# Многопоточность-2

Егор Суворов

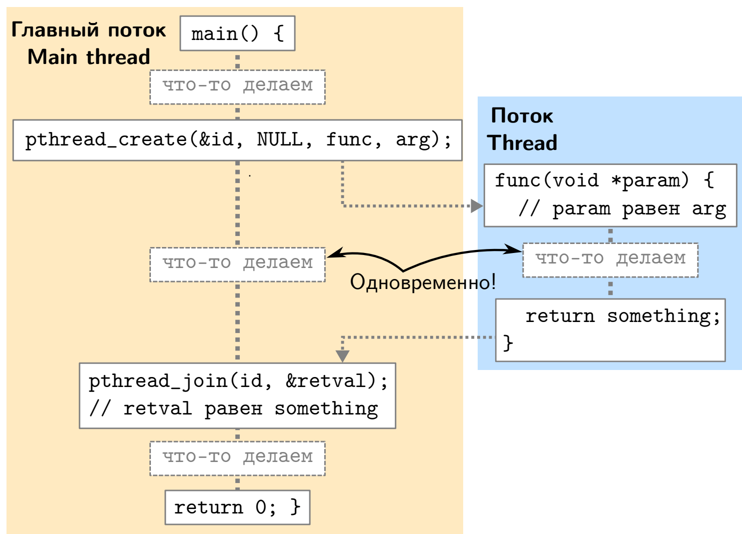
Курс «Парадигмы и языки программирования», подгруппа 3

Понедельник, 13 ноября 2017 года

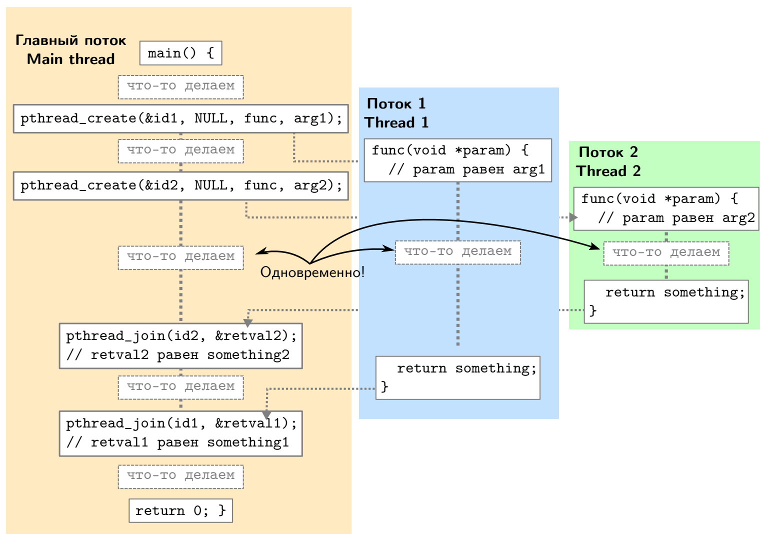
# План занятия

1. Напоминание
   - Потоки, гонки, мьютексы
   - Не пытайтесь повторить это дома
2. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
3. Домашнее задание
4. Бонус

# Жизненный цикл потоков



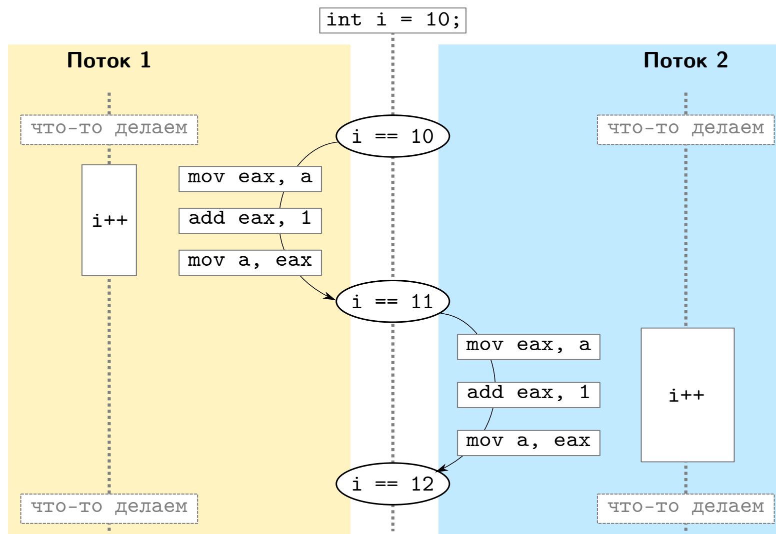
# Потоков бывает много



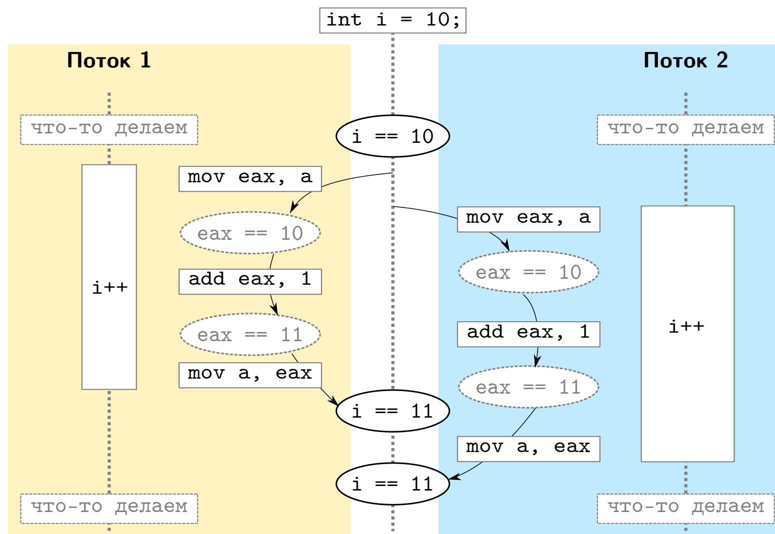
# Напоминание про потоки

- Потоки выполняют код независимо и параллельно
- Так удобно писать код, который работает над несколькими вещами одновременно
- Если несколько ядер процессора — ещё и получается быстрее
- Внутри одного *процесса* работает несколько *потоков*
- У всех потоков внутри процесса общая память
- Но лучше к общей памяти не обращаться (см. далее — *гонки*)

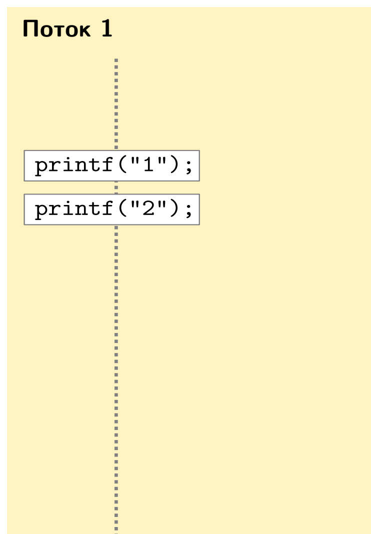
# Гонка данных (повезло)

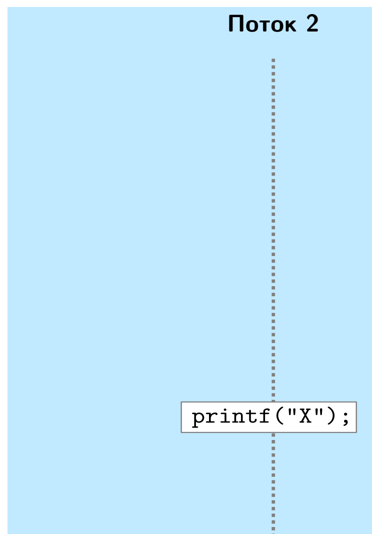


# Гонка данных (не повезло)

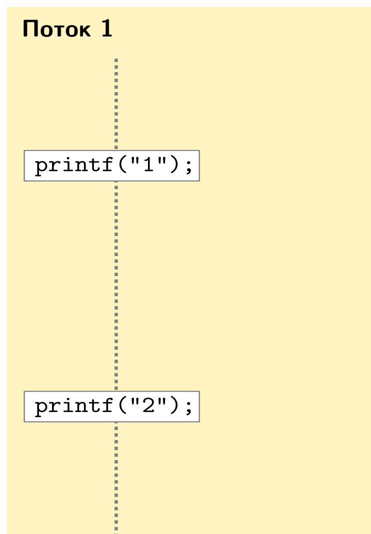


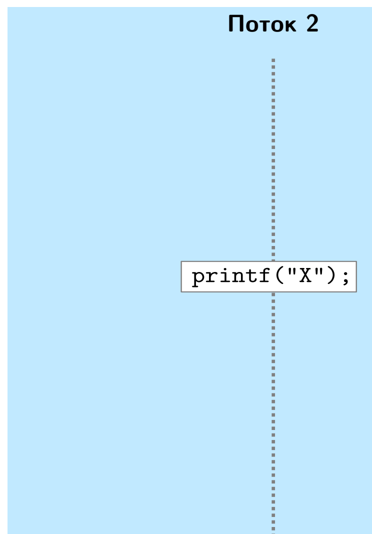
# Гонка ресурсов (повезло)



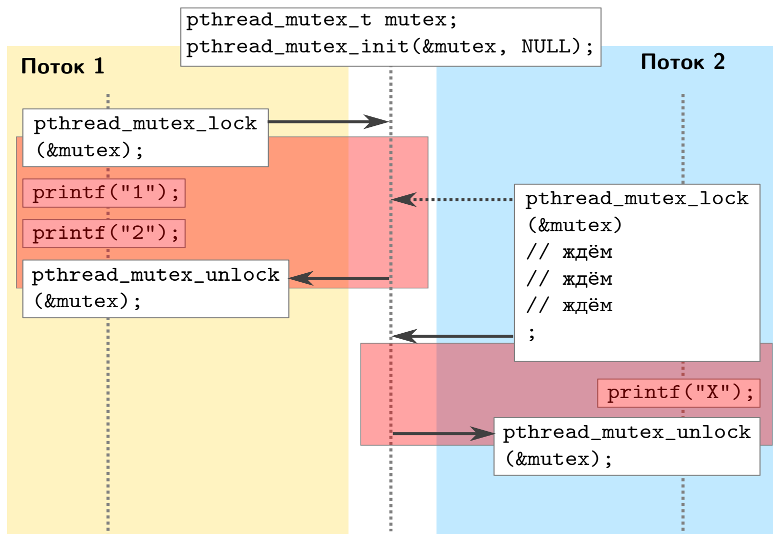



# Гонка ресурсов (не повезло)



# Гонка ресурсов (как правильно)



# Напоминание про гонки

- Пока не знаем ничего атомарного, кроме *захвата* или *освобождения* мьютексов
- Если хотим сделать операцию атомарной — *защищаем* мьютексом
- Операции между разными потоками могут как угодно перемешиваться
- Между двумя атомарными операциями могут вклиниться другие, если не защитить
- Захват и освобождение — медленные операции

1. Напоминание
   - Потоки, гонки, мьютексы
   - Не пытайтесь повторить это дома
2. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
3. Домашнее задание
4. Бонус

# Загадка

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

```
int data;
void* worker(void*) {
    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*) {
    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*) {
    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*) {
    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

Измените код:

```
volatile int data;  // Обозначили переменную volatile.
void* worker(void*) {
    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

Измените код:

```
volatile int data;  // Обозначили переменную volatile.
void* worker(void*) {
    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. Напоминание
   - Потоки, гонки, мьютексы
   - Не пытайтесь повторить это дома
2. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
3. Домашнее задание
4. Бонус

# Зачем

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

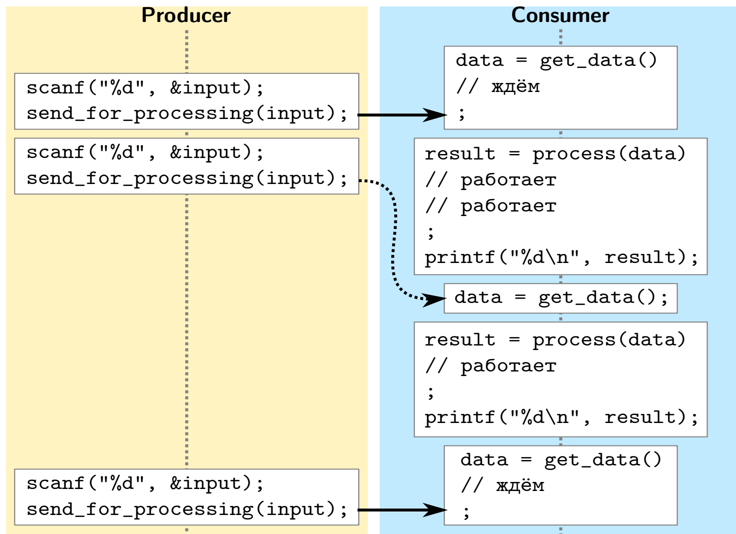
Классическая задача:

- 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.

# Producer-Consumer



# 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

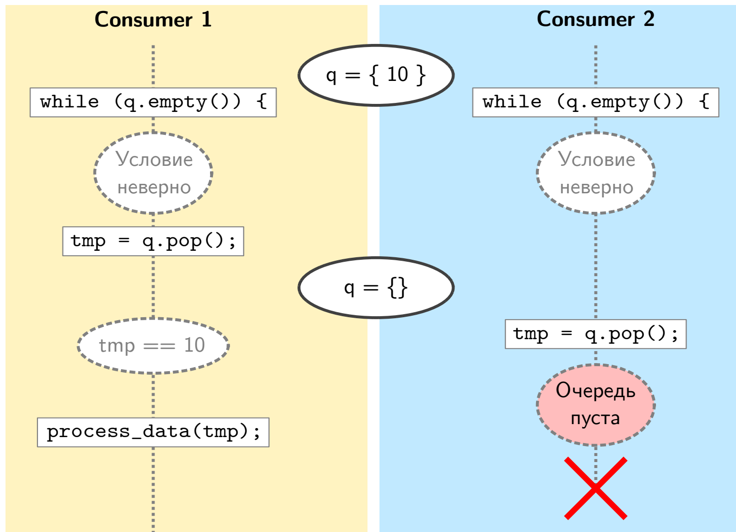
Producer:

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

Consumer:

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

# Есть гонка



# 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:

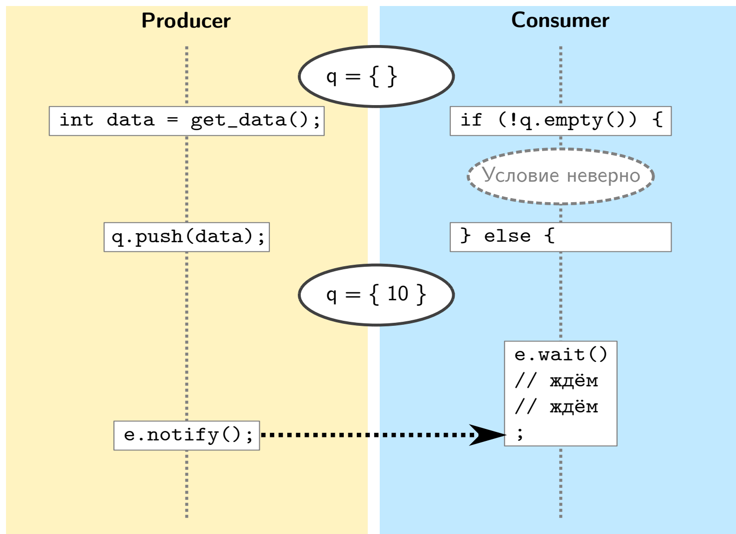
- `e.wait()` — усыпляет поток.
- `e.notify()` — будит уснувший поток.

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

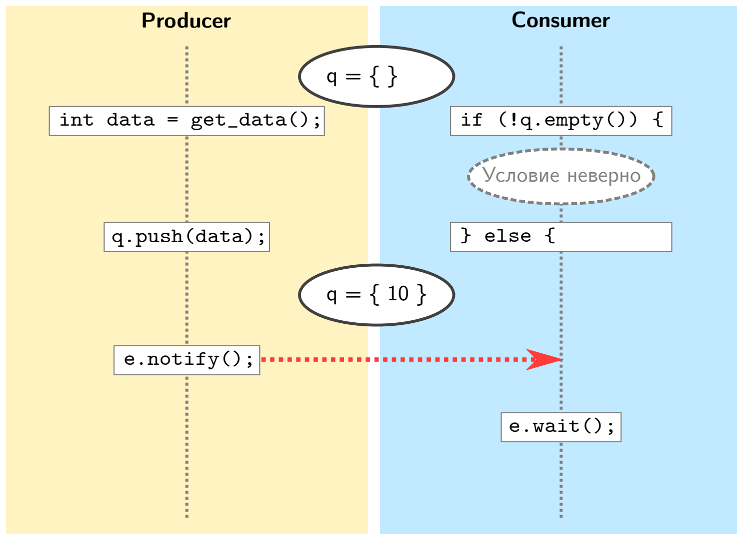
```
// Consumer
while (true) {
  if (!q.empty()) {
    process_data(q.pop());
  } else {
    e.wait();
  }
}
```

Есть ли проблемы в коде выше? Проблемы есть.

# Повезло



# Не повезло



# Ну вы поняли

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);
}
```

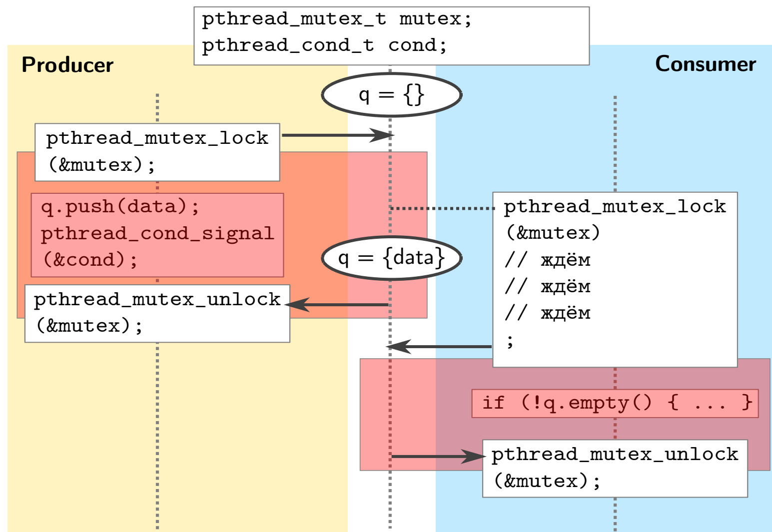
```
// Consumer
while (true) {
  pthread_mutex_lock(&m);
  if (!q.empty()) {
    process_data(q.pop());
  } else {
    e.wait();
  }
  pthread_mutex_unlock(&m);
}
```

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

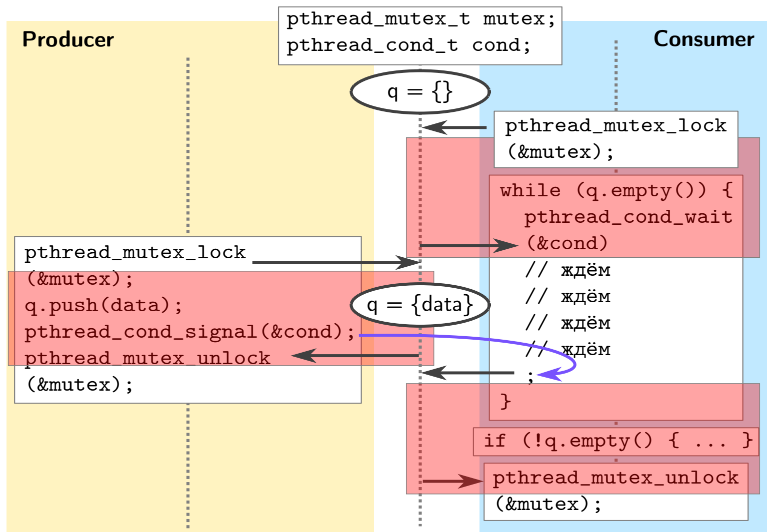
# Условные переменные

- 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.

# Событие наступило до ожидания



# Событие наступило после ожидания



# Свойства условных переменных

- 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 (`while`, а не `if`).

# 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);
```

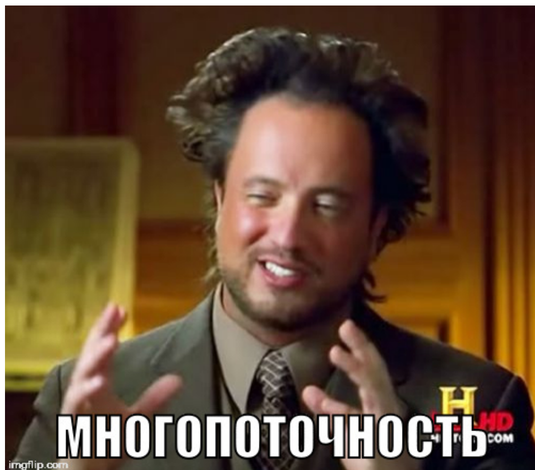
# Ожидание условия

```
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 pthread_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 pthread_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

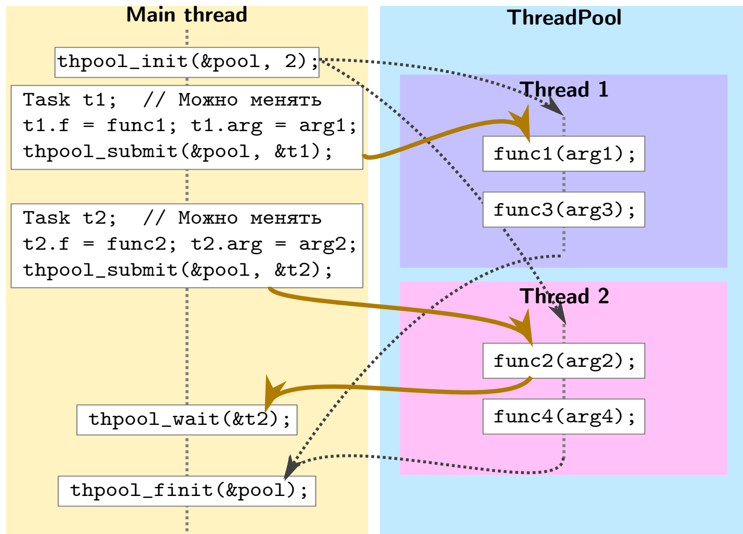
- Условные переменные нужны там и только там, где поток ждёт некоторое условие.
- А это условие всегда защищено ровно одним мьютексом (почему?)
- Соответственно, условная переменная тоже защищена ровно одним мьютексом.
- Условие всегда надо проверять в цикле.
- `pthread_cond_wait` — это лишь оптимизация. 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.

1. Напоминание
   - Потоки, гонки, мьютексы
   - Не пытайтесь повторить это дома
2. Обмен сообщениями
   - Простая реализация
   - События
   - Условные переменные
3. Домашнее задание
4. Бонус

# 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.

# Иллюстрация



# Тонкости

- 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.
- Рекомендую перед реализацией quick sort очень аккуратно прописать кто и чем владеет.
- Возможно, придётся немного изменить интерфейс ThreadPool — задавайте вопросы!

# Пример использования

```
void foo(void* arg_) {
    printf("got %d\n", arg_); free(arg_);
}
int main() {
    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 (задавайте вопросы!).
- Выбирать средний элемент в quick sort можно как угодно.
- Неасимптотические оптимизации quick sort не нужны.
- Есть ещё куча замечаний в самом задании.

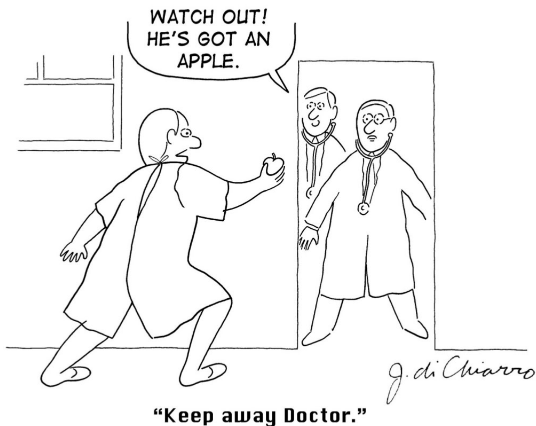
1 Напоминание
- Потоки, гонки, мьютексы
- Не пытайтесь повторить это дома

2 Обмен сообщениями
- Простая реализация
- События
- Условные переменные

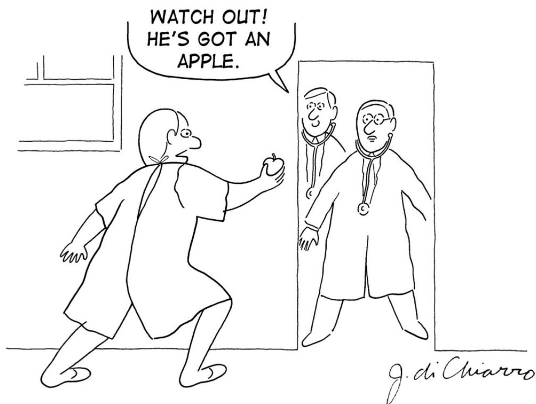
3 Домашнее задание

4 Бонус

# Как отлаживать



# 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. Он хорош в нахождении некоторых гонок данных, *происходящих во время выполнения*.
- Аналогично можно использовать Valgrind.
- На Windows можно поставить виртуальную машину с Linux и запускать там.
- Задавайте вопросы!