Теория и практика многопоточного программирования

ЛЕКЦИЯ 11. РАЗДЕЛЯЕМЫЕ СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ ОБСЛУЖИВАНИЯ

В прошлый раз...

Типы замков

Подходы к синхронизации

Темы лекции

Обзор [параллельных] структур данных

Особенности реализации замков

Зачем?

От критических секций к высокоприоритетным задачам и задачам реального времени

- Важна не столько скорость алгоритма, сколько предсказуемость времени исполнения
- Важна предказуемость масштабирования, обоснованная теоретически
- Решений множество – важно уметь выбирать алгоритм под условия (даже если алгоритмы теоретически идентичны)

О «параллельных» структурах

Структура данных это:

- Собственно структура – определённым образом связанные области памяти
- Алгоритмы обслуживания данных. Чаще всего это CRUD

Параллельные структуры/классы обслуживают взаимодействие множества [однотипных] потоков, поэтому чаще всего это:

- Разделяемые счётчики
- **Разделяемые контейнеры объектов**
- Вспомогательные объекты (**locks**, …)

Параллельные алгоритмы – часто модификация последовательных алгоритмов с дополнительными особенносиями и проверками

Контейнеры объектов

Типы контейнеров:

- **Множества** (Set) – *put(key, item), get(key), contains(key)*
- **Пулы** (Pool) – *put(item), get()*
  - Очереди
  - Стеки
  - *Очереди с приоритетом* put(item, **priority**), get()

NB! Алгоритмическая сложности операций CRUD в первую очередь определяется тем, какая организация (**структура)** данных находится в основе, а затем уже какие методы синхронизации применяются

Взгляд изнутри: массивы

Массив

- Очень долгое изменение размера
- Очень долгое (*O(N)*) «полное» удаление
- Очень **быстрый** поиск по ключу
- Удобен как контейнер фиксированной длины

Поэтому

- Много читать, мало писать и удалять
- Правило «90 / 9 / 1»
- Хорош для **множеств,** но может быть вспомогательной структурой для пулов

Массивы: хэш-таблицы

Принцип «*дешёвая память, дорогое время*» и свойства массива делают его хорошо подходящим для структур с **натуральным параллелизмом –** hash-таблиц

Два типа хэш-таблиц:

- 1 ячейка – 1 элемент (open address hash set)
- 1 ячейка – несколько элементов

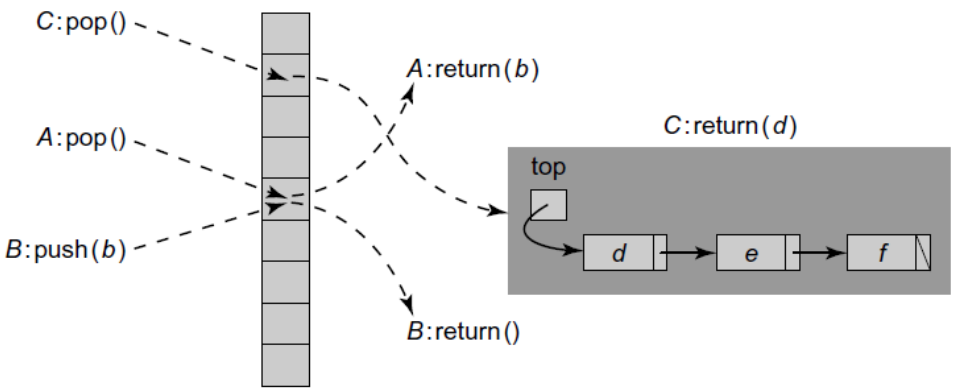
Применение подходов синхронизации:

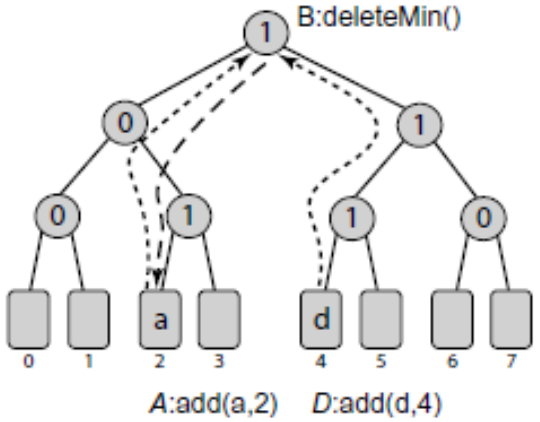
- Stripped hash set
- Lock-free hash set
  - вместо поиска – в ячейке lock-free логарифмическое множество.
  - Опорный массив быстрых переходов на связный список
- Cuckoo hash set
  - Два массива и 2 хэш-функции
- Комбинации: stripped cuckoo hash set

Массивы: стэк и очередь

Специфическое применение:

- «взаимоуничтожение вызовов» в стеке (call elimination)
- Наивная очередь с приоритетом
  - Медленные поиск (*O(N)*)
- Очередь с приоритетом и опорным фиксированным двоичным деревом





Взгляд изнутри: (сортированный) связный список

Связный список

- CRUD – *O(N)*
- Не требует перевыделения памяти
- Непосредственная (в известное место) вставка очень быстрая
- Идеальна для стеков и очередей
- ***Для очереди с приоритетом удаление O(1)*

Поэтому

- Хорошо подходит для **пулов**
- При использовании специальных модификаций превращается в O(*log(N)*) структуру
- Используется для собственных реализаций Garbage Collector’ов.

Связный список: подходы

Основной принцип коллекций на связных списках:

- Объект присутствует в множестве, если он достижим из головы списка

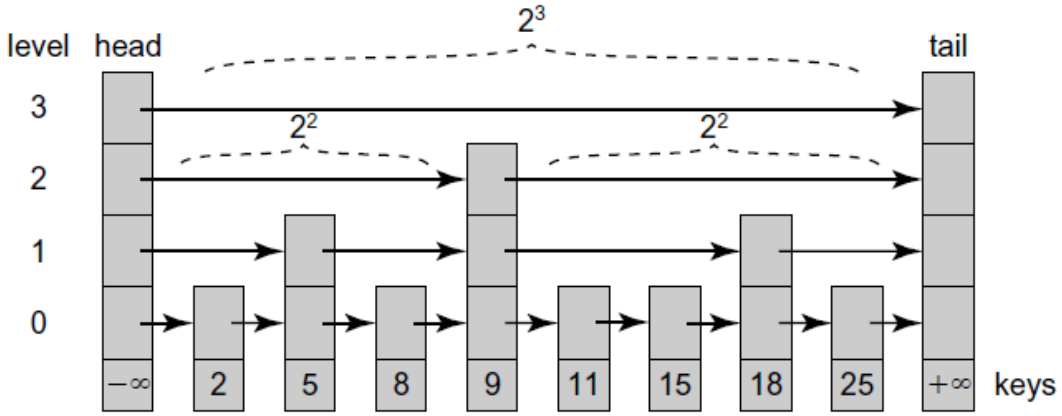
Fine-grained: попарное блокирование – в теории избегать «борьбы» (проблема «дальних» элементов)

Хорошо сочетается с *optimistic-* и *lazy-* подходами

Связный список: модификации

Skip list:

- Вероятностная структура
- O(*log(N)*)*
- Самобалансировка – альтернатива деревьям



Применение:

- Множество
- Очередь с приоритетом – быстрее реализации на куче, так как «приоритетный» элемент часто находится в начале

Параллельные особенности:

- Замыкание ссылок при вставке снизу вверх
- Удаление lazy

Связный список: особенности параллелизма

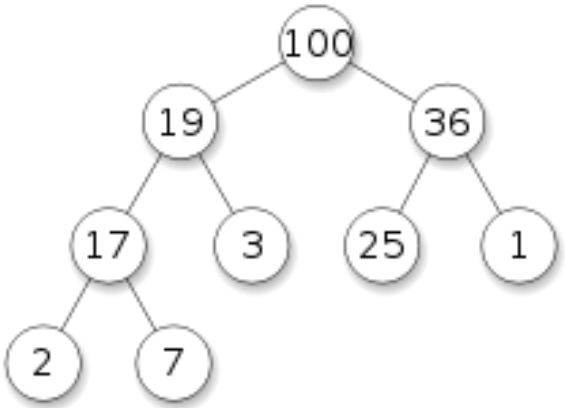
Конкурентное изменение ссылок в связном списке может приводить к возникновению проблемных ситуаций:

- *Lazy*: конфликт при вставке+удалении. Решается при помощи **атомарных помечаемых ссылок** (atomic markable references). Возвожна реализация через «bit stealing»
- *ABA* (например, при создании GC): **ссылки с подписью** (stamped reference)

Взгляд изнутри: деревья

Деревья (бинарные, кучи):

- CRUD = O(*log(N)*)
  - Heap: Read = O(1)
- Требуют перестроения



Поэтому:

- Хороши для множеств «90 / 9 / 1»
- Кучи хорошо подходят для очереди с приоритетом (*среди log(N) решений – единственная линеаризуемая, остальные – согласованы по периодам покоя*)
- Бинарные деревья хороши для индексации

Экзотический случай

Синхронная очередь

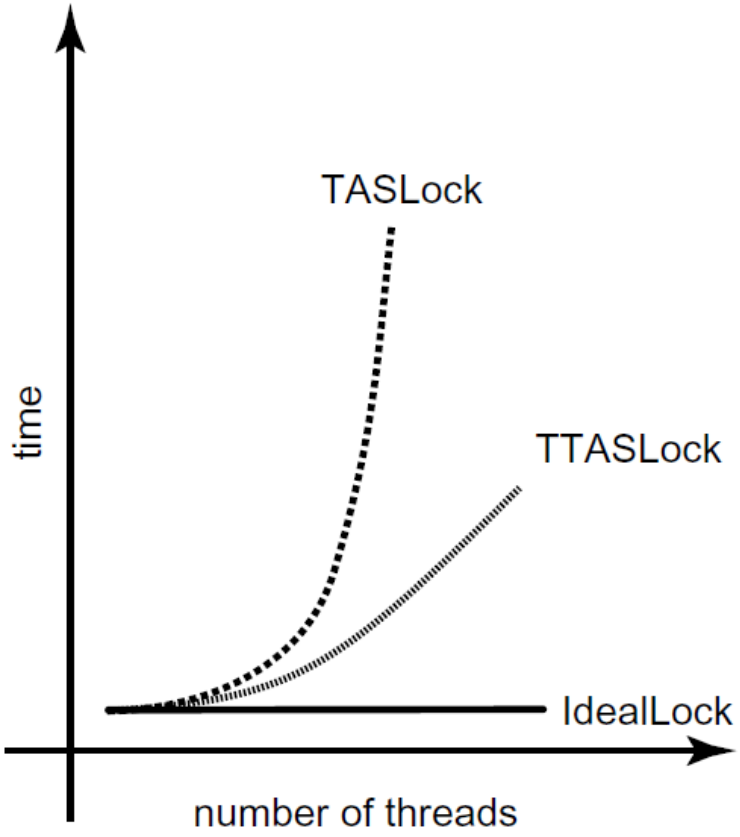
- Очередь с условием, что «писатель» не может продолжать работу, пока его элемент не будет прочитан.
- В таком случае достаточно одной ячейки для «ожидающего» элемента

Замки и ожидание

Внезапно: алгоритм Peterson lock / Bakery lock не являются корректными из-за того, что полагаются на sequential consistency программы. Решается memory barriers, но их «стоимость» сопоставима с CAS.

Замок сам может быть причиной плохого масштабирования

Идеальный замок имеет константное время обращения



Что нужно знать про замки

Существует 2 подхода к ожиданию:

- Spinning
- Sleeping

Spinning. Global spinning vs local spinning (in-cache):

- ```
  while (occupied.getAndSet(true)) { };
  return;
  ```

- ```
  while (true) {
      while (occupied.get()) { };
      if (!occupied.getAndSet(true)) return;
  }
  ```

Что нужно знать про замки

Local spinning

- При большом количестве потоков – очереди/массивы с заявками.
- «Выходящий» поток отовещает следующий

Sleeping

- Иногда лучше замереть (back off) на небольшое время, чтобы дать конкуренту завершить операцию.
- Правило ожидания: чем больше безуспешных попыток взять замок, тем выше contention, тем дольше нужно ждать