Your Mission: Implement a Weighted Round Robin Scheduler in C

Data Structures. Pepe Gallardo, 2025.

Dpto. Lenguajes y Ciencias de la Computación. University of Málaga

---

Introduction: The Heart of an Operating System

Welcome, future systems engineer! Every time you use a computer, an operating system (OS) is working tirelessly behind the scenes, managing countless tasks at once. One of its most critical jobs is deciding which task gets to use the CPU and for how long. This process is managed by a **task scheduler**.

In this project, you will build your own **Weighted Round Robin (WRR)** scheduler. The WRR algorithm is a smart and fair way to manage multiple tasks by giving them CPU time based on their assigned priority or "weight."

Using the C programming language, you will dive into the world of system-level programming. We will use a **singly linked circular list** as our core data structure to create a scheduler that is both elegant and efficient. By the end of this project, you will not only have a functional scheduler but also a deep, practical understanding of how operating systems bring order to chaos. Let's get started!

Understanding the Foundation: The Singly Linked Circular List

Before we build the scheduler, let's master its foundation. A singly linked circular list isn't your average list; it's a dynamic, continuous loop of data perfect for a scheduler that runs forever.

Key Components:

1. **The Node:** Each element in our list is a `Node`. A node contains two things:
   - The data itself (in our case, all the information for a single task).
   - A pointer that acts like an arrow, pointing to the *next* node in the list.
2. **The Circular Nature:** Unlike a standard list that ends with a `NULL` pointer, the last node in our list points right back to the first one. This creates a circle, an endless loop that our scheduler can traverse indefinitely.
3. **The Anchor (`p_last`):** To make managing our list easy, we will only keep track of one node: the **last** one. By holding a pointer to the last node (`p_last`), we can efficiently access both the end of the queue (to add new tasks) and the beginning (since `p_last->p_next` points to the first task).

This circular structure will be the engine of our scheduler.

The Algorithm: Weighted Round Robin

Now, let's see how our scheduler will use this circular list to conduct a fair and efficient performance.

The WRR Algorithm: A Fair Distribution of Time

1. **Setting the Stage:** Each task enters our circular queue, bringing with it a **weight**. This weight determines its **quantum**—the amount of time it gets to run.
2. **The Performance Begins:**
   - The scheduler picks the first task in the queue.

- This task takes the spotlight, running for its assigned time quantum.
- Once its time is up, it gracefully moves to the back of the queue.
- The next task in line steps forward for its turn.

3. **The Never-Ending Cycle:** This simple, powerful process repeats, ensuring every task gets its moment on stage, with a duration proportional to its weight.

A Practical Example

Imagine three tasks: **T1** (weight 2), **T2** (weight 1), and **T3** (weight 3).

1. **Initial Queue:** `[T1(2), T2(1), T3(3)]` (T1 is at the front)

2. **The Performance:**

   - T1 runs for 2 time units. The queue becomes: `[T2(1), T3(3), T1(2)]`
   - T2 runs for 1 time unit. The queue becomes: `[T3(3), T1(2), T2(1)]`
   - T3 runs for 3 time units. The queue becomes: `[T1(2), T2(1), T3(3)]`

3. **The Cycle Repeats:** The queue is now back to its original order, and the entire cycle begins again. This rhythmic pattern ensures CPU time is distributed fairly according to each task's weight.

Visualizing Our Scheduler

An empty scheduler is just a `NULL` pointer. A scheduler with tasks is a pointer (`p_last`) to the last node. Here's a diagram of a scheduler with three tasks, with the first node containing a task whose ID is 1:

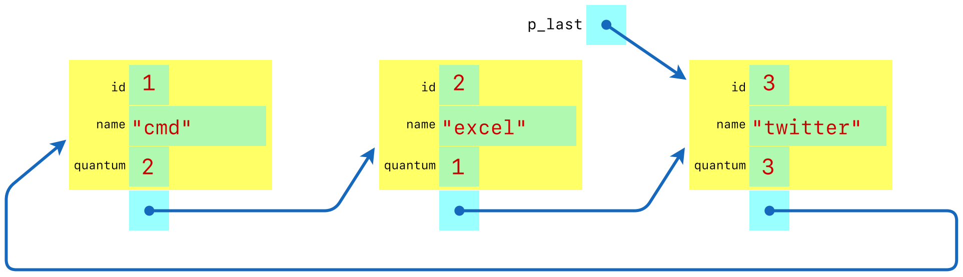

Figure 1: Queue of Tasks as a Circular Linked List

The cyan boxes represent pointers to nodes, while the yellow boxes represent tasks. Each node contains a task and a pointer to the next node, forming our circular structure. The `p_last` pointer always points to the last node, acting as our anchor in this circular structure. In this diagram, `p_last` points to the node with Task 3. The circular nature means that `p_last->p_next` points to the first node, containing Task 1.

Your Blueprint: Crafting the Scheduler in C

It's time to translate theory into code. Let's define the two main components of our system: the `Task` and the `Scheduler`.

The `Task`: Our Fundamental Unit

First, let's define what a task is. Each task will have three attributes:

- **ID**: A unique `unsigned int` identifier.

- **Name**: A character string for the task's name.
- **Quantum**: An `unsigned int` representing its assigned time quantum (its weight).

This is defined in the `Task.h` header file:

```
#ifndef TASK_H
#define TASK_H

#include <stddef.h>

#define MAX_NAME_LEN 20          // Maximum length for a task's name

struct Task {
  unsigned int id;               // A unique identifier
  char name[1 + MAX_NAME_LEN];   // The task's name
  unsigned int quantum;          // Its assigned time quantum
};

// Function prototypes for task operations
struct Task* Task_new(unsigned int id, const char name[], unsigned int quantum);
void Task_free(struct Task** p_p_task);
struct Task* Task_copyOf(const struct Task* p_task);
void Task_print(const struct Task* p_task);
#endif
```

Your first job is to implement the C source file (`Task.c`) for these functions:

- `Task_new`: Creates a new task on the heap. It must validate that the `name` is not longer than `MAX_NAME_LEN`.
- `Task_free`: Deallocates a task from the heap and sets the original pointer to `NULL` to prevent dangling pointers.
- `Task_copyOf`: Creates a new, independent copy of a given task on the heap.
- `Task_print`: Prints a task's details to standard output. The format should be:

```
Task(ID: <id>, Name: <name>, Quantum: <quantum>).
```

The `Scheduler`: Orchestrating Our Tasks

Now for the main event. The scheduler is represented by a pointer to the **last** node in our circular list. Each node in the list contains a task.

Here is the `Scheduler.h` header file:

```
#ifndef SCHEDULER_H
#define SCHEDULER_H

#include "Task.h"
#include <stddef.h>

struct Node {
  struct Task task;     // The task to be executed
  struct Node* p_next; // A pointer to the next task in line
};

// Function prototypes for scheduler operations
struct Node* Scheduler_new();
```

```c
size_t Scheduler_size(const struct Node* p_last);
void Scheduler_clear(struct Node** p_p_last);
struct Task* Scheduler_first(const struct Node* p_last);
void Scheduler_enqueue(struct Node** p_p_last, const struct Task* p_task);
void Scheduler_dequeue(struct Node** p_p_last);
void Scheduler_print(const struct Node* p_last);
#endif
```

Let's break down your implementation tasks (file `Scheduler.c`):

- `Scheduler_new`: Returns an empty scheduler (which is simply a `NULL` pointer).
- `Scheduler_size`: Returns the number of tasks currently in the scheduler.
- `Scheduler_clear`: Removes all tasks from the scheduler, freeing all associated memory.
- `Scheduler_first`: Returns a new copy of the first task in the queue without removing it. Important: The caller is responsible for freeing this copy later using `Task_free()`.
- `Scheduler_enqueue`: Adds a new task to the end of the queue. This involves creating a new node and updating the `p_last` pointer.
- `Scheduler_dequeue`: Removes the first task from the queue and frees the node's memory. This is called after a task completes its quantum.
- `Scheduler_print`: Displays all tasks in the queue, starting with the first.
  - Empty scheduler: prints `Scheduler()`
  - Non-empty: prints:
    - Header: `Scheduler(`
    - Then, for each task in order:
      - a newline, two spaces, then the task using `Task_print`
      - a comma after each task except the last
    - After the last task: a newline followed by `)`. There is no trailing newline after the closing parenthesis.

  Example (three tasks):

  ```
  Scheduler(
    Task(ID: 1, Name: T1, Quantum: 2),
    Task(ID: 2, Name: T2, Quantum: 1),
    Task(ID: 3, Name: T3, Quantum: 3)
  )
  ```

Assertions and error handling (read this before coding) In this assignment, all invalid conditions must be checked using assertions in the corresponding functions. Do not print error messages; instead, assert with a clear message so failures are caught immediately during the execution and by the test suite.

- Use `assert` for these cases (non-exhaustive):
  - Non-optional pointer parameters, in functions such as `Task_print`, `Scheduler_enqueue` and `Scheduler_dequeue`, must be non-NULL.
    - Examples:
      - `assert(p_task != NULL && "Task_print: NULL p_task");`
      - `assert(p_p_last != NULL && "Scheduler_enqueue: NULL p_p_last");`
      - `assert(p_task != NULL && "Scheduler_enqueue: NULL p_task");`
  - Operations that require a non-empty scheduler, such as `Scheduler_first` and `Scheduler_dequeue`, must assert that the scheduler is not empty (i.e., `p_last` is not `NULL`).

        * Examples:
            · `assert(p_last != NULL && "Scheduler_first: empty scheduler");`
            · `assert(p_last != NULL && "Scheduler_dequeue: empty scheduler");`

- Input validation in functions such as `Task_new`:
    - `assert(name != NULL && "Task_new: NULL name");`
    - `assert(strlen(name) <= MAX_NAME_LEN && "Task_new: name too long");`
    - `assert(quantum > 0 && "Task_new: quantum must be > 0");`
    - Memory allocation must succeed:
        * Example: after `malloc`, assert that the pointer to the allocated memory is not `NULL`.

This policy keeps contracts explicit and aligns with the tests, which expect assertion failures for invalid usage or memory allocation errors.

A Key Concept: Why Use a Pointer to a Pointer (`T**`)? In this assignment, some functions take a pointer to a pointer because they must modify the caller's pointer value (not just the data the pointer points to).

- Scheduler operations with `struct Node** p_p_last` (`Scheduler_enqueue`, `Scheduler_dequeue`, `Scheduler_clear`), where `p_p_last` is the address of the `p_last` pointer which points to the last node in the scheduler: these may need to change `p_last` itself. For example, when inserting into an empty list `p_last` goes from `NULL` to the new node; when removing the only node, `p_last` goes back to `NULL`. Passing the address of `p_last` lets the function update the anchor in the caller.

- Task deallocation with `struct Task** p_p_task` (`Task_free`), where `p_p_task` is the address of the `p_task` pointer which points to the task being freed: this frees the task (after asserting the pointers are non-`NULL`) and then sets the caller's pointer to `NULL` to avoid dangling pointers. Passing the address allows the callee to null out the pointer held by the caller.

General rule: we pass `T**` when the callee must reassign the caller's `T*` (e.g., set it to a new object or to `NULL`). See the provided demonstration program (`Demo.c`) for examples of how to use these functions.

Your Implementation Playbook: Tips for Success

As you build your scheduler, you are not working in the dark. We have provided a complete framework to help you test, debug, and validate your code. Here's how to make the most of it.

1. Understand the Dual-Mode Main Program

Your project's entry point is `Main.c`. This file is designed to run in one of two modes, which you can switch between by editing the file:

- **Demo Mode (`RUN_DEMO`)**: Runs a simple demonstration program from `Demo.c`. This shows a high-level example of how your scheduler functions are intended to be used together.
- **Test Mode (`RUN_TESTS`)**: Runs a comprehensive set of unit tests from `tests/Suite.c`. These tests will rigorously check every function you write for correctness, edge cases, and memory management.

To switch between modes, open `Main.c` and change the `#define` directives at the top before compiling and running your program:

To run the demonstration:

```
#undef RUN_TESTS
#define RUN_DEMO
```

To run the unit tests (this should be your default):

```
#define RUN_TESTS
#undef RUN_DEMO
```

2. Start with the Demo, Then Live in the Tests

We recommend the following workflow:

1. **Study the Demo (`Demo.c`)**: Before you write any code, read through `runDemo()`. It shows you exactly how a user would interact with your scheduler: creating it, adding tasks, peeking at the first task, and clearing it. This will give you a clear picture of the final goal.
2. **Use the Test Suite (`Suite.c`)**: This is your most important tool. We have written dozens of tests that check every aspect of your implementation. There is a test suite for each function you will implement. A correct scheduler **must pass all of these tests**. As you develop, compile and run the tests constantly. They will fail at first, but as you implement each function correctly, you will see the tests begin to pass one by one. This is the best way to track your progress and catch bugs immediately.

3. Master the Edge Cases

The test suite will check for many tricky scenarios, but it's crucial for you to think like a tester. As you write each function, ask yourself "what if?":

- What if I add a task to an **empty scheduler**?
- What if I remove the **only task** from the scheduler? Does `p_last` become `NULL`?
- What if I call `Scheduler_first` or `Scheduler_dequeue` on an **empty scheduler**? (Your code should assert and stop, as the tests expect).
- What if the task name is more than `MAX_NAME_LEN` characters long?

Handling these cases gracefully is the difference between code that just works sometimes and code that is truly robust.

4. Be a Memory Guardian

C gives you complete control over memory, which is both a power and a great responsibility.

- **Every `malloc` needs a `free`**: For every piece of memory you allocate for a `Task` or a `Node`, you must ensure it is eventually deallocated.
- **Avoid Dangling Pointers**: After freeing memory, set the pointer to `NULL`, just as `Task_free` is designed to do.

The good news is that the provided test suite is designed to help you with this. It automatically tracks memory allocations and deallocations. If your code has a memory leak, the tests will fail and tell you! This is an invaluable tool for writing clean, professional-grade C code.

Conclusion: Reflecting on Your Creation

By completing this project, you will have built a core component of an operating system from scratch. This is no small feat. You will have translated an abstract algorithm into a concrete, working program, gaining invaluable experience in C programming, data structures, and memory management.

The principles of resource allocation and scheduling you've applied here are fundamental to computer science. Take pride in your work, and get ready to code. May your tasks always run smoothly