Approach Rationale

I chose a double-ended queue implemented using a dynamic array that allows for dynamic resizing using a circular buffer over a linked list implementation for the following reasons:

A dynamic array allows for efficient insertion and deletion/removal or tasks from both the front and back ends of the queue, satisfying requirements of this project.

The circular buffer allows O(1) insertions and deletions as required for RUN command.

This implementation could have been done using a linked list approach, however, this would have required much more complicated memory management and lead to slower access times. It also would have increased the cost as a linked list will require managing linked nodes and pointers.

Class Design

Class cpu

This class was designed to hold the functions responsible for assigning, running, and redistributing tasks as well as shutting down cores.

Public variables and functions:

- **void spawn_task(int P_ID); //function to add task P_ID to queue of least busy core**
  Parameters: int P_ID
  Return Type: n/a
  Rationale for design: This function takes in task P_ID, if it is valid, it uses a for loop to go through the cores in the cpu instance and find the one with the least amount of tasks. It then pushes the task to the back of this cores queue using the push_back() function.
- **void run_task(int C_ID); //function to run task from specific core C_ID**
  Parameters: int C_ID
  Return Type:n/a
  Rationale for design: This function takes the first task in a cores queue and runs it if valid. It then checks if the size of the core is 0, if yes, it uses a for loop to find the core with the most tasks in its queue and pushes the back most task of the busiest core to the current empty core using the push_back() function.
- **void sleep_core(int C_ID); //function to reassign core C_ID tasks and make it sleep**
  Parameters:int C_ID)
  Return Type:n/a
  Rationale for design: This function checks if the core is valid, if it's empty there will be no tasks to reassign. If the core is not empty, it takes the back most task and uses a for loop to find the core with the least tasks. push_back() is then called to push the task to the least busy core.
- **void shutdown(); //function to clear tasks for all cores**
  Parameters: n/a
  Return Type:n/a

Rationale for design: This function uses a for loop to iterate through the cores, the while loop then allows the front task to be popped off as long as it exists. If for any core, the size is less than or equal to 0, there are no tasks to remove.

Private variables and functions:

- **int number_cores;**
  Should not be able to manipulate from a user standpoint
- **deque* cores;**
  Pointer also must not be manipulated by users

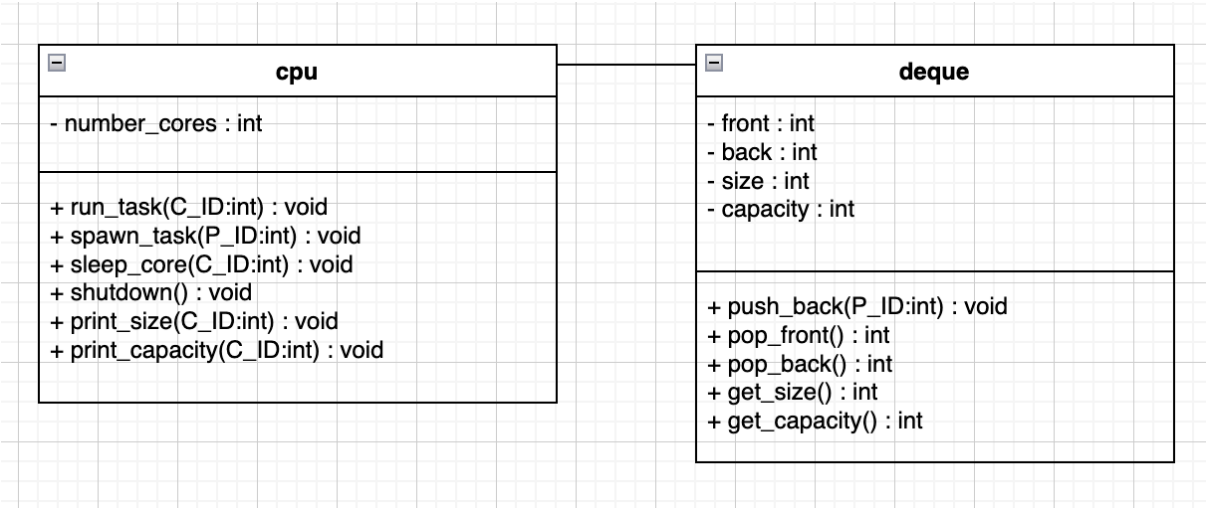
Class deque

This class will manage the array of tasks for each core. It will handle the resizing of the array as well as adding and removing tasks from both ends (front and back) of the queue of tasks.

Public variables and functions:

- **void push_back(int P_ID);**
  Parameters: int P_ID
  Return Type:n/a
  Rationale for design: This function puts task P_ID at the back of the queue, uses circular indexing then updates the size of the queue. Lastly, it checks if capacity needs to be doubled.
- **int pop_front(); //function to remove task from front and return it**
  Parameters: n/a
  Return Type:int
  Rationale for design: This function takes the task at the front of the queue, uses circular indexing and updates the size. Lastly, it checks if the capacity needs to be halved and returns the P_ID of the task that was popped off the front.
- **int pop_back(); //function to remove task from back and return it**
  Parameters: n/a
  Return Type:int
  Rationale for design: This function uses circular indexing, takes the task at the back, updates size and then checks if capacity needs to be halved. The P_ID is returned as an integer.
- **void resize(int new_capacity); //function to resize array as required**
  Parameters: n/a
  Return Type:n/a
  Rationale for design: This function copies the current contents of the array to a new one using a for loop and circular indexing, it then deletes the old array and updates the necessary values.

UML Diagram



Runtime Analysis

RUN command

Runtime is O(1) because the queue will have tasks in it, the task to run will simply be popped off the front making runtime O(1). We are also told to treat it as O(1) since the number of cores (N) is constant.

SPAWN command

Best case: O(1)

This is the best case when a task is added to the back of a task queue without resizing.

Worst case: O(C)

The number of cores is constant meaning any operation that involves iterating over these cores is O(1), assuming N<<C, N=O(1).

The worst case for the SPAWN command is when the array needs to be resized, resizing takes time O(C) as tasks will need to be copied to a new task array. Therefore the worst time is O(C).