Лекция 4. Архитектурные особенности современных компьютеров. Упорядоченность памяти. Атомарность.

Содержание

- Видимость результатов работы команды.
- Модель упорядоченности (непротиворечивости) доступа к памяти.
- Атомарность, атомарные примитивы.

Особенности системы команд Intel

- В основном регистровые команды.
- Регистров мало.
- Обращение к памяти затратно.
- Команды атомарны и транзакционны.
- Исполнение конвейерно и каждая команда проходит все этапы.
- Существуют векторные операции.
- Операции исполняются суперскалярно.

Что значит *атомарность*?

- Команда исполняется целиком или не исполняется совсем. (восстановимость)
- Результаты исполнения команды недоступны до её завершения. (невидимость)

Завершение команды и её результаты.

Команды могут завершать исполнение в различное время. Порядок завершения команд в общем случае может нарушаться.

```
int x = z * 12;
int y = k + 6;
```

Вероятно, результат `x` будет получен позже результата `y`.
Как мы об этом узнаем?

Завершение команды и её результаты.

Команды могут завершать исполнение в различное время. Порядок завершения команд в общем случае может нарушаться.

```
int x = z * 12;
int y = k + 6;
```

Вероятно, результат `x` будет получен позже результата `y`.
Как мы об этом узнаем?
Только при попытках считать значения `x` и `y`.
Должны ли `x` и `y` находиться в памяти?

Завершение команды и её результаты.

Команды могут завершать исполнение в различное время. Порядок завершения команд в общем случае может нарушаться.

```
int x = z * 12;
int y = k + 6;
```

Вероятно, результат `x` будет получен позже результата `y`.
Как мы об этом узнаем?
Только при попытках считать значения `x` и `y`.
Должны ли `x` и `y` находиться в памяти?
Нет. Компилятор для использования может поместить их в регистры.

Завершение команды и её результаты

- Процессор **может** не исполнять команды, результаты которых не наблюдаются.
- Эффективное время исполнения таких инструкций равно нулю.
- Как только требуется наблюдать результат операции, эффективное время исполнения становится ненулевым.
- Это относится не ко всем инструкциям, только к тем, которые не имеют побочных эффектов.
- Инструкции чтения/записи памяти побочными эффектами обладают.

Результаты исполнения команды

- Компиляторы в режиме оптимизации удаляют команды с ненаблюдаемыми результатами.

```
int main() {
    int a,b,c,d;
    scanf("%d %d", &a, &b);
    c = a + b;
    d = a + b;
    printf("%d\n", c);
}
```

```
    leaq    L_.str(%rip), %rdi
    leaq    -4(%rbp), %rsi
    leaq    -8(%rbp), %rdx
    xorl    %eax, %eax
    callq   _scanf
    movl    -8(%rbp), %esi
    addl    -4(%rbp), %esi
    leaq    L_.str1(%rip), %rdi
    xorl    %eax, %eax
    callq   _printf
```

Работа с памятью. Пример. Продолжение.

Если компилировать без оптимизации, всё корректно.

Скомпилируем пример в ассемблер без флагов оптимизации.

```
void prefetch(long *big, int SIZE) {
   for (long i=0; i<SIZE; i+=512) {
      long mp = big[i];
   }
}
```

```
        pushq   %rbp
        movq    %rsp, %rbp
        popq    %rbp
        retq
```

Компилятор обнаружил, что переменная `mp` не используется и удалил код.

Кто виноват? Что делать?

Работа с памятью. Пример. Продолжение.

Добавить спецификатор *volatile*.

```
void prefetch(volatile long *big,
  int SIZE) {
   for (long i=0; i<SIZE; i+=1024){
      long mp = big[i];
   }
}
```

```
    pushq   %rbp
    movq    %rsp, %rbp
    xorl    %eax, %eax
    testl   %esi, %esi
    jle     LBB1_3
    movslq  %esi, %rcx
LBB1_2:
    movq    (%rdi,%rax,8), %rdx
    addq    1024, %rax
    cmpq    %rcx, %rax
    jl      LBB1_2
LBB1_3:
    popq    %rbp
    retq
```

Спецификатор `volatile`

- Спецификатор `volatile` запрещает *компилятору* использовать какую-либо оптимизацию при работе с указанным адресом памяти.
- При любом обращении к этому адресу требуется физическое обращение.

Порядок записи в память

- Операция обращения к памяти есть широковещательный запрос шине. Операции записи буферизуются и могут осуществляться в удобное для процессора время.
- Актуализация (физическая запись в память) может произойти самостоятельно, при операции чтения или по исполнению барьера.
- Барьером может быть специфическая машинная команда `sfence`, `mfence` или `lfence` или сериализующая команда (`xchg`, `cpuid`, команды с префиксом `lock`, команды `in`/`out` и другие).

Имеется много режимов работы с памятью.

- *WriteThrough*. Операция записи в память завершается, как только контроллер памяти записал данные физически.
- *WriteBack*. Операция записи в память завершается, как только данные переданы контроллеру памяти. Контроллер памяти самостоятельно решает, когда и как производить запись физически.

Понятие *умозрительное чтение* — запрос к памяти на чтение может не выполняться физически.

Регистровые команды

В системе команд имеются такие операции

```
inc     [eax]
add     ebx, gvar
```

Время их исполнения настолько велико, что в документации Intel для программирования они не рекомендуются для ряда процессоров. Вместо них предлагается писать так:

```
;replace inc     [eax]
mov edx, [eax]
inc edx
mov [eax], edx

; replace add     ebx, gvar
mov edx, gvar
add ebx, edx
mov gvar, edx
```

Каждая из команд перед операцией с областью памяти требует использования временного регистра.

Регистров мало

- В 32-разрядном режиме регистров всего 7.
- Для ряда операций требуются специфические регистры.
- Для деления требуется загрузить два регистра, `EAX` и `EDX` для делимого (регистр `EDX` содержит знак делимого).
- Два-три регистра обычно требуются для индексных операций.
- В скомпилированном коде присутствует фальшивое разделение регистров (регистр содержит попеременно значение различных переменных).

Обращение к памяти затратно

- Компилятор старается осуществить как можно больше действий с регистрами.
- Имеется несколько уровней кэш памяти.

Команды атомарны и транзакционны

- Команда исполняется целиком или не исполняется совсем (транзакционность).
- Существуют команды, которые выполняются много тактов.
- Внешние прерывания могут прервать исполнение команды и передать управление обработчику прерываний.
- Обработчики прерываний исполняют минимальную необходимую работу и переводят запрос в очередь для обработки.
- Обработчик прерывания — машинный код, программа, и он изменяет процессорные регистры и память.
- После завершения обработки прерывания исполнявшаяся команда возобновляет исполнение таким образом, что оказывается невозможным определить, прерывалось ли исполнение команды.

Команды: атомарность и транзакционность

- Промежуточные результаты будут недоступны до её завершения.
- Современные компиляторы «длинные инструкции» не генерируют.
- В программах, откомпилированных старыми компиляторами такое встречается.
- Старые компиляторы это делать могли.

```
rep movsw
```

```
@loop:
  tst  ecx,ecx
  je   @endloop
  mov scratch, ds:[esi]
  add  esi,4
  mov es:[edi], scratch
  add  edi,4
  dec ecx
  jmp @loop
@endloop:
```

Атомарность исполнения команды

- Если `ECX` имеет большое значение, команда исполняется миллисекунды.
- Вероятность возникновения прерывания велика.
- Что мы увидим в регистрах `EDI`, `ESI`, `ECX` в обработчике прерывания?
- То же, что и перед началом исполнения.
- Что мы увидим после исполнения?
- Изменённые регистры.

Атомарность операций

- Отдельные команды атомарны.
- Даже простое изменение переменной требует нескольких команд.
- Для простого увеличения `volatile` переменной компилятор выдаст несколько команд.

```
; x += 10;
mov eax,[ebp+4]
add eax, 10
mov [ebp+4],eax
```

- Это чревато проблемами при многопоточном исполнении.

Атомарность при многопоточном исполнении

- Два потока исполняют этот код.

```
; *x += 10;
mov eax, [esi]
add eax, 10
mov [esi],eax
```

```
; *x += 10;
mov eax, [esi]
add eax, 10
mov [esi],eax
```

- На многоядерном процессоре проблемы очевидны.
- Так ли они очевидны на одноядерном?

Атомарные инструкции и атомарные операции

- С нашей точки зрения мы исполняем атомарную **операцию**.
- С точки зрения процессора исполняется несколько атомарных **инструкций**.
- Атомарных инструкций мало и они не все соответствуют операциям в нашем понимании.

Атомарные инструкции и атомарные операции

Требуются инструкции атомарного изменения данных.
Что исполняется атомарно?

1. Атомарное чтение и запись
   1. Все чисто регистровые операции
   2. Чтение/запись байта
   3. Чтение/запись выровненных данных, 16, 32 и 64 битных
   4. На современных: любые кэшируемые данные в линейке кэша
2. Инструкции `XCHG`
   `xchg [ebp+16], eax`
   - читает `[ebp+16]` в `scratch`
   - пишет `eax` в `[ebp+16]`
   - пишет `scratch` в `eax`

Многопоточная работа с памятью

Новый взгляд на память: память — устройство, обрабатывающее сообщения.

- Запись в память — посылка сообщения с адресом и значением.
- Чтение из памяти — посылка сообщения с адресом и получение сообщения со значением.

Диаграмма работы с памятью

Различные потоки нагружают память сообщениями.

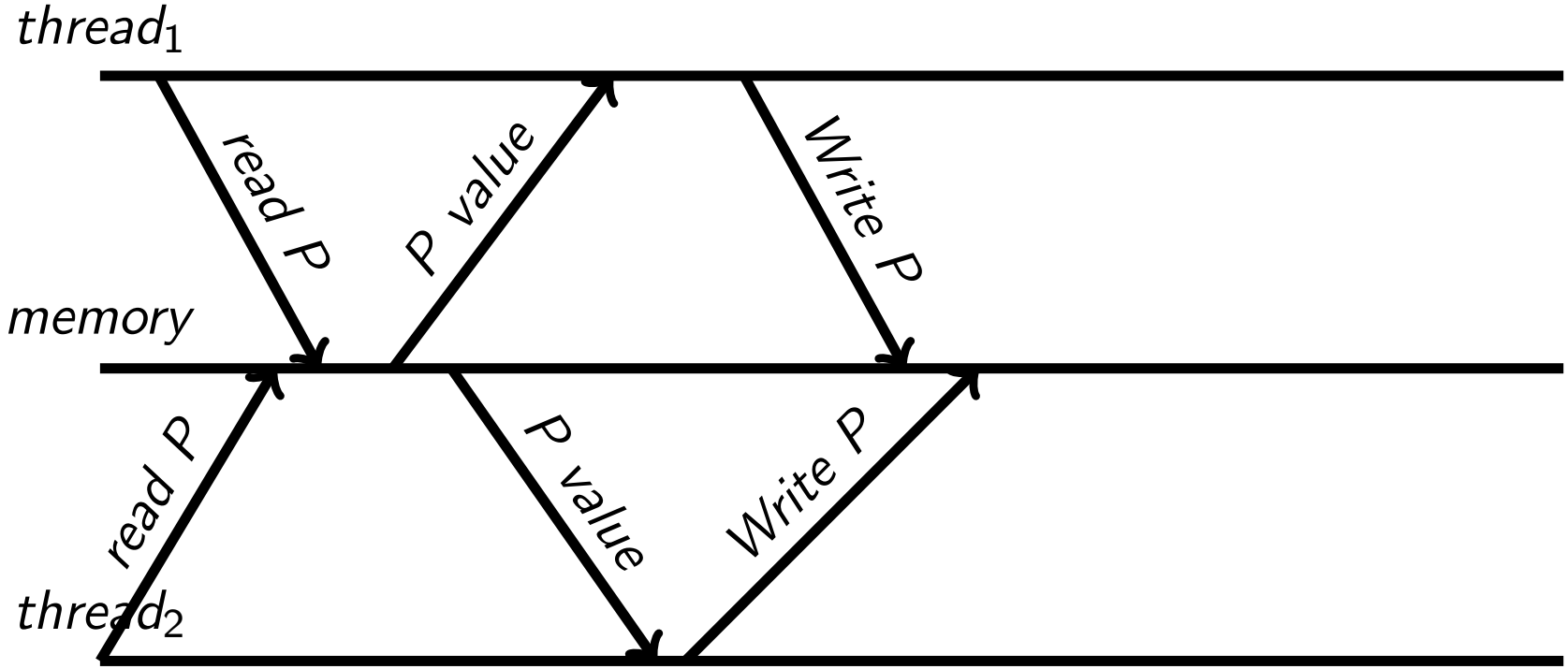


Диаграмма работы с памятью

При начале операции $Thread_1$ в красной зоне возникает конфликт.

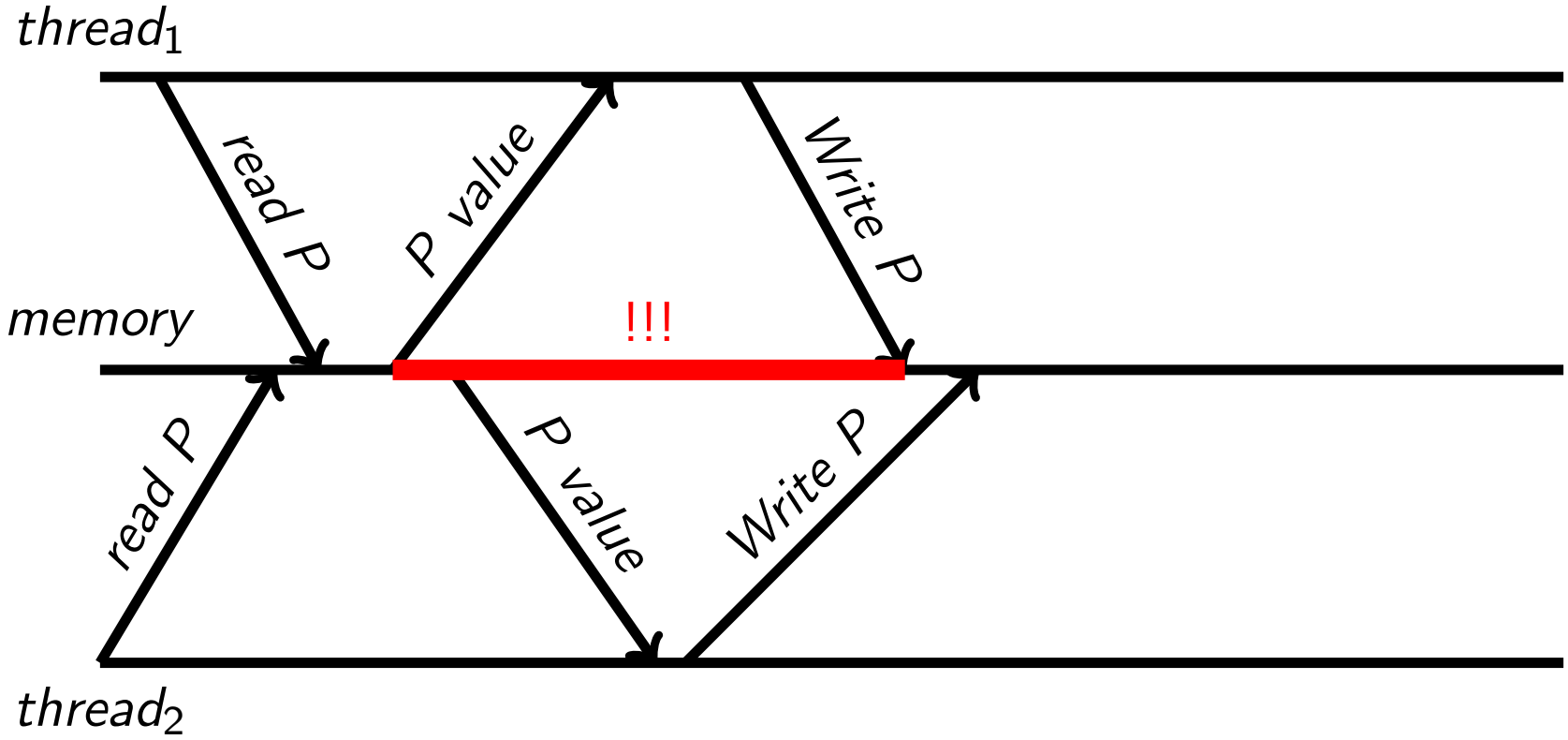


Диаграмма работы с памятью

Любая операция $Thread_2$ в этой зоне недетерминирована.

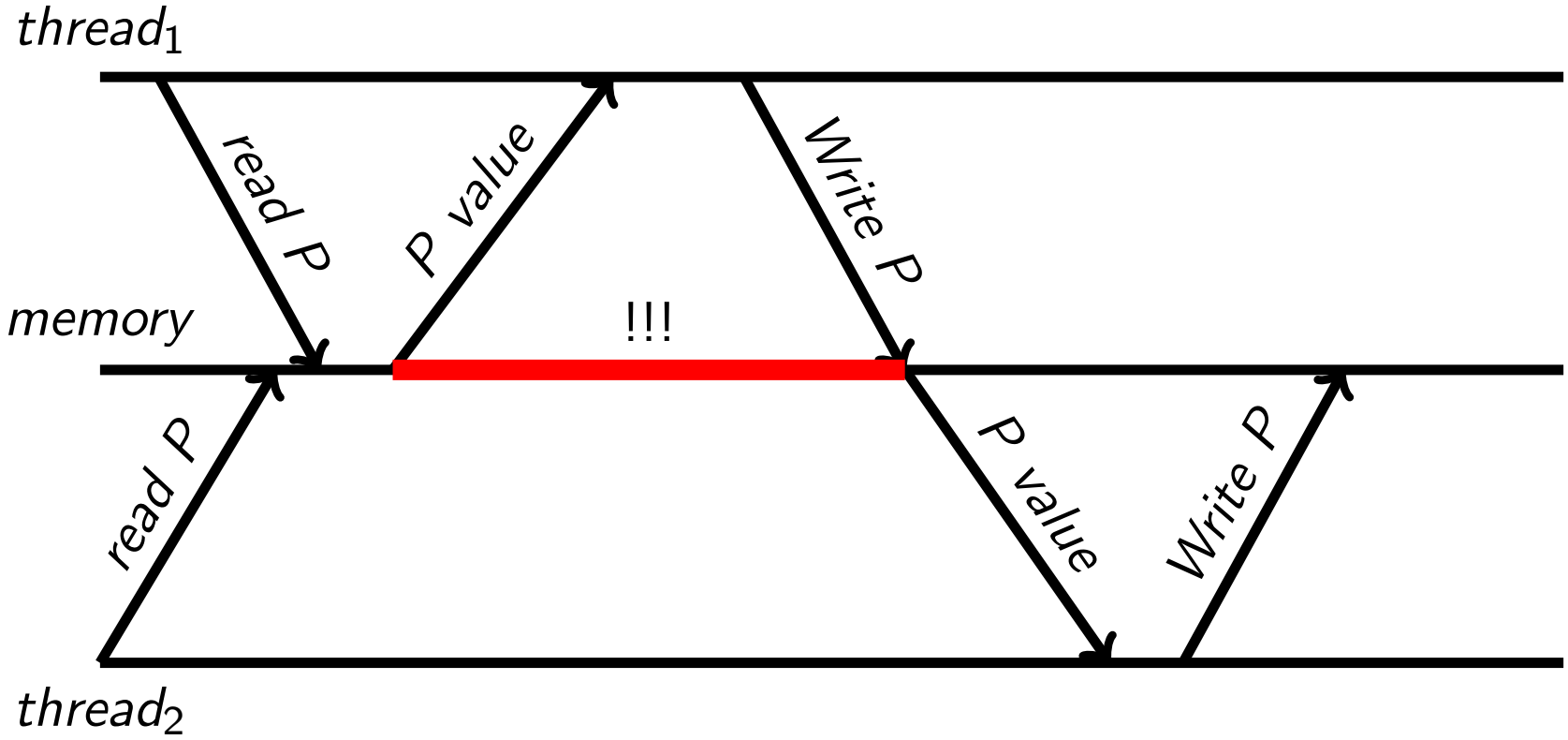
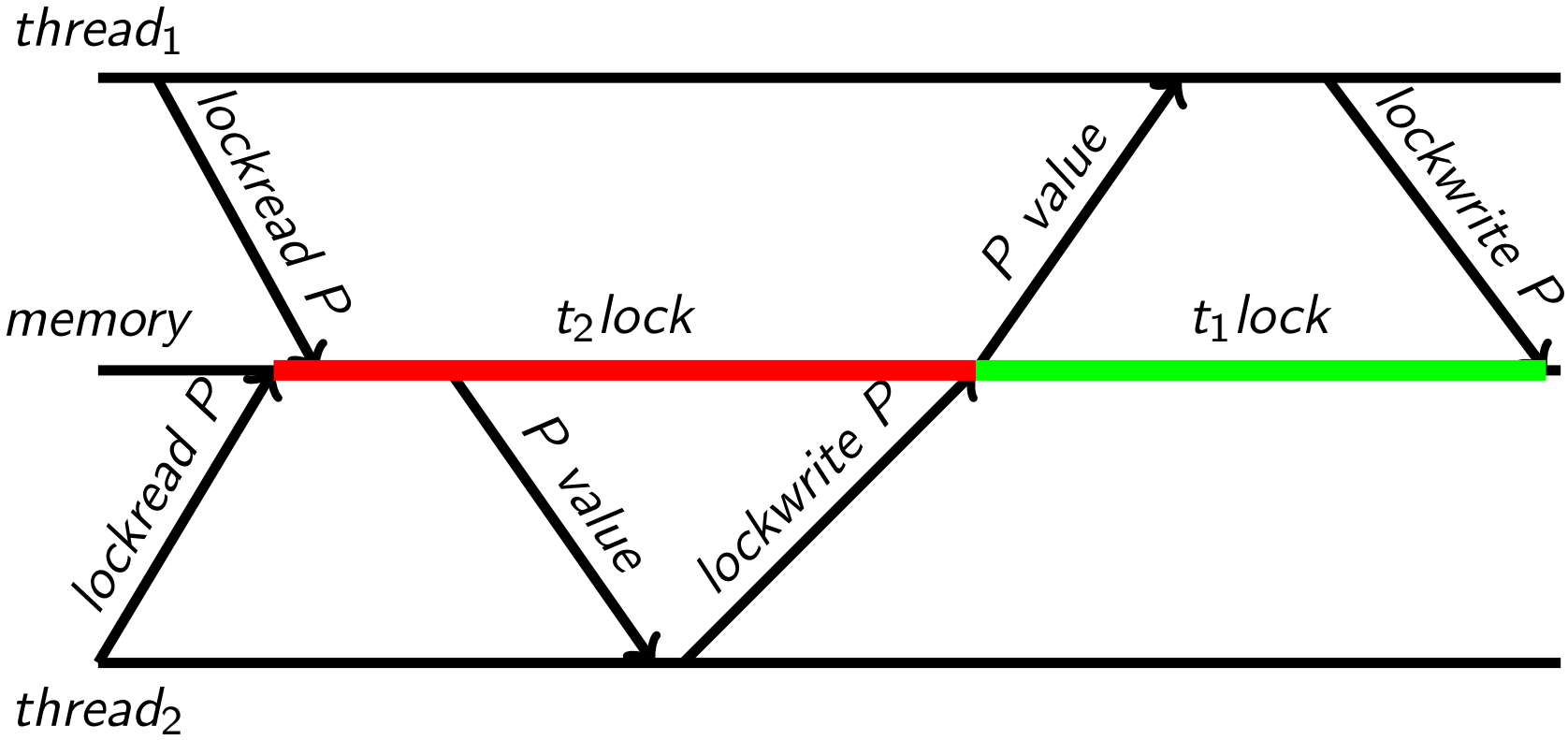


Диаграмма работы с памятью

Правильный порядок исполнения блокирующих операций.



Семантика блокировки префикса `lock`

Машинная команда с префиксом `lock`:

- Завершает все операции записи на шине памяти.
- Запрещает все операции с памятью до завершения машинной команды.
- Исполняет команду и посылает результат в память.
- Как только результат команды достиг памяти, она разблокируется.

Блокирующиеся операции в gcc/c++/clang

- Операции делятся на пре- и пост-

  `type = one of hardware scalar integer types`

```
type __sync_fetch_and_add(type *p, type value);
type __sync_fetch_and_sub(type *p, type value);
type __sync_fetch_and_or(type *p, type value);
type __sync_fetch_and_and(type *p, type value);
type __sync_fetch_and_xor(type *p, type value);
type __sync_fetch_and_nand(type *p, type value);
```

  Функции возвращают значение переменной *перед* изменением.

Блокирующиеся операции в gcc/c++/clang

- Вариант пост-операций:

```
type = one of hardware scalar integer types
type __sync_add_and_fetch(type *p, type value);
type __sync_sub_and_fetch(type *p, type value);
type __sync_or_and_fetch(type *p, type value);
type __sync_and_and_fetch(type *p, type value);
type __sync_xor_and_fetch(type *p, type value);
type __sync_nand_and_fetch(type *p, type value);
```

Функции возвращают значение переменной *после* изменения.

Пример исполнения INC/DEC

Инструкции ++ и -- не атомарны. Программа работает неверно.

```
using mytype = unsigned long;
void *func_add(void *arg) { //  thread 1
  volatile mytype *p = (mytype *)arg;
  for (int i = 0; i < 10000000; i++) {
    (*p)++;
  }
  return NULL;
}

void *func_sub(void *arg) { //thread 2
  volatile mytype *p = (mytype *)arg;
  for (int i = 0; i < 10000000; i++) {
    (*p)--;
  }
  return NULL;
}
```

Пример исполнения INC/DEC

Используются атомарные операции. Программа работает верно.

```
using mytype = unsigned long;
void *func_add(void *arg) {
  volatile mytype *p = (mytype *)arg;
  for (int i = 0; i < 10000000; i++) {
    __sync_fetch_and_add(p, 1);
  }
  return NULL;
}

void *func_sub(void *arg) {
  volatile mytype *p = (mytype *)arg;
  for (int i = 0; i < 10000000; i++) {
    __sync_fetch_and_sub(p, 1);
  }
  return NULL;
}
```

Атомарные операции в Visual C++

Используется семейство intrinsic-функций Interlocked*.

```
LONG InterlockedAdd(LONG volatile *p, LONG value);
LONG InterlockedSub(LONG volatile *p, LONG value);
LONG InterlockedAnd(LONG volatile *p, LONG value);
LONG InterlockedOr(LONG volatile *p, LONG value);
LONG InterlockedXor(LONG volatile *p, LONG value);
LONG InterlockedBitTestAndComplement(LONG volatile *p,
LONG value);
```

Всего 131 функция!

Особенности реализации атомарных операций

- Технически атомарные операции реализуются через префикс `lock`, которые блокирует шину на время исполнения операции.

Атомарные операции в C++11

C++11 созрел для реализации переносимых атомарных инструкций.

```
#include <atomic>
using namespace std;
using mytype = unsigned long;
void *func_add(void *arg) {
  auto *p = (atomic<mytype> *)arg;
  for (int i = 0; i < 10000000; i++) {
    (*p)++;
  }
  return NULL;
}
```

```
  pushq   %rbp
  movq    %rsp, %rbp
  movl    $10000000, %eax
LBB1_1:
  lock
  incq    (%rdi)
  addl    $-1, %eax
  jne     LBB1_1
  xorl    %eax, %eax
  popq    %rbp
  retq
```

Спасибо за внимание.

Следующая тема — языки программирования и многопоточность. Конкурентный доступ.