ТЕОРИЯ И ПРАКТИКА МНОГОПОТОЧНОГО ПРОГРАММИРОВАНИЯ

Тема 3

Система команд, атомарность и порядок

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

Тема

- Краткий обзор специфики выполнения системы команд Intel x86
- Видимость результатов работы команды
- Модель упорядоченности (непротиворечивости) доступа к памяти,
- Атомарность, атомарные примитивы

Система команд x86/x64

- Многие операции модификации памяти проводятся через регистры (за некоторыми исключениями), в несколько стадий
- Большинство команд будут выполнены «целиком» или не выполнены вовсе (не путать с атомарностью!) - транзакционность
- Черный ящик-автомат: есть вход, и есть выход, причем выход принимает конечное число значений.
- Нельзя ни подсмотреть что там делается, ни разобрать процесс на стадии (а что такое 50% записи одного байта? Биты по одному «переползают» в результат?)
- Наблюдаемость результатов работы одной инструкции только путем выполнения других инструкций (результат неизвестен до момента выполнения другой инструкции).

Упорядоченность

- Когда оператор или инструкция «заканчиваются»?
- 2 должно быть перед 3
- Это гарантируется платформой и компилятором
- Порядок 5 и 6 не важен, компилятор и/или платформа может переставить местами
- Промежуточное значение после 5 может быть ненаблюдаемо вторым потоком (например, в регистре)
- Неявно возникает «наблюдатель»?

```
    …
1   int a=2,b;
2   a++;
3   b = a+1;
    …
4   int a=1,b=2;
5   a++;            // a == 2
6   b++;            // b == 3
7   a += b;         // a == 5
    …
```

Упорядоченность

- Как заставить «соблюдать условия»?
- Компилятор: volatile?
  - Если значение нужно, никогда не буферизовать в регистре, а читать из памяти
  - Никак не помогает синхронизации (хотя, казалось бы...)
  - И даже может быть компилятором проигнорировано (см пример) – практически все его указания не обязательны

```
void myFunction(void)
{
    int b;
    extern volatile int a;
    …
    b = a;
    return;
}
```

Упорядоченность

- Если заставили компилятор работать как захотели, уже все хорошо?
- Обращение к памяти: широковещательный запрос к шине
- Write Combining буфер (x86)
  - Записи: всегда в порядке выполнения команд
  - Чтение: может быть переставлено
- Гарантия порядка – барьер
  - явный или неявный
  - чтение-запись
- Часть команд пишут «мимо кеша», SSE

Примеры x86 барьеров

- Встроенный: XCHG
- Префикс lock где разрешен, LOCK CMPXCHG
- Явный: SFENCE/LFENCE/MFENCE/CPUID
  - sfence: store, sfence, mfence, serialization (cpuid)
- Через настройку типа памяти через PAT/MTRR (WB mode – тогда xcgh будет блокировать только свою линию кеша)

Упрощенно, x86

- Чтение: в любом порядке, может быть умозрительными
- Чтение может «обойти» (закончиться позже) буферизованную запись (из не одной и той же памяти)
- Запись: почти всегда в порядке программы
- Запись может быть буферизована
- Запись: не умозрительно –только для реально исполненых инструкций
- Данные из буфера записи могут быть переданы инструкции чтения напрямик внутри процессора
- И чтение, и запись не могут «обойти» любых инструкций ввода-вывода, инструкций с префиксом LOCK или сериализующих инструкций
- Чтение не может «обойти» MFENCE/LFENCE.
- Запись не может «обойти» MFENCE/SFENCE.

- Для SMP системы
- Отдельные процессоры используют правила, приведенные выше
- Записи одного процессора наблюдаются в том же порядке всеми остальными
- Записи отдельных процессоров на системную шину не упорядочены по отношению к друг другу.

- PREFETCH (предварительная загрузка) никак не упорядочивается MFENCE/LFENCE/SFENCE

Влияние порядка

Поток 1

```
threadOne = 1;              // место для барьера
victim = 1;                 // место для барьера
while (threadTwo && victim == 1)
   ;
// critical section
threadOne = 0;

// то, что мы ожидаем
[threadOne] <- 1
[victim] <- 1
reg0 <- [threadTwo]
reg1 <- [victim]

// то, что мы можем получить
reg0 <- [threadTwo]
[threadOne] <- 1
[victim] <- 1
reg1 <- [victim] // в критическую секцию!
```

Поток 2

```
threadTwo = 1;              // место для барьера
victim = 2;                 // место для барьера
while (threadOne && victim == 2)
   ;
// critical section
threadTwo = 0;

// то, что мы ожидаем
[threadTwo] <- 1
[victim] <- 2
reg0 <- [threadOne]
reg1 <- [victim]

// то, что мы можем получить
reg0 <- [threadOne]
[threadTwo] <- 1
[victim] <- 2
reg1 <- [victim] // в критическую секцию!
```

эффекты архитектуры

- Пример: Write Combining buffer
  - Является «выделенными» линиями «рядом» с кэшом
  - Слабая упорядоченность (не кэширована, протокол когерентности не работает, возможны умозрительные чтения, записи могут быть задержаны на шине)
  - Формируется по ходу записи и буферизует их до некоторых событий
  - Если хотя бы байт не записан из процессора, то переписывание в память будет выполняться «частичными» транзакциями, отрезками по 8 байт
  - Если WC 64 байта, то 16 записей по 4 байта (слову) запишутся в память единой транзакцией!
  - Но, если мы пишем 63 байта? До 8 (!) частичных транзакций! В произвольном порядке (и между разными WC так же порядок произвольный). Пример – «невыровненная» запись RMW инструкциями.

Атомарность

- Набор действий, которые могут комбинироваться только так, что для всех наблюдателей они представляют собой единую операцию с двумя возможными результатами – успехом или неудачей
- Ключевое понятие – «видимость» (и «невидимость»)
- Условия атомарности:
  - *Невидимость* - до момента завершения всех операций никто в системе (никакой процесс) не может наблюдать факт произведения изменений
  - *Восстановимость* - если операция по какой либо причине не завершилась успешно, то состояние системы должно быть полностью восстановлено к моменту до начала работы атомарной операции
- Атомарность, видимость и упорядоченность – разные понятия (см volatile и порядок)

Атомарность в x86

- Аппаратно гарантируется атомарность ≥ 486:
  - Чтение или запись байта
  - Чтение или запись слова, выровненного на 16-битовую границу
  - Чтение или письмо двойного слова, выровненного на 32-битовой границе.
- Аппаратно гарантируется атомарность ≥ Pentium:
  - Чтение или запись четверного слова, выровненного на 64-битовую границу
  - 16-битовый доступ к не кэшированным участкам памяти, которые входят целиком в 32-битовую шину данных.
- Аппаратно гарантируется атомарность ≥ P6:
  - Не выровненный 16-, 32-, и 64-битовые доступ к кэшированным участкам памяти, которые помещаются целиком в линию кэша
- Аппаратно НЕ гарантируется атомарность по умолчанию (только специальными средствами)
  - Доступ к кэшированной памяти, которая пересекает границы шины, линий кэша или страницы

Атомарные примитивы

- Чтение и запись
  - Mov
- Проверить-и-установить (test-and-set)
  - Xchg (встроенный lock)
- Извлечь-и-добавить (fetch-and-add или read-modfy-write)
  - Xadd
- Сравнить-и-поменять-местами (compare-and-swap)
  - CMPXCHG
- Загрузить-и-зарезервировать/условно-сохранить (load-link/store-conditional)
  - Отсутствует в x86

Скорость примитивов

- Если форсируют блокировку шины, то...
- Если форсируют упорядоченность операций (fence), то...
- Если используют WC буфер, то...
- Лучше всего – если ничего не используют

Выводы

- Есть много сложных понятий – упорядоченность операций, атомарность, видимость результатов для процессора и т.д.
- Большинство этих понятий практически независимы друг от друга, их нельзя путать (атомарность например не влечет упорядоченности)
- Есть «потаенные» устройства, работа с которыми влияет на поведение программ, в первую очередь с точки зрения производительности
- Неучет порядка операций в памяти может привести к неправильной работе внешне корректного алгоритма (один и тот же код может работать по разному на разных платформах – например, x86 & Sparc!)
- Все платформы, в том числе и x86, содержат аппаратно поддерживаемые атомарные примитивы синхронизации, на основании которых строятся более сложные примитивы (и технологии).

(c) А. Тормасов, 2010-11 г.

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