**Пояснения**.

Честно говоря, сама формулировка задачи мне показалась странной. Конечно, можно реализовать метод, вставляющий в список узел с заданным номером, но если одновременно могут выполняться несколько потоков, реализующих вставки и удаления, то понятие номера теряет всякий смысл. Чтобы это предотвратить, нужно, чтобы каждый поток блокировал весь список, но тогда непонятно, при чем здесь гранулярность.

По этой причине я реализовал список, имеющий два метода.

* Метод удаления узла, условие нахождения удаляемого узла может быть любым.
* Метод вставки узла, условие нахождения позиции для вставки может быть любым.

**Описание решения**.

Код решения содержится в mutex\_list.cpp. Объект класса Node<T> представляет узел списка, каждый элемент содержит зачение типа T. Сам список представлен классом List<T>.

Метод insert вставляет в список узел, содержащий значение value типа T, found – предикат, задающий условие поиска позиции для вставки, если условие выполнено, то новый узел помещается перед найденным.

Метод remove удаляет из списка узел, параметр value и предикат found задают условие поиска удаляемого узла, если узел найден, то он удаляется, и метод возвращает true, в противном случае возвращает false.

**Аргументация в пользу потокобезопасности**.

Попробую объяснить, почему я считаю этот код потокобезопасным.

Сначала рассмотрим наиболее общий случай, т.е. тот случай, когда список разблокирован. В этом случае может удаляться узел, не являющийся начальным, и может проиводиться вставка узла перед существующим узлом. Заметим следующее.

* При удалении узла блокируется удаляемый узел и предшествующий ему узел. Модифицируется узел, предшествующий удаляемому.
* При вставке блокируются оба узла, между которыми выполняется вставка. Модифицируется узел, после которого выполняется вставка.

Пусть поток пытается удалить узел B (см. рис. 1). Важно, чтобы во время выполнения операции не был удален C или модифицирован A. Для удаления C нужно заблокировать С, который заблокирован, также невозможно модифицировать A, т.к. A также заблокирован.

A

C

B

Рис. 1. Удаление узла.

Рассмотрим вставку узла между A и B (см. рис.2). Важно, чтобы во время выполнения этой операции не были модифицированы или удалены A и B, но их удаление или модификация невозможны, т.к. они оба заблокированы.

A

B

N

Рис. 2. Вставка узла.

Расмотрим теперь тот случай, когда заблокирован список. В этой ситуации может выполняться только другой поток, не блокирующий список. Поток, заблокировавший список, может выполнять удаление начального узла или вставку узла перед начальным. Заметим следующее.

* При удалении начального узла блокируется список и начальный узел, модифицируется список.
* При вставке узла перед начальным узлом блокируется список и начальный узел, модифицируется список.

List

B

A

Рис. 3. Удаление начального узла.

Расмотрим удаление начального узла A (см. рис. 3). В этой ситуации список заблокирован и не может быть модифицирован другим потоком. B (если он существует) также не может быть удален, т.к. для его удаления нужно заблокировать A, но A заблокирован.

List

A

N

Рис. 4. Вставка начального узла.

Рассмотрим вставку начального узла. В этой ситуации список не может быть модифицирован, т.к. он заблокирован. A также не может быть удален, т.к. он заблокирован.

Очевидно также, что при добавлении узла в конец списка последний узел также не может быть удален или модифицирован, т.к. он заблокирован.

**Описание теста**.

Размеется тест не может гарантировать потокобезопасность решения, но все же позволяет предполагать, что решение работает корректно. И вообще, доказательства доказательствами, а протестировать надо.

В тесте создается ситуация, в которой в список различными потоками постоянно добавляются и удаляются узлы, причем позиция добавлния узла узла и позиция удаления узла выбирается случайным образом. Если размер списка растет несильно, то возрастает вероятность возникновения конфликтов между потоками за обладание списком в целом или его отдельными узлами.

В тесте используетcя List<int>. Условием вставки является то, что следующий узел содержит значение, не меньшее нового значения, таким образом список оказывается упорядоченным по возрастанию.

Функция add\_Nodes вставляет в список numb случайных чисел в диапазоне от 0 до 100. Поскольку значение числа случайно, то позиция вставки в список также оказывается случайной. Каждое вставленное в список значение случайного числа помещается в пул.

Функция remove\_Nodes удаляет из списка numb узлов с заданным значением. Значение выбирается из пула, удаленное значение удаляется из пула. Если пул пуст, то выбирается значение -1, заведомо отсутствующее в списке, и делается попытка удалить отсутствующий узел. Таким образом, каждый раз происходит удаление одного узла (если только пул не успел стать пустым), причем позиция удаляемого узла также оказывается случайной.

Пул значений, хранящихся в списке по существу представляет собой очередь. Тем не менее, я не стал использовать std::queue, чтобы операция помещения в очередь и операция извлечения из очереди выполнялись, как можно быстрее, я реализовал очередь в виде простого массива. В этом случае помещение в очередь и извлечение из очереди – просто доступ к некоторому элементу массива и увеличение счетчика. Чем меньше время, расходуемое на выполнение сторонних операций, тем больше времени в сумме тратится на операции вставки и удаления узлов, следовательно возрастает вероятность конфликтов. Я не стал делать разделяемый пул критической секцией, потому что указатели p и g только увеличиваются, и задача состоит только в том, чтобы g “не догнал” p, а дополнительный mutex сильно увеличил бы время доступа к пулу.

В тесте сначала в список помещается 10 узлов, затем запускается поток adder, выполняющий добавление 1000 узлов и поток remover, выполняющий удаление 1000 узлов. Выводятся следующие данные.

|  |  |
| --- | --- |
| Счетчик | Значение |
| inserted\_nodes | Число вставленных узлов (всегда 1010) |
| removed\_nodes | Число удаленных узлов (всегда 1000) |
| not\_found\_nodes | Число не найденных узлов |
| after\_finish | Число узлов, удаленных после завершения потока adder |
| list\_counter | Число конфликтов за обладание списком |
| node\_counter | Число конфликтов за обладание узлом |

Результат теста, как и следовало ожидать меняется от прогона к прогону.

Как правило, наблюдаются две типичные ситуации.

1. Величина not\_found\_nodes оказывается большой, величина after\_finish - малой (обычно 0). Это означает, что remover “догнал” adder.
2. Величина not\_found\_nodes оказывается малой (обычно 0), величина after\_finish - большой. Это означает, что поток adder находится “впереди” потока remover, а remover отстает.

Видно, что число конфликтов за доступ у списку и за доступ к узлу во многих случаях оказывается достаточно большим (десятки и сотни). Тем не менее, все конфликты успешно разрешаются, потому что сохраняется структура списка. По завершении теста может быть успешно выполнен подсчет числа узлов и вывод значений.