ТЕОРИЯ И ПРАКТИКА МНОГОПОТОЧНОГО ПРОГРАММИРОВАНИЯ

Тема 8

Формальное описание программ

Д.ф.-м.н., профессор А.Г. Тормасов
Базовая кафедра «Теоретическая и Прикладная Информатика», МФТИ

Тема

- Формальное описание программ с использованием понятий событий, «истории», сериализованной истории, эквивалентности, легальности, линеаризации.
- Теоремы о композиционности и неблокируемости линеаризуемости.
- Условный прогресс.
- Свобода от ожидания и свобода от блокировок

Необходимость

- Для анализа необходимы математические модели
- Для моделей необходима формализация понятий и процессов (их хода)

Поток исполнения thread

- Является конечным автоматом (state machine)
- Недетерминированный конечный автомат (НКА) - это пятерка $M = (Q, T, D, q_0, F)$, где
  - Q - конечное множество состояний;
  - T - конечное множество допустимых входных символов (входной алфавит);
  - D - функция переходов (отображающая множество $Q \times (T \cup \{e\})$ во множество подмножеств множества Q), определяющая поведение управляющего устройства;
  - $q_0 \in Q$ - начальное состояние управляющего устройства;
  - $F \subseteq Q$ - множество заключительных состояний.

Конфигурация автомата

- Конфигурация автомата M есть пара $(q, \omega) \in Q \times T^*$, где
  - q - текущее состояние управляющего устройства
  - ω - цепочка символов на входной ленте, состоящая из символа под головкой и всех символов справа от него
  - Конфигурация (q_0, ω) называется начальной
  - конфигурация (q, e), где $q \in F$ - заключительной (или допускающей).

События

- Переходы между состояниями - события (event)
- События считаются одномоментными
  - события не являются одновременными
    - Упорядочены: одно следует за другим
    - Произвольный порядок, если не уверены
    - событие a предшествует событию b: $a \rightarrow b$

Интервалы

- *интервал* I(a,b):
  - продолжительность времени, прошедшего между событиями a и b, причем a → b
- Интервал I(a,b) предшествует интервалу I(c,d) если b→ c
  - I(a,b) → I(c,d)
- Интервалы, не связанные отношением → , будем называть соисполняемыми (concurrent)

Определения

- Определение
  «*Критической секцией*» (critical section, CS) называется последовательность кода, которая может исполняться только одним потоком исполнения в один момент времени.

- Определение
  Свойство «*Взаимное исключение*» (mutual exclusion) системы потоков присутствует, если критические секции любых разных потоков не перекрываются: $CS_A \rightarrow CS_B$ или $CS_B \rightarrow CS_A$.

Процесс исполнения

- Определение
«*Зависанием*» (starvation) называется остановка исполнения потока в своем исполнении на неопределенное время

- Определение
Свойство «*Отсутствие зависаний*» (starvation freedom/ lockout freedom) присутствует, если каждое обращение к методам обязательно завершается.

- Определение
Свойство «*Неблокируемости*» (nonblocking) для потоков означает, что блокировка (задержка) одного потока не может задержать другие потоки.

Свойство «неблокируемость»

- реализация объекта является неблокируемой, если:
  - некий процесс закончит все операции за конечное число шагов
  - вне зависимости от относительной скорости работы других процессов!
- Неблокируемость системы в целом гарантирует ее прогресс вне зависимости от индивидуальных задержек, остановок, сбоев и т.д. ее частей
- Этим свойством могут обладать алгоритмы и условия

Свойство «отсутствия зависаний»

- каждый поток, который вызывает метод блокирования объекта lock(), когда нибудь попадет в критическую секцию
- Но когда?
- Разобьем всю процедуру вызова метода блокировки lock() на
  - «вход» - D (строго конечное число шагов, «ограниченный от ожидания прогресс»)
  - «ожидание» - W (нет ограничений)

Свобода...

- Определение «*Свобода от взаимной блокировки*» (Deadlock freedom) в системе потоков:
  - Если какой либо поток попытался взять блокировку lock(), то некоторый поток его получит
  - Если некоторый поток вызвал блокировку lock(), но не сможет никогда ее получить, то другие потоки должны завершить бесконечное количество критических секций

! «Отсутствие зависаний» подразумевает «свободу от взаимной блокировки»

Условия Коффмана

- Условия, при которых наступает взаимная блокировка (Коффман, 1971)
  - Условие взаимного исключения, то есть ресурс не может в один момент времени быть использован более чем одним потоком,
  - Условие удержания и ожидания, то есть поток захватывает какие то ресурсы и ожидает, когда сможет захватить еще ресурсы, не освобождая уже захваченные,
  - Условие неперераспределяемости, или невозможности принудительного освобождения уже захваченного ресурса кем либо, кроме захватившего его потока,
  - Условие взаимного кругового ожидания, или существования «круга», в котором каждый участник ждет одного или более ресурсов, захваченных другими участниками «круга».

Определения…

- Определение
  Система потоков обладает свойством «*честности*» (fairness) по отношению к последовательности «входов» D_A и D_B процесса блокировки любых двух потоков A и B с критическими секциями CS_A и CS_B в том случае, если A закончил исполнение входа до того как B начал свой вход, то критическая секция A не может быть выполнено позже чем в B:
  если $D_A \rightarrow D_B$ то $CS_A \rightarrow CS_B$ для любых A и B из системы.
  - «первый пришел – первый обслужился»
    - с точки зрения выполнения процедуры «входа»
  - если два входа двух потоков соисполняемы, то их последовательность не определена

Определения…

- Определение
  Свойство «*свобода от ожидания*» (wait-free) присутствует, если выполнение метода заканчивается за конечное количество шагов без каких либо процедур ожидания
  - Свободный от ожидания всегда неблокируемый (но не наоборот!)

- Определение
  Свойство «*ограниченный от ожидания прогресс*» (bounded wait-free progress) присутствует, если выполнение метода заканчивается за конечное и ограниченное количество шагов без каких либо процедур ожидания
  - «улучшенный» вариант свободы от ожидания

Определения…

- Определение
  Метод является «*свободным от блокировок*» (lock-free), если гарантированно, что неограниченно часто вызываемый конкретный метод закончится за конечное число шагов
- Определение
  Объект имеет в «последовательном» описании:
  - «состояние»
  - вызываемый «метод», имеющий события
    - «*начало вызова*» (invocation)
    - «конец вызова» (response)
- Определение
  Объект является «*покоящимся*» (quiescent) если в данный момент времени нет ожидающих вызовов методов объекта.

Метод

- <x.m(a*)A> - invocation
  событие «начало вызова» метода m объекта x с последовательностью аргументов a* в потоке A

- <x:t(r*)A> - response
  событие «конец вызова», где t является или Ok или именем исключения, а r* представляет собой последовательность результирующих значений

Определения…

- Определение
  Метод является тотальным, если он определен для каждого состояния объекта.

- Определение
  Метод является частичным, если он не является тотальным.

  Пример: для неограниченной последовательной очереди метод «добавить в очередь» является тотальным так как это можно сделать для любого состояния очереди, тогда как метод «взять из очереди» является частичным так как он не определен если очередь пуста.

  для любого соисполнения для любых незавершенных вызовов тотальных методов существует события «конец вызова», согласованные по периодам покоя – по сути это неблокирующее условие корректности (из определения)

Принципы

- **Принцип 1**
  Вызовы методов должны происходить в режиме «по одному за раз» в последовательном режиме.

- **Принцип 2**
  Вызовы методов, разделенные периодами «покоя», должны давать результат в порядке их реального вызова.

Определения…

- Определение
  Вызовы методов, удовлетворяющие одновременно принципам 1 и 2, называются согласованными по периодам покоя (quiescent consistency).
  - Если объект становится покоящимся, то результат его исполнения до этого момента оказывается эквивалентен упорядоченной последовательности завершенных вызовов его методов

Определения…

- Определение
  Свойство *корректности* P является *композиционным* для системы объектов в том случае, если каждый из объектов в системе удовлетворяет P, то и система в целом удовлетворяет P.
  - Свойство упорядоченной согласованности является композиционным.
- **Принцип 3**
  Результаты вызовов методов должны происходить в том порядке, который определен программой.
- Определение
  Вместе принципы 1 и 3 определяют свойство корректности под названием «*упорядоченной согласованности*» (sequential consistency).
  - Оно не является композиционным.

упорядоченная согласованность

A: `---|+q.Put(x)+|-------|+q.Get(y)++|-----`

B: `---------------|+q.Put(y)+|----------`

- Выполнение упорядоченно согласованно
  - Несмотря на то что казалось бы нарушен порядок доступа к FIFO
  - Оба Put не связаны порядком программы (в разных потоках), можно переставлять

A: `-|+p.Put(x)+|------------|+q.Put(x)+|-------------|+p.Get(y)+|-----------`

B: `------------|+q.Put(y)+|-------------|+p.Put(y)+|-----------|+q.Get(x)+|-`

- Выполнение упорядоченно несогласованно!
  - p и q сами по себе упорядоченно согласованы, тогда как общее исполнение – НЕТ

упорядоченная согласованность

A: `-|+p.Put(x)+|------------|+q.Put(x)+|-------------|+p.Get(y)+|-----------`

B: `------------|+q.Put(y)+|-------------|+p.Put(y)+|-----------|+q.Get(x)+|-`

- Почему?
- Так как p – FIFO и A Get(y) из p, то y должно быть помещено в очередь ДО x

$$p.put(y)_B \rightarrow p.put(x)_A$$

- Аналогично,

$$q.put(x)_A \rightarrow q.put(y)_B$$

- Но по порядку исполнения программы должно быть

$$p.put(x)_A \rightarrow q.put(x)_A \text{ и } q.put(y)_B \rightarrow p.put(y)_B$$

- В результате – цикл!

Определения…

Определение

Метод является «*свободным*» (obstruction free) если, будучи вызванным изолированным (когда другие потоки не выполняют шагов), при любых условиях вызова заканчивается за конечное число шагов.

Определение

Метод является *линеаризуемым*, если результат его применения имеет место непосредственно между вызовом и его завершением.

- Каждый линеаризуемый метод является упорядоченно согласованным, но не каждый упорядоченно согласованный метод является линеаризуемым.

Определение

Точка линеаризации – это то место где результат исполнения метода применяется к его данным (имеет эффект).

- Пример точки линеаризации для программирования на базе блокировок – каждая критическая секция метода. Для программирования без блокировок точка линеаризации обычно та, где результат применения метода к его данным становится видимым другим методам.

Комментарий

- *Упорядоченная согласованность* предоставляет возможность переупорядочивать потенциально перекрывающиеся операции в виде некоторой упорядоченной последовательности.
  - Причем, порядок операций внутри одного потока не может меняться!
- *Упорядоченная согласованность* является хорошим способом описания отдельных систем, где композиционность не существенна (скажем, аппаратная память).
- Упорядоченная согласованность и согласованность по периодам покоя никак не соотносятся друг с другом

Комментарий

- *Линеаризуемость* подразумевает введение жесткого порядка, а вернее, упорядоченности операций в смысле «было до» (happens-before).
- *Линеаризуемость* является хорошим способом описания больших программных систем, где компоненты должны быть реализованы и верифицированы независимо.
- *Линеаризуемость*, так же как и упорядоченная согласованность, является неблокирующей. Боле того, линеаризованность, так же как и согласованность по периодам покоя, является композиционной – составленный из линеаризуемых объектов объект также является линеаризуемым.

История

Определение

«*История*» H представляет собой конечную последовательность событий типа «начало вызова» и «конец вызова» для методов. Подисторией S истории H является подпоследовательность событий из H.

- Пусть «конец вызова» «*соответствует*» «началу вызова» если они имеют одни и те же объект и поток.

Определение

Вызов метода в истории H есть пара соответствующая событиям «начало вызова» и следующее соответствующее ей событие «конец вызова» в H.

Определение

Если событию «начало вызова» нет соответствующего события «конец вызова», то вызов метода является *ожидающим*.

История...

Определение
«*расширение*» истории H есть история, содержащая оригинальную историю и 0 или более событий типа «конец вызова» для ожидающих вызовов.

Определение
«*завершенная*» история истории H complete(H) называется подпоследовательность всех пар «соответствующих» событий истории H, без ожидающих вызовов.

История...

- История H является упорядоченной («сериализованной»), если
  - первое событие истории есть «начало вызова»
  - каждое событие типа «начало вызова» (возможно, за исключением последнего), имеет непосредственно следующее за ним «соответствующее» событие «конец вызова».
- Подистория потока H|A (H по A) есть подпоследовательность всех событий H, в которых поле имя потока равно A. Аналогично определяется подистория объекта H|x.
- Пусть две истории H и H' системы потоков эквивалентны, если для любого потока A из системы H|A = H'|A.
- История «оформлена» (well formed) если каждая подистория потока упорядочена.
  - Хотя все подистории потоков оформленной истории упорядочены, подистории ее объектов не обязаны быть таковыми.

Предшествование

- вызов метода m_0 предшествует вызову метода m_1 в истории H $\mathbf{m_0 \rightarrow_H m_1}$, если:
  - m_0 закончился раньше, чем стартовал m_1 (событие «конец вызова» метода m_0 будет предшествовать событию «начало вызова» метода m_1)

- $\mathbf{m_0 \rightarrow_x m_1}$
  - m_0 предшествует m_1 в подистории H|x.

Легальность

Определение
Упорядоченной спецификацией объекта будем называть набор упорядоченных историй объекта.

Легальная

```
<f.открыть("file.exe")A>
                    <f:Ok(handle)A>
<f.записать(handle ,data)A>
                    <f:Ok()A>
<f.закрыть(handle)A>
                    <f:Ok()A>
```

Нелегальная

```
<f.открыть("file.exe")A>
                    <f:Ok(handle)A>
<f.закрыть(handle)A>
                    <f:Ok()A>
<f.записать(handle ,data)A>
                    <f:Ok()A>
```

Определение
Упорядоченная история H является легальной если для каждого объекта его подистория легальна для объекта.

Линеаризация истории

Определение

История H является *линеаризуемой*, если ее расширение H' и существует легальная упорядоченнная история S (далее называемая *линеаризацией* H) так, что

- complete(H') = S
- если $m_0 \rightarrow_H m_1$, то $m_0 \rightarrow_S m_1$.

- если вызов состоялся, но пока не вернулся, то результат его работы может быть уже использован
- если один метод предшествует другому в истории, то это же должно быть правдой и в ее линеаризации
- S не является уникальной для любой конкретной H!

композиционность линеаризуемости

Теорема (композиционность линеаризуемости)

Для того, чтобы H была линеаризуемой, необходимо и достаточно, чтобы для каждого объекта x H|x являлось линеаризуемым.

Доказательство:

Необходимость – оставим в качестве упражнения.

Достаточность: пусть существует H|x, и H’ есть ее расширение. По индукции:

Если там только один вызов, то теорема доказана.

m – самый последний вызов метода для x в H'|x по отношению к $\rightarrow$ (не существует m’ такого, что $m \rightarrow_H m'$).

Пусть G' есть H', из которой убран этот вызов m, то есть H’ = G’m.

По индукции, G' линеаризуемо в некую сериализованную историю S', и оба H и H' линеаризуемы в S'm.

неблокируемость линеаризуемости

Теорема (неблокируемость линеаризуемости)

Пусть inv(m) – вызов тотального метода (событие типа «начало вызова»).
Если <x inv P> есть неоконченный вызов в линеаризуемой истории H, то существует событие «конец вызова» <x res P> такое, что H <x res P> является линеаризуемым.

Доказательство:

Пусть S есть произвольная линеаризация H. Если она включает в себя <x res P>, то теорема доказана, так как S есть также и линеаризация H <x res P>.

Если нет, то <x inv P> не входит в S, поскольку, по определению, линеаризация не содержит незавершенных вызовов.

Так как метод тотальный, то существует событие «конец вызова» <x res P> такое, что S′ = S <x inv P> <x res P> является легальным.

S', однако, есть линеаризация H <x res P>, то есть также линеаризация H.

- **линеаризуемость сама по себе никогда не заставляет поток со сделанным вызовом заблокироваться!**
- **линеаризуемость программы есть соответствующее условие корректности для систем, где соисполнение и ответы в реальном времени являются существенными**

Блокирующий или нет?

- один и тот же объект может быть организован как блокирующий и как не блокирующий.
  - Очередь с единым блоком:
    - Блокирующая - поток, пытающийся захватить блок, может ждать на вызове lock() вечно
    - Неблокирующая - вызов lock() не блокируется, а возвращает «занят» - реализация будет «свободна от ожидания».

- Свободные от ожидания алгоритмы могут оказаться неэффективными или неудобными в реализации.
  - Очевидно, что свобода от ожидания → свобода от блокировок
  - свобода от блокировок не исключает ситуации с наличием зависаний из-за неполучения ресурса каким либо потоком
  - Тем не менее, такая ситуация может на практике быть маловероятной, и тогда может оказаться, что быстрая реализация без блоков окажется более приемлемой чем более медленная реализация со свободой от ожидания.

Условный прогресс

- Свобода от ожидания и свобода от блокировок
  - гарантируют, что процесс вычисления в целом будет прогрессировать
- «свобода от взаимной блокировки» и «отсутствие зависаний»
  - прогресс зависит от гарантий платформы - условен
    - когда ОС гарантирует, что каждый поток, вошедший в любую критическую секцию, когда-нибудь из нее выйдет
    - лучше если за какое то ограниченное время!

Свобода...

- свобода от блокировок → свободный метод (но не обратное!)
- свободный метод не гарантирует прогресс при соисполнеии
  - Потоки могут блокировать друг друга (активная блокировка возможна)
- «оптимистические» методы
  - предполагающие, что каждый участник работает в одиночестве, без влияния соседей, и детектирующее это влияние в конце работы – «все будет хорошо»?
- Свободный от блокировок метод обеспечивает глобальный прогресс для каждого шага любого потока (нет активной блокировки)
  - Но, каждый поток может зависать !
  - Может быть инверсия приоритетов и конвоирование

свободный от блокировок

```
void push(int t)
{
        Node* node = new Node(t);
        do {
                node->next = head;
        } while (!cas(&head, node, node->next));
}
bool pop(int& t)
{
        Node* current = head;
        while(current)
        {
                if(cas(&head, current->next, current))
                {
                        t = current->data; // problem?
                        return true;
                }
                current = head;
        }
        return false;
}
// где ошибка в этом коде?
```

Свобода...

- свобода от ожидания
  - все потоки завершают работу вне зависимости от поведения друг друга за конечное число шагов, без ожидания
  - Не может быть инверсии приоритетов
- ограниченный от ожидания прогресс
  - существует определенная граница для числа шагов
  - Самое сильное условие, лучше некуда (почти)!
  - Не ограничивает в использовании циклов и неопределенных ветвлений, но обычно требует доказательства для оценки

Свобода...

Определение

Если свободный от ожидания алгоритм не зависит в своей сложности от количества потоков, то он называется независимым от совокупности

- Сложность – чтото типа O(1) (или от числа потенциальных конфликтов?)

Пример свободного от ожидания

```
// Реализация свободной от ожидания FIFO очереди
// использует одномерный массив HEAD
// объектов типа FetchAndInc (инициализированы 0)
// и двумерный массив ITEMS объектов Swap (NONE)
// один регистр ROW (0)
// размерность O(n) (второе измерение – ∞)
// n – число потоков

Static enq_row, tail;

void Put(int x)
{
   val = Swap(Items[enq_row, tail], x);
   if (val == T)
   {
      enq_row++;
      tail = 0;
      Swap(Items[enq_row, tail], x);
      Write(ROW, enq_raw);
   }
   tail++;
}
```

```
// один писатель, и множество (n) читателей
//

int Get(void)
{
   deq_row = Read(ROW);
   head = FetchAndInc(HEAD[deq_row]);
   val = Swap(Items[deq_row, head], T);
   if (val == NONE)
   {
      return S;
   }
   else
   {
      return val;
   }
}
```

Свобода !???

- Свободные от ожидания алгоритмы – среди самых «хороших»
- Условные примитивы (CAS) – самые «хорошие» (как будет показано далее, самые «мощные»)
- Так давайте реализовывать алгоритмы этими примитивами и получим идеальную реализацию?
- Увы...
- lock cmpxchng – медленный, блокирует шину
- Их (ячеек, ими обслуживаемых) надо ОЧЕНЬ много для реализации
  - Класс Visible(n) - включают в себя все объекты, которые поддерживают операции, которые должны осуществить видимые кем то записи до своего окончания
    - счетчики, стеки, очереди, обменные операции swap, извлечь-и-добавить (fetch-and-add)
  - Существует доказательство, что для класса Visible(n) надо не менее n условных регистров
  - Для них существует O(1) реализация через не-условные примитивы (например, fetch-and-inc)!

Отношение понятий

Неблокирующие условия (безусловный прогресс)

- независимый от совокупности — population oblivious
- ограниченный от ожидания прогресс — bounded wait-free
  - свобода от ожидания — wait-free
    - свобода от блокировок — lock-free
      - свободный метод (условен!) — obstruction free

Блокирующие условия (условный прогресс)

- отсутствие зависаний — starvation freedom
  - свобода от взаимной блокировки — deadlock freedom

Отношение понятий

- линеаризуемый — linearizable
  - упорядоченно согласованный — sequential consistency

Никак не соотносятся:

- упорядоченно согласованный
- согласованный по периодам покоя

Выбор желаемых свойств

- Свойства зависят от
  - Требований приложения
  - Характеристик платформы исполнения
- Абсолютные свободы от ожидания и блокировок
  - Хороши в теоретических свойствах
  - Работают везде
  - Могут давать гарантии реального времени для звука-видео и тд
- Условные свободы от зависаний, взаимоблокировок, в изоляции
  - Полагаются на гарантии платформы
  - Часто проще в реализации и быстрее в работе

Выводы

- Введено множество понятий, описывающих ход выполнения программы в зависимости от разнообразных условий
- Обсуждены их отношения друг с другом и степень удобства для программиста
- Увы, идеала нет. Часто если примитив хорош, то реализация неэффективна по каким то критериям
- При выборе свойства своего алгоритма надо четко представлять себе условия его работы, вероятности коллизий и их типы, требования в худшем-лучшем-среднем и другие условия работы

(c) А. Тормасов, 2010-11 г.

Базовая кафедра «Теоретическая и Прикладная Информатика» ФУПМ МФТИ
tor@ crec .mipt .ru_

Для коммерческого использования курса просьба связаться с автором.