Bounded object pool

Довольно часто в lock-free алгоритмах возникает необходимость в пуле некоторого фиксированного объема. Раз алгоритм lock-free, то и пул тоже должен быть по крайней мере lock-free. Пул — это не очередь и не стек, хотя эти структуры данных могут быть пулом. Пул — это класс с двумя интерфейсными функциями:

- `T * get()` - возвращает любой объект из пула или `nullptr`, если пул пуст
- `bool put( T * )` - помещает объект в пул, возвращает `true`, если успешно (пул не переполнен) или `false`, если пул полон.

Как правило, конструктор пула заполняет его некими объектами. В этом существенное отличие от классической очереди или стека, которые создаются пустыми. И вообще, нормальное состояние пула — «пул полон», тогда как для очереди - «очередь пуста».

Пул применяется для хранения некоторых преаллоцированных ресурсов (вспомогательных объектов), которые разделяются между потоками. Например, монитор синхронизации может иметь пул mutex'ов, которые используются в реализации lock-based fine-grained дерева. Эти mutex'ы используются для синхронизации доступа к вершинам дерева при вставке/удалении. Число узлов дерева может быть огромно, создавать для каждого узла свой mutex нерационально (не забудем, что mutex – системный ресурс, число которого может быть ограничено), тогда как число одновременно работающих с деревом потоков невелико, то есть в каждый момент времени нам нужно не 1 млн mutex'ов, а 10 — 20. Здесь может быть использован пул mutex'ов: мы берем из пула mutex, прикрепляем его к узлу дерева, лочим, делаем модификацию поддерева, разлочим и возвращаем mutex в пул.

В настоящее время (libcds 2.1.0) в качестве пула в библиотеке используется алгоритм bounded очереди Дмитрия Вьюкова. Он быстр, но имеет один существенный недостаток — он не является линеаризуемым: при почти полной очереди `push()` может быть неудачным, хотя место в очереди ещё есть. То есть эта очередь не может стабильно работать в режиме «пул полон».

Требуется: найти/придумать и реализовать быстрый алгоритм lock-free/wait-free bounded pool:

```
template <typename T>
class Pool
{
    size_t capacity_;
public:
    Pool( size_t capacity ): capacity_( capacity ) {}
    T * get(); // return nullptr if pool is empty
    bool put( T * obj );  // return false if pool is full

    bool empty() const;
    bool full() const;
    size_t capacity() const { return capacity_; }
};
```

Методы `get()` и `put()` не должны распределять памяти.

Критерий корректности: реализация должна успешно пройти такой тест на успешную работу в состоянии «пул почти полон» (псевдокод):

```
Pool<void *> pool(256);
std::atomic<size_t> nGetError(0);
std::atomic<size_t> nPutError(0);

void thread_func()
{
     for ( int i = 1; i <= 1000000; ++i ) {
          if ( !pool.put( &i ))
               nPutError.fetch_add(1, std::memory_order_relaxed);
          void * p = pool.get();
          if ( p == nullptr )
               nGetError.fetch_add( 1, std::memory_order_relaxed);
     }
}

void main()
{
      size_t const thread_count = 16;

      size_t const initial_size = pool.capacity() - thread_count;

      // initialize pool
      for ( size_t i = 1; i <= initial_size; ++i )
            pool.put(  &initial_size /* put anything */ );

      // run working threads
      for ( int i = 0;  I < thread_count; ++i )
          run_thread( thread_func );
      wait_for_all_threads_done();

      // no put/get error
      assert( nPutError.load() == 0);
      assert( nGetError.load() == 0 );
}
```