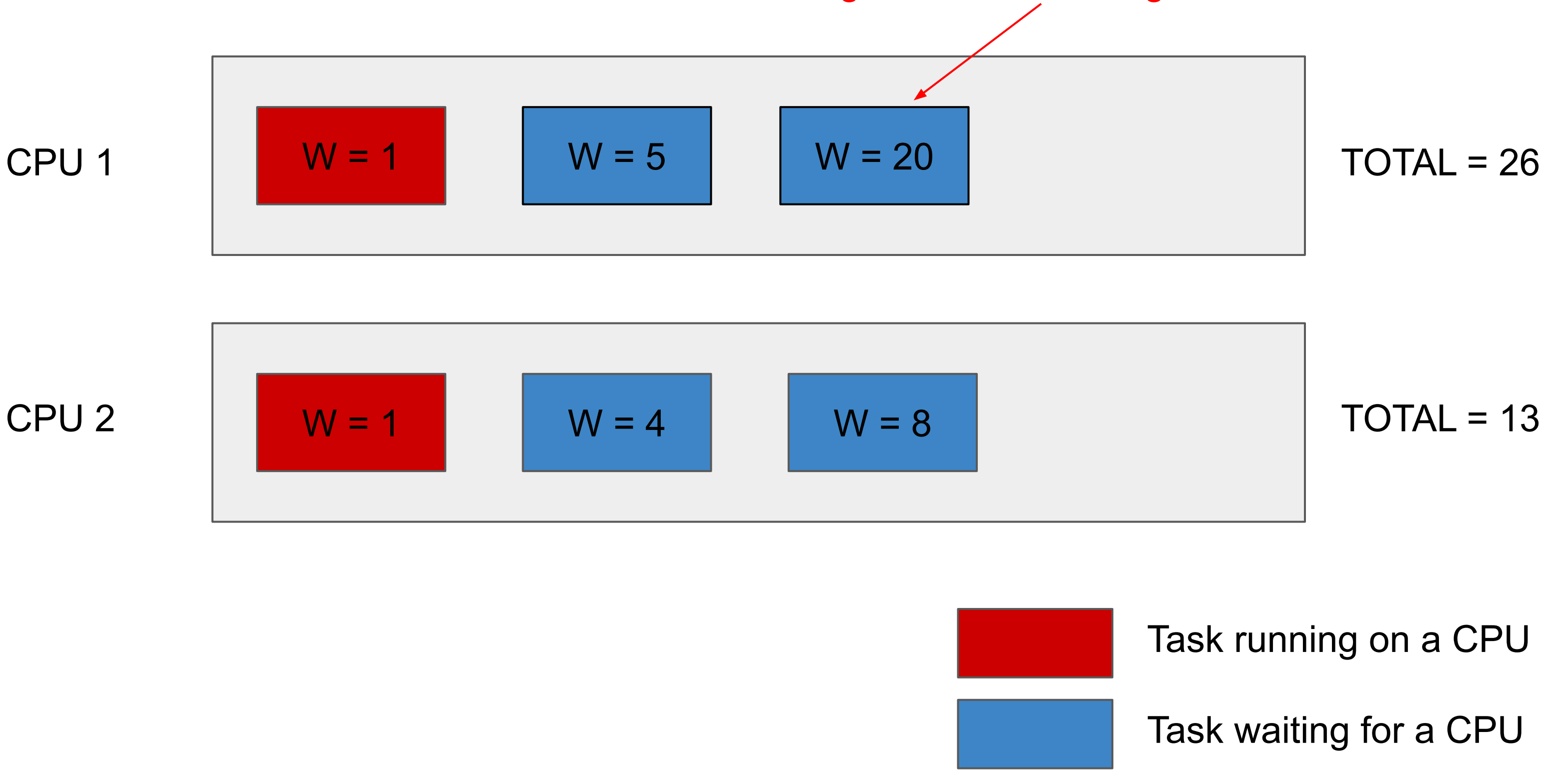
CPU 2 — W = 1 — W = 4 — W = 8 — TOTAL = 13

🟥 Task running on a CPU

🟦 Task waiting for a CPU

Load Balancing Example

This task cannot be migrated because it will make the weight sum of CPU 2 greater than that of CPU 1



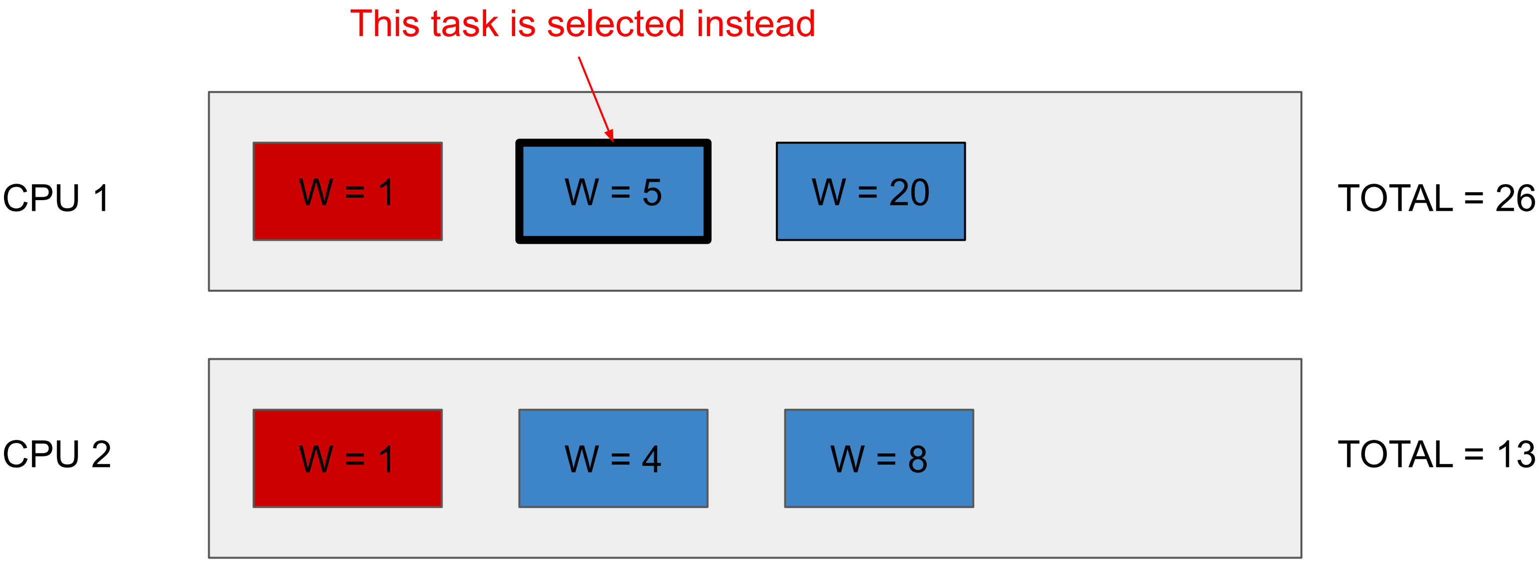
CPU 1 — W = 1 — W = 5 — W = 20 — TOTAL = 26

CPU 2 — W = 1 — W = 4 — W = 8 — TOTAL = 13

🟥 Task running on a CPU

🟦 Task waiting for a CPU

Load Balancing Example



This task is selected instead

CPU 1

W = 1

W = 5

W = 20

TOTAL = 26

CPU 2

W = 1

W = 4

W = 8

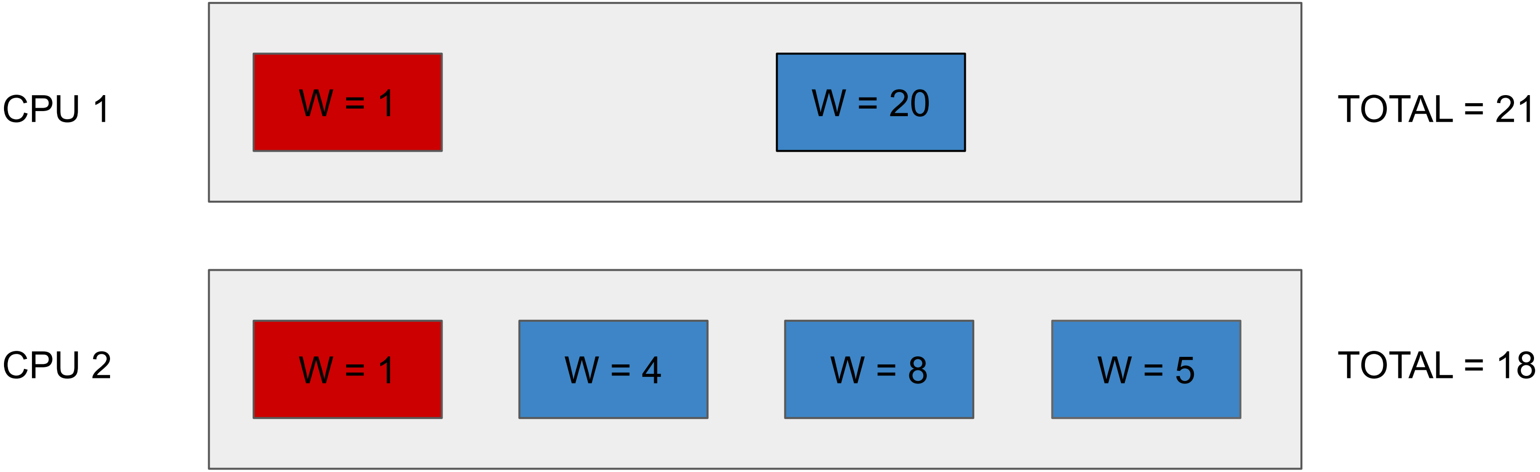
TOTAL = 13

🟥 Task running on a CPU

🟦 Task waiting for a CPU

Load Balancing Example

After migration



CPU 1

W = 1

W = 20

TOTAL = 21

CPU 2

W = 1

W = 4

W = 8

W = 5

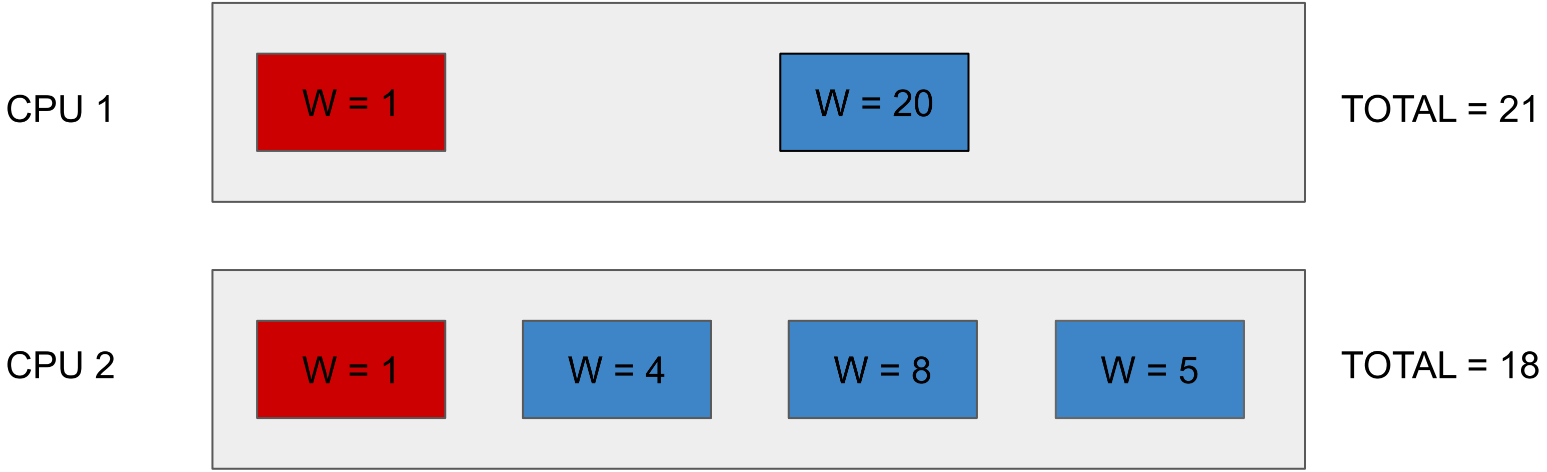
TOTAL = 18

🟥 Task running on a CPU

🟦 Task waiting for a CPU

Load Balancing Example

After 2000 ms, the scheduler attempts to migrate a task from CPU 1 to CPU 2 again



CPU 1

W = 1

W = 20

TOTAL = 21

CPU 2

W = 1

W = 4

W = 8

W = 5

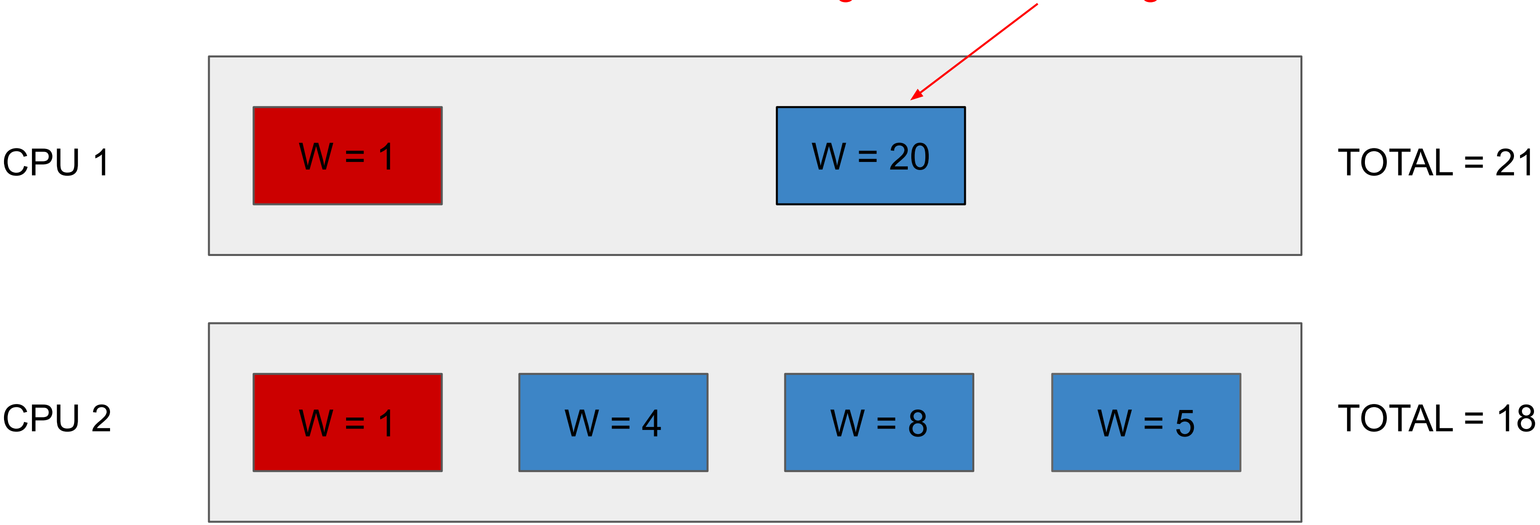
TOTAL = 18

🟥 Task running on a CPU

🟦 Task waiting for a CPU

Load Balancing Example

This task cannot be migrated because it will make the weight sum of CPU 2 greater than that of CPU 1

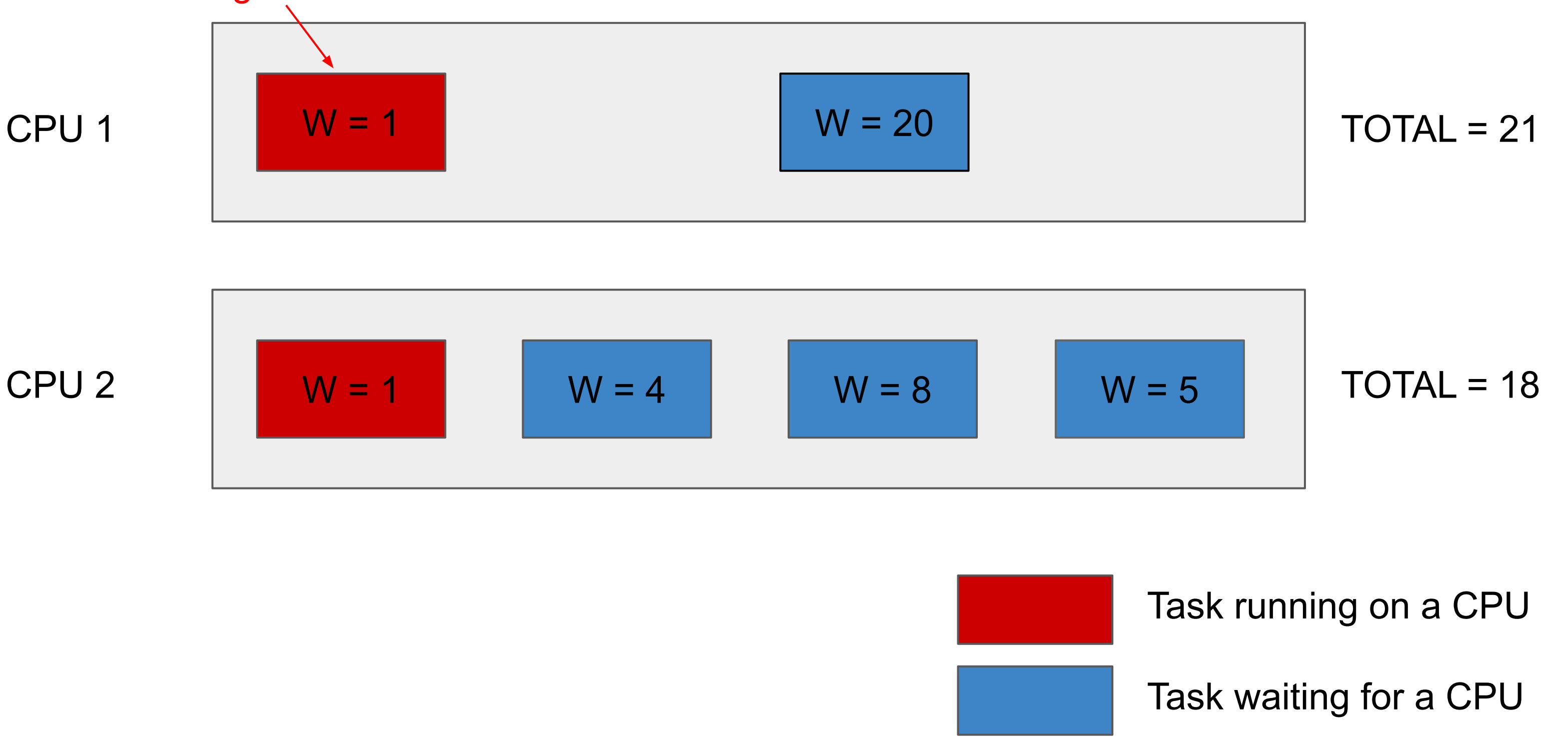


🟥 Task running on a CPU

🟦 Task waiting for a CPU

Load Balancing Example

This task cannot be migrated since it is running on a CPU



CPU 1: W = 1, W = 20 — TOTAL = 21

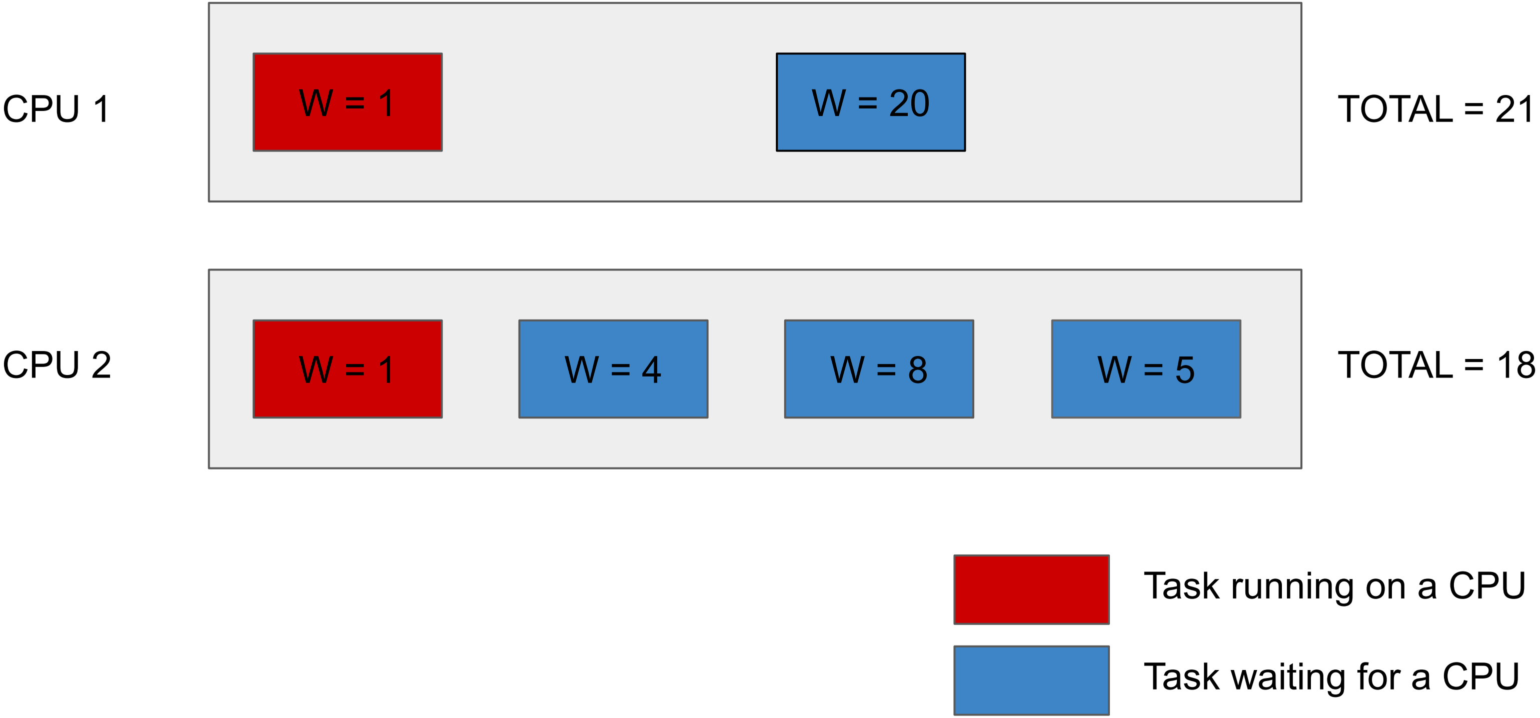
CPU 2: W = 1, W = 4, W = 8, W = 5 — TOTAL = 18

🟥 Task running on a CPU

🟦 Task waiting for a CPU

Load Balancing Example

Load balancing is **skipped**
The scheduler retries after 2000 ms



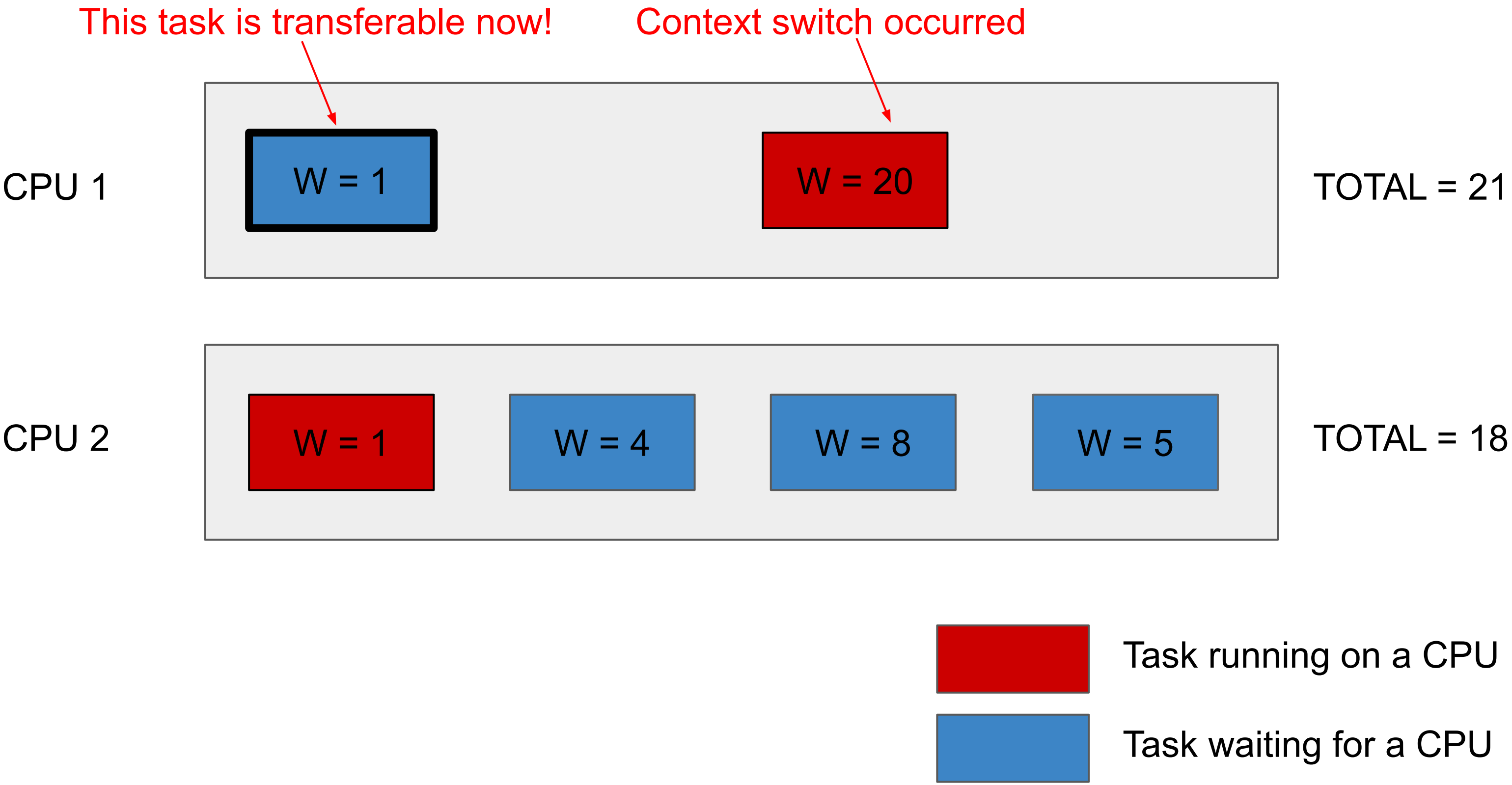
CPU 1: W = 1, W = 20 — TOTAL = 21

CPU 2: W = 1, W = 4, W = 8, W = 5 — TOTAL = 18

🟥 Task running on a CPU

🟦 Task waiting for a CPU

Load Balancing Example



This task is transferable now!

Context switch occurred

CPU 1

W = 1

W = 20

TOTAL = 21

CPU 2

W = 1

W = 4

W = 8

W = 5

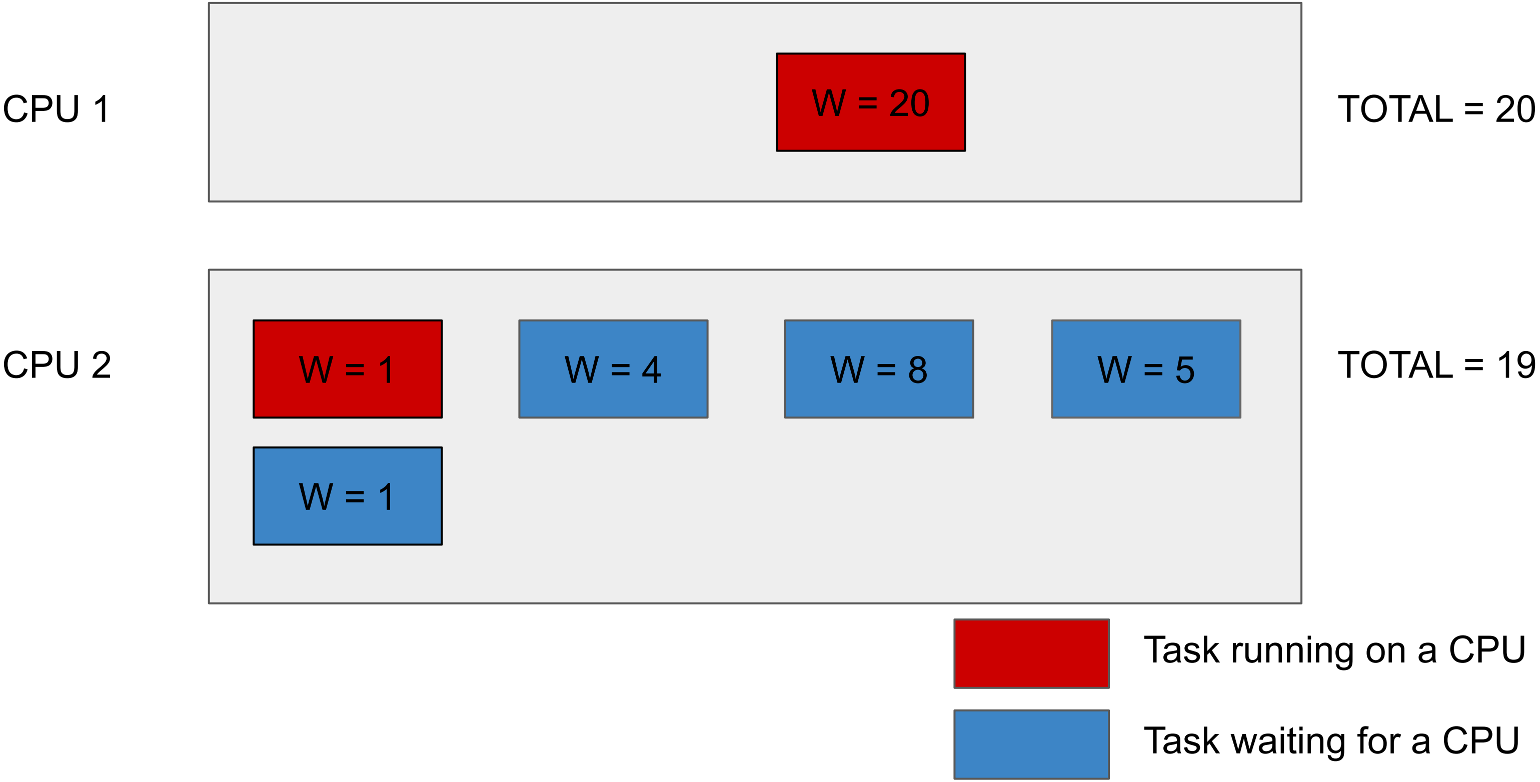
TOTAL = 18

🟥 Task running on a CPU

🟦 Task waiting for a CPU

Load Balancing Example

After migration



CPU 1

W = 20

TOTAL = 20

CPU 2

W = 1

W = 4

W = 8

W = 5

W = 1

TOTAL = 19

🟥 Task running on a CPU

🟦 Task waiting for a CPU

Scheduler Implementation

Preliminaries

- Check `arch/arm64/configs/tizen_bcmrpi3_defconfig`
  - We have already set `CONFIG_SCHED_DEBUG=Y` and `CONFIG_SCHEDSTATS=Y`
  - These options should be enabled in order to debug your scheduler
- (Optional) Modify `kernel/sched/debug.c` to print necessary information about WRR scheduler
  - Plus, modifying `kernel/sched/stats.c` to print statistics will be helpful

Implementation Overview (1)

- Define necessary constants and data structures
  - `include/linux/sched.h`
  - `include/uapi/linux/sched.h`
  - ...
- Register a new scheduling class for WRR and implement necessary functions in **`kernel/sched/wrr.c`**
- Make every normal task (including **`swapper`** and **`kthreadd`**) assigned to the WRR scheduler

Implementation Overview (2)

- Modify **`kernel/sched/core.c`** to support WRR
    - e.g.) trigger load balancing function every 2000 ms, …
- Register some function signatures in `kernel/sched/sched.h`
- Implement two system calls
    - `sched_setweight` and `sched_getweight`
- Check that your scheduler performs load balancing correctly

Constants & Data Structures

- Define `SCHED_WRR` as **7** (We've done this in the patch)
  - `include/`**`uapi`**`/linux/sched.h`
- Define some fields for WRR scheduler in `struct task_struct`
  - See how other schedulers like RT, CFS, … keep their run-time data
  - Store WRR weight, time slice, and other necessary metadata
- Define a run queue for tasks under the WRR scheduler
  - `struct rq` will have to maintain some information about the WRR run queue
  - What kind of information should be stored here?
  - Should this have a locking mechanism?

Registering Scheduler

- Declare and define **`wrr_sched_class`** in `kernel/sched/sched.h` and in `kernel/sched/wrr.c`
    - Take a look at `kernel/sched/fair.c` & `kernel/sched/rt.c`
    - The priority should be RT > WRR > CFS
- Implement necessary functions for `wrr_sched_class`
    - `enqueue_task`, `dequeue_task`, …
    - You don’t need to implement all the functions to make it work
- Define other necessary functions for load balancing and debugging

Modifying `kernel/sched/core.c`

- The Linux scheduler is NOT fully pluggable: the code implicitly assumes that there are exactly four predefined scheduling classes (i.e. DL, RT, CFS, IDLE) in the kernel
- You will need to hack `kernel/sched/core.c` at various points in order to incorporate the WRR scheduler into the kernel
  - Initialize WRR run queue
  - Make `SCHED_WRR` policy valid
  - Manage forked tasks
    - A child should follow the same scheduling policy of its parent
  - ...

Debugging

- Reminder: You should turn on `CONFIG_SCHED_DEBUG`
- You might want to modify `kernel/sched/debug.c` to check whether your WRR scheduler works correctly
- Scheduling information is written to `/proc/sched_debug`

System Calls

- You all know how to implement system calls!
- Authentication is essential in `sched_setweight`
  - Increasing weight: administrator only
  - Decreasing weight: process owner & administrator only
  - Check uid and euid
- Nothing difficult here :)

Load Balancing (1)

- How do I check the remaining time slice or figure out when to trigger load balancing?
- Take a look at the function `scheduler_tick`
  - `kernel/sched/core.c`
  - Called every tick
- Tick frequency: HZ
  - A macro which represents the number of ticks in a second

Load Balancing (2)

- How do I check the remaining time slice or figure out when to trigger load balancing? (cont'd)
- `scheduler_tick`
- Tick frequency: HZ
- `jiffies`
  - A global variable containing the number of ticks after system boot
  - unsigned long - beware of overflow!
  - There are some useful macros you should know
    - `time_after`, `time_before`, `time_after_eq`, `time_before_eq`
  - More things: http://www.makelinux.net/ldd3/chp-7.shtml

Load Balancing (3)

- How do I prevent race condition while load balancing?
- Note that `scheduler_tick` is called by every CPU
  - You need to make sure that only one CPU is working on load balancing at any time
- Think carefully about synchronization issues

Experiment

- Main question: How does the WRR weight affect the performance of a batch job?
- Write a test program `trial` that calculates the prime factorization of a prime number using the trial and division method
- Measure the turnaround time of `trial` with varying weights
  - You can spawn multiple processes and average their turnaround times to get clearer results

Misc Comments

- It is natural that the WRR scheduler is slow
  - When the shell is not responding, just wait for a while
- `rcu_read_lock` when iterating over CPU cores
- This is the hardest project so far, and the only project that you may not be able to finish on time, so start early!

Design Review

- Conversation with the TA about your design and plan
- Check the eTL notice
- Due: 4/30 (Fri)

About submission (IMPORTANT!)

- Don't be late!
  - TAs will clone all repositories exactly at the deadline
- Submit code
  - Your team’s private project 3 repo (swsnu/project3-hello-scheduler-team-n), master branch
  - README: description of your implementation, how to build, results of your experiments, and lessons learned
- Submit slides and demo (n is your team number)
  - Email: osspr2021@gmail.com
  - Title: [Project 3] Team n
  - Attachments: `team-n-slides.{ppt,pdf}, team-n-demo.{mp4,avi,…}`
    - One slide file, one demo video!

Q & A