Presentation 1

Non-blocking algorithms

Lock-freedom гарантирует прогресс в системе

Treiber Lock-Free Stack

Michael-Scott Lock-Free Queue

Presentation 2.1

Model happens before

H - Исполнение программы - множество операций: чтение и запись ячеек памяти

Отношение ->_H - это happens before

e ->_H f - означает, что e произошло строго до f, это отношение транзитивно, антирефлексивно и асимметрично

Две операции параллельны, если !(e ->_H f) & !(f ->_H e)

Модель глобального времени: каждая операция - временной интервал

"Произошло до" на практике: современные яп предоставляют операции синхронизации: std::atomic в C++11, volatile в Java, потоки и тд.

Модель памяти определяет то, каким образом исполнение операций синхронизации создает отношения happens before

Конфликты и гонки данных.

Две операции над одной переменной, одной из которых является запись называются конфликтующими, они не коммутируют в модели чередования. Если они происходят параллелльно, то такая ситуация называется data race. Программа, в любом допустимом исполнении которой нет гонок данных называется корректно синхронизированной.

H|P - Сужение исполнения на поток P. Если сужение на все потоки является последовательным, тогда такое исполнение называется правильным, иначе неправильным.

Объединение всех сужений на потоки - это po - program order

H|x - Сужение исполнения на объект - множество всех операций исполнения над объектом x. В правильном исполнении оно необязательно последовательно

Последовательное исполнение является допустимым, если выполнены все последовательные спецификации объектов, те. сужения на все объекты последовательны + не бывает такого что x.w(1), затем x.r:0

Как же определить допустимость параллельного исполнения? Надо сопоставить эквивалентное последовательное. Тут в игру вступают условия согласованности.

Последовательная согласованность

Исполнение п.с. если можно сопоставить эквивалентное ему допустимое последовательное исполнение, сохраняющее po. Иначе говоря, у тебя есть потоки на временной шкале, и ты можешь все операции двигать как хочешь и куда хочешь, лишь бы порядок в рамках одного потока сохранялся

П.с. на каждом объекте не влечет п.с. всего исполнения

Линеаризуемость

Исполнение линеаризуемо, если можно сопоставить эквивалентное ему допустимое последовательное, сохраняющее порядок happens before

Свойства линеаризуемости

- В линеаризуемом исполнении каждой операции e можно сопоставить точку линеаризации t(e) \in R, так что время всех операций различно, и e -> f => t(e) < t(f)
- Линеаризуемость локальна, т.е. линеаризуемость исполнения на каждом объекте эквивалентна линеаризуемости в целом
- Операции над линеаризуемыми объектами называют атомарными, они происходят как бэ мгновенно и в определенном порядке
- В глобальном времени исполнение линеаризуемо тогда и только тогда, когда точки линеаризации могут быть выбраны так что t_inv(e) < t(e) < t_res(e)
- Исполнение системы, выполняющей операции над линеаризуемыми объектами можно анализировать в модели чередования

Presentation 2.2

Построение линеаризуемого объекта. Хотим построить линеаризуемый объект, например стек, так что все операции над ним (push, pop) линеаризуемы

Можно использовать mutex

```
class Stack:
  Mutex mutex
  ...

def push:
  mutex.lock()
  ...
  mutex.unlock()

def pop:
  mutex.lock()
  ...
  mutex.unlock()
```

То, что между lock и unlock - критическая секция (CS). Так как критические секции не выполняются параллельно, то их исполнение линеаризуемо

Попробуем реализовать lock и unlock

```
// Подобная реализация lock и unlock плоха: несколько потоков могут оказаться в CS

shared boolean want

def lock:
  while want:
    wait()
  want = true

def unlock:
  want = false

// Эта реализация работает, но нет никакой гарантии прогресса


threadlocal int id
shared boolean want[2]

def lock:
  want[id] = true
  while want[1 - id]:
    wait()

def unlock:
  want[id] = false
```

Алгоритм Петерсона

Чет ничего не получается, поэтому приведем корректный алгоритм блокировки, аж сразу обобщенный на N потоков

```
threadlocal int
shared int
shared int
id // 0 to N-1
level[N]
victim[N]
def lock:
  for j = 1..N-1:
  level[id] = j
  victim[j] = id
  while exist k: k != id and level[k] >= j and victim[j] == id:
    wait()
def unlock:
  level[id] = 0
```

Что можно сказать про этот алгоритм

- Гарантирует взаимное искоючение, отсутствие взаимной блокировки и отсутсвие голодания
- Не очень честный
    - в худшем случае O(N^2) - время ожидания
    - лучше сделать O(N)

Для этого есть алгоритм Лампорта который корректен и обладает всеми свойствами прогресса. Более того, он обладает свойством FCFS(first come first served) - это сильнее чем линейное ожидание

Тонкая и двухфазная блокировка

Грубая блокировка - блокируем всю операцию целиком. Очень неэффективно по времени. Зато любой последовательный объект можно линеаризовать.

Тонкая блокировка - блокируем отдельно операции над какими-то объектами

В примере со стеком у нас была грубая блокировка. Теперь можно сделать тонкую используя два мьютекса - один для top, второй для массива. Но если отдельно менять top под topMutex, а потом массив под arrayMutex, программа будет некорректная. Так как будут возможны нелинеаризуемые исполнения. Для обеспечения линеаризуемости используются двухфазные блокировки

Алгоритм 2-Phase-Locking

1. Взять блокировки на все необходимые объекты
2. Выполнить операцию
3. Отпустить все взятые блокировки (в любом порядке)

Но и тут могут быть проблемы. Может возникнуть deadlock, для его обнаружения может быть использован граф ожидания

Поэтому есть еще и иерархическая блокировка. Упорядочим все блокировки выстроив их в иерархию, и всегда будем захватывать сначала более приоритетные блокировки. Тогда deadlock невозможен

Presentation 3

Безусловные условия прогресса

- Отсутствие помех (obstruction-freedom). Если несколько потоков пытаются выполнить операцию, то любой из них должен выполнить ее за конечное время если все остальные остановить.*
- Lock-freedom. Если несколько потоков птыаются выполнить операцию, то хотя бы один должен выполнить ее за конечное время.
- Wait-freedom. Если поток птыается выполнить операцию, то он должен выполнить за конечное время.

Используя блокировку мы не можем получить obstruction-freedom или lock-freedom

Регистры

Объект лежащий в основе общения потоков между собой

- Безопасный регистр - гарантирует получение последнего записанного значения, если операция чтения не параллельна операции записи
- Регулярный регистр - при чтении выдает либо последнее значение, либо одно из тех, которые пишутся
- Атомарный - исполнение линеаризуемо

Дальше строим всякие более сложные регистры..

Presentation 4

Говорим про JMM - Java Memory Model

Атомарный доступ

Доступ ко всем базовым типам атомарен кроме long и double. Если приписать к ним volatile, то для них тоже атомарен.

Последовательная согласованность

Если в программе нет гонок, то она п.с.

Поймем что такое happens before с точки зрения JMM

PO - программный порядок

SO - это некий порядок на операциях синхронизации

SW - это некий подпорядок ограниченный конкретными парами, например volatile rw, lock unlock и тд.

Happens before (HB) = (SW ∪ PO)^+

Таким образом чтения в JMM могут увидеть либо последнюю запись в HB, либо что-то еще через гонку

JMM: out-of-thin-air значения запрещены, то есть если прочитали значение, значит кто-то до нас его записал

Presentation 5

Задача о консенсусе

```
class Consensus:

def decide(val):
  ...
  return decision
```

Условия задачи

- Каждый поток использует класс Консенсус единожды
- Согласованность: все потоки должны вернуть одно и то же значение из метода decide
- Обоснованность: выходное значение должно быть входным значением одного из потоков
- Wait-free

Консенсусное число - если с помощью класса атомарных объектов C можно реализовать консенсусный протокол для N потоков (и не больше), то говорят, что у класса C консенсусное число равно N.

Атомарный регистры имеют консенсусное число 1.

RMW(Read - Modify - Write) регистры - умеют атомарно менять значение и возвращать старое
Консенсусное число такого регистра больше/равно двум

Универсальный объект - у которого конс. число равно бесконечности, например CASRegister

Presentation 6.1

Какая-то хрень

Presentation 6.2