# Многопоточность-1

Егор Суворов

Курс «Парадигмы и языки программирования», подгруппа 3

Понедельник, 23 октября 2017 года

# План занятия

1. Параллельные вычисления
   - Зачем
   - Как
2. Практические грабли
   - Простое приложение на pthread
   - Состояние гонки
   - Гонка данных
   - Взаимное исключение
3. Владение ресурсами

1. Параллельные вычисления
   - Зачем
   - Как
2. Практические грабли
   - Простое приложение на pthread
   - Состояние гонки
   - Гонка данных
   - Взаимное исключение
3. Владение ресурсами

# 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 бóльшего 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:

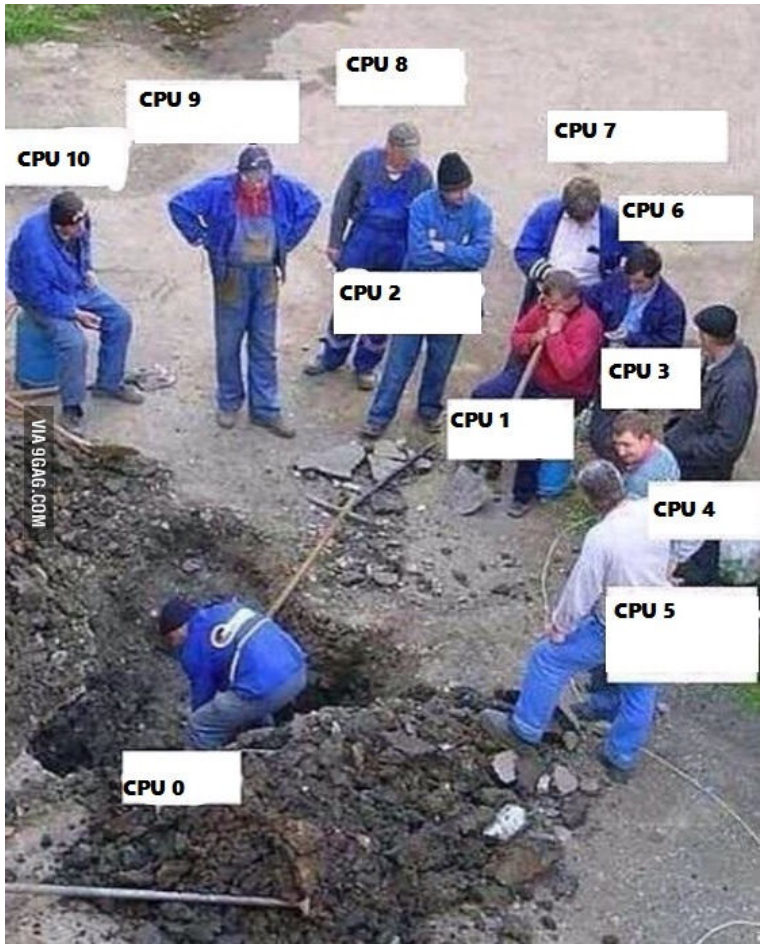
```
char buf1[100], buf2[100];
fread(file_on_disk1, 1, sizeof buf1, buf1);
fread(file_on_disk2, 1, sizeof buf2, buf2);
```

- Некоторые не параллелятся:

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

- Надо писать специальные алгоритмы для распределённых вычислений.
- Иногда перебор лучше умного решения.

# 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 языках 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

3 Владение ресурсами

# Что такое pthread

- Стандартный интерфейс 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() {
    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;
}
```

# Упражнение: сборка кода

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

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

# 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.

# Несколько замечаний про C/C++

- На языках C и 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, то в языке C++ следует не писать его имя, например, `void* work(void*) {`.
- Из функции всегда надо что-то вернуть (хотя бы `NULL`).
- Никогда не начинайте название переменной с нижнего подчёркивания!
- `(void*)123` — *плохая* идея для передачи числа.

# 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`? Предполагаем, что процесс при этом не умрёт.

# 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`? Предполагаем, что процесс при этом не умрёт. 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
3. Владение ресурсами

# Упражнение

1. Возьмите код.
2. При желании можете скачать Makefile.
3. Убедитесь, что на экран выводится строка.
4. Запустите второй поток, который выводит на экран другую строку.
5. Найдите место, где первая строка сменяется второй.
6. 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

1. Добавьте снятие снимков в свой счётчик.
2. Убедитесь, что все значения теперь чётные.
3. Запустите второй поток-счётчик, который тоже увеличивает `data`.
4. 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
3. Владение ресурсами

# 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*) {
    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*) {
    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.

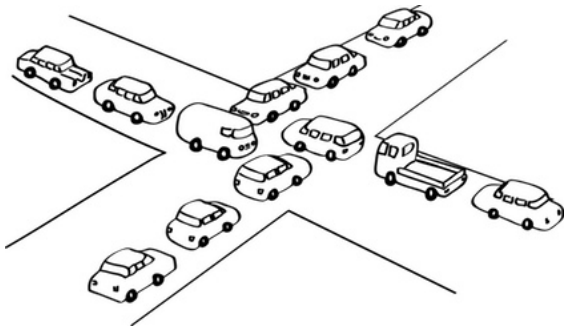
Исправленный двойной счётчик, исправленный writeln.

# 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
3. Владение ресурсами

# Что такое «ресурс»?

- Программы управляют большим количеством *ресурсов*:
  - Динамически выделяемая память
  - Открытые файлы
  - Сетевые соединения
  - Общение с какой-нибудь простой железкой (сканер штрих-кодов)
- Обычно у *ресурсов* есть следующие свойства:
  - К нему можно обращаться по некоторому адресу, указателю, номеру, handle.
  - Этот handle можно получить из функции *открытии* ресурса.
  - Если открываем ресурс несколько раз — получаем либо ошибку, либо каждый раз новый handle, либо вообще разные ресурсы.
  - Handle обязательно надо *закрыть*, когда ресурс больше не нужен, причём ровно один раз.
  - Один handle предназначен для выполнения *последовательных операций*. Обычно он хранит внутри себя какую-то информацию про предыдущие операции (например, позицию в файле).

# Проблемы с ресурсами

- Все следующие ситуации являются проблемными:
  - Handle не закрыли (утечка памяти/ресурсов).
  - Handle закрыли дважды (неопределённое поведение).
  - Ресурс используют для двух независимых цепочек операций (они смешиваются, получается чушь).
  - Ресурс используют после закрытия handle.
- Примеры того, за чем надо следить:
  - Выделенная при помощи `malloc`/`new`/`new[]` память освобождается ровно один раз при помощи `free`/`delete`/`delete[]`.
  - Память освобождается тем же методом, что и создавалась.
  - Открытый файл закрывается ровно один раз.
  - Любой созданный поток либо переводится в detached-режим, либо на нём ровно один раз вызывают `pthread_join`.
  - Два потока не пытаются читать из одного файла.
  - Два потока не пытаются одновременно читать и писать в одну переменную.
- Это всё дополнительные инварианты.

# Концепция «владения»

- Инвариант:
  - У любого ресурса в любой момент времени в любой точке программы есть ровно один «владелец» (поток, функция, кусок кода — разные уровни детализации).
  - Только владелец может закрыть ресурс и он обязан это сделать ровно так, как требует ресурс.
  - Только владелец имеет доступ к ресурсу (но иногда эта часть ослабляется).
- Как ведёт себя владение:
  - При создании объекта его владельцем становится создатель.
  - Текущий владелец может явно передать владение другому месту программы.
  - Объект можно скопировать, тогда получится две копии, у каждой — свой владелец.
  - Объект можно временно «одолжить» кому-нибудь: он сможет с ним работать и передавать дальше, но не сможет закрыть.

# Пример

Немного упрощая:

```
int main() {
  string x = "xxx"; // 1. main() владеет x.
  string res = foo(x);
  // 6. main() получил res во владение от foo().
  // 7. main() удаляет x и res, как владелец.
}
string foo(string &s) {
  // 3. foo() "одолжило" s по ссылке.
  string res = s + "foo";  // 4. foo() создал res.
  return res;  // 5. foo() передал владение res вызвавшему.
}
```

# Плохой пример

```
int main() {
  char *s = malloc(2);  // 1. main() владеет s.
  s[0] = 'x'; s[1] = 0;
  s = foo(s);
  // 5. Результат foo() лежит в s, во владении у main()
  free(s);  // 6. Освободили результат foo().
  // 7. А старое s уже потеряно - утечка. Мы его передали
  //    foo(), а foo() не в курсе.
}
char* foo(const char *s) {
  // 2. foo лишь "одалживает" s и только из него читает.
  char* res = malloc(strlen(s) + 1);  // 3. Владеем res.
  strcpy(res, s);
  return res;  // 4. Передаём res во владение вызвавшему.
}
```

# Последствия

- С такими инвариантами упрощаются проблемы с освобождением, race condition и остальным.
- В некоторых языках есть разного вида конструкции, которые форсируют этот инвариант:
  - RAII в C++: если объект выделен на стеке, то им владеет текущая функция. Отсюда растут «умные указатели».
  - Обычно в C++ никогда напрямую не используются операции «открыть ресурс» или «закрыть ресурс» — всё через RAII (`ifstream` вместо `FILE*`, скажем).
  - Язык Rust: в синтаксисе надо явно указывать, кто чем владеет и как передаёт.
- Концепция имеет смысл даже в языках со сборкой мусора (Python, Java, JavaScript): там мы всё равно должны закрывать файлы и всё ещё есть race conditions.

# Примеры ресурсов

- Динамическая память
- Мьютексы pthread (надо создавать и уничтожать)
- Потоки (надо либо join, либо передавать владение ОС при помощи detach)
- `FILE*` (закрываем руками), `ifstream` (компилятор C++ всегда закрывает за нас)
- Стандартные потоки ввода и вывода
- Переменные (впрочем, с локальными всё просто)

# Практический вывод

- У всех параметров и возвращаемых значений всегда указывайте, кто и когда чем владеет. Особенно в многопоточных приложениях.
- Владение — это свойство *вызываемой* функции.
- Как закрывать ресурс — это свойство *ресурса* (например, разные указатели может быть надо закрыть по-разному).
- Параметр можно либо временно одолжить, либо передать во владение.
- Возвращаемое значение либо передаётся вызываемой функции, либо ей одалживается на время.
- Если нам что-то одолжили, мы не имеем права это удалять или куда-то сохранять (потому что нам одолжили лишь временно).