Work Stealing Dequeue Project discussion

Discussion on: ⏰17/4/22 5.45pm ~ 6.00pm
Aim:
In this document, we aim to talk about Work Stealing Dequeues in general and how we have planned out the project on a high level. We will talk about work distributions and how work should be distributed for maximum performance.

Why work stealing dequeues are needed?

It is a well accepted fact that if work in a multi-processor system is not distributed properly, then we would not be gaining the maximum efficiency. Two ways in which we do this is

1. Work Yielding
2. Work Stealing

In the case of work yielding, loaded processes/threads will use precious clock cycles to check if some other process/thread can take some work from them. Obviously, this is inefficient. Doing so would increase the load of loaded processors and, if all processes are busy, would not be fruitful.

Therefore, the commonly accepted approach is work stealing, where a thread which is not loaded will steal some work from a thread/process which is loaded. In order to do this, we use a *linearizable Data structure* called the **Work Stealing Dequeue** (*linearizable Dequeue* used for managing tasks and allowing for Work Stealing).

High Level structure of the project

DEFINING THE TASK:

We have created a class for the task. This class will mimic the effect of an independent task on the

```
class Task
    {
        std::exponential_distribution<> m_waiting_time;
        std::default_random_engine m_seeder;

    public:
        /**
         * @brief Construct a new Task object
         *
         */
        Task()
        {
            // 3.7 is just a randomly chosen number for init
            m_waiting_time = std::exponential_distribution<>(3.7);

        }

        void run()
        {
            std::this_thread::sleep_for(time);
            // time is expected execution time of the task
        }
    };
```

THE ABSTRACT DATA STRUCTURE:

The Work Stealing DEQueue has mainly 3 APIs.

1. `pushBottom`
2. `popBottom`
3. `popTop`

Notice how there is no `pushTop` API. This is because we do not need it for this task. New Tasks are added through the `pushBottom` API. Threads take tasks it has added through the `popBottom` API and the '*stealing*' happens through the `popTop` API.

```
class WSDequeue
{
public:
    virtual void pushBottom(std::shared_ptr<task::Task> task) = 0;

    virtual std::shared_ptr<task::Task> popBottom() = 0;

    virtual std::shared_ptr<task::Task> popTop() = 0;
};
```

BOUNDED WORK STEALING DEQUEUE:

In this implementation, the runnable tasks are picked up from a finite Queue where a threads can pick them. The idea here is to use `compareAndSet` function in case of stealing. The purpose of this is that each task having its unique stamp value. Now consider one thread trying to steal any task from other thread, at any index suppose 5. But for some reason this thread gets stalled and other thread steals this task and the top becomes 4. The victim adds a new task so the top again becomes 5. Now if the initial thief comes back to steal the task at index 5, the former task residing at index 5 is gone and new task sits there. Hence here the stamps are helpful in distinguishing that this task is different than the task it was initially trying to steal.

`PopTop` works by checking if there is only one element in the queue, if there are more, it simply pops the task and increases the top. `PopBottom` works by checking if there is only one element in the queue, if there are more, it simply pops the task and decreases the bottom. If there is only one task in the queue the `PopBottom` of the thread and `PopTop` from other thread might collide and hence the use of `compareAndSet` is made to reset the bottom to 0 as the queue is empty now.

UNBOUNDED WORK STEALING DEQUEUE:

In this implementation, there is no need to use stamps for preventing the problem described in the bounded implementation. Also there is no need to reset the bottom to zero once the queue is empty. The `pushBottom` method on finding that the queue is full, creates a bigger array and copies the tasks into the new array and the indexing is done on basis of modulus of the size of array. In case of only one task left in the array, the `popBottom` first decrements bottom and now top and bottom become equal. If `popBottom` finds out that the bottom and top are equal , the top is incremented by 1 by either the `popTop` or `popBottom`. The winner takes the task and finally the bottom is incremented. Now the top and bottom are equal hence the queue is empty.