Class Design

For this project, I decided to use 3 classes based on how the project guideline seemed to separate tasks into 3 parts: a deque, core, and cpu class. A cpu class can have multiple cores, and each core contains a deque. I could have combined the core and deque classes but I found how deque was fundamentally a concept itself so keeping them seperate allowed me to implement the deque easily and then add the core functions on top of it.

The deque was made using a dynamic circular array to store integers in a buffer that can be accessed from the front and back. When full, it doubles capacity. When 1/4, it halves capacity.
int* arr: dynamically allocated circular array
int cap: current capacity
int size: current number of tasks
int front: index of the logical front

Cores each contain a deque and wrap around it to implement tasks like stealing.
int id: the core's index
dq dq1: the core's task queue

CPU manages all the cores, using a fixed array of core pointers to objects in core. Handles all the work stealing logic here as this is the control center or "brain" for the cores.
core cores**: array of dynamically allocated cores
int num: number of cores
bool active: whether the CPU has been initialized with ON

UML Diagram

cpu	core	dq
- cores : core** - num : int - active: bool	- id : int - dq1 : dq	- arr : int* - cap : int - size : int - front : int
+ cpu() + ~cpu() + on(n:int) : bool + off() : void + spawn(pid:int, cid:int&) : bool + run(cid:int, pid:int&, empty:bool&) : bool + sleep(cid:int) : bool + shutdown() : bool + size(cid:int, out:int&) : bool + capacity(cid:int, out:int&) : bool + isOn() : bool	+ core(i:int) + getId() : int + addTask(pid:int) : void + runTask() : int + stealTask() : int + taskCount() : int + getCap() : int + hasTasks() : bool	+ dq(startCap:int=4) + ~dq() + pushBack(x:int) : void + pushFront(x:int) : void + popBack() : int + popFront() : int + getSize() : int + getCap() : int + isEmpty() : bool - resize(newCap:int) : void

Function Design

Deque key functions:
pushBack, pushFront: Add tasks at either end
popBack, popFront: Remove tasks while maintaining circular indexing
resize(): Doubles or halves capacity depending on usage

Core key functions:
addTask(int): push to back of the deque
runTask(): pop from the front
stealTask(): pop from the back when another core is idle
taskCount() and getCap(): provide statistics for load balancing

CPU key functions:
findLeast(): decides where to assign tasks by finding the least-loaded core
findMost(int skip): chooses where to steal from by finding the most-loaded core
sleep(int cid): redistributes tasks one by one, recomputing least load each time
shutdown(): removes all remaining tasks in core order (0 to N−1) to match the spec
spawn(int pid, int &cid): returns true/false to signal validity (instead of throwing errors)
run(int cid, int &pid, bool &empty): passes back both the task ID and a flag if core was empty

Runtime Analysis

Deque push/pop	O(1)	Uses modular arithmetic indexing, constant work
Deque resize	O(C) (C = old cap)	Copies all elements when capacity changes
Spawn	O(C) worst case (C = old cap)	Usually O(1), but resizing makes it O(C) in the worst case
Run	O(1)	Pops a single element, optional steal involves constant work (no loops)
Sleep	O(T × N) worst case	For T tasks and N cores, each reassignment recomputes least-loaded core
Shutdown	O(A) (A = all tasks)	Pops and prints every remaining task once

[1] OpenAI, "ChatGPT (GPT-5)," *chat.openai.com*, accessed Oct. 5 2025.
Spelling, grammar & portions of UML description were corrected with assistance from ChatGPT.