

Параллелизм

«Операционные системы»

Д.В. Иртегов

ФИТ/ФФ НГУ

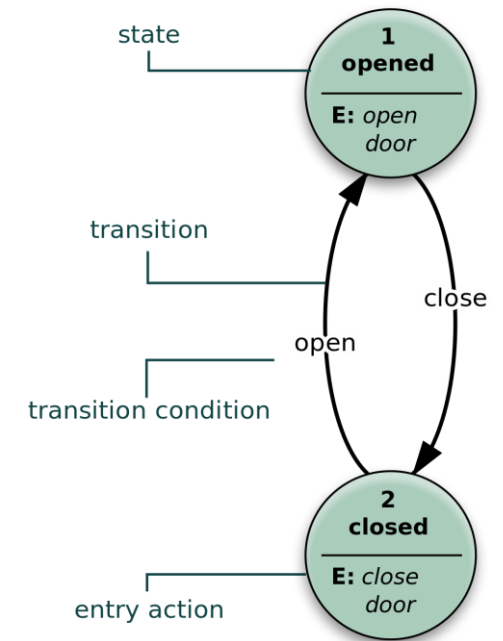
2020

Где вы сталкиваетесь с параллельным программированием

- Сети
- Распределенные системы
 - Базы данных
 - Бизнес-приложения
 - Системы поддержки разработки ПО (например, VCS)
- Многомашинные кластеры
 - Вычислительные задачи: решение дифуров, машинное обучение
 - Масштабируемые веб-приложения
 - Системы поддержки разработки ПО (CI/CD)
- Многоядерные процессоры
- Приложения, ориентированные на ввод-вывод
 - Прерывания
 - Пользовательский интерфейс
 - Игры
 - Приложения реального времени
- То есть, практически везде

Теоретические проблемы параллельного программирования

- Последовательную программу можно представить как конечный автомат (state machine)
- Параллельную программу так представить нельзя:
 - переходы между состояниями происходят не одновременно
 - Эта же проблема возникает при сопряжении цифровых устройств с разными тактовыми частотами
 - Есть обобщения конечных автоматов, например, сети Петри
 - ФИТ будет проходить их в курсе «Параллелизм»



Практические проблемы параллельного программирования

- Race conditions (ошибки соревнования, «гонки»)
 - Разные сценарии наложения во времени параллельных сопрограмм будут давать разные результаты
 - Пример: два человека приходят покупать последний билет
 - Если некоторые из этих разных результатов ошибочны, это и есть ошибка соревнования
 - Пример: оба купили билет на одно и то же место
 - Или: билет никто не купил, но он считается проданным
- Практически не «ловятся» тестированием

Ошибки соревнования

- Главным образом, возникают из-за разделяемых данных
- Пример: параллельное увеличение счетчика

```
ld r0, r1  
inc r1  
st r0, r1
```

```
ld r0, r1  
inc r1  
st r0, r1
```

- Другой пример: вставка в сортированный массив параллельно с поиском в этом же массиве

Целостность данных

- Еще говорят, согласованность (consistency)
- Требования предметной области:
 - Бухгалтерия: дебет-кредит == сальдо
 - Физика: сумма всех энергий должна сохраняться
- Требования алгоритмов обработки:
 - Записи должны иметь уникальные идентификаторы
 - Строки должны заканчиваться нулем
 - Массив должен быть отсортирован
 - Дерево должно быть сбалансировано
- В общем случае, неформализуемое понятие

Критическая секция

- Участок кода (или время исполнения такого участка кода)
 - Который либо нарушает целостность разделяемой структуры данных
 - Либо полагается на целостность такой структуры (при том, что эта структура может быть изменена кем-то еще)
- Я обычно говорю о критических секциях как о проблеме, но это первый шаг к решению проблемы соревнования
- Если вы не думаете о критических секциях, ваша программа вся может быть одной гигантской критической секцией (хотя бы часть данных все время не целостна)
- Если вы о них подумали и выделили в коде, значит, у вас есть периоды, когда данные целостны

Подходы к решению

- Выделение критических секций и блокировки
 - Пример: блокировка участков файлов
- Отказ от изменяемых разделяемых данных
 - Иммутабельные данные: чистое ФП, Java Streams, Python tuples
 - Гармоническое взаимодействие
- Copy-modify-merge
 - Заставить человека сливать конфликты вручную (VCS, Lotus Notes)
 - Транзакции (откат при конфликте)
- Lockless программирование
 - Переделать алгоритм так, чтобы он не нарушал целостность (обычно невозможно) или не полагался на нее (иногда получается)

Блокировки


- Давайте свяжем с критической секцией флаговую переменную
- Когда кто-то сидит в этой секции, `flag==true`
- Когда никого нет, `flag==false`

```
while (flag) {}  
flag=true;
```

Блокировки

- Давайте свяжем с критической секцией флаговую переменную
- Когда кто-то сидит в этой секции, `flag==true`
- Когда никого нет, `flag==false`

```
while (flag) {}  
flag=true;
```



- Поздравляю, мы получили критическую секцию из единственной булевой переменной
- Видимо, самая простая из возможных критических секций

Алгоритм Деккера

Нить 1

```
while (flag2) {}  
flag1=true
```

Нить 2

```
while (flag1) {}  
flag2=true
```

Аппаратное взаимoisключение

- x86: lock prefix
- Может использоваться с большинством команд, работающих с памятью, например, `lock inc [eax]`
- Работает в режиме монопольного захвата шины
- Атомики в Java и C реализованы через этот префикс
- Специальная команда `xchg reg, mem` (eXCHange)

```
spin:    mov eax, 1
         xchg eax, flag
         tst eax
         jnz spin
```

Аппаратное взаимное исключение

- ARM: команды ldrex/strex
- ldrex r1, [r0] – читает значение из адреса r0, помещает его в регистр r1 и взводит монитор
- strex r2, r1, [r0] – пытается записать значение в защищенную монитором ячейку памяти
 - Если с момента ldrex ячейка не менялась, происходит запись в память
 - Если менялась (конфликт), записи не происходит, и в r2 записывается 1
 - В любом случае, монитор сбрасывается
- Можно реализовать не только атомарные операции, но и целые транзакции (правда, там есть ограничение по длине кода)

Атомарное исключение

- Так или иначе, все процессоры, пригодные для многопроцессорной/многоядерной работы, имеют средства для реализации атомарной функции Compare_And_Set (CAS)
- Взаимное исключение на основе CAS:
`while(CAS(flag, 1)) {}`
- Это называется spinlock
- Спинлоки используются в ядрах ОС
- На спинлоках можно реализовать более сложные атомарные операции

Что такое «примитив»

- Примитив – это переменная непрозрачного типа, над которой определен некоторый набор операций
- Вы можете использовать эти операции, но вам не следует работать с объектом мимо этих операций
- В случае примитивов синхронизации – потому что есть риск выполнить операцию неправильно и словить ошибку соревнования
- Примерно то же самое, что объект с инкапсуляцией
- Но слово «примитив» было придумано задолго до ООП, поэтому в старых книжках используется это слово

Более сложные примитивы синхронизации

- Спинлок — это холостой цикл, а холостой цикл — это плохо
- Спинлоки пригодны только для защиты коротких критических секций
- Но они позволяют объединять операции в атомарные блоки!
- Давайте объединим в атомарный блок проверку флаговой переменной и засыпание (блокировку нити)?
- Большинство примитивов синхронизации так и сделаны: семафоры, мутексы, блокировки чтения-записи, критические секции Win32, синхронизованные блоки Java, да тыщи их

Проблемы программирования на блокировках

- Мертвые блокировки (deadlock)
- Голодание
- Главная проблема:
границы критических секций может определить только программист
 - Целостность данных— неформализуемое понятие
- Люди ошибаются, а ошибки определения границ секций не ловятся тестированием

Мертвая блокировка

```
// Thread 1
synchronized(lock1) {
    synchronized(lock2) {
        // stuff
    }
}
```

```
// Thread 2
synchronized(lock2) {
    synchronized(lock1) {
        // stuff
    }
}
```

Как избежать

- Захват в определенном порядке
 - Если ваша программа структурирована по слоям, возникает естественным образом
 - Но все равно за этим надо следить
- Возвращать ошибку при возникновении цикла,
 - Может вести к живой блокировке (холостому циклу на блокировке)
 - Когда вы подозреваете дэдлок, вы должны снять все блокировки
 - А для этого, как минимум, надо знать про них про все (противоречит инкапсуляции)
- Групповой (транзакционный) захват
 - Надо знать про все блокировки, которые вам понадобятся
 - Несовместимо с инкапсуляцией

Голодание

- Если вы используете упорядоченный захват, внешние (первые по порядку) блокировки приходится держать дольше
- Избегайте глубоко вложенных блокировок (легко сказать...)
- При групповом захвате, возможен сценарий, когда нить не может захватить всю группу, при том, что большинство семафоров большую часть времени свободны
- В блокировках чтения-записи, блокировка на запись может ждать очереди дольше

Гармоническое взаимодействие

- Вместо примитивов синхронизации мы используем примитивы синхронизованной передачи данных
- Разделяемых данных нет
- Критических секций нет
 - точнее, они спрятаны в операциях примитива, но они простые и для них можно доказать корректность
- Если вы передаете в трубу или сокет несогласованные данные, это легко ловится тестированием
- Можно получить большинство преимуществ параллелизма, не сталкиваясь с его главным недостатком

Примитивы гармонического взаимодействия

- Трубы
- Сокеты
- Очереди сообщений
- VMS mailboxes (псевдоустройства, которые можно использовать как очередь сообщений)
- Transputer links (что-то вроде небуферизованной трубы)
- Go channels

Go channels

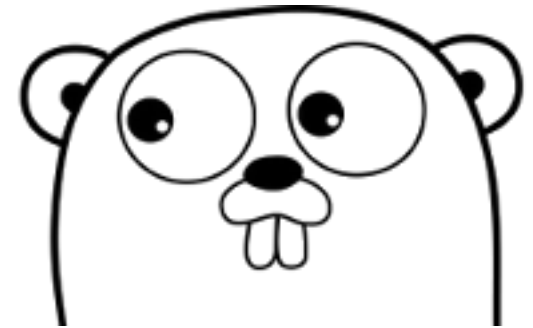
```
ch <- v    // Send v to channel ch.  
v := <-ch  // Receive from ch, and  
           // assign value to v.
```



Go channels - select

```
package main
import "fmt"
func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```



Проблемы гармонического взаимодействия

- Вы работаете с копиями данных
 - Нужна дополнительная память
 - Накладные расходы на копирование
 - Ваши данные отстают от актуального положения дел
- Что делать, если вам нужен произвольный доступ к данным?
 - Данных много, а вам нужна только небольшая их часть, и вы заранее не знаете, какая
- Что делать, если вам нужны разделяемые данные
 - Например, продажа билетов

Одно из развитий идеи гармонического взаимодействия

- Что, если передавать по каналам не данные, а запросы?
Команды или даже целые программы
- На самом деле, многие реальные приложения так и работают
 - Пример: SQL