1 Function Design

`resizeTasks`
params - `int newCapacity`
returns - `void`
Resizes the `tasks` array by `newCapacity` while preserving the elements. Assigns the value of `newCapacity` to `capacity`.

`pushFront, pushBack`
params - `int taskId`
returns - `void`
Pushes a new `taskId` the front/back of the deque. If the deque is full after pushing, resize by `capacity * 2`.

`popFront, popBack`
returns - `void`
Pops an existing `taskId` from the front/back of the deque. If `currSize` reaches $\frac{1}{4}$ of `capacity` after popping, resize by `capacity / 2`.

`findCoreIdWithLeastTasks, findCoreIdWithMostTasks`
returns - `int`
Returns the core id with the least/most amount of assigned tasks from the array of cores.

`findCoreIdWithLeastTasksExcluding`
params - `int C_ID`
returns - `int`
Returns the core id with the least amount of assigned tasks from the array of cores, excluding the specified `C_ID`.

2 Runtime

The
`findCoreIdWithLeastTasks`, `findCoreIdWithLeastTasksExcluding`, `findCoreIdWithMostTasks`
methods each have a runtime of `O(numCores)`, where `numCores` is the number of cores in the current CPU instance. Each method has a loop that runs `numCores` times to do element comparison and return value update. The computation per iteration is done in `O(1)`. Thus, the total loop is done in `O(numCores)`.

The `resizeTasks` method has a runtime of `O(C)` when the deque is full, where C is the capacity of the core's deque.
A `newTasks` array with `newCapacity` set to twice the original capacity is created (done in `O(1)` time).
Then it copies each element from the current `tasks` to `newTasks`, which executes a loop that runs `C` times to access each `tasks` element (done in `O(C)` time).
Then, it assigns the value of `newTasks` to `tasks`, and the value of `newCapacity` to `capacity` (done in `O(1)` time).
The dominant time complexity for this method is `O(C)`.

The `RUN` command has a runtime of `O(1)`, presuming no resizing occurs. The associated method is `runTask`.
Out of range: early return. Done in `O(1)` time.
Core is empty: find core id with most tasks (done in `O(numcores)` time). Steal work from that core if it is not empty (done in `O(1)` time). Early return.
Run next task: get the front task id from the deque, and pop the task. Done in `O(1)` time.
Core is empty after running task: find core id with most tasks (done in `O(numcores)` time). Steal work from that core if it is not empty (done in `O(1)` time).
Runtime: `O(numCores)` is the dominant time complexity. By asymptotic analysis, this is considered `O(1)`, since `numCores` is a constant value after CPU initialization.

The `SPAWN` command has a worst-case runtime of `O(C)`, where C is the capacity of the core's deque. The associated method is `spawnTask`.
Worst-case: find core id with least tasks, push to the back of the deque with resizing:
Finding the core id with the least amount of tasks is done in `O(numCores)` time.
Resizing the deque when full is done in `O(C)`.
`O(C)` dominates `O(numCores)` in asymptotic analysis, since `O(numCores)` is considered `O(1)` here.