Лекция 8. Формальное описание программ.

Содержание

- Понятия события, сериализации событий, эквивалентности и линеаризации.
- Виды условного прогресса.
- Свобода от ожидания и свобода от блокировок.

Основные формальные понятия.

- Чтобы проанализировать исполнение параллельных программ, требуется модель исполнения.
- Чтобы определить модель, нужно формализовать понятия.

Основные понятия: модель потока

- Поток — недетерминированный конечный автомат (НКА).
- НКА — пятёрка

$$M = (Q, T, D, q_0, F),$$

где:
Q — множество состояний автомата
T — входной алфавит автомата
D — функция переходов

$$Q \times (T \cup \{e\}) \to \{Q_0, \ldots, Q_l\} : Q_i \subseteq Q$$

D может быть детерминистической и недетерминистической.
$q_0 \in Q$ — начальное состояние
$F \subseteq Q$ — множество конечных состояний

Конечный автомат

- Состояние НКА определяется своей конфигурацией.
- Конфигурация НКА

$$M : (q, \omega) \in Q \times T^*,$$

где:

- q — текущее состояние устройства управления
- ω — строка всех символов на автоматной ленте под головкой и правее
- q_0, ω — начальная конфигурация
- q, e — заключительная конфигурация

Конечный автомат: отображение на вычислительную машину

- Q — состояние памяти и регистров процессора.
- q_0 — точка начала исполнения программы (*entry point*)
- D — событие появления команды t_i или некоторого условия. Происходит изменение состояния (*transition*).

```
#include <thread>
#include <atomic>
#include <stdio.h>
#include <vector>
std::atomic<int> a, b; // состояние: переменные
void f1() {
    a = 0xF; // событие: изменение переменной
    while (a != b); // событие:  ожидание условия.
    printf("OK\n");
}
void f2(int c) {
    b |= c; // Событие: изменение переменной
}
int main() {
   b.store(0);
   std::vector<std::thread> v(5);
   v[4] = std::thread(f1);
   for (int i = 0; i < 4; i++)
      v[i] = std::thread(f2, 1 << i);
   for (auto &t: v) t.join();
   printf("Thats all\n");
}
```

События

- Переходы между состояниями $q_i \to q_j$ есть *события*.
- События *одномоментны* (квантование)
- Два события a и b не являются *одновременными*.
- Если событие a предшествует событию b, то это записывается как $a \to b$.
- События могут быть *неупорядочены* ($a \not\to b \& b \not\to a$).
- Отношение порядка $a \to b$ транзитивно. Если $a \to b$ и $b \to c$, то $a \to c$.

Интервалы исполнения

- Интервал $I(a, b)$ есть отрезок времени между событием a и последующим событием b, $a \to b$.
- Интервал $I(a, b)$ предшествует интервалу $I(c, d)$, если $b \to c$. Тогда $I(a, b) \to I(c, d)$
- Если $I(a, b) \not\to I(c, d)$ и $I(c, d) \not\to I(a, b)$, то $I(a, b)$ и $I(c, d)$ — *соисполняемые*

Ещё немного формализации

- *Критическая секция* CS — последовательность кода, которая может исполняться только одним потоком в один момент времени.
- *Взаимное исключение* — ситуация, когда $CS_a \to CS_b \vee CS_b \to CS_a$.
- *Зависание* (starvation) — остановка исполнения потока по внешним причинам на неопределённое время.
- *Отсутствие зависания* (starvation freedom) — свойство метода при каждом обращении приводить к завершению.
- *Неблокируемость* (nonblocking) — блокировка одного потока не задерживает исполнение других.

Неблокируемость

- Реализация объекта неблокируема, если:
  - любой поток завершит все свои операции за конечное время независимо от относительной скорости исполнения других потоков.
  - Неблокируемость гарантирует прогресс системе в целом. В отдельных частях при этом возможны задержки и сбои.

Отсутствие зависания

- Поток, вызывающий метод *lock* формирует критическую секцию и когда-либо попадает в неё.
- Разобьём метод *lock* на фазу входа D и фазу ожидания W.
- В фазе D имеется ограниченное количество шагов, ограниченный от ожидания прогресс.
- В фазе W ограничений на длину нет.

Система потоков обладает свойством *честности* по отношению к последовательности D_a и D_b процесса блокировки любых потоков a и b из системы с критическими секциями CS_a и CS_b, если выполняется утверждение: из $D_a \to D_b$ следует $CS_a \to CS_b$.

Классификация безтупиковых алгоритмов

- wait-free — свободный от ожидания
- lock-free — свободный от блокировок
- obstruction-free — свободный от задержек
- deadlock-free — свободный от тупиков алгоритм

wait-free алгоритмы

- Независимо от других потоков наблюдается продвижение.
- Обычно используются блокирующие память операции.

```
GCC:     __sync_fetch_and_add/sub/or/xor/and/nand
MS:      InterlockedIncrement/Decrement...
C++11    atomic_increment/decrement
```

Пример алгоритма: увеличение счётчика использования объекта.

```
someClass::someClass() {
    ...
    __sync_fetch_and_add(referenceCount, 1);
}

someClass::~someClass() {
    ...
    if (__sync_sub_and_fetch(referenceCount, 1) == 0) {
        ... delete unused shared object
    }
}
```

lock-free алгоритмы

- Гарантируется, что в группе потоков, связанных с какой-либо межпоточной синхронизацией хотя бы один продвигается вперёд независимо от внешних факторов.
- Остальные потоки могут в это время активно ожидать, то есть, непродуктивно использовать процессорное время.
- Обычно используются CAS-примитивы.
- Может наблюдаться инверсия приоритетов и конвоирование.

```
GCC:    __sync_val_compare_and_swap
MS:     InterlockedCompareExchange
C++11: atomic_compare_exchange_weak/strong
```

lock-free алгоритмы: пример

```
void LFStack::push(item *p) {
    item *newhead;
    do {
        newhead = head;
        p->next = newhead;
    } while (__sync_val_compare_and_swap(
        head, newhead, p));
}
```

obstruction-free алгоритмы

- Поток продвигается вперёд только тогда, когда нет конкуренции со стороны других потоков.
- Система в целом при большой взаимной конкуренции потоков не продвигается вперёд.
- Возможна живая блокировка (live-lock).
- Сами алгоритмы могут быть быстрее других.
- Некоторые алгоритмы можно реализовать лишь так (*double-linked list*).

Свойства алгоритмов

Одни и те же алгоритмы могут иметь реализации разных классов.
Пример: работа с очередью.

- Пусть очередь Q использует единую блокировку.
- Вариант 1. Поток, вызывающий блокирующий *lock* может ожидать бесконечно.
- Вариант 2. Поток, вызывающий неблокирующий *lock* может в случае неуспеха установить флаг исключительной ситуации «занято». Эта реализация wait-free.

Ещё формальные определения

- Все современные вычислительные системы асинхронны внутри.
- Что из проявлений асинхронности влияет на исполнение программ?
- Что из проявлений асинхронности влияет на поведение программистов?

Термины для объектов

Для каждого из объектов программы определены термины:

- *состояние* (state)
- *вызываемый метод* (method)
- *начало вызова* (invocation)
- *конец вызова* (response)

Вызов любого метода — протяжённый по времени процесс.

Описание исполнения программ

- Объект является *покоящимся* (quiescent), если для него нет ожидающих вызовов методов объекта.

Принципы описания процесса исполнения кода:

1. Вызовы методы должны происходить в режиме «один за раз» в последовательном режиме.
2. Если методы разделены периодами покоя, то результат должен быть аналогичен результату в порядке их реального вызова.
3. Результаты вызовов методов должны получаться в порядке, который определён программой.

Это — согласование *по периодам покоя* (quiescent consistency)

Тотальный и частичный методы

Это — два взаимоисключающих свойства.

- Метод является *тотальным*, если он определён для любого из состояние объекта.
- Метода является *частичным*, если он не является тотальным.

Для объекта «неограниченная последовательная очередь» метод `Insert` является тотальным, а метод `Remove` является частичным.

Композиционность

Свойство композиционности по свойству P есть сохранение этого свойства в системе, каждый элемент которой обладает свойством P.
Свойство согласованности по периодам покоя является композиционным.
Принципы 1 и 3 определяют свойство *упорядоченной согласованности*. Оно не является композиционным.

Линеаризуемость

- Метод является линеаризуемым, если изменение данных как результат его применения имеют место между вызовом и его завершением.
- Линеаризуемый $\rightarrow$ упорядоченно согласованный.
- Упорядоченно согласованный $\not\rightarrow$ линеаризуемый.
- *Точка линеаризации* — место, где имеют эффект действия, явившиеся следствием изменения данных.
- Каждая критическая секция метода — точка линеаризации.
- Для обычного кода — точка линеаризации есть место, где результат становится видимым другим методам.
- Линеаризуемость: описание больших программных систем для верификации их взаимодействия.
- Линеаризуемость композиционна.

Упорядоченная согласованность

- Позволяет упорядочивать потенциально перекрывающиеся операции в виде упорядоченной последовательности.
- Порядок операций внутри одного потока меняться не должен.
- Хороший способ описания систем, где не существенна композиционность.
- Согласованность по периодам покоя $\neq$ упорядоченной согласованности.
- Аппаратная память.

Согласованность по периодам покоя

- Удовлетворяет пунктам 1 и 2 принципам процесса исполнения кода.
  - Вызовы методы должны происходить в режиме «один за раз» в последовательном режиме.
  - Если методы разделены периодами покоя, то результат должен быть аналогичен результату в порядке их реального вызова.
- Параллельные вычисления основанные на сообщениях: кластерная математика.

Идеальные алгоритмы?

- Панацея ли «свободные от ...» алгоритмы?
  1. Они могут оказаться неэффективными. `CAS` — дорогая операция.
  2. Они могут оказаться трудно реализуемыми.
- `lock-free` $\rightarrow$ `wait-free`
- `wait-free` $\not\rightarrow$ `lock-free`.
- В `wait-free` возможны ситуации с недополучением ресурса потоком.
- Может оказаться, что `lock-free` окажется быстрее `wait-free`.
- `wait-free` реализации требуют $O(N)$ памяти, где N — число потоков.

Условный прогресс

- `wait-free` и `lock-free` дают безусловный прогресс системе.
- `obstruction-free` даёт условный прогресс. Должны быть гарантии от планировщика ОС или оборудования.

Определить класс алгоритма

```
void push(int t) {
    node *n = new node(t);
    do {
        n->next = head;
    } while (!CAS(&head, n, n->next));
}
bool pop(int &t) {
    node *c = head;
    while (c != NULL) {
        if (CAS(&head, c->next, c)) {
            t = c->data;
            return true;
        }
        c = head;
    }
    return false;
}
```

Спасибо за внимание.

Следующая тема — задача о консенсусе.