Project 1 Design Document

1. CPU Class Design

1.1 Overview

The CPU class represents a multi-core processor capable of managing tasks across multiple cores.

1.2 Attributes

All attributes for the CPU class are private since they should not be altered or accessed directly by the user. Instead, they are manipulated and accessed by member functions.

- `coreCount` : An integer representing the number of cores.
- `cores` : A pointer to an array of Deque objects, each representing a core's task queue.

1.3 Functions

Functions were made public or private depending on if they are required to be callable by the user. Constructor and destructor are public. Each possible input command has a function associated with it; these functions are all public since they must be callable from main. All helper functions are private since they should only be callable from within the class, from other class functions.

1.3.1 Public Functions

- `CPU()` : Trivial constructor. Sets `coreCount` to 0 and `cores` to `nullptr` .
- `~CPU()` : Destructor. Loops through cores and calls the Deque destructor for each core. Deallocates the cores array.
- `on(int n)` : Initializes the CPU with n cores. Fails if cores already exists.
- `shutdown()` : Loops through cores and pops all tasks from each core, starting from the front. Fails if there are no tasks among all cores.
- `spawn(int taskId)` : Adds a task to the back of the queue of the core with the least tasks. Fails if the task ID is nonpositive.
- `run(int coreId)` : Pops the first task in the queue of the specified core. Fails if core ID is not valid. Does not pop a task if the core's queue is empty. If the core's queue is now empty, reassigns a task from the back of the queue of the core with the least tasks.
- `sleep(int coreId)` : Reassigns tasks from the specified core to the core with the least tasks, starting from the back of the deque. The target core is recalculated with every iteration.
- `size(int coreId)` : Returns the number of tasks in the specified core's queue.
- `capacity(int coreId)` : Returns the capacity of the specified core's queue.

1.3.2 Private Functions

- `getCoreWithLeastTasks(int exception)` : Loops through cores and checks the size of each core to find the core with the least tasks. Skips over the core with ID `exception` .
- `getCoreWithMostTasks(int exception)` : Loops through cores and checks the size of each core to find the core with the most tasks. Skips over the core with ID `exception` .

2. Deque Class Design

2.1 Overview

The Deque class implements a dynamic circular double-ended queue used to manage tasks for each core.

2.2 Attributes

All attributes for the Deque class are private to ensure encapsulation and prevent direct manipulation of the internal state.

- `front` : An integer representing the index of the front element.
- `back` : An integer representing the index of the back element.
- `size` : An integer tracking the current number of elements.
- `capacity` : An integer representing the current capacity of the array.
- `array` : A pointer to a dynamically allocated integer array storing the elements.

2.3 Functions

Functions are made public or private based on whether they need to be called from outside the class. The constructor, destructor, and main operations are public, while helper functions are private.

2.3.1 Public Functions

- `Deque()` : Constructor. Initializes an empty deque with initial capacity 4. Sets `front` to 0, `back` to -1, and `size` to 0.
- `~Deque()` : Destructor. Deallocates the dynamically allocated array.
- `pushBack(int val)` : Adds an element to the back of the deque. Increments size and calls `checkSpaceAndResize()` .

- `popFront()` : Returns the front element. Decrements size and calls `checkSpaceAndResize()` .
- `popBack()` : Returns the back element. Decrements size and calls `checkSpaceAndResize()` .
- `getSize()` : Returns the current number of elements.
- `getCapacity()` : Returns the current capacity of the deque.

2.3.2 Private Functions

- `resize(int newCapacity)` : Dynamically allocates a new array with the specified capacity. Copies all elements (front -> back) from existing array to the start of the new array. Deallocates existing array and updates pointer to the new array. Updates member variables `front` , `back` , and `capacity` .
- `checkSpaceAndResize()` : Checks size against capacity. If size has reached capacity, calls `resize()` with double the current capacity. If size has reached a quarter of capacity, calls `resize()` with half the current capacity.
- `positiveMod(int n)` : Handles circular indexing with positive modulo.

3. Runtime Analyses

This section proves the time complexities of the `run()` and `spawn()` functions to be $O(1)$, assuming that the number of cores is a constant (or $O(C)$).
Many of the member functions of the CPU and Deque classes used in `run()` and `spawn()` run in $O(C)$ or better:

- `Deque->getSize()` is $O(1)$ since it simply returns a member variable.
- `Deque->popFront()` , `Deque->popBack()` , and `Deque->pushBack()` are all $O(1)$; changing a value in an array and updating `size` are both constant-time operations.
- `CPU->getCoreWithMostTasks()` and `CPU->getCoreWithLeastTasks()` are both $O(C)$; getting member variables and comparing them runs in constant-time.

3.1 `run()`

The `run()` function is designed to be $O(1)$ time complexity, assuming the number of cores is constant.

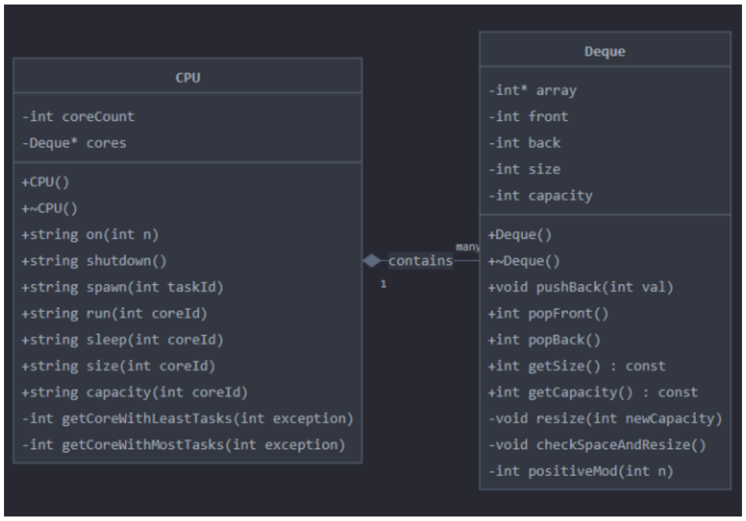
- Most operations in `run()` are $O(1)$. This includes core ID validation, checking if the core's size, popping and pushing tasks, and all boolean comparisons.
- The only exception is finding the core with the most tasks. As proven above, this is $O(C)$ and therefore dominates the runtime.
- There are no other loops or other operations that would contribute to a greater time complexity.
  Therefore, the overall time complexity of the `run()` function is $O(C)$, given the assumption that the number of cores is constant.

3.2 `spawn()`

The `spawn()` function is designed to be O(1) time complexity, assuming the number of cores is constant.

- Task ID validation and pushing a task to a core's deque are both $O(1)$.
- Finding the core with the least tasks. As proven above, this is $O(C)$ and therefore dominates the runtime.
- There are no other loops or other operations that would contribute to a greater time complexity.
  Therefore, the overall time complexity of the `spawn()` function is $O(C)$, given the assumption that the number of cores is constant.

UML Diagram


CPU
-int coreCount
-Deque* cores
+CPU()
+~CPU()
+string on(int n)
+string shutdown()
+string spawn(int taskId)
+string run(int coreId)
+string sleep(int coreId)
+string size(int coreId)
+string capacity(int coreId)
-int getCoreWithLeastTasks(int exception)
-int getCoreWithMostTasks(int exception)
contains
1
many
Deque
-int* array
-int front
-int back
-int size
-int capacity
+Deque()
+~Deque()
+void pushBack(int val)
+int popFront()
+int popBack()
+int getSize() : const
+int getCapacity() : const
-void resize(int newCapacity)
-void checkSpaceAndResize()
-int positiveMod(int n)