ECE250 P1: *Work Stealing*

1. Class Design

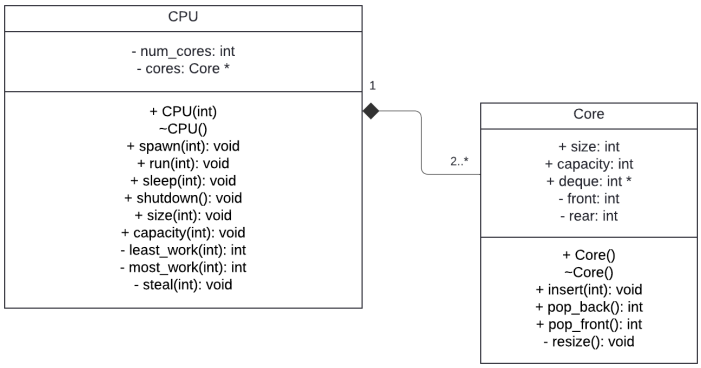
Class 1: *Core*

Purpose	The Core class plays a significant role in the core management of the CPU. It is used to encapsulate functions and store variables specific to a core.
Design Rationale	- *Efficient use of space*: A circular deque was implemented to handle core operations. This data structure reuses space by essentially connecting the back and front of the array. The program simply 'wraps around' to the beginning of the array to insert elements, allowing the queue to use available space efficiently, without needing to resize the array. - *Constant time operations*: Insertion and removal of elements at either the front/ rear of the array will take O(1), since shifting elements to perform operations is not required.
Variables	*Private:* int front, int rear - Manage the internal state of the core's task queue. Remains private to ensure external classes do not need to manipulate these pointers. All manipulation of these pointers must be done by the class's member functions. *Public*: int size, int capacity - They are public since CPU class needs to directly access them when handling specific commands like SIZE and CAPACITY.

Class 2: *CPU*

Purpose	Responsible for managing multiple cores, distributing tasks among them, and coordinating operations.
Design Rationale	- *Task distribution*: Designed the CPU class to act as a load balancer. It is responsible for distributing tasks dynamically, selecting the core with the least amount of work. - *Scalability:* Having a CPU class that maintains a global view of all available cores and workloads, ensures that if necessary, multiple CPU classes can run at the same time, if requirements change.
Variables	*Private:* int num_cores - Stores the number of cores available in the CPU. Declared as a private variable to restrict direct access from outside the class.

2. Unified modeling language (UML) diagram



3.Function Design

Core	**+ Resize(): void;** Streamlines array resizing by automatically checking and resizing after each insertion or deletion, eliminating the need for separate checks elsewhere. **- pop_front(); int, pop_back(): int; , insert(int): void** Implements basic deque functions, with pop returning an int for element access in the CPU class, and insert() returning void, while insert_front() is omitted as unnecessary.
CPU	Each input command has an allocated function. **- least_work(int): int, -most_work(int):int** Returns the index of the CPU with the least/most work, excluding the specified core (passed through parameter), and evaluates all cores if no index is passed, enabling reusable and encapsulated functionality. **- steal(int): void** Implements work-stealing by transferring tasks from the busiest core to the specified core, with a void return type for task redistribution.

4. Runtime

The RUN function includes comparisons, task stealing, and pop_front(). Task stealing has 1 comparison, pop_back(), insert(), and most_work().

$Let\ T_1\ be\ the\ constant\ time\ for\ comparisons, pop\_front(),\ pop\_back(),\ insert(),\ assuming\ no\ resizing$
$Let\ T_2\ be\ the\ constant\ time\ for\ each\ iteration\ in\ most\_work()$

$$f(n) = T_1 + C\,T_2,\ C = \#\ of\ cores$$

$Since\ we\ have\ an\ initial\ fixed\ number\ of\ cores,\ ,$

$$f(n) = T_3,\ (T_3\ is\ constant\ time)$$

$\therefore f(n) = O(1)$

The SPAWN function includes 1 comparison, determining core with least work, and resizing if necessary.

$Let\ T_1\ be\ the\ constant\ time\ for\ comparisons,\ and\ least\_work()$
$Let\ T_2\ be\ the\ constant\ time\ for\ each\ iteration\ in\ insert()$
$Worst\ case\ runtime\ for\ insert(),\ includes\ resizing\ the\ cores\ (iterate\ and\ copy\ tasks\ in\ deque)$

$$f(n) = T_1 + C\,T_2,\ C = \#\ of\ tasks\ in\ core$$

$Since\ C\ depends\ on\ input\ size,$
$\therefore f(n) = O(C)$