ECE250 Project 1: Work Stealing

Due Date: Monday, October 7, 2024 @11:00pm to the Dropbox on Learn

Overview

The goal of this project is to implement a deque class to simulate a multi-core CPU. You must create your own deque class for this project, using proper object-oriented design principles for the rest of the project. **You are not permitted to use any classes from the STL C++ library in this project[1].**

Understanding Work Stealing

A common type of system is one in which multiple equivalent "workers" have to process a stream of incoming tasks. Workers are assigned tasks, and they maintain a queue of tasks they must work on. When a new task is available, it is assigned to the worker who has the least work to do. As workers work on their tasks, they remove them from the front of their queue. If a worker runs out of tasks to do, it finds the worker with the most tasks and "steals" a task from the back of their queue. By doing this we can ensure that workers always have something to do unless there are no tasks available for anyone, and we ensure the work is split as evenly as possible. The implementation of work stealing algorithms such as this one are very well suited to using deques.

Important Technical Note: In reality, both workers and tasks are not exactly equivalent in some work stealing systems such as a modern CPU – tasks may be higher or lower priority whereas workers may be more or less capable. We are ignoring this complication as: (1) it makes the project much harder and (2) there are lots of work stealing systems in which workers and tasks *are* equivalent[2].

Required Deque Functionality

In this project, you must implement your deque as a resizable array of integers (where each integer represents the ID of a task to be performed). The deque has a "capacity" (the total number of integers that can be stored in it) and a "size" (the number of integers currently stored in it). Whenever size reaches capacity, the capacity must be doubled and the array resized, with all elements being copied to the new array. Whenever size reaches ¼ of capacity, the capacity must be halved and the array resized, with all elements being copied to the new array. The minimum capacity is 2.

Program Design and Documentation

You must use proper Object-Oriented design principles to implement your solution. You will create a design using classes which have appropriately private data members and appropriately public services (member functions). It is not acceptable to simply make all data members public. **Structs are not permitted.** All input validation must be handled within your class(es), not your main function.

You are required to write documentation for your design according to the instructions posted to Learn.

Input/Output Requirements

You must create a cpp file that contains your main function. This program must read commands from standard input and write to standard output. The program must respond to the commands shown in Table1 below. The outputs listed in the "Output" column must appear as shown. For instance, the first row has "success" as an output. This means that the string "success", written in lowercase, is expected, followed by a newline.

[1] std::string and all of <iostream> are permitted

[2] Such as the ece-nebula GPU compute cluster. Contact mike (mstachow@uwaterloo.ca) if you are interested in working with our free-to-use GPUs, set up for undergrads!

Table 1: Testing Commands

Command	Parameters	Description	Output
ON	N	Creates a new CPU instance with N cores. Cores will be assigned indices from 0 to N-1. Initialize all queue capacities to 4. You may assume that all input files always begin with a valid ON command with N>1.	**success** if no CPU instance has been created yet **failure** if a CPU instance already exists.
SPAWN	P_ID	Spawns a new task to run with ID given as the strictly positive integer P_ID. Assign it to the core with the lowest amount of work. In case there is a tie, assign it to the core with the smallest index.	**core i assigned task P_ID** where i is the index of the core to which the work was assigned **failure** if P_ID is nonpositive
RUN	C_ID	Runs the next task of the core with ID given by C_ID. This means that this core pops the next task from the front of its queue, if possible. If the queue is empty or If this core's queue is empty after running its last task, the CPU should assign it work from the core with the largest amount of work to do by popping that core's job off the back of its queue and inserting it into the queue of core C_ID. In case there is a tie for cores with most work to do, steal work from the core with the smallest index. The core does not run this stole task yet.	**core C_ID is running task P_ID** where P_ID is the P_ID of the task, stored in the queue **core C_ID has no tasks to run** if the core's queue is empty **failure** if C_ID is outside of the range [0, N-1]
SLEEP	C_ID	Puts core C_ID to sleep by distributing its remaining tasks to the remaining cores. It does so by assigning tasks to the core with the least amount of work repeatedly until all tasks are reassigned. Tasks are removed from the back of the queue. Each time a task is reassigned, the core with the least work is recomputed. Nothing special needs to be done to sleeping cores. If a new SPAWN command subsequently assigns a task to it, for instance, it will enqueue that task and is capable of running it. This is how a core "wakes up".	**task P_ID0 assigned to core C_IDj task P_ID1 assigned to core C_IDk…** where P_ID* are the task IDs being reassigned and C_ID* are the cores to which they are assigned **core C_ID has no tasks to assign** if core C_ID's queue is empty **failure** if C_ID is outside of the range [0,N-1]
SHUTDOWN		Pops all remaining tasks off of all remaining queues, starting with the lowest index core and going in order. It is possible to have commands after this command and they will run as normal. Imagine that the CPU has been shutdown by this command, then some time passes, and then it has been	**deleting task P_IDx from core C_IDj deleting task P_IDy from core C_IDj deleting task P_IDz from core C_IDk…** where P_ID* are the task IDs being deleted and C_ID* are the cores from which they are deleted **no tasks to remove** if no core has any remaining tasks

		powered on and more commands are run.	
Command	**Parameters**	**Description**	**Output**
SIZE	C_ID	Prints the number of tasks currently in the queue of core C_ID	**size is S** where S is the number of elements in the queue if C_ID is valid **failure** if C_ID is not in the range [0,N-1]
CAPACITY	C_ID	Prints the current capacity of the queue of core C_ID	**capacity is C** where C is the capacity **failure** if C_ID is not in the range [0,N-1]
EXIT		Final command of all input files	This command produces no output

The expected runtime of the RUN command is O(1) presuming no resizing occurs. SPAWN must have a worst-case runtime of at most O(C) where C is the capacity of the core's queue. Prove that these runtimes are met in your design document.

Valgrind and Memory Leaks

5% of the grade of this project will be allocated to memory leaks. We will be using the Valgrind utility to do this check. The expected behaviour of Valgrind is to indicate 0 errors and 0 leaks possible, with all allocated bytes freed. To test your code with Valgrind, presuming you are using an input file test01.in, you would use the following command:

```
valgrind ./a.out < test01.in
```

Test Files

Learn contains some sample input files with the corresponding output files. The files are named test01.in, test02.in and so on with their corresponding output files named test01.out etc.

Submitting your Program

Once you have completed your solution and tested it comprehensively on your own computer or the lab computers, you must transfer your files to the eceUbuntu server and test there. **A makefile is required for this project since the exact source structure you use will be unique to you.** We perform automated testing on this platform, so if your code works on your own computer but not on eceUbuntu it will be considered incorrect.

Once you are done your testing you must create a compressed file in the tar.gz format, that contains only the following:

- A typed document, **maximum of two pages**, describing your design. Submit this document in PDF format. The name of this file should be xxxxxxxx_design_pn.pdf where xxxxxxxx is your maximum 8-character UW user ID and n is the project number.
- A cpp file containing your main() function. Your class implementations may not go into this file.
- Required header files that you created and any additional cpp files that you created.
- A makefile, named Makefile, with commands to compile your solution and create an executable. This makefile will be run using the "make" utility. Do not specify an output file name, the default of a.out is used in the automated testing.

The name of your compressed file should be xxxxxxxx_pn.tar.gz, where xxxxxxxx is your UW ID as above and n is the project number.