## Work Stealing (Project 1)

### Overview

The Work Stealing Project aims to implement a task management system with multiple cores capable of managing and redistributing tasks dynamically. The design focuses on efficiency, scalability, and optimal load balancing among cores. This document outlines the key design decisions, class structures, and algorithms employed in the implementation.

### Class Design

- Reason for using two classes

  The decision to use three classes, Deque, Core and CPU, was based on the principles of separation of concerns, which shows that each class has a clear responsibility. The designs allow for easier testing and modification. For example, you can test the Deque or Core independently of the CPU class. Each class encapsulates its own data and methods, protecting the integrity of the data and promoting cleaner, more maintainable code.

  **Deque class:** The rationale of this class is that tasks can be added to be the back when created and removed from the front when executed, while still allowing for efficient popping from the back when tasks are stolen.

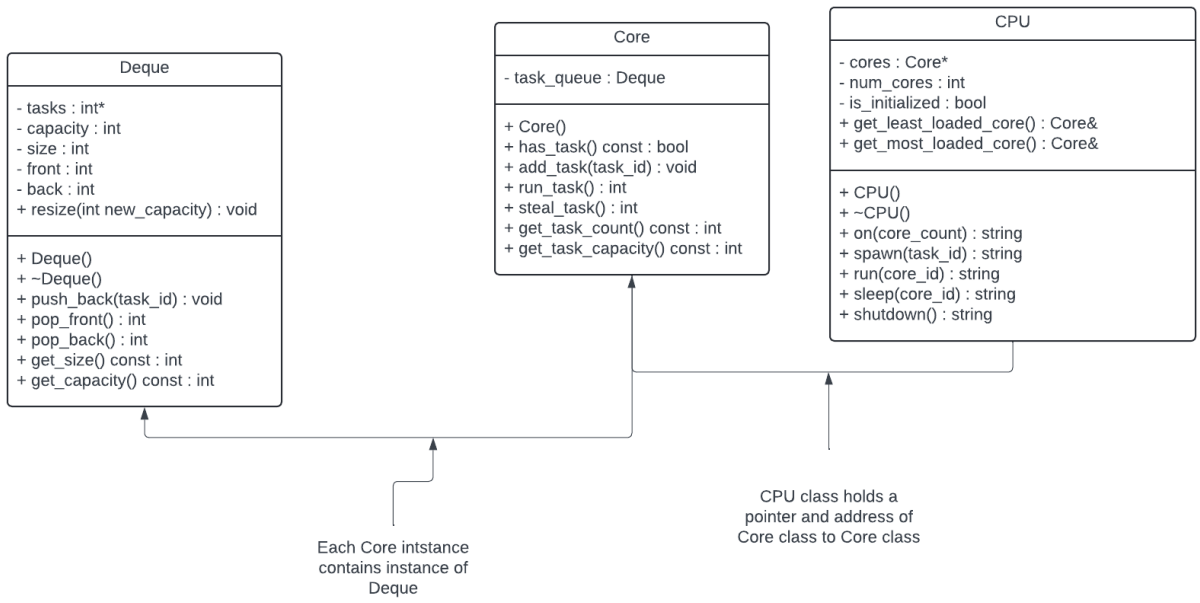
  **Core class:** Involves in a computational core that holds a deque of tasks assigned to it. Each core operates independently, making it easy to manage tasks and allows for straightforward task distribution strategies (like stealing tasks from overloaded cores).

  **CPU class:** As his class manages multiple cores and orchestrates task distribution and execution, a centralized CPU class facilitates core management, load balancing, and task assignments across cores, which is crucial for an efficient work-stealing mechanism.

- Private Attributes of Robot and Map
    - Deque object stores the tasks for the core. Making it private ensures that only the Core class can manage the task queue, preventing direct access by external objects since Deque class is declared in Core class as a private attributes.
    - ‘cores’ is a pointer to a dynamically allocated array of Core objects. Keeping it private, the CPU class ensures that cores pointer can only be modified

inside of the class, preserving the integrity of task distribution and execution logic.

**UML Design**




**Function Design**

- Key Functions:
    - void Deque::push_back(int task_id): Efficiently adds a task while managing dynamic resizing.
    - int Core::run_task(): Handles the execution of a task while maintaining the state of the task queue.
    - std::string CPU::spawn(int task_id): Implements a load balancing strategy by assigning tasks to the least loaded core.

**Runtime**

- It is generally O(1) runtime. For push_back() in Deque operations, it's average O(1) due to amortized cost of resizing and O(n) when resizing occurs. For run_task() and steal_task() in Core operations, it's O(1) for executing the next task and as it only involves pointer adjustments. For spawn() and run() in CPU operations, it's O(n) to find the least loaded core among all available cores and when attempting to steal tasks from the most loaded core, respectively.