Project 1 Design Document

Kelly Kim
ECE 250
7 October 2024

This is the design document for ECE250 Project 1 Work Stealing.
Using a single header file (cpu.h) for the project keeps things organized and straightforward, reducing complexity. Since the components are closely related, multiple header files can create confusion about which methods interact with which member variables. Additionally, this approach can lead to faster compilation times, as the compiler processes fewer files.

Public:

- CPU(int n)
- ~CPU()

Both constructor and destructor are public because it allows the user to create and manage CPU objects. The constructor allows the user to create instances of CPU and the destructor cleans up the resources when they are done using.

- void on (int n)
- void spawn (int p_id)
- void run (int c_id)
- void sleep (int c_id)
- void shutdown()
- void size (int c_id)
- void capacity (int c_id)
- bool exit();

By making these methods public, users will be able to easily manage tasks by adding, running, or putting cores to sleep. It allows them to monitor the state and control how all the tasks and cores work.
On function will have a void return type printing out “success” if the CPU instance has been created.
Spawn/Run functions will have a void return type because they will be printing where the task went/task running instead of returning any values.
Sleep function will have a void return type because it will not be returning, but printing where the tasks in the given core are getting moved when it is put to sleep.
Shutdown function will have a void return type because it will be printing the deleted tasks from the core without returning anything.
Size/Capacity functions will have a void return type, not returning but printing the size/capacity of the given core.
Exit function will be used to control when the program stops. Returning true for the program to exit, returning false for the program to keep going.

- bool exitsign = true;

This variable changes to false when the exit function is called to exit the program.

- void resize (int coreNum)
- int minTask (int coreNum)
- int maxTask (int coreNum)

Resize function will be used to resize the core’s capacity if the core’s size goes beyond or below a certain value of the capacity. It will be a void return type because it does not need to return anything to the user, it just modifies the state of the object.

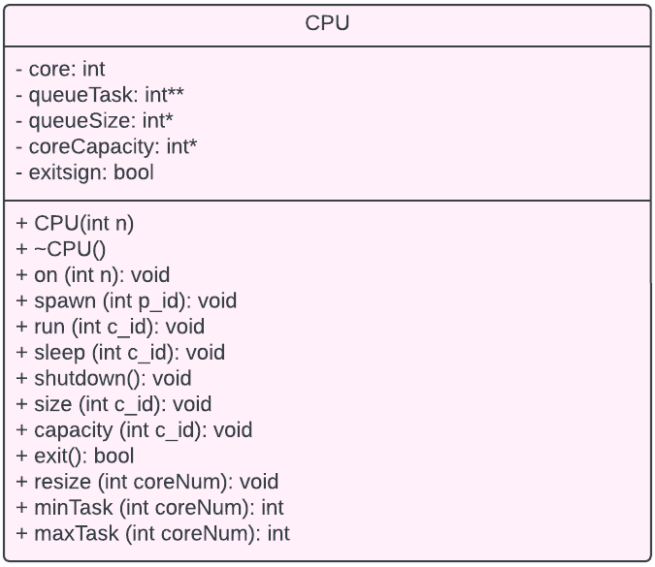
minTask function will find the core with the lowest amount of tasks, which will be later used in the sleep function. It will have an int return type because it will be returning the core number with the lowest task.
maxTask function will find the core with the highest number of tasks, which will be used in the run function. It will have an int return type because it will be returning the core number with the highest task.

Private:

- int core
- int** queueTask
- int * queueSize
- int* coreCapacity

Core will be representing the number of cores there are. It will be private to ensure that it can only be accessed through public member functions as well as to have more control over how the variable is used.
queueTask will be private to make sure that the tasks are managed solely within the class itself. By restricting access, the user would have to go through other public functions to add tasks, etc.
queueSize/coreCapacity will be private so no external codes can change these variables. All changes must go through public functions, allowing more control of the system.

UML DIAGRAM:



Running Time:
Run function (assuming resize function does not get called) first checks the bounds and according to its condition, it either runs the task and moves every task in the queue forward or it does not run anything at all. If they are moving the tasks forward, this will happen through for loops which will iterate through C times (C=number of cores). Since the number of cores are constant, the run time for the run function would be O(1).
Spawn function first checks the bounds and according to its condition, it will either assign a task to the core or print failure. If it is assigning a task to the core, it will try to find the core with the lowest task using a for loop, performing constant-time operations during each iteration. The total time complexity of the loop can be expressed as $number\ of\ core \times O(1) = O(number\ of\ cores)$. As the loop iterates, even if the queue size of the core reaches C (C=capacity of the core's queue), it still checks their sizes up to maximum capacity. As the capacity of the core's queue starts to increase to a very large number (∞) checking the capacity of the core's queue is more significant than checking the size. Therefore, the runtime of the spawn function becomes O(C) (C=capacity of the core's queue).