1 Executor use

The idea of this executor is the following. We have many tasks to execute. Some of them can be executed in parallel, while some subset of them have to be executed sequentially. We may execute in parallel tasks from two different subsets, but within each subset, no two tasks should be executed in parallel. To achieve this, we can use a `SerialExecutor` per subset.

All those serial executors have the same parent executor (think CPU thread scheduler for instance). Then each task of a subset can be added to the serial executor corresponding to this subset, and the serial executor will make sure that all tasks that were added to it are executed sequentially.

Example use case : We have a server with multiple sockets. We want to make sure that all tasks reading or writing to the same socket are executed sequentially (so as not to have two threads writing to the same socket at the same time). This can be achieved easily by having a serial executor per socket, and adding all tasks related using a socket to its corresponding serial executor.

2 Implementation

```
class SerialExecutor{
  Executor parent_;
  size_t scheduled_; //atomic
  UnboundedQueue queue_; //multiple producers single consumer

  SerialExecutor(parent){
    parent_ = parent;
    queue = newQueue();
        scheduled_ = 0;
  }

  void add(Func func){
    queue_.enqueue(func);
    parent_->add({this->run()});
  }
  void run(){
    if(scheduled_.fetch_add(1, acquire) > 0)
      return;
    do {
      Func func = queue_.dequeue();
      func();
    } while(scheduled_.fetch_sub(1, release) > 1);
  }
}
```

The `SerialExecutor` has a queue of tasks. Whenever a new task is added through the method `add`, the task is added to this queue, and we add to the parent executor the task of running the serial executor.

When the serial executor is run (through the method `run`), it first increments `scheduled_`. If `scheduled_` was 0, then it is the only thread runnning this serial executor at the moment, so it can starts taking elements from the queue and executing them one after the other. It keeps doing so until it runs out of elements to work on (that is to say when `scheduled_` becomes 0). Note that `scheduled_` is incremented only after an element has been added on the queue, and only one thread can pop elements from the queue before decrementing `scheduled_`, making sure we never try to take elements from an empty queue.

The release and acquire synchronization on reading `scheduled_` ensure synchronisation between the different threads executing task added to this serial executor.

3 Some properties

What we could try to prove about this example:

- If we model the functions `func` added to the queue as simply requiring full access to some variable `x`, we can check that synchronization is enough to send `acc(x, write)` from on task to another;

- We can also check that only one thread can take elements from the queue (another `acc(queueAsConsumer, write)` access ?)

- Using some specifications about the queue, trying to prove that all elements are executed (queue is multiple producers/single consumer thread safe).

4 Another atomic location

The full implementation uses a second atomic location `keepAliveCounter` that is used to monitor when can the serial executor safely be deleted. I removed it here to first have a proof without.