Лекция 10

Потокобезопасные структуры данных

- Безопасная совместная работа с данными нескольких потоков
- Concurrency

Загадка

Эффективность как функция от размера задачи и количества обрабатывающих элементов

n	$p = 1$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
64	1.0	*0.80*	0.57	0.33	0.17
192	1.0	0.92	*0.80*	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	*0.80*	0.62

Гранулярность и степень детализации блокировки

- Размер области данных или ресурсов, которые защищает одна блокировка
- Синонимы
- Термин "гранулярность блокировки" встречается немного чаще, но "степень детализации блокировки" также широко используется и понятен
- Крупная (грубая) гранулярность / Низкая степень детализации: Одна блокировка защищает большой участок данных
- Мелкая (тонкая) гранулярность / Высокая степень детализации: Множество блокировок защищают небольшие участки данных

Инвариант

- **Инвариа́нт** – термин, обозначающий нечто неизменяемое. Конкретное значение термина зависит от той области, где он используется
- В программировании:
  - Инвариант цикла – условие, которое остается истинным до (и после) каждой итерации
  - Инвариант функции – условие, которое должно быть истинным перед вызовом функции и остается истинным после ее завершения
  - **Инвариант класса/объекта** – условие, которое описывает допустимое состояние объекта класса и должно быть истинным после завершения конструктора и оставаться истинным после вызова любого публичного метода класса
  - Инвариант алгоритма – условие, которое сохраняется на протяжении всего выполнения алгоритма

Использование инвариантов

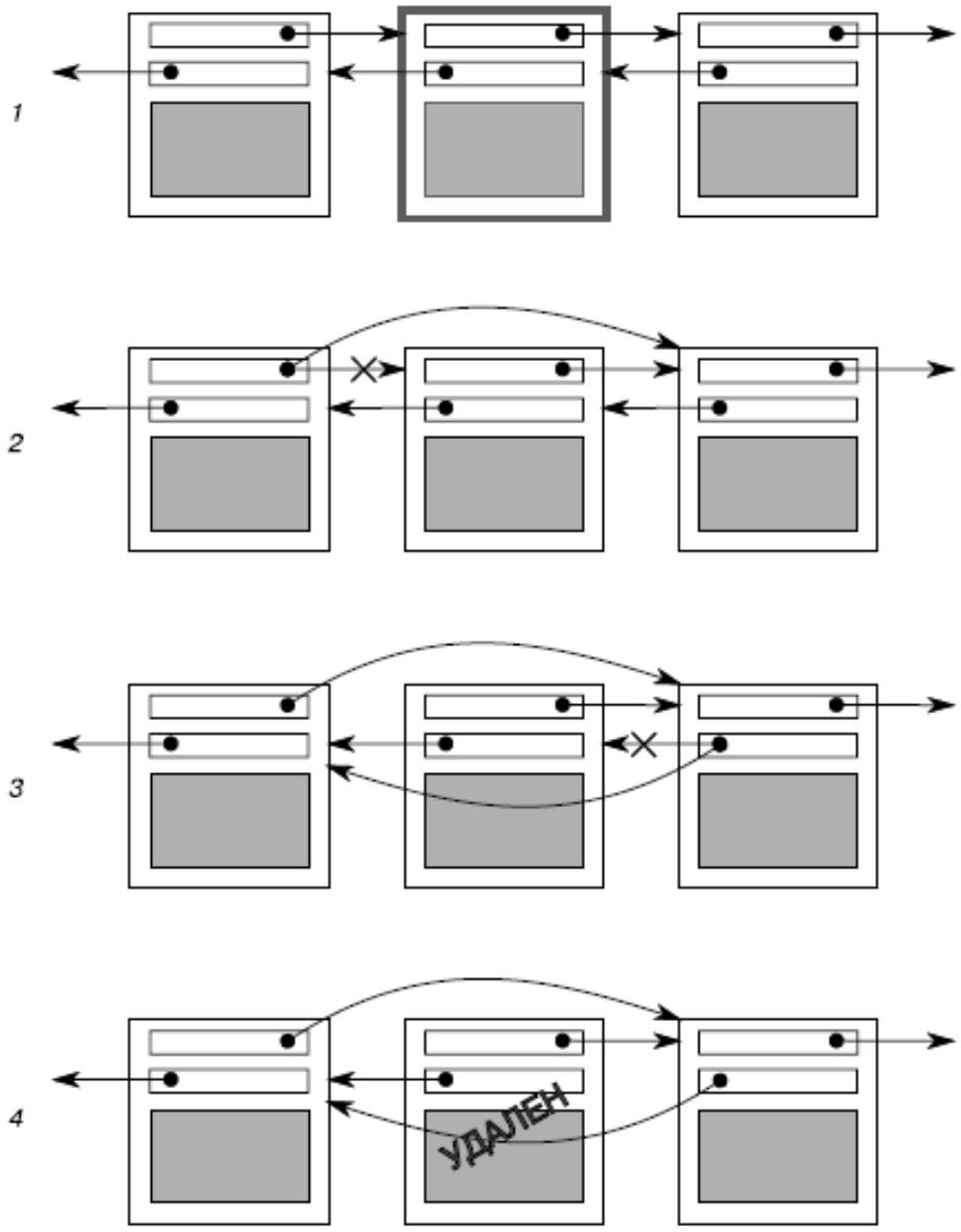
- Корректность: Инварианты помогают доказывать корректность алгоритмов и программ
- Обслуживаемость: Инварианты облегчают понимание кода
- Отладка: Нарушение инварианта указывает на ошибку в коде
- Проектирование: Четкое определение инвариантов помогает в проектировании надежных и предсказуемых систем
- Верификация: Инварианты используются в формальной верификации программного обеспечения для автоматического доказательства корректности кода

Примеры инвариантов

- В цикле поиска минимального элемента в массиве: Переменная min_element содержит индекс минимального элемента в массиве от arr[0] до arr[i], где i - индекс текущей итерации цикла.
- В стеке: Переменная top указывает на верхний элемент стека или равна -1, если стек пуст.
- В банковском счете: Баланс счета всегда должен быть больше или равен нулю (если не допускается овердрафт).
- В алгоритме сортировки: Элементы arr[0] до arr[i] отсортированы.
- В двоичном дереве поиска: Для каждой вершины дерева все ключи в левом поддереве меньше ключа в этой вершине, а все ключи в правом поддереве больше ключа в этой вершине.

Инвариант и состояние гонки

Этапы удаления узла из двусвязного списка



Проблема нарушенных инвариантов

Основные проблемы многопоточности

?

Основные проблемы многопоточности

- Состояние гонки (Race condition)
- Взаимная блокировка (Deadlock)
- Голодание (Starvation/Kesource starvation)
- Живая блокировка (Livelock)

Дать определения

Подходы к обеспечению потокобезопасности

1. Заключить структуру данных в механизм защиты
2. Изменить конструкцию структуры данных и ее инвариантов так, чтобы модификации вносились в виде серии неделимых изменений, каждая из которых сохраняет инварианты – программирование без блокировок (lock-free programming) – нелегко реализовать
3. Обновлении структуры данных в форме транзакции, так же как это делается при обновлениях баз данных

Блокировки

- *Взаимное исключение (Mutual Exclusion)*:
  - ✓ synchronized (Java)
  - ✓ Мьютексы (Mutexes)
  - ✓ Ограничения: грубая детализация блокировки, возможность взаимных блокировок
- *Чтение-запись блокировки (Read-Write Locks)*:
  - ✓ Разделение доступа для читателей и писателей
  - ✓ Улучшение параллелизма по сравнению с обычными блокировками в сценариях, где чтение происходит чаще, чем запись
- *Блокировка с соответствующей степенью детализации (Fine-Grained Locking)*:
  - ✓ Разделение структуры данных на части и использование отдельных блокировок для каждой части
  - ✓ Повышение параллелизма, но и увеличение сложности

Программирование без блокировок

- *Атомарные переменные* (Atomic Variables)
- *Безблокировочные структуры данных* (Lock-Free Data Structures)

  Использование атомарных операций для обеспечения потокобезопасности без блокировок
- *Неизменяемость* (Immutability)

  Создание объектов, состояние которых не может быть изменено после создания
- *Потокоизолированность* (Thread Confinement)

  Предоставление каждому потоку собственной копии данных

Программная транзакционная память

- Транзакционная память — технология синхронизации конкурентных потоков
- Она упрощает параллельное программирование, выделяя группы инструкций в атомарные транзакции
- Конкурентные потоки работают параллельно, пока не начинают модифицировать один и тот же участок памяти
- Если происходит конфликт данных, транзакция отменяется
- Оптимистичный подход

Программная транзакционная память

Преимущества:

- ✓ Относительная простота использования (заключение целых методов в блок транзакции)
- ✓ Полное отсутствие блокировок и взаимных блокировок
- ✓ Повышение уровня параллелизма, а следовательно, и производительности

Ограничения:

- ✓ При неправильном использовании возможно падение производительности и некорректная работа
- ✓ Ограниченность применения — в транзакции нельзя выполнять операции, действие от которых невозможно отменить
- ✓ Сложность отладки — поставить breakpoint внутри транзакции невозможно

Состояние гонки, присущее интерфейсам

Интерфейс адаптера контейнера std::stack

Element access	
top	accesses the top element (public member function)
Capacity	
empty	checks whether the container adaptor is empty (public member function)
size	returns the number of elements (public member function)
Modifiers	
push	inserts element at the top (public member function)
push_range (C++23)	inserts a range of elements at the top (public member function)
emplace (C++11)	constructs element in-place at the top (public member function)
pop	removes the top element (public member function)
swap (C++11)	swaps the contents (public member function)

```
stack<int> s;

if (! s.empty())
{
   int const value=s.top();
   s.pop();
   do_something(value);
}
```

Выбор структуры данных

- Выбор структуры данных может стать …
- Если предполагается обращаться к структуре данных сразу из нескольких потоков, то либо …, либо …
- Один из вариантов заключается в применении для защиты …, а другой – …
- При разработке структур данных, допускающих конкурентный доступ, следует обратить внимание на два аспекта: … и …

Завершить предложения

Обеспечение потокобезопасности структуры данных

- Нужно гарантировать, что ни один поток не сможет столкнуться с нарушением инвариантов структуры данных из-за действий другого потока
- Следует воспрепятствовать возникновению состояния гонки, присущего интерфейсу структуры данных, предоставив функцию для завершенных операций, а не для их поэтапного выполнения
- Нужно обратить внимание на поведение структуры данных при наличии исключений и обеспечить соблюдение инвариантов
- Следует свести к минимуму вероятность взаимной блокировки при использовании структуры данных, ограничив область действия блокировок и по возможности исключив вложенные блокировки

Обеспечение конкурентного доступа к данным

Перечень вопросов, на которые разработчик структуры данных должен ответить сам себе:

- Может ли область действия блокировок ограничиваться таким образом, чтобы некоторые части операции выполнялись за пределами блокировки?
- Можно ли разные части структуры данных защитить разными мьютексами?
- Всем ли операциям требуется одинаковый уровень защиты?
- Можно ли простым изменением структуры данных расширить возможности конкурентного доступа, не затрагивая при этом семантику операций?

Потокобезопасные структуры данных

Обзор

Java: Потокобезопасные коллекции

- Блокирующие структуры данных
- Неблокирующие структуры данных

Источник

Кей Хорстманн, Глава 12

Практические рекомендации

1. Выбирайте правильный уровень гранулярности блокировок:
   - Слишком мелкие → накладные расходы
   - Слишком крупные → низкая параллельность
2. Предпочитайте готовые потокобезопасные структуры:
   - Они тщательно тестированы и оптимизированы
3. Избегайте блокировок при чтении:
   - Используйте неблокирующие алгоритмы или RW-блокировки
4. Минимизируйте время владения блокировкой:
   - Выполняйте только необходимые операции внутри критических секций
5. Правильно обрабатывайте исключения:
   - Всегда освобождайте блокировки (используйте RAII в C++ или try-finally в Java)

Вопрос

От чего зависит выбор между блокирующими и неблокирующими структурами данных?