Многопоточность

Егор Суворов

Курс «Парадигмы и языки программирования», подгруппа 3

Среда, 19 октября 2016 года

План занятия

1. Параллельные вычисления
   - Зачем
   - Как
2. Практические грабли
   - Простое приложение на pthread
   - Состояние гонки
   - Гонка данных
   - Взаимное исключение
   - Не пытайтесь повторить это дома
3. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
4. Бонус
5. Домашнее задание

1. Параллельные вычисления
   - Зачем
   - Как
2. Практические грабли
   - Простое приложение на pthread
   - Состояние гонки
   - Гонка данных
   - Взаимное исключение
   - Не пытайтесь повторить это дома
3. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
4. Бонус
5. Домашнее задание

Скорость вычислений

- Хочется обрабатывать всё бóльшие объёмы информации всё быстрее.
- Пример: эмуляция движений воздуха на планете Земля за ближайшие 48 часов (прогноз погоды).
- Пока работает эмпирический закон Мура: каждые два года плотность транзисторов удваивается.
- Раньше это означало увеличение частоты процессора в два раза.
- Уже нет: процессор с частотой 2.8 ГГц был представлен в 2004 году (Pentium 4 Prescott).
- С тех пор скорость работы повышалась, но другими способами: размер кэша, скорость памяти, периферии...
- Уже уткнулись в ограничения размера процессора из-за скорости света.

Параллелизм

- Иногда можно работать быстрее, не увеличивая частоту, распараллелив команды:

```
int x = a * b * 10;   // Нужен блок умножения.
int y = a / b;        // Нужен блок деления.
```

- Процессоры умеют это автоматически детектировать без участия программистов.
- Компиляторы умеют передвигать операции так, чтобы процессору было проще.
- В последние годы активно появляются многоядерные процессоры: впихнуть второе ядро оказалось проще оптимизации физических процессов.
- Также можно использовать мощь бо́льшего числа компьютеров (Folding@Home).

На домашнем компьютере

Идеи распараллеливания полезны и где-то, кроме ускорения:

- Обычные задачи дома не требуют большой вычислительной мощи:
    - Процесс обычно ждёт реакции пользователя, диска или сети.
    - Вычисления длятся не больше нескольких секунд.
- Хочется свободно переключаться между приложениями и слушать музыку в фоне.
- Если есть ресурсоёмкая задача, нестрашно, если она будет выполняться чуть медленнее.
- На телефоне одно ядро может целиком отрисовывать нетормозящий интерфейс, а другое — производить вычисления.

Параллельные алгоритмы

- Некоторые алгоритмы параллелятся просто:

```
int sum = 0;
for (int x : values) sum += x;
```

- Некоторые — естественно и на уровне железа:

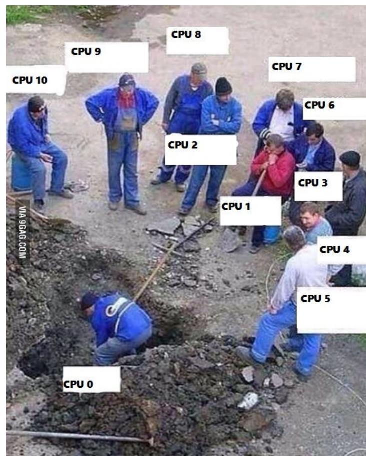
```
char buf1[100], buf2[100];
fread(file_on_disk1, 1, sizeof buf1, buf1);
fread(file_on_disk2, 1, sizeof buf2, buf2);
```

- Некоторые не параллелятся:

```
int steps = 0;
for (int x = 1; x != 0; x = f(x)); steps++;
```

- Надо писать специальные алгоритмы для распределённых вычислений.

Иллюстрация



Простое добавление ядер не увеличивает производительность!

В прикладном хозяйстве

- Современные ОС различают *потоки* и *процессы*.
- Процесс — это обычно одно приложение (браузер, IDE, веб-сервер...), у которого может быть много потоков.
- Например: у браузера один поток на вкладку; у веб-сервера — один поток на клиента.
- Изначально у процесса есть только один поток (*главный*), он может создавать другие.
- Более строго: процесс — это некоторое множество потоков, у которых общая память и другие ресурсы (открытые файлы).
- Поток — это что-то, выполняющее некий код (есть отдельный стек, свои данные в регистрах процессора, свой код).

С точки зрения программиста

- На разных ОС разные методы для работы с процессами или потоками.
- Напрямую API уровня ОС, как обычно, никто не использует.
- В языке высокого уровня (Java, Python) обычно есть соответствующая библиотека.
- Также есть другие классические библиотеки и стандарты:
  - pthread — Posix Thread, стандарт в C. Будем использовать.
  - OpenMP — высокоуровневое распараллеливание для C/C++/Fortran.
  - CUDA — вычисления на графических картах (ядер тысячи, но они умеют меньше, чем CPU).

Типичный псевдокод-1

```
void draw() {
    while (true) {
        wait_for_events();
        process_updates();
        process_mouse_events();
        repaint();
    }
}
int main() {
    Thread draw_thread(draw);
    draw_thread.start();
    // ...
    add_rectangle(10, 10, 30, 40);
    // ...
}
```

Типичный псевдокод-2

```
void process_client(Client client) {
    string request = client.read();
    string answer = "I've got " + request;
    client.write(answer);
}
int main() {
    while (true) {
        Client client = get_next_client();
        Thread(process_client, client).start();
    }
}
```

Типичный псевдокод-3

```
void merge_sort(int l, int r) {
    if (l + 1 == r) return;
    Thread t1(merge_sort, l, (l + r) / 2);
    Thread t2(merge_sort, (l + r) / 2, r);
    t1.start(); t2.start(); // Запускаем потоки.
    t1.join(); t2.join();   // Ждём завершения.
    merge(l, r);
}
```

1 Параллельные вычисления
- Зачем
- Как

2 Практические грабли
- Простое приложение на pthread
- Состояние гонки
- Гонка данных
- Взаимное исключение
- Не пытайтесь повторить это дома

3 Обмен сообщениями
- Простая реализация
- События
- Условные переменные

4 Бонус

5 Домашнее задание

Что такое pthread

- Стандартный интерфейса функций для работы с потоками (POSIX Threads).
- Есть реализации под Windows, Linux и другие ОС.
- Стандарт при разработке программ на C.
- Имена функций и типов начинаются с `pthread_`.
- В Linux можно получить справку, набрав `man <имя функции>` в консоли.
- Под остальными — то же самое, но в гугле.

Пример кода

```
void* worker(void* arg) {
    printf("Hello from thread! arg=%d\n", *(int*)arg);
    *(int*)arg += 10;
    return arg;
}
int main(void) {
    pthread_t id;
    int data = 1234;
    assert(pthread_create(&id, NULL, worker, &data) == 0);
    void* retval;
    assert(pthread_join(id, &retval) == 0);
    assert(retval == &data);
    printf("data is %d\n", data);
    return 0;
}
```

Как живут потоки

- При создании потока при помощи pthread_create указывается функция и её аргумент — один указатель на что угодно.
- Вернуть функция тоже может указатель на что угодно.
- Поток завершается, когда функция делает return или pthread_exit.
- Указатель на поток хранится в переменной типа pthread_t.
- При создании потока он сразу начинает выполняться.
- pthread_join делает следующее:
  1. Ждёт окончания работы потока.
  2. Освобождает все ресурсы потока (стек).
  3. Возвращает то, что вернула функция потока.
- Когда main делает return 0 или вы вызываете exit(0), умирает весь процесс со всеми потоками.
- Но в main можно сделать pthread_exit, если очень хочется, тогда процесс не умрёт, пока живы потоки.

Упражнение: сборка кода

1. Качаем решение с GitHub.
2. `gcc 01-simple.c -o 01-simple -std=gnu99 -Wall -Wextra -Werror -pthread` или аналог в вашей IDE.
3. `./01-simple`
4. Ожидаемый вывод:

```
Hello from thread! arg=1234
data is 1244
```

Несколько замечаний про C

- На языке C лучше включить все предупреждения компилятора (warnings), в GCC это делают ключи `-Wall`, `-Wextra`.
- Если вы включили предупреждения — их лучше сразу трактовать как ошибки (`-Werror`), иначе быстро научитесь их игнорировать.
- Если аргумент функции не используется, после него в GCC надо писать `__attribute__((unused))`.
- Из функции всегда надо что-то вернуть (хотя бы `NULL`).
- Никогда не начинайте название переменной с нижнего подчёркивания!
- Константы задаются при помощи `#define SOME_CONST (value)`.
- Встроенного типа `bool` нет, надо подключить `#include <stdbool.h>`

Иллюстрация

Несколько замечаний про pthread

Про потоки и pthread:

- Использовать `void* arg` и возвращаемое значение для передачи данных необязательно.
- Вся память внутри процесса одинаково доступна всем потокам на чтение и запись.
- `void* arg` возникает только тогда, когда надо запустить потоки на разных данных.
- Что произойдёт, если мы забудем `join` и `main` завершится до начала `worker`?

Несколько замечаний про pthread

Про потоки и pthread:

- Использовать `void* arg` и возвращаемое значение для передачи данных необязательно.
- Вся память внутри процесса одинаково доступна всем потокам на чтение и запись.
- `void* arg` возникает только тогда, когда надо запустить потоки на разных данных.
- Что произойдёт, если мы забудем `join` и `main` завершится до начала `worker`? Неопределённое поведение — `worker` попытается изменить переменную `data`, которая уже исчезла.

Кто освобождает ресурсы?

На самом деле в pthread есть два типа потоков: joinable и detached.
Joinable:

- Тип по умолчанию.
- На таком потоке должен быть ровно один раз вызыван метод `pthread_join`, который освободит ресурсы и сообщит, что поток вернул.
- Если не вызвать — ресурсы не будут освобождены до конца программы.
- Если вызвать дважды — второй вызов может уронить программу или вернуть неверный результат.

Detached:

- Система автоматически освободит ресурсы как только поток завершится.
- Нельзя вызывать `pthread_join` и получать возвращаемое значение — его негде хранить после окончания работы.

В других системах

- Joinable/detached также используется в Java.
- В Windows (не в pthread под Windows!) другая концепция:
  - Указатель на поток — сложный объект, который надо запрашивать у ОС и освобождать (как `FILE*`), а не просто переменная.
  - Ресурсы потока освобождаются, когда он завершился и на него больше нет указателей.
  - Нет разделения joinable/detached.
  - Если кто-то может спросить состояние потока — у него есть указатель, значит, ресурсы потока ещё не освобождены.

Упражнение

1. Измените код так, чтобы data стала глобальной переменной (после этого arg не нужен).
2. Вызовите pthread_detach на втором потоке после запуска.
3. Убедитесь, что программа упала.
4. Уберите вызов pthread_join и printf из основного потока.
5. Убедитесь, что второй поток не всегда успевает отработать.
6. Добавьте вызов pthread_exit в main.
7. Убедитесь, что после приложение перестало закрываться до окончания работы всех потоков.

Код

1. Параллельные вычисления
   - Зачем
   - Как
2. Практические грабли
   - Простое приложение на pthread
   - Состояние гонки
   - Гонка данных
   - Взаимное исключение
   - Не пытайтесь повторить это дома
3. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
4. Бонус
5. Домашнее задание

Упражнение

- Возьмите код.
- При желании можете скачать Makefile.
- Убедитесь, что на экран выводится строка.
- Запустите второй поток, который выводит на экран другую строку.
- Найдите место, где первая строка сменяется второй.
- Удивитесь.

Код

Объяснение

- Потоки выполняют команды «одновременно». Если есть доступ к общему ресурсу (экран), то порядок не определить.
- Поэтому символы выводятся вперемешку.
- *Состояние гонки* (*race condition*) — это когда результат работы зависит от того, в каком порядке потоки выполняли команды.
- Самая популярная ошибка у начинающих.
- Операция называется *атомарной*, если она всегда выполняется «за один такт», то есть другие потоки не видят её частично выполненной.
- `writeln` выше не атомарна.

Упражнение

1. Сделайте счётчик:
    - Второй поток в цикле увеличивает глобальную переменную `data` до $N = 5 \cdot 10^8$.
    - Основной поток (`main`) выводит на экран текущее значение `data` в цикле $M = 1000$ раз.
    - Отключите оптимизации компилятора (ключ `-O2` или схожий не нужен).
2. Убедитесь, что программа выводит на экране увеличивающиеся значения, а в конце — N.
3. Поиграйте со значением M, чтобы убедиться, что в конце всегда выводится N.
4. Сделайте так, чтобы основной поток выводил на экран только чётные значения `data` и увеличьте M.
5. Что теперь происходит?

Мой код: счётчик, чётный счётчик.

Объяснение

Возможная последовательность действий:

- Основной поток: `if (data % 2 == 0)` → `true`.
- Второй поток: `data++`.
- Основной поток: `printf`.

Как исправить?

Объяснение

Возможная последовательность действий:

- Основной поток: `if (data % 2 == 0)` → `true`.
- Второй поток: `data++`.
- Основной поток: `printf`.

Как исправить?

- Можно на каждой итерации записать значение `data` в локальную `data_snapshot` (снимок) и работать с ним.
- Работает только если чтение одной переменной атомарно.
- Не работает, если у нас много переменных мы не можем сделать атомарный снимок (классическая задача).

Иллюстрация

Упражнение

- Добавьте снятие снимков в свой счётчик.
- Убедитесь, что все значения теперь чётные.
- Запустите второй поток-счётчик, который тоже увеличивает `data`.
- Что произошло?

Мой код: счётчик со снимками, два счётчика.

Объяснение

1. На уровне железа `data++` происходит так:
   - Считай значение `data` из памяти.
   - Прибавь единицу.
   - Положи `data+1` на то же место в памяти.
2. Порядок операций между разными потоками произвольный.

Операция	data
1: read → 10	10
1: +1 → 11	10
1: write(11)	11
2: read → 11	11
2: +1 → 12	11
2: write(12)	12

Операция	data
1: read → 10	10
1: +1 → 11	10
2: read → 10	10
2: +1 → 11	10
1: write(11)	11
2: write(11)	11

А что вообще атомарно?

Полезные советы

- Что атомарно — очень сильно зависит от платформы, языка и ключей компиляции («модель памяти»).
- Не пытайтесь угадать.
- Не пытайтесь самостоятельно писать код, зависящий от атомарности.
- В некоторых языках бывает `AtomicInteger` и похожие структуры.
- За ними тоже надо аккуратно следить, обычно не используют.

1. Параллельные вычисления
   - Зачем
   - Как
2. Практические грабли
   - Простое приложение на pthread
   - Состояние гонки
   - Гонка данных
   - Взаимное исключение
   - Не пытайтесь повторить это дома
3. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
4. Бонус
5. Домашнее задание

Как избежать гонок

- Можно обозначить кусок кода как *критическую секцию* (critical section).
- Каждую критическую секцию может выполнять максимум один поток.
- Если весь доступ к общим данным обозначить как критическую секцию, то он станет де-факто атомарным.
- С каждой критической секцией ассоциируется *блокировка* (lock).
- При входе в секцию блокировку надо *взять/захватить* (acquire).
- При выходе из секции блокировку надо *отпустить* (unlock/release).
- Другие названия блокировок: монитор (monitor), мьютекс (mutex, **mut**al **ex**clusion).
- Обычно реализованы на уровне ОС и все операции с ними медленные.

Некорректный пример

```
int data;
void* worker(void* arg __attribute__((unused))) {
    pthread_mutex_t m;
    pthread_mutex_init(&m, NULL);
    for (int i = 0; i < N; i++) {
        pthread_mutex_lock(&m);
        data++;
        pthread_mutex_unlock(&m);
    }
    pthread_mutex_destroy(&m);
    return NULL;
}
```

Код.

Корректный пример

```
int data;
pthread_mutex_t m;
void* worker(void* arg __attribute__((unused))) {
    for (int i = 0; i < N; i++) {
        pthread_mutex_lock(&m);
        data++;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}
// ...
  pthread_mutex_init(&m, NULL);
// ...
  pthread_mutex_destroy(&m);
// ...
```

Упражнение

1. Добавьте мьютекс в двойной счётчик.
2. Уменьшите N на несколько порядков (мьютексы сильно замедляют программу).
3. Убедитесь, что всегда выводится $2N$.
4. Добавьте мьютекс в writeln.

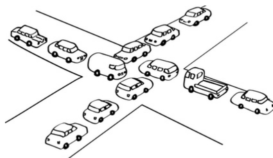
Исправленный двойной счётчик, исправленный writeln.

Блокировка

- pthread_mutex_lock блокируется и ждёт, пока блокировка не станет доступна для захвата.
- Есть ли проблемы в следующем псевдокоде?

```
void thread1() {
    m1.lock(); m2.lock();
    // ...
    m2.unlock(); m1.unlock();
}
void thread2() {
    m2.lock(); m1.lock();
    // ...
    m1.unlock(); m2.unlock();
}
```

Взаимная блокировка



- Может случиться проблема:
  1. Поток 1 захватывает `m1`.
  2. Поток 2 захватывает `m2`.
  3. Поток 1 не может захватить `m2`.
  4. Поток 2 не может захватить `m1`.
  5. Оба потока встали в *deadlock* (*взаимная блокировка*).
- Решение: всегда брать блокировки в одном и том же порядке. Тогда можно доказать, что deadlock такого вида не образуется.
- Если ввести линейный порядок на мьютексах не получается, у вас проблема.

Мьютексы — не панацея

```
void inc() {  // Атомарна.
    m.lock(); data++; m.unlock();
}
void double_inc() {  // Не атомарно.
    inc(); inc();
}
void check() {
    m.lock();
    assert(data % 2 == 0);
    m.unlock();
}
```

- Код выше может упасть, даже если вызывать только `double_inc`.
- Мьютекс в простом случае лишь добавляет атомарности.
- Две атомарные операции подряд, как и раньше, атомарную вместе не образуют.

Reentrant

```
void inc() {  // Атомарна.
    m.lock(); data++; m.unlock();
}
void double_inc() {  // Атомарно?
    m.lock(); inc(); inc(); m.unlock();
}
```

- `double_inc` заблокируется, так как `inc` попробует взять мьютекс второй раз.
- Есть специальный вид мьютексов, которые позволяют захватывать себя ещё раз из того же потока, называется *reentrant*.
- Их обычно не используют — они сложнее в реализации.

Решение

Вынести специальную функцию для внутреннего использования, которая предполагает, что блокировка уже взята:

```
void inc_lock_held() { data++; }  // Приватная
void inc() {
    m.lock(); inc_lock_held(); m.unlock();
}
void double_inc() {
    m.lock();
    inc_lock_held(); inc_lock_held();
    m.unlock();
}
```

В ООП функция `inc_lock_held` обязательно была бы приватной.

Блокировка — ресурс

Есть ли проблема?

```
void inc_special() {
    m.lock();
    if (condition()) return;
    data++;
    m.unlock();
}
```

Блокировка — ресурс

Есть ли проблема?

```
void inc_special() {
    m.lock();
    if (condition()) return;
    data++;
    m.unlock();
}
```

- Есть: блокировка может быть не отпущена в случае досрочного `return`.
- Тогда `m` больше никто никогда не захватит и все следующие вызовы `inc_special` не смогут начаться.
- Вам поможет привычка всегда освобождать взятые ресурсы (файлы, память, *мьютекс*).
- Или специальный синтаксический сахар вроде RAII в C++, или try-with-resources/synchronized-блоков в Java.

Синтаксический сахар

Идиома *RAII*:

```
void inc_special() {
    ScopedLock lock(m);  // Берём блокировку в конструкторе
    if (condition()) return;
    data++;
}  // Освобождаем в деструкторе.
```

synchronized-блоки в Java (каждый получит мьютекс):

```
void inc_special() {
    synchronized {
        if (condition()) return;
        data++;
    }
}
```

Резюме

- Атомарные операции в потоках могут выполняться в любом порядке, если их не синхронизировать.
- Вы обычно не знаете, что атомарно, а что нет.
- *Любой* доступ к общим ресурсам должен быть *защищён* (guarded) блокировкой.
- В частности, любая переменная, доступная из нескольких потоков, должна быть защищена *ровно* одной блокировкой (почему?).
- Блокировки надо брать всегда в одном и том же порядке.
- Если пишете потокобезопасный объект, блокировку надо брать на самой «верхней» операции, которая должна быть атомарной.
- Дубовый способ переделки *потоконебезопасной* структуры в безопасную: создать один мьютекс и брать его на каждую операцию со структурой.
- Мьютексы сильно тормозят, лучше вообще избегать общего доступа к данным (и меньше шанс набагать).

1. Параллельные вычисления
   - Зачем
   - Как
2. Практические грабли
   - Простое приложение на pthread
   - Состояние гонки
   - Гонка данных
   - Взаимное исключение
   - Не пытайтесь повторить это дома
3. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
4. Бонус
5. Домашнее задание

Загадка

Что произойдёт при запуске кода? Предполагаем, что запись и чтение int атомарны.

```
int data;
void* worker(void* arg __attribute__((unused))) {
    for (;;) {
        data++;
    }
}
// ...
    while (data < 100);
    printf("Done\n");
// ...
```

Загадка

Что произойдёт при запуске кода? Предполагаем, что запись и чтение int атомарны.

```
int data;
void* worker(void* arg __attribute__((unused))) {
    for (;;) {
        data++;
    }
}
// ...
    while (data < 100);
    printf("Done\n");
// ...
```

- Race condition отсутствуют.

Загадка

Что произойдёт при запуске кода? Предполагаем, что запись и чтение int атомарны.

```
int data;
void* worker(void* arg __attribute__((unused))) {
    for (;;) {
        data++;
    }
}
// ...
    while (data < 100);
    printf("Done\n");
// ...
```

- Race condition отсутствуют.
- Он зависнет.

Загадка

Что произойдёт при запуске кода? Предполагаем, что запись и чтение int атомарны.

```
int data;
void* worker(void* arg __attribute__((unused))) {
    for (;;) {
        data++;
    }
}
// ...
    while (data < 100);
    printf("Done\n");
// ...
```

- Race condition отсутствуют.
- Он зависнет.
- И никогда не выведет Done.

Разгадка

Как обычно в C/C++.

Подробная разгадка

- Компилятор по умолчанию ничего про потоки не знает.
- Очевидно, что `while (data < 100);` переменную `data` изменить не может.
- Соответственно, переменная `data` никак измениться не может.
- Значит, `data < 100` всегда истинно, можно заменить на `true`.
- Получаем бесконечный цикл.

volatile

Изменим код:

```
volatile int data;  // Обозначили переменную volatile.
void* worker(void* arg __attribute__((unused))) {
    for (;;) {
        data++;
    }
}
// ...
    while (data < 100);
    printf("Done\n");
// ...
```

- volatile говорит компилятору честно сохранять/читать значение этой переменной из памяти каждый раз, когда написано.
- Есть ли проблемы?

volatile

Изменим код:

```
volatile int data;  // Обозначили переменную volatile.
void* worker(void* arg __attribute__((unused))) {
    for (;;) {
        data++;
    }
}
// ...
    while (data < 100);
    printf("Done\n");
// ...
```

- volatile говорит компилятору честно сохранять/читать значение этой переменной из памяти каждый раз, когда написано.
- Есть ли проблемы? Пока нет.

Reordering

Эквивалентны ли два куска кода?

```
int data, finished;
// ...
data = 123;
finished = 1;
```

```
int data, finished;
// ...
finished = 1;
data = 123;
```

Reordering

Эквивалентны ли два куска кода?

```
int data, finished;
// ...
data = 123;
finished = 1;
```

```
int data, finished;
// ...
finished = 1;
data = 123;
```

- Эквивалентны. Оптимизатор тоже так считает.

Reordering

Эквивалентны ли два куска кода?

```
int data, finished;
// ...
data = 123;
finished = 1;
```

```
int data, finished;
// ...
finished = 1;
data = 123;
```

- Эквивалентны. Оптимизатор тоже так считает.
- И может переставить местами: всё равно никто не заметит.

Reordering

Эквивалентны ли два куска кода?

```
int data, finished;
// ...
data = 123;
finished = 1;
```

```
int data, finished;
// ...
finished = 1;
data = 123;
```

- Эквивалентны. Оптимизатор тоже так считает.
- И может переставить местами: всё равно никто не заметит.
- А что, если в другом потоке было так?

```
if (finished) {
    printf("%d\n", data);
}
```

Иллюстрация

Reordeing возвращается

- Даже если одна переменная помечена как `volatile`, компилятор может изменить порядок записи/чтения.
- А вот если обе — не может. Проблема решена?

Reordeing возвращается

- Даже если одна переменная помечена как `volatile`, компилятор может изменить порядок записи/чтения.
- А вот если обе — не может. Проблема решена?
- В процессоре тоже есть оптимизатор.
- Он тоже может переставлять инструкции как захочет, а `volatile` действует только на компилятор.
- Есть специальные ассемблерные инструкции («барьеры памяти»), которые действуют на процессор.
- Не надо сразу пытаться в этом разобраться.
- `volatile` не предназначен для многопоточности, он нужен для других целей (memory-mapped I/O).
- В любом случае, иногда один поток может встретить состояние, которое было бы невозможно получить, исполняя инструкции последовательно.

А что же mutex?

- В разных языках/библиотеках разные модели памяти (потом должны подробно рассказать про Java).
- Обычно везде считается, что в следующих случаях происходит (почти) полная синхронизация памяти между двумя потоками:
  1. A взял мьютекс, который B недавно отпустил (возможно, его брал ещё кто-то).
  2. A создал поток B.
  3. A подождал завершения потока B.
- Все нужные барьеры памяти и прочее уже вшиты внутрь мьютексов и работы с потоками.

Пример

```
// Thread 1
started = true;
m.lock(); data++; m.unlock();
finished1 = true;
finished2 = true;
// Thread 2
m.lock(); m.unlock();
if (finished2) {
    assert(started);     // Верно
    assert(data > 0);    // Верно
    assert(finished1);   // Может быть неверно
}
```

Если убрать из второго потока мьютекс — ничего не знаем.

Резюме

- Если вы что-то не защитили мьютексом, можно огрести из-за reordering, даже если всё «очевидно должно работать».
- Если всё защищено мьютексом и вы ничего не предполагаете о происходящем за пределами критических секций — не о чем беспокоиться.
- Ничего сложнее «взяли один глобальный мьютекс перед операцией, отпустили в конце» обычно не требуется (в том числе в дз).
- Любой сколько-нибудь более сложный контроль требует понимания модели памяти.
- Все проблемы — от общих ресурсов (переменные, файлы, экран). Поэтому стараются минимизировать их количество.
- Нет общих ресурсов — нет проблем.

1. Параллельные вычисления
   - Зачем
   - Как
2. Практические грабли
   - Простое приложение на pthread
   - Состояние гонки
   - Гонка данных
   - Взаимное исключение
   - Не пытайтесь повторить это дома
3. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
4. Бонус
5. Домашнее задание

Зачем

Довольно часто потоки не совсем независимы, а хотят взаимодействовать между собой.
Классическая задача:

- Есть очередь задач.
- Один поток генерирует данные (producer) и добавляет их в очередь.
- Второй поток должен брать добавленные данные по очереди (consumer) и что-то с ними делать.

Например:

- Первый поток ждёт ввода с клавиатуры и кладёт считанные данные в буфер.
- Второй поток выполняет введённые команды (которые могут занять долгое время).
- Мы хотим уметь вводить команды, даже если предыдущая ещё выполняется.

Потокобезопасная очередь

```
class ThreadsafeQueue {
    ThreadsafeQueue() { pthread_mutex_init(&m, NULL); }
    ~ThreadsafeQueue() { pthread_mutex_destroy(&m); }
    void push(int x) {
        pthread_mutex_lock(&m);
        q.push(x);
        pthread_mutex_unlock(&m);
    }
    int pop() { ... }
    bool empty() { ... }
private:
    pthread_mutex_t m;
    queue<int> q;
};
```

Первая попытка

Producer:

```
while (true) {
    int data = get_data();
    q.push(data);
}
```

Consumer:

Первая попытка

Producer:

```
while (true) {
    int data = get_data();
    q.push(data);
}
```

Consumer:

```
while (true) {
    while (q.empty()) {
    }
    process_data(q.pop());
}
```

Проблемы

- Если несколько COnsumer'ов, то есть race condition.
- Consumer активно ждёт событие от первого и тратит процессорное время.
- Даже если ничего не происходит, программа потребляет 100% CPU.
- Consumer постоянно берёт и отпускает mutex, мешая producer'у.

Проблемы

- Если несколько COnsumer'ов, то есть race condition.
- Consumer активно ждёт событие от первого и тратит процессорное время.
- Даже если ничего не происходит, программа потребляет 100% CPU.
- Consumer постоянно берёт и отпускает mutex, мешая producer'у.
- А если добавить задержку в consumer (проверять только каждые X мс), то сильно увеличится задержка в обработке.
- Без новых *примитивов синхронизации* не обойтись.

Новый примитив

Введём примитив `Event` с двумя методами:

- `e.wait()` — усыпляет поток.
- `e.notify()` — будит уснувший поток.

```
// Producer
while (true) {
  int data = get_data();
  q.push(data);
  e.notify();
}
```

```
// Consumer
while (true) {
  if (!q.empty()) {
    process_data(q.pop());
  } else {
    e.wait();
  }
}
```

Есть ли проблемы в коде выше?

Новый примитив

Введём примитив `Event` с двумя методами:

- `e.wait()` — усыпляет поток.
- `e.notify()` — будит уснувший поток.

```
// Producer
while (true) {
  int data = get_data();
  q.push(data);
  e.notify();
}
```

```
// Consumer
while (true) {
  if (!q.empty()) {
    process_data(q.pop());
  } else {
    e.wait();
  }
}
```

Есть ли проблемы в коде выше? Проблемы есть.

Ну вы поняли

Чуть подробнее

1. Consumer проверяет `!q.empty()`
2. Producer добавляет данные.
3. Producer вызывает `e.notify()`, а будить некого.
4. Consumer вызывает `e.wait()` и засыпает навечно.

Что делать?

Первый подход

- Можно сказать, что если в момент вызова `e.notify()` никто не спит, то будет разбужен следующий попытающийся уснуть.
- Другими словами, у `Event` теперь есть состояние: просигналили или нет.
- `e.notify()` — устанавливает флаг «просигналили» и будит все потоки.
- `e.wait()` — ждёт, пока флаг установят (или не ждёт, если уже установлен) и сбрасывает его.
- Решает задачу producer-consumer.
- Используются в Windows API.

Однако:

- Дополнительное состояние вносит сложность — за ним надо следить и добавлять инвариант.
- В pthread не входят и под Linux обычно не используются.

Второй подход: добавим мьютексов?

```
// Producer
while (true) {
  int data = get_data();
  pthread_mutex_lock(&m);
  q.push(data);
  e.notify();
  pthread_mutex_unlock(&m);
}
```

```
// Consumer
while (true) {
  pthread_mutex_lock(&m);
  if (!q.empty()) {
    process_data(q.pop());
  } else {
    e.wait();
  }
  pthread_mutex_unlock(&m);
}
```

Теперь race condition отсутствует.

Второй подход: добавим мьютексов?

```
// Producer
while (true) {
  int data = get_data();
  pthread_mutex_lock(&m);
  q.push(data);
  e.notify();
  pthread_mutex_unlock(&m);
}
```

```
// Consumer
while (true) {
  pthread_mutex_lock(&m);
  if (!q.empty()) {
    process_data(q.pop());
  } else {
    e.wait();
  }
  pthread_mutex_unlock(&m);
}
```

Теперь race condition отсутствует. Зато есть deadlock: producer не может ничего писать, пока consumer спит.

Условные переменные

- Нам нужна атомарная операция «отпусти мьютекс и жди события».
- Такой примитив синхронизации в pthread (и вообще много где) называется *условная переменная* (conditional variable).
- Смысл: условная переменная — это способ оповещать потоки о *возможном* изменении некоторого *условия*, защищённого мьютексом.
- Ожидание пассивное, ресурсы CPU не тратятся.
- На каждое условие создаётся условная переменная.
- Поток, изменивший условие, может разбудить либо все ожидающие потоки (`broadcast`), либо один случайный (`signal`).
- Бывают spurious wakeup — система иногда может разбудить ждущий поток, даже если никто не вызывал `signal`/`broadcast`.
- Поэтому важно проверять условие после пробуждения.

Создание

Точно так же, как и мьютекс:

```
pthread_cond_t cond;
pthread_cond_init(&cond);
// ...
pthread_cond_destroy(&cond);
```

Оповещение

```
pthread_mutex_t m;
pthread_cond_t cond; // GUARDED_BY(m)
bool some_condition; // GUARDED_BY(m)
// ...
pthread_mutex_lock(&m);
// Следующие две строки в любом порядке.
some_condition = true;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&m);
```

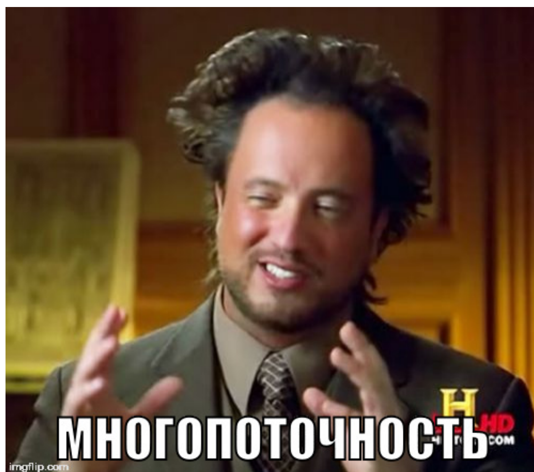
Ожидание условия

```
pthread_mutex_t m;
pthread_cond_t cond; // GUARDED_BY(m)
bool some_condition; // GUARDED_BY(m)
// ...
pthread_mutex_lock(&m);
while (!some_condition) {
    // Атомарно снимает мьютекс и начинает ожидание
    pthread_cond_wait(&cond, &m);
    // После выхода из cond_wait мьютекс снова захвачен.
}
pthread_mutex_unlock(&m);
```

Упражнение

1. Возьмите реализацию с producer-consumer с GitHub.
2. Запустите и убедитесь, что на каждую введённую строчу отзывается второй поток: сначала сразу, а потом через две секунды.
3. Убедитесь, что если во время ожидания второго потока ввести новую строчку, то на неё второй поток тоже среагирует.
4. Убедитесь, что если во время ожидания ввести две новых строчки, то будет обработана только последняя.
5. Задайте все вопросы по коду; поймите, зачем нужна и что делает каждая строчка.
6. Есть ли проблемы в этом коде?

Конечно, есть!



Проблема

Вот тут возникает race condition:

```
while (true) {
    fgets(str, sizeof str, stdin);
    pthread_mutex_lock(&m);
    str_available = true;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&m);
}
```

Проблема

Вот тут возникает race condition:

```
while (true) {
    fgets(str, sizeof str, stdin);
    pthread_mutex_lock(&m);
    str_available = true;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&m);
}
```

- fgets меняет буфер, который также читается из другого потока.
- Значит, буфер должен быть защищён мьютексом на всех стадиях.
- Если поменяем fgets и pthraed_mutex_lock местами, то будет deadlock: consumer не может читать данные, пока producer ждёт.

Проблема

Вот тут возникает race condition:

```
while (true) {
    fgets(str, sizeof str, stdin);
    pthread_mutex_lock(&m);
    str_available = true;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&m);
}
```

- `fgets` меняет буфер, который также читается из другого потока.
- Значит, буфер должен быть защищён мьютексом на всех стадиях.
- Если поменяем `fgets` и `pthraed_mutex_lock` местами, то будет deadlock: consumer не может читать данные, пока producer ждёт.
- Правильно сначала считать в локальную переменную, а потом скопировать в буфер. Код.

Резюме

- Условные переменные нужны там и только там, где поток ждёт некоторое условие.
- А это условие всегда защищено ровно одним мьютексом (почему?)
- Соответственно, условная переменная тоже защищена ровно одним мьютексом.
- Условие всегда надо проверять в цикле.
- `pthread_cond_wait` — это лишь оптимизация. Если её убрать, программа должна остаться корректной.
- Никакого внутреннего состояния у условной переменной нет, из-за этого она просто реализуется в ОС, но программисту надо самому явно формулировать условие, которого ждёт поток.

Необязательное упражнение на дом

Реализуйте Windows Event через conditional variable:

- Объект «событие» с методами `wait` и `notify`.
- В каждый момент не более одного потока ждёт (`wait`).
- Метод `notify` либо пробуждает ждущий поток, либо делает так, что следующий метод `wait` мгновенно завершится.

Можно создать отдельную папку в репозитории и прислать код на проверку (в теме — `[add-161019]`).
Сигнатура:

- `typedef ... event_t;`
- `event_init(event_t*)`
- `event_wait(event_t*)`
- `event_notify(event_t*)`
- `event_destroy(event_t*)`

1. Параллельные вычисления
   - Зачем
   - Как
2. Практические грабли
   - Простое приложение на pthread
   - Состояние гонки
   - Гонка данных
   - Взаимное исключение
   - Не пытайтесь повторить это дома
3. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
4. Бонус
5. Домашнее задание

Стандартные паттерны-1

- **Пул потоков** (thread pool):
  - Создание потоков на короткие задачи — это очень неэффективно.
  - Поддерживается пул из некоторого числа потоков (примерно по числу ядер).
  - Задачу можно отправить в пул, она выполнится в одном из потоков, когда тот освободится.
  - Ограничивает число одновременно выполняющихся задач.
- **Блокировка чтения-записи** (readers-writer lock)
  - Как мьютекс, но позволяет потоку указать режим доступа: «чтение» или «запись».
  - Читателей может быть сколько угодно.
  - Если кто-то пишет, то другие потоки ждут.
  - Ускоряет доступ к редко меняющимся данным.

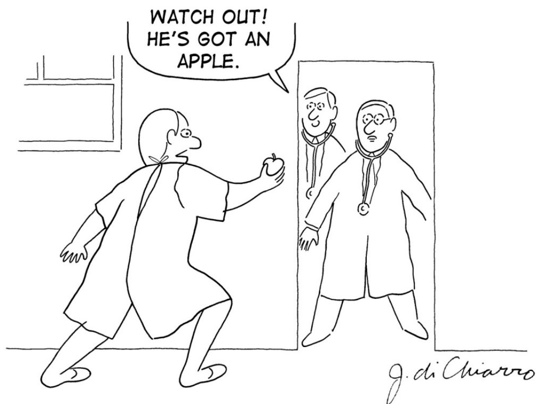
Стандартные паттерны-2

- **Акторы** (actors)
  - Небольшие потоки, не имеющие общих ресурсов.
  - Для обмена информацией посылают друг другу неизменяемые *сообщения*.
  - Вся синхронизация сконцентрирована в системе обмена сообщениями.
- **Неблокирующие** (lock-free) структуры данных
  - Работают поверх атомарных операций и моделей памяти.
  - Не требуют блокировок или мьютексов.
  - Работают быстрее, так как многие операции атомарны в железе и не требуют вмешательства ОС.

Как отлаживать

"Keep away Doctor."

Как отлаживать



An apple a day keeps the doctor away.
Лучше предотвращать, чем отлаживать.

Причина

- Многопоточные баги обычно тесно связаны с порядком выполнения операций.
- Операции выполняются в разном порядке каждый запуск, под отладчиком, в разном коде.
- Очень сложно ловить баг «за руку».
- Корректная работа на куче тестов не означает отсутствие багов.

Как предотвращать

- Явно расставляйте инварианты в комментариях: что чем защищено, в каком порядке захватывать мьютексы.
- Нарисуйте на бумажке все возможные состояния системы и проверьте, что инварианты выполняются.
- Минимизируйте количество мьютексов, если нет проблем со скоростью работы.
- Не используйте для синхронизации ничего, кроме мьютексов (в частности, явных `sleep` в программе быть не должно).

Как тестировать

- Запускайте на больших тестах, в которых потоки работают медленно и часто происходит переключение.
- Если вы под 64-битным Linux — используйте thread sanitizer (добавьте ключи `-g -fsanitize=thread -O2 -fPIC -pie`). Он хорош в нахождении некоторых гонок данных, *происходящих во время выполнения*.
- Для аналогичных целей можно использовать Valgrind.
- На Windows можно поставить виртуальную машину.

Пример вывода Thread Sanitizer-1

На примере с двумя счётчиками находит гонку сразу, во время первых выводов на экран:

```
==================
WARNING: ThreadSanitizer: data race (pid=13170)
  Read of size 4 at 0x7f31e00e12cc by thread T2:
    #0 worker .../08-two-threads.c:10 (...)
    #1 <null> <null>:0 (libtsan.so.0+0x000000032d69)

  Previous write of size 4 at 0x7f31e00e12cc by thread T1:
    #0 worker .../08-two-threads.c:12 (...)
    #1 <null> <null>:0 (libtsan.so.0+0x000000032d69)

  Location is global 'data' of size 4 at ... (...)
```

Пример вывода Thread Sanitizer-2

Также указывает, где были созданы соответствующие потоки:

```
Thread T2 (tid=13173, running) created by main thread at:
  #0 pthread_create <null>:0 (libtsan.so.0+0x000000047f23)
  #1 main .../08-two-threads.c:20 (...)

Thread T1 (tid=13172, finished) created by main thread at:
  #0 pthread_create <null>:0 (libtsan.so.0+0x000000047f23)
  #1 main .../08-two-threads.c:19 (...)

SUMMARY: ThreadSanitizer: data race .../08-two-threads.c:10
```

1. Параллельные вычисления
   - Зачем
   - Как
2. Практические грабли
   - Простое приложение на pthread
   - Состояние гонки
   - Гонка данных
   - Взаимное исключение
   - Не пытайтесь повторить это дома
3. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
4. Бонус
5. Домашнее задание

Общая идея

- Вам надо реализовать Thread Pool (почти как в Java).
- Это нечто, что хранит несколько потоков, готовых выполнять любые задачи, которые отправляют в thread pool.
- Число потоков фиксируется при создании.
- В пул можно отправлять задачи (функция + аргумент), они должны выполняться.
- Задачи могут быть отправлены в любой момент (и когда есть свободный поток, и когда нет).
- Задачи тоже могут отправлять задачи в поток (это не должно ни на что влиять).
- Можно подождать завершения задачи (т.е. пока она начнёт и закончит выполняться).
- После всего надо распараллелить quick sort при помощи thread pool.

Как всё хранится

- Структуру `ThreadPool` вы целиком реализуете сами как хотите.
- В структуре `Task` обязательно должно лежать описание задачи (функция + её аргумент).
- Наверняка вам захочется добавить в `Task` что-то ещё, чтобы можно было ждать её завершения.
- Память под структуры `ThreadPool` и `Task` выделяет тот, кто пользуется ThreadPool.

Пример использования

```
void foo(void* arg_) {
    printf("got %d\n", arg_); free(arg_);
}
int main(void) {
    ThreadPool pool;
    thpool_init(&pool, 2);  // Создаём пул на два потока.
    Task tasks[100];
    for (int i = 0; i < 100; i++) {
        tasks[i].f = foo;
        int* arg = malloc(sizeof(int));
        *arg = i; tasks[i].arg = arg;
        thpool_submit(&pool, &tasks[i]);
    }
    thpool_finit(&pool);  // Ожидает все задачи.
}
```

Самые важные замечания

- Не должно быть race condition и dead locks в любом виде.
- Не должно быть утечек памяти.
- Нельзя активно ждать событий в цикле, тратя процессорное время.
- Thread Pool должен быть независим от реализации quick sort.
- При увеличении числа потоков в thread pool сортировка должна становиться быстрее.
- Выбирать средний элемент в quick sort можно как угодно.
- Неасимптотические оптимизации quick sort не нужны.
- Есть ещё куча замечаний в самом задании.