Не моргай!

Соловьёв Владимир Валерьевич
Huawei, НГУ, СУНЦ
vladimir.conwor@gmail.com
vk.com/conwor

Откуда вы знаете, что, когда вы отворачиваетесь, столы за вашими спинами не превращаются в кенгуру?

Бертран Рассел

Эта презентация отображается какой-то программой

Эта презентация отображается какой-то программой

И сейчас она ждёт, когда я нажму кнопку переключения на следующий слайд

Эта презентация отображается какой-то программой

И сейчас она ждёт, когда я нажму кнопку переключения на следующий слайд

Что-то типа scanf или cin >>

Эта презентация отображается какой-то программой

И сейчас она ждёт, когда я нажму кнопку переключения на следующий слайд

Что-то типа scanf или cin >>

```
int action = waitButtonClick();
```

```
int action = waitButtonClick();  // user input

if (action == 42) {              // computing
  ...
}
```

Это действие выполняют периферийные устройства

```
int action = waitButtonClick();  // user input

if (action == 42) {              // computing
  ...
}
```

Это действие выполняют периферийные устройства

А процессор стоит на месте, пока я не нажму на кнопку

```
int action = waitButtonClick();  // user input

if (action == 42) {              // computing
  ...
}
```

Это действие выполняют периферийные устройства

А процессор стоит на месте, пока я не нажму на кнопку

Так давайте нагрузим его другими задачами!

Многозадачность

Одновременное выполнение компьютером нескольких задач

Многозадачность

кусков кода

Одновременное выполнение компьютером нескольких ~~задач~~

Многозадачность

кусков кода

Одновременное выполнение компьютером нескольких ~~задач~~

Два способа:

1) Несколько процессоров (**ядер**) в одном компьютере (многопроцессорность)

2) Ядро регулярно переключается между кусками кода (переключение контекста)

Переключение контекста

Переход ядра от одной задачи к другой

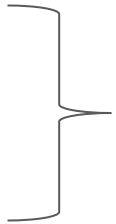
Состояние текущей задачи (регистры, указатель на стек, указатель на текущую команду, ...) сохраняется

Состояние следующей задачи загружается

Причины переключения

1) Код ожидает ввода данных

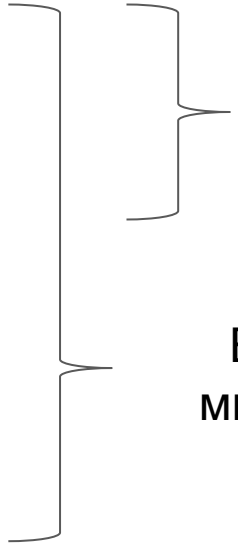
2) Код добровольно отдаёт ядро (уходит поспать)



Кооперативная многозадачность

Причины переключения

1) Код ожидает ввода данных
2) Код добровольно отдаёт ядро (уходит поспать)
3) ОС принимает решение забрать ядро у кода



Кооперативная многозадачность

Вытесняющая многозадачность

Кооперативная многозадачность

Требует слаженной работы всех программ ⇒ неприменима в системах общего назначения (персональный компьютер)

Вытесняющая многозадачность

ОС определяет **квант времени** (несколько миллисекунд)

По истечение этого кванта ОС принудительно останавливает код, забирает ядро и меняет ему контекст на другой код

Применяется во всех современных системах

Кто делает всю работу?

ОС в связке с процессором

Переключение контекстов, определение приоритетов, распределение периферийных устройств, ...

В простых случаях для программиста многозадачность незаметна

После пробуждения код продолжает работать, как раньше

Окружающий программу мир

Многослойный

- локальные переменные
- глобальные переменные
- динамическая память
- файлы
- прочие объекты ОС (сокеты, ...)

Окружающий программу мир

Многослойный

- локальные переменные
- глобальные переменные
- динамическая память
- файлы
- прочие объекты ОС (сокеты, ...)

Личное пространство программы

Окружающий программу мир

Многослойный

- локальные переменные
- глобальные переменные
- динамическая память

Личное пространство программы

- файлы
- прочие объекты ОС (сокеты, ...)

Пространство, **разделяемое** между программами

Проклятье многозадачности

В пространстве, которое одна программа разделяет с другими, может происходить что угодно

Проклятье многозадачности

В пространстве, которое одна программа разделяет с другими, может происходить что угодно

```
saveSomethingInFile(“foo.txt”);
...
...
...
loadSomethingFromFile(“foo.txt”);
```

Проклятье многозадачности

В пространстве, которое одна программа разделяет с другими, может происходить что угодно

Другая программа пришла и переписала содержимое файла foo.txt

```
saveSomethingInFile(“foo.txt”);
...
...
...
loadSomethingFromFile(“foo.txt”);
```

Стоит моргнуть

Стоит моргнуть

Проклятье многозадачности

И даже моргать необязательно - в многопроцессорной системе это может произойти без остановки вашего кода

Проклятье и дар многозадачности

Проклятье и дар многозадачности

Раз в разделяемом пространстве может происходить что угодно

Значит там может происходить и контролируемое **взаимодействие!**

Пример

Одна программа (**worker**) что-то непрерывно считает и пишет промежуточные результаты в файл

Вторая программа (**printer**) регулярно выводит содержимое файла пользователю

Зачем так?

Для эффективности

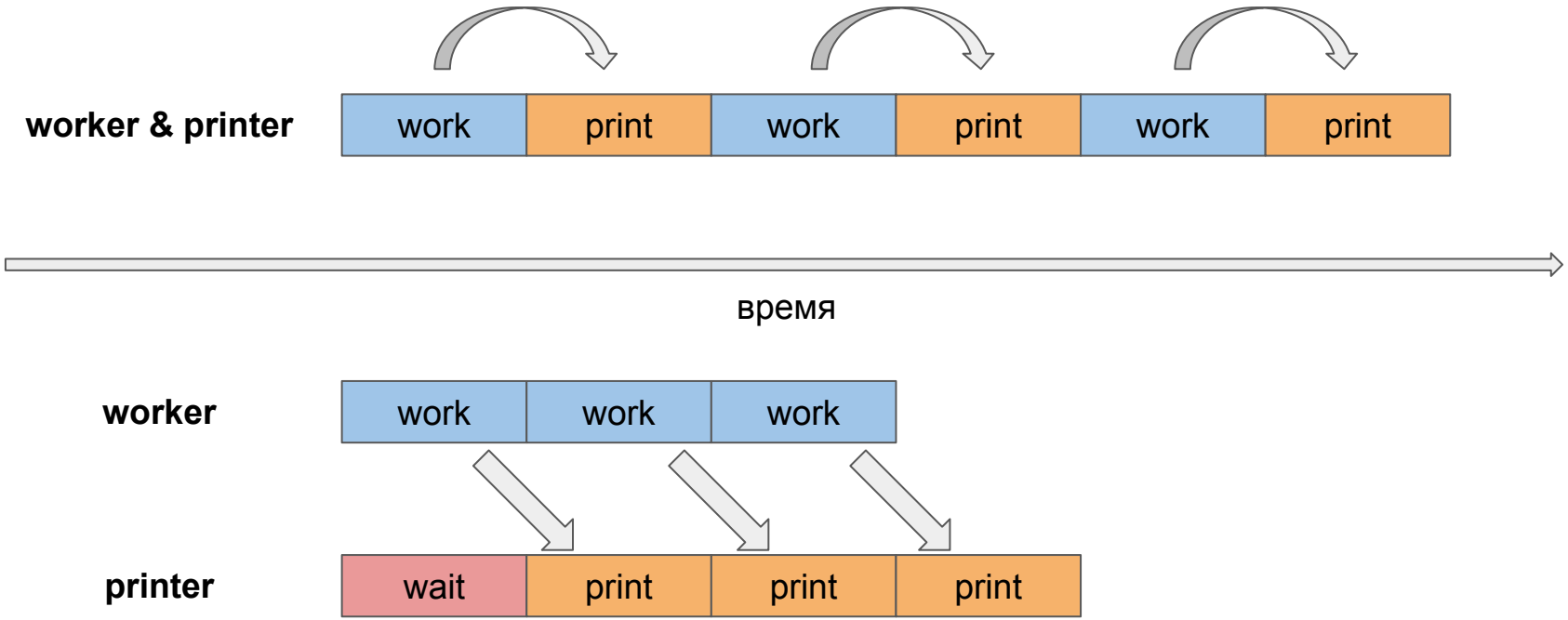
Программа worker не тратит время на вывод данных

Для простоты реализации каждой из них

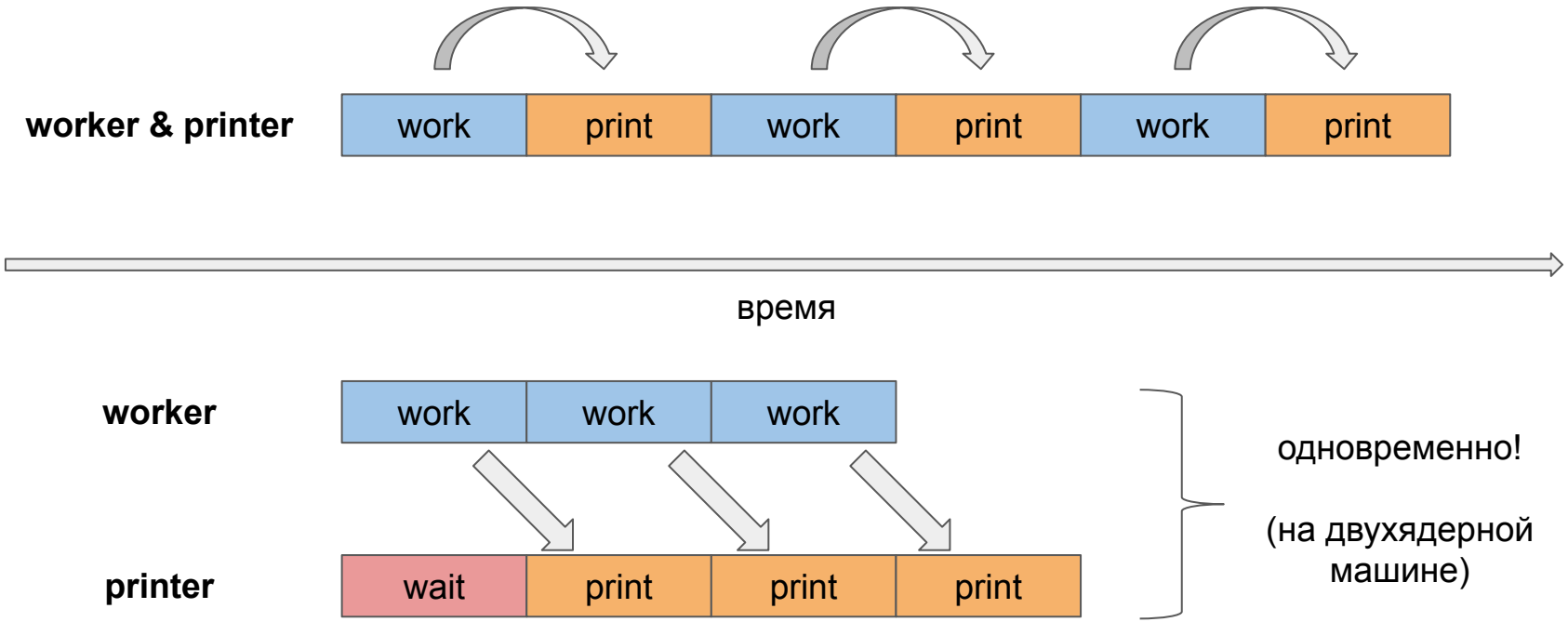
Printer может быть разным (консольный, графический, сетевой, ...)

Переделка printer не требует переделки worker

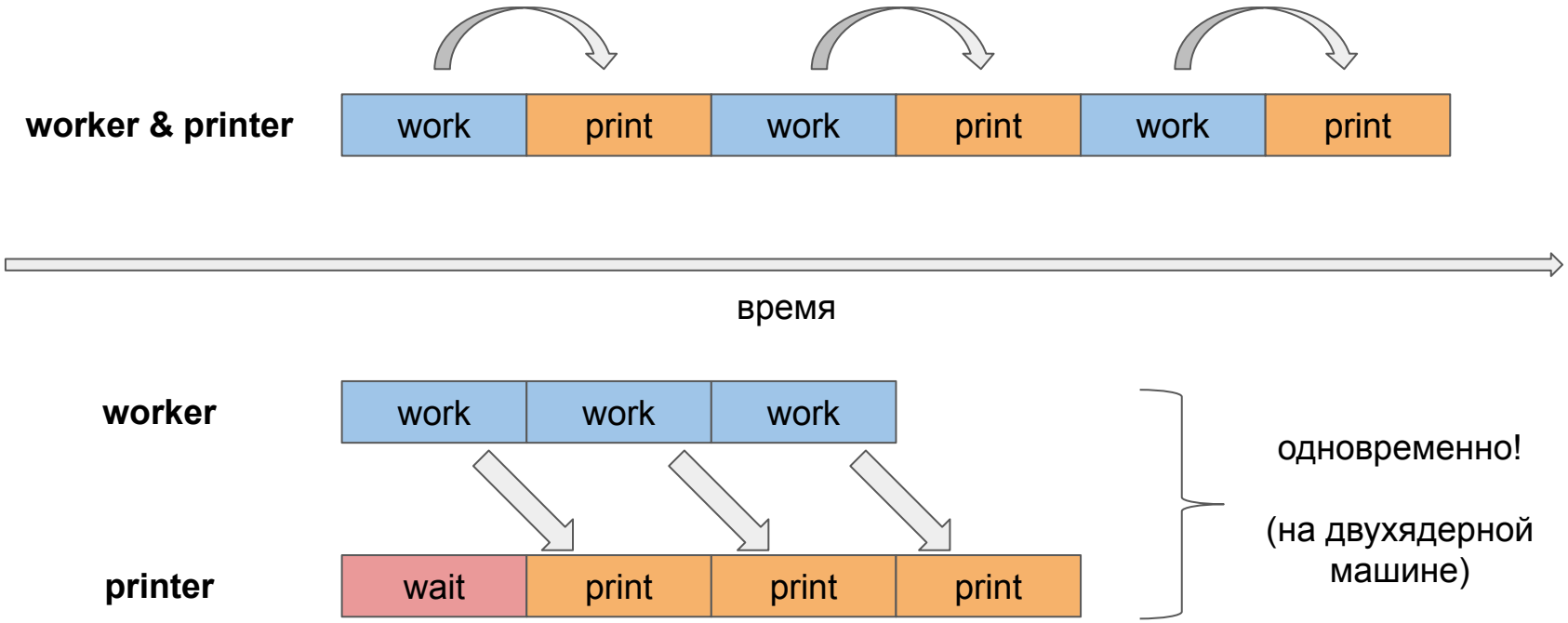
Пример


worker & printer
work
print
work
print
work
print
время
worker
work
work
work
printer
wait
print
print
print

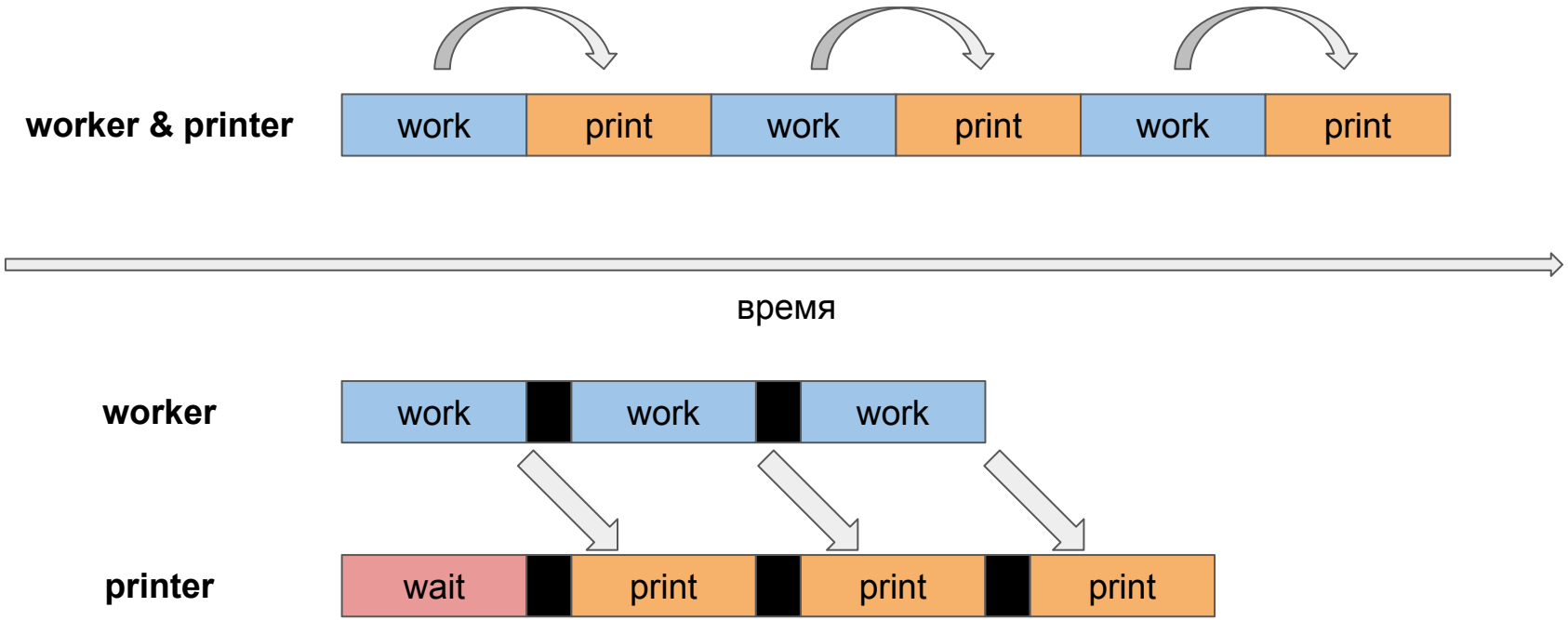
Пример


worker & printer
work
print
work
print
work
print
время
worker
work
work
work
printer
wait
print
print
print
одновременно!
(на двухядерной машине)

Что упустили?



Файловые операции



Эффективно ли это?

Эффективно ли это?

На одноядерной машине - нет

- Накладные расходы на переключение контекстов и на файловые операции

А на многоядерной?

Эффективно ли это?

На одноядерной машине - нет

- Накладные расходы на переключение контекстов и на файловые операции

А на многоядерной?

- Зависит от соотношения времён, затрачиваемых на работу, вывод и файловые операции

Эффективно ли это?

Файлы - это очень медленно, объём *work* должен быть действительно большим для того, чтобы это было эффективно

Есть и другие, более эффективные механизмы взаимодействия программ

Сигналы, сокеты, каналы, …

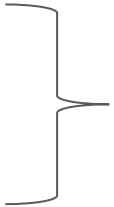
Тем не менее, все они работают через ОС, что создаёт существенные накладные расходы

А что если?

- память программы

- файлы

- прочие объекты ОС (сокеты, ...)



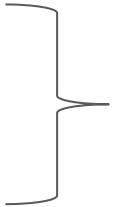
Пространство, разделяемое между программами

А что если?

- память программы

- файлы

- прочие объекты ОС (сокеты, ...)



Пространство, разделяемое между ~~программами~~ кусками кода

А что если?

- память программы

- файлы

- прочие объекты ОС (сокеты, ...)

Пространство, разделяемое между ~~программами~~ кусками кода

Многопоточность

Существование в одном процессе нескольких кусков кода, которые могут исполняться одновременно - **потоков**

Since C++11

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(chrono::milliseconds(1000));
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(chrono::milliseconds(1000));
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```

Добровольная отдача ядра другому потоку на 1000 миллисекунд (минимум)

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(chrono::milliseconds(1000));
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```

Добровольная отдача ядра другому потоку на 1000 миллисекунд (минимум)

Если этого не сделать, поток успеет сказать очень много слов за квант времени

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(...);
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```

Создание и запуск нового потока, ассоциированного с объектом ping

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(...);
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```

Создание и запуск нового потока, ассоциированного с объектом ping

Указатель на функцию, с которой начинается стек вызовов нового потока

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(...);
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```

Создание и запуск нового потока, ассоциированного с объектом ping

Указатель на функцию, с которой начинается стек вызовов нового потока

Значения параметров, передаваемых в функцию speaker

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(...);
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```

Аналогично создаём и запускаем второй поток

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(...);
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```

Аналогично создаём и запускаем второй поток

Это делает основной поток программы (начавшийся с функции main)

одновременно

С тем, как поток ping уже начал свою жизнь

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(...);
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```

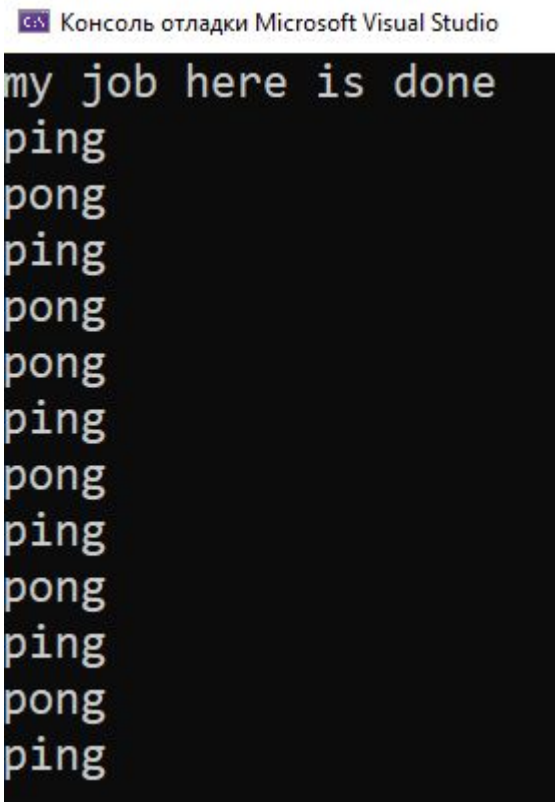
Репортуем об окончании работы

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(...);
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```

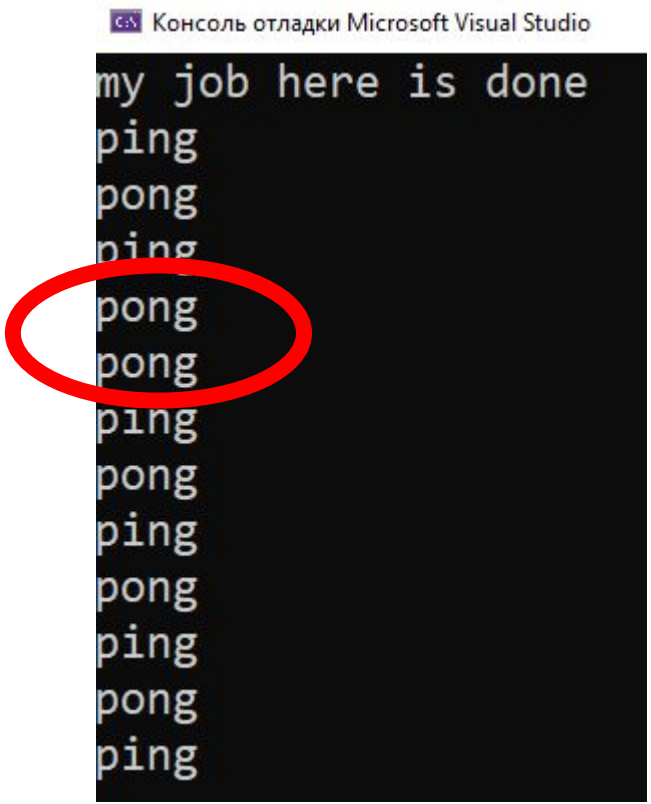
Если основной поток выйдет из функции main, это приведёт к вызову деструкторов для объектов ping и pong, которые ассоциированы с исполняющимися потоками

Так делать нельзя (другие способы решения проблемы - позже)

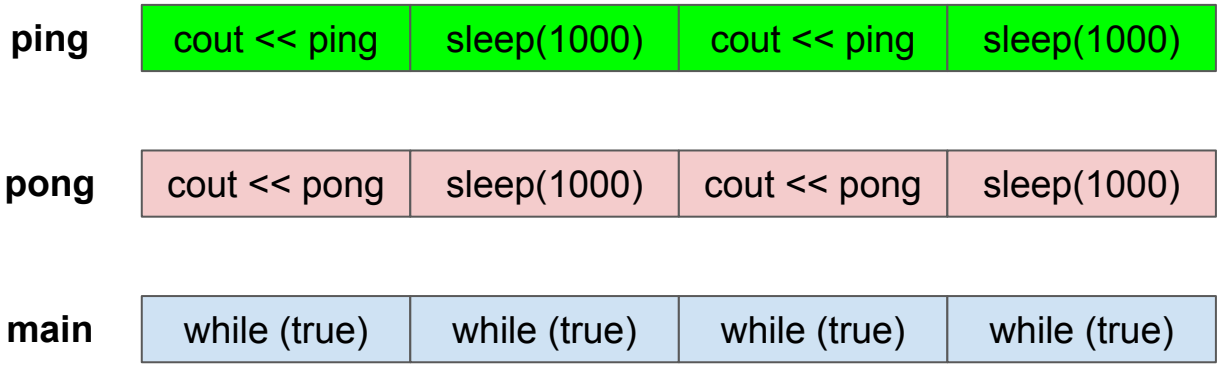
```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(...);
    }
}

void main() {
    thread ping(speaker, "ping");
    thread pong(speaker, "pong");

    cout << "my job here is done" << endl;
    while (true) {}
}
```



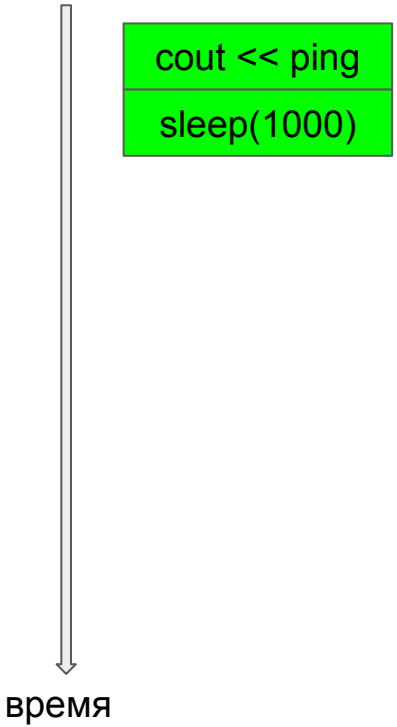
Как так то?!


Консоль отладки Microsoft Visual Studio
my job here is done
ping
pong
ping
pong
pong
ping
pong
ping
pong
ping
pong
ping

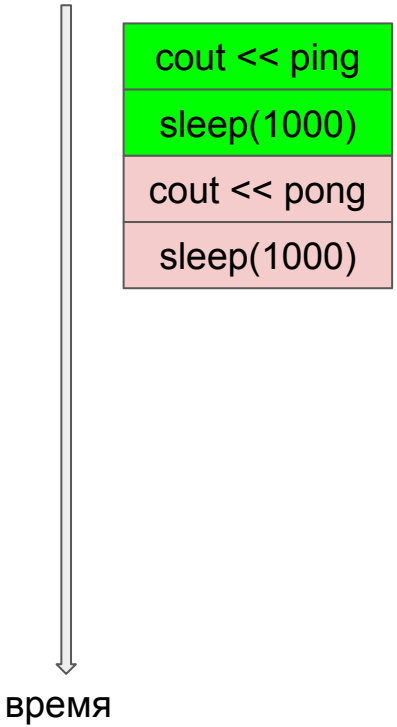
Например, вот так



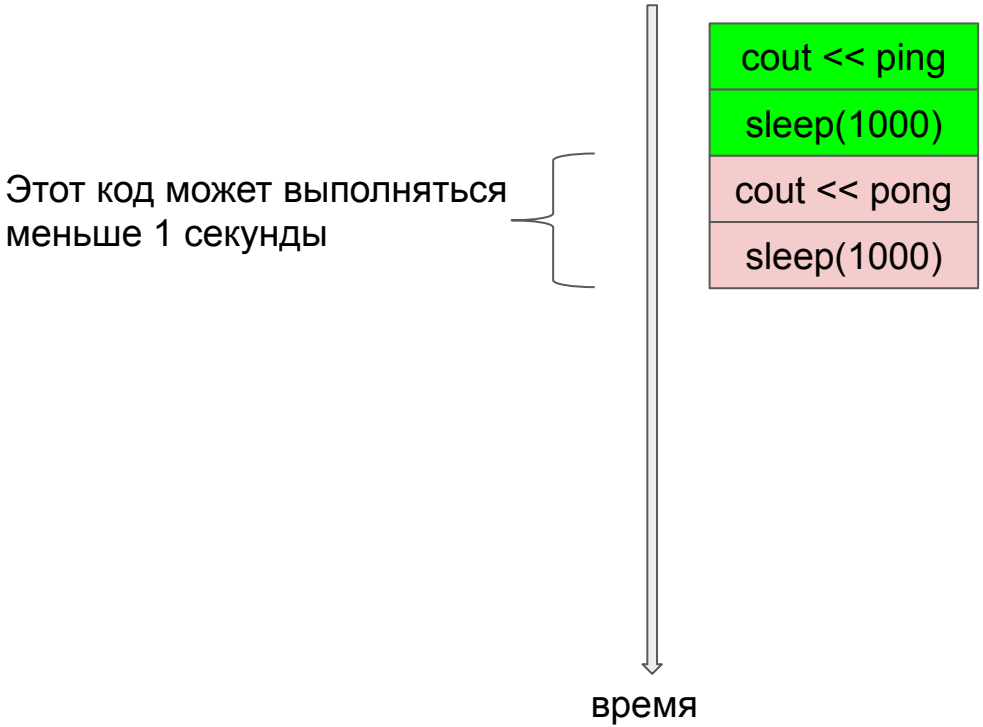
Например, вот так



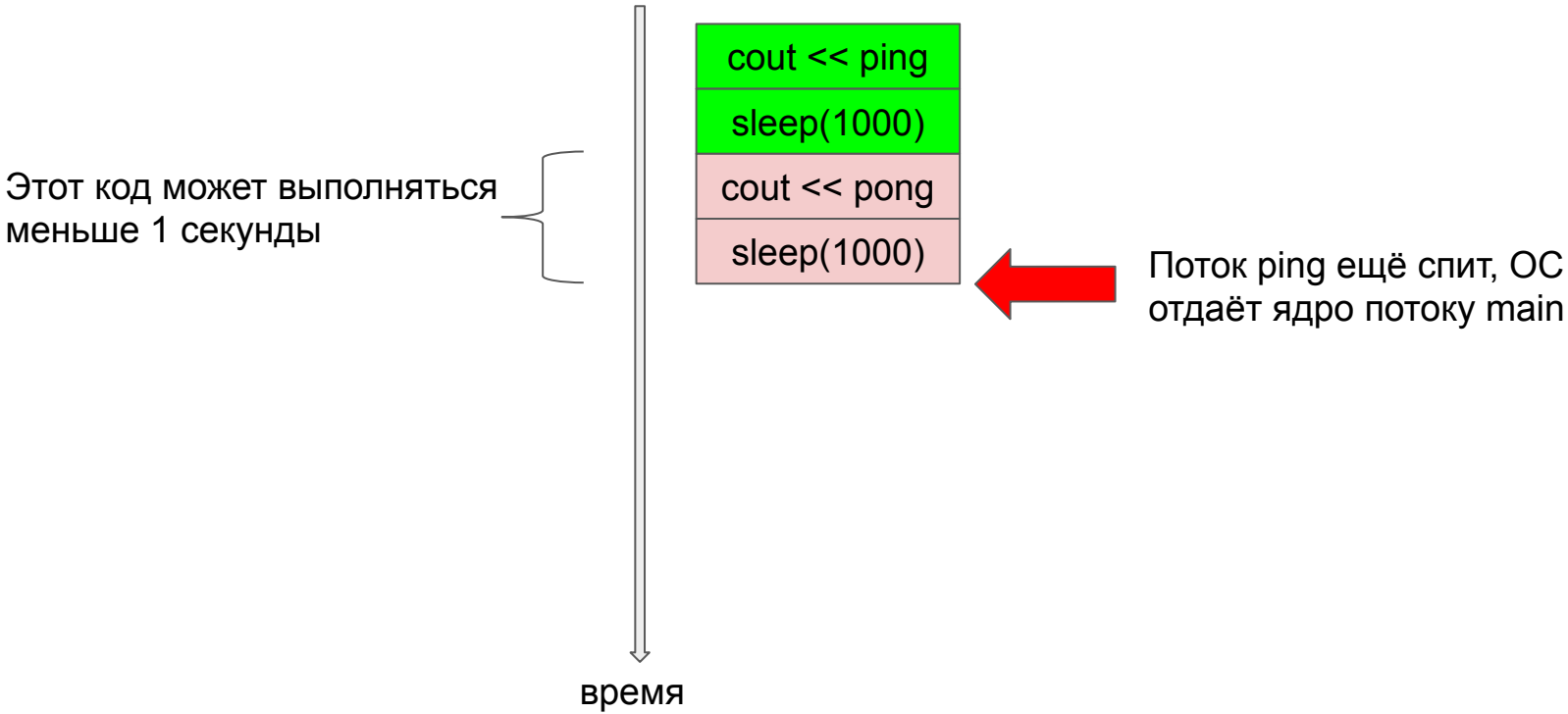
Например, вот так



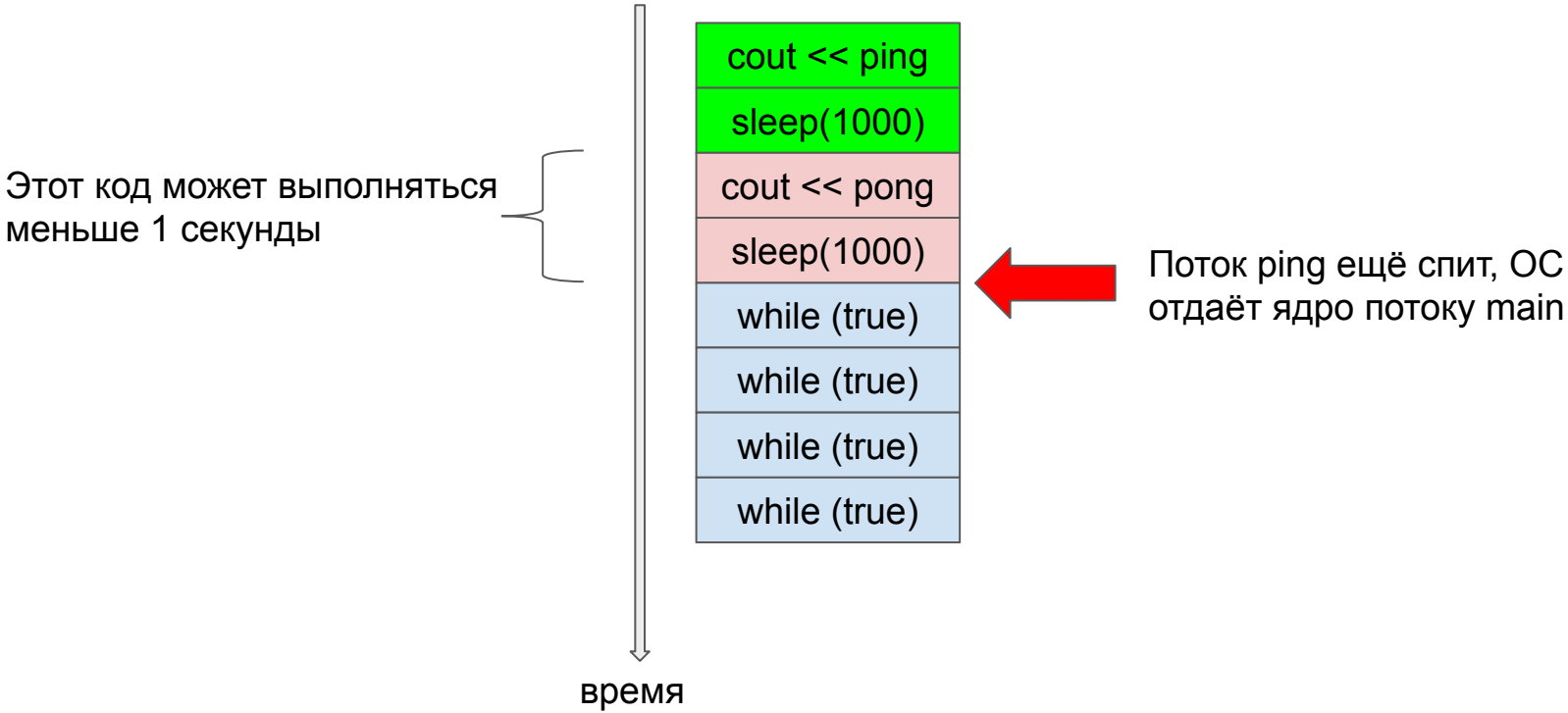
Например, вот так



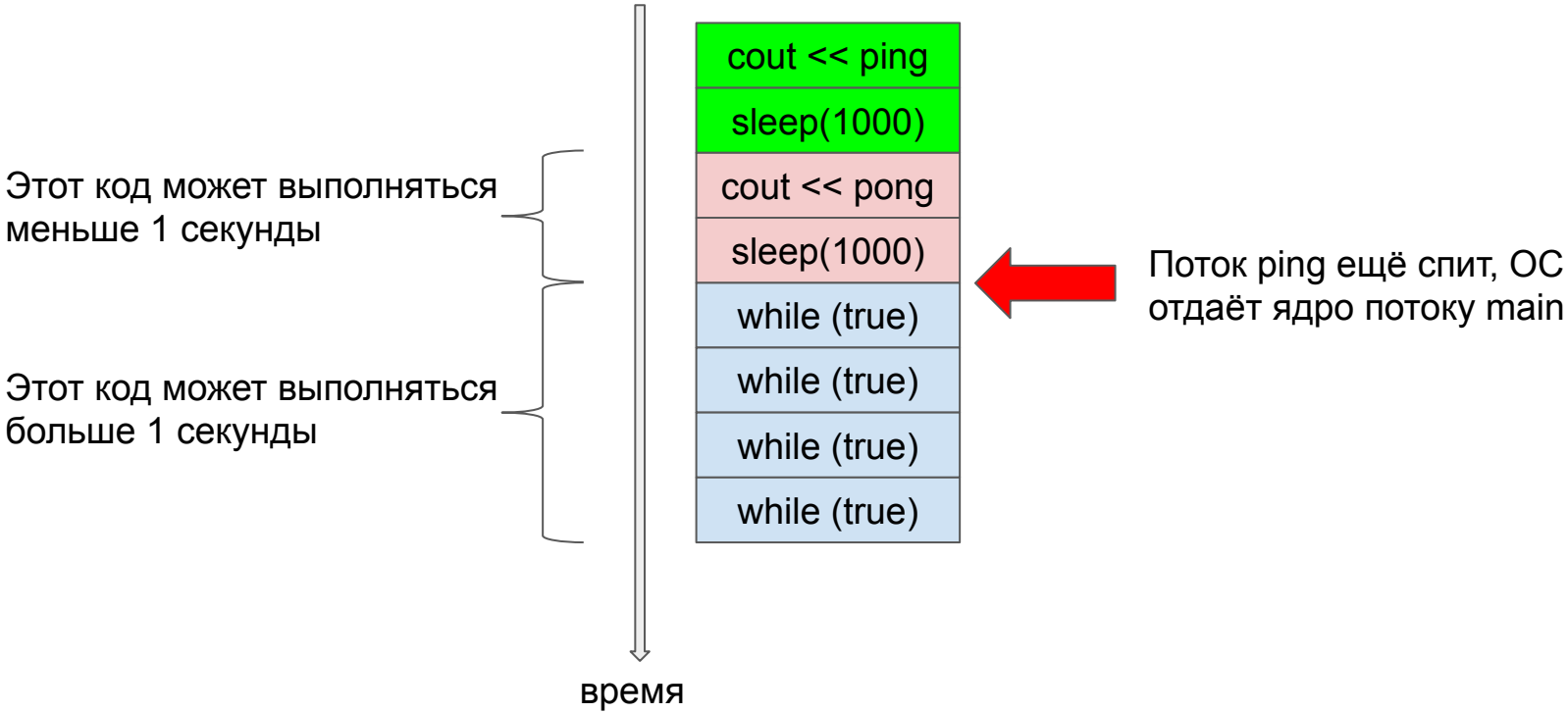
Например, вот так



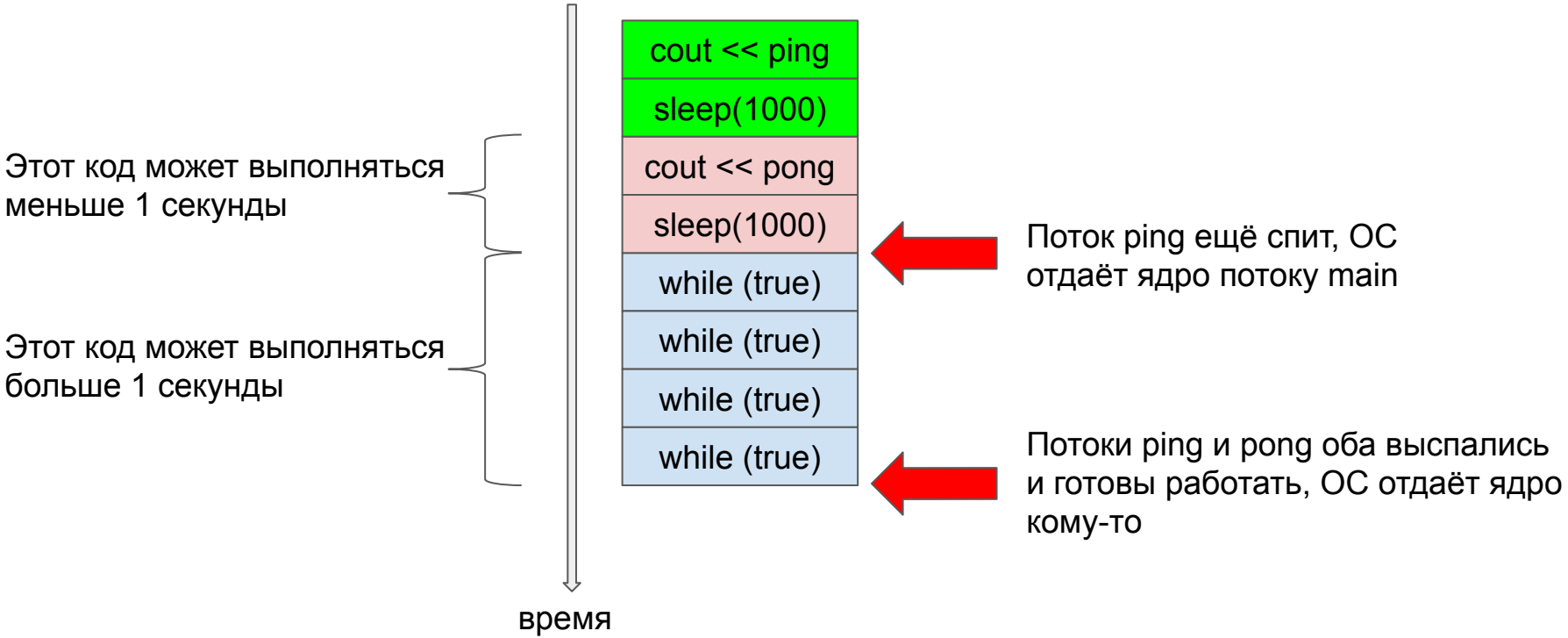
Например, вот так



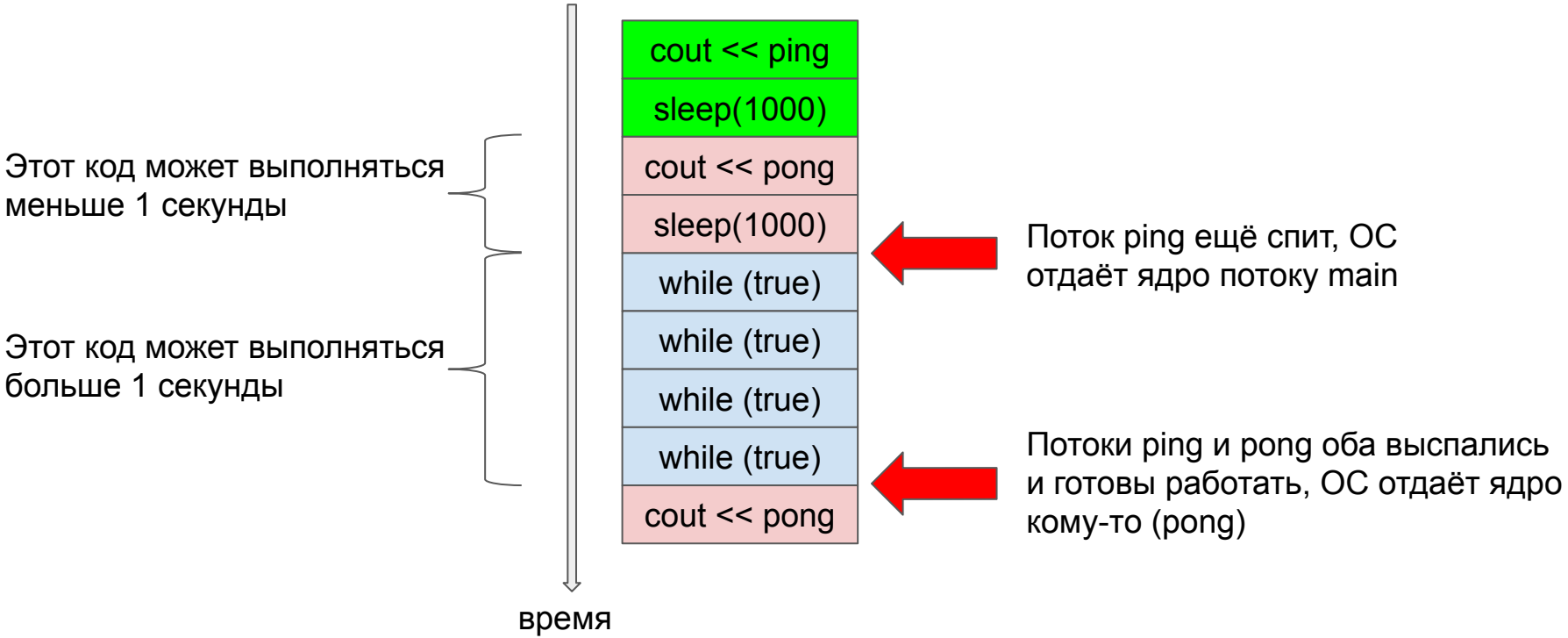
Например, вот так



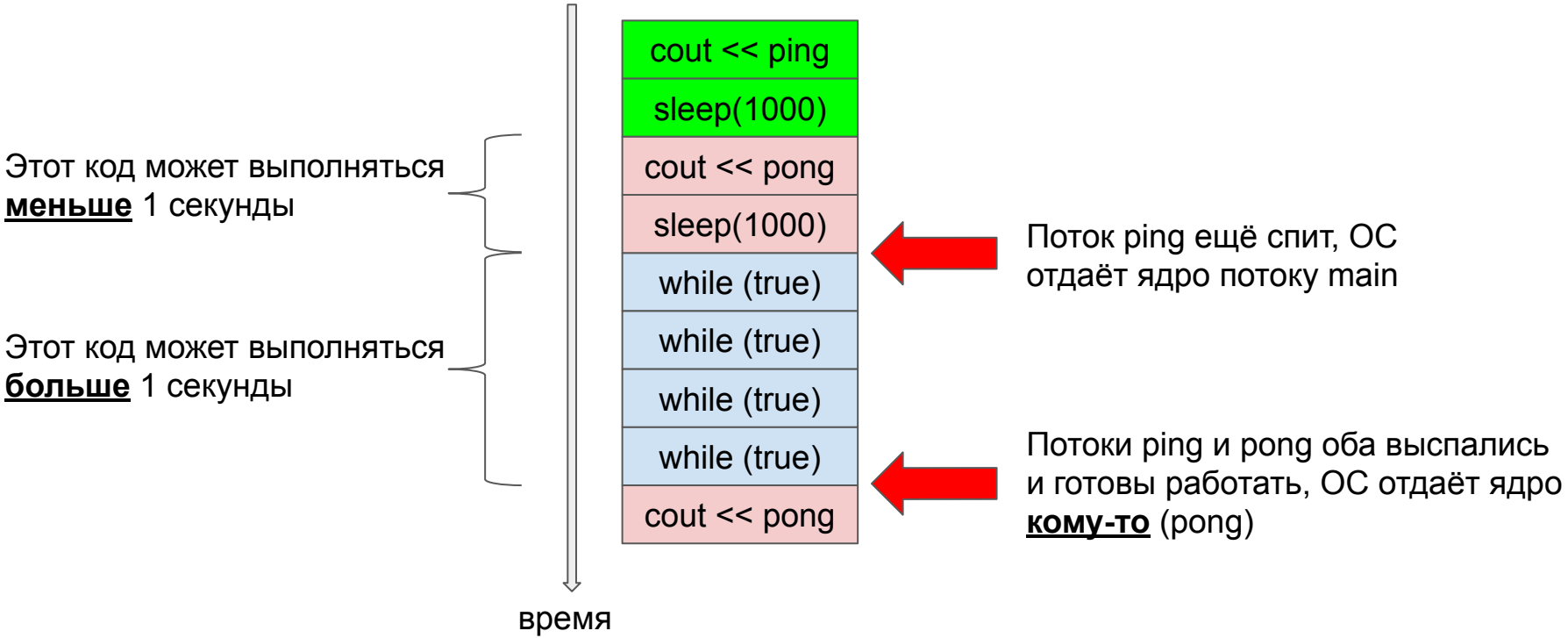
Например, вот так



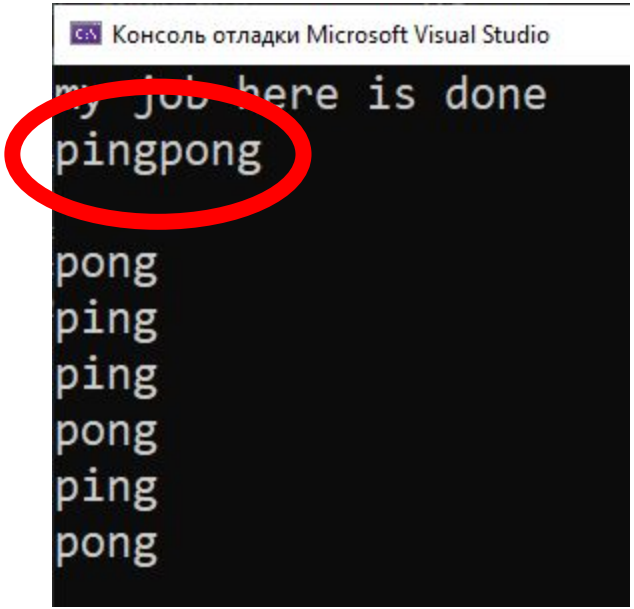
Например, вот так



Или как-то иначе



Дальше хуже



Один раз на двадцать запусков

Дальше хуже

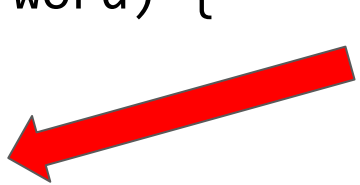
```
void speaker(string word) {
    while (true) {
        cout << word << endl;
        this_thread::sleep_for(...);
    }
}
```

Дальше хуже

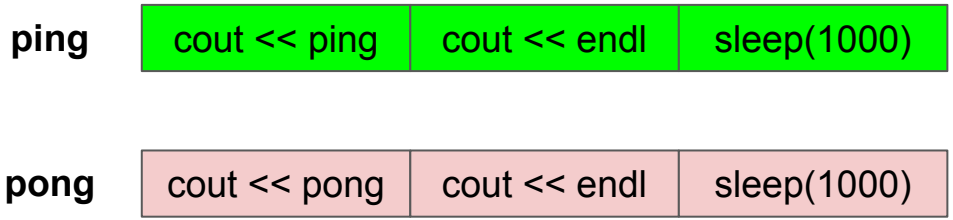
```
void speaker(string word) {
    while (true) {
        cout << word;
        cout << endl;
        this_thread::sleep_for(...);
    }
}
```

Дальше хуже

```
void speaker(string word) {
    while (true) {
        cout << word;
        cout << endl;
        this_thread::sleep_for(...);
    }
}
```



Дальше хуже

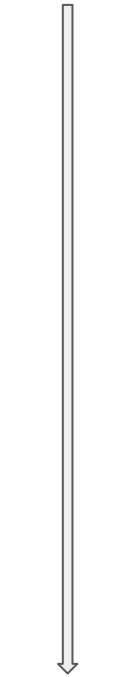

ping
cout << ping
cout << endl
sleep(1000)
pong
cout << pong
cout << endl
sleep(1000)

Дальше хуже

ядро

время

Дальше хуже



время

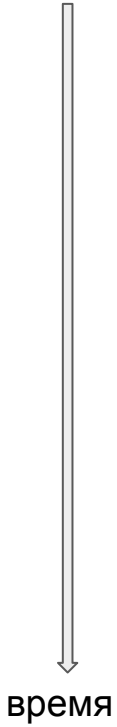
ядро №1

ядро №2

Дальше хуже


ядро №1
ядро №2
cout << ping
cout << endl
sleep(1000)
cout << pong
cout << endl
sleep(1000)
время

Дальше хуже



ядро №1

```
cout << ping
cout << endl
sleep(1000)
```

ядро №2

```
cout << pong
cout << endl
sleep(1000)
```

cout - общий!

вывод неопределён

Время неподконтрольно

Переключение контекста может происходить в произвольных точках кода

Если программист хочет, чтобы части его потоков исполнялись в определённом порядке - это его проблемы

Конструкции на ожиданиях (sleep) не работают!

А пространство?

А пространство?

Дар многозадачности усиливается

Память единая для всего процесса, то есть общая для всех потоков

Взаимодействие может осуществляться через память

А пространство?

Дар многозадачности усиливается

Память единая для всего процесса, то есть общая для всех потоков

Взаимодействие может осуществляться через память

Проклятие тоже усиливается

Неожиданные изменения теперь могут произойти в ваших объектах, массивах, глобальных переменных, ...

Как с этим жить?

Как с этим жить?

1) Контролировать **разделяемость** памяти

Как с этим жить?

1) Контролировать **разделяемость** памяти

Теоретически, вся память процесса разделена между потоками, но если исключить доступ по случайным адресам, то можно управлять доступностью каждого объекта из каждого потока

Как с этим жить?

1) Контролировать **разделяемость** памяти

2) Контролировать процесс изменений разделённой памяти

Как с этим жить?

1) Контролировать **разделяемость** памяти

2) Контролировать **процесс** изменений разделённой памяти

Протекающий во времени, с непредсказуемыми моментами переключения контекстов

Контроль разделяемости

Локальные переменные

У каждого потока - свой стек вызовов, фреймы и локальные переменные

Локальные переменные

У каждого потока - свой стек вызовов, фреймы и локальные переменные

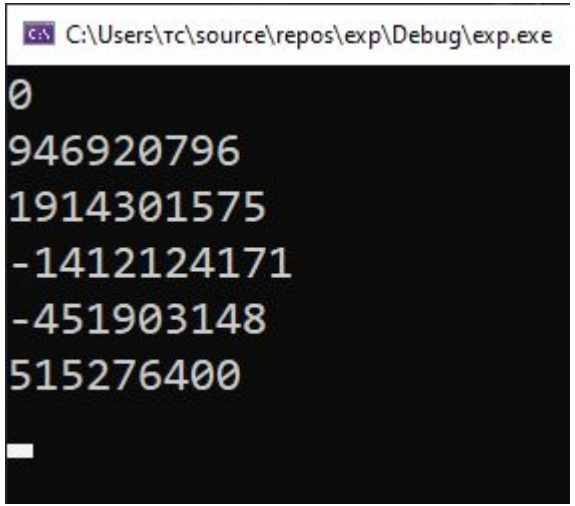
Наименее разделённая память - в обычном режиме значения локальных переменных потока не изменяются между квантами времени

Локальные переменные

У каждого потока - свой стек вызовов, фреймы и локальные переменные

Наименее разделённая память - в обычном режиме значения локальных переменных потока не изменяются между квантами времени

```
void foo(int* ptr) {
    while (true) {
        (*ptr)++;
    }
}

void main() {
    int x = 0;
    thread thr(foo, &x);

    while (true) {
        cout << x << endl;
        this_thread::sleep_for(...);
    }
}
```



Глобальные переменные

По умолчанию разделены между потоками

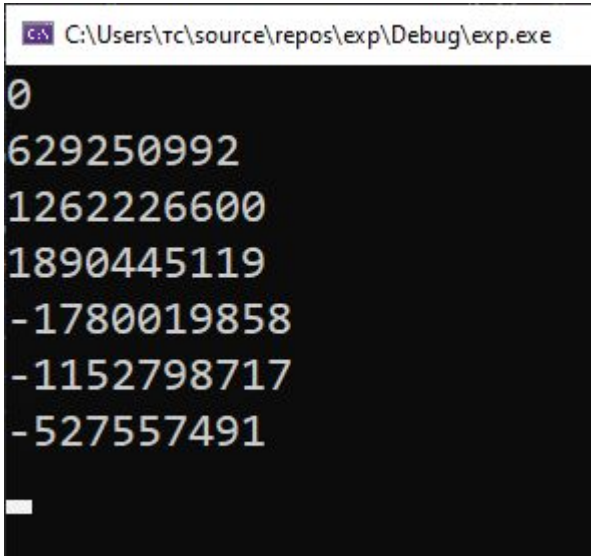
Thread-local storage

```
thread_local int foo;
```

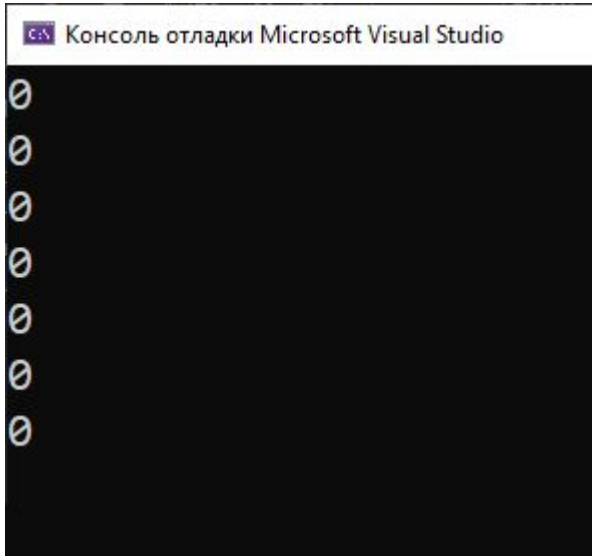
Для каждого потока создаётся своя версия этой переменной

В обычном режиме доступ к ней есть только у её потока

```
int x;

void foo() {
    while (true) {
        x++;
    }
}

void main() {
    thread thr(foo);

    while (true) {
        cout << x << endl;
        this_thread::sleep_for(...);
    }
}
```



```
thread_local int x;

void foo() {
    while (true) {
        x++;
    }
}

void main() {
    thread thr(foo);

    while (true) {
        cout << x << endl;
        this_thread::sleep_for(...);
    }
}
```



Динамическая память

Объект создаётся в операторе new, и единственный указатель возвращается вызвавшему коду

Дальнейшее распространение зависит от программиста

Достижимость объекта из потока

Множество объектов, достижимых из потока, вычисляется:

1) Параметры, переданные в поток
2) Глобальные переменные
3) Объекты, на которые есть указатели или ссылки из других достижимых объектов

Контроль разделяемости

Меньше глобальных переменных, по возможности - thread_local

Аккуратная передача параметрами в поток только необходимых объектов

Неизменяемые объекты

Неизменяемые объекты

Проклятие многозадачности - в непредсказуемых изменениях

Если объект не может измениться - проблемы нет

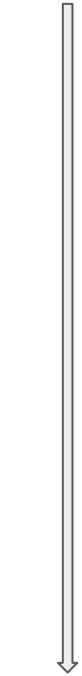
Ок, я максимально уменьшил разделяемую память. Теперь всё безопасно?

Контроль процесса изменений

Ок, я максимально уменьшил разделяемую память. Теперь всё безопасно?

Нет. Теперь нужно определить процесс изменений этой памяти.

Дальше хуже



время

ядро №1

```
cout << ping
cout << endl
sleep(1000)
```

ядро №2

```
cout << pong
cout << endl
sleep(1000)
```

cout - общий!

вывод неопределён

worker & printer

```
struct Point {
    int x;
    int y;
};

Point result;
```

```
void worker() {
    while (...) {
        int x = ...;
        int y = ...;

        result.x = x;
        result.y = y;
    }
}
```

```
void printer() {
    while (...) {
        int x = result.x;
        int y = result.y;

        cout & sleep
    }
}
```

worker & printer

```
struct Point {
    int x;
    int y;
};

Point result;
```

разделяемая переменная

```
void worker() {
    while (...) {
        int x = ...;
        int y = ...;

        result.x = x;
        result.y = y;
    }
}
```

```
void printer() {
    while (...) {
        int x = result.x;
        int y = result.y;

        cout & sleep
    }
}
```

worker & printer

```
struct Point {
    int x;
    int y;
};

Point result;
```

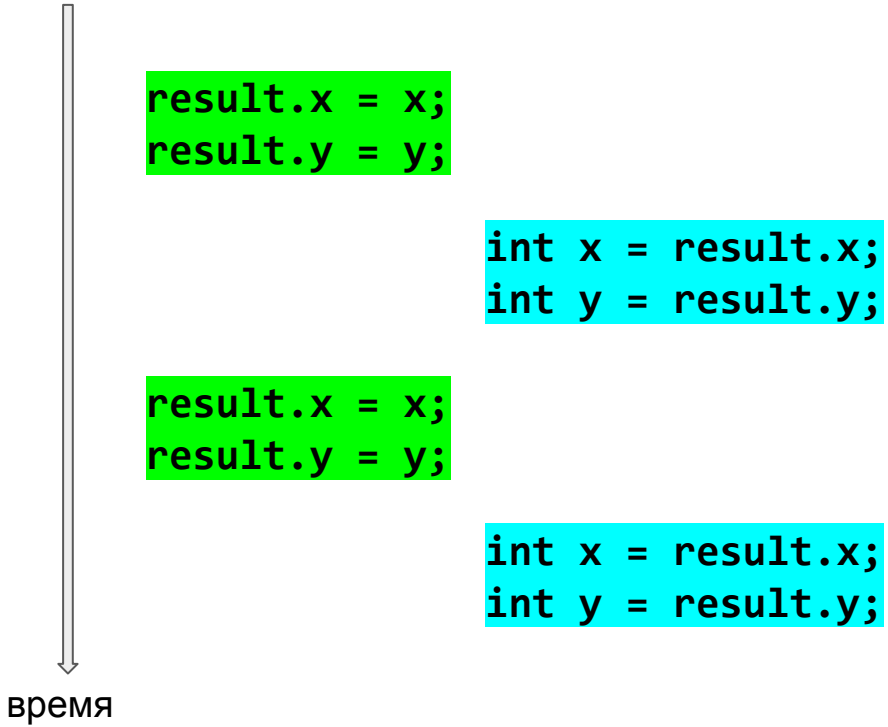
разделяемая переменная

```
void worker() {
    while (...) {
        int x = ...;
        int y = ...;

        result.x = x;
        result.y = y;
    }
}
```

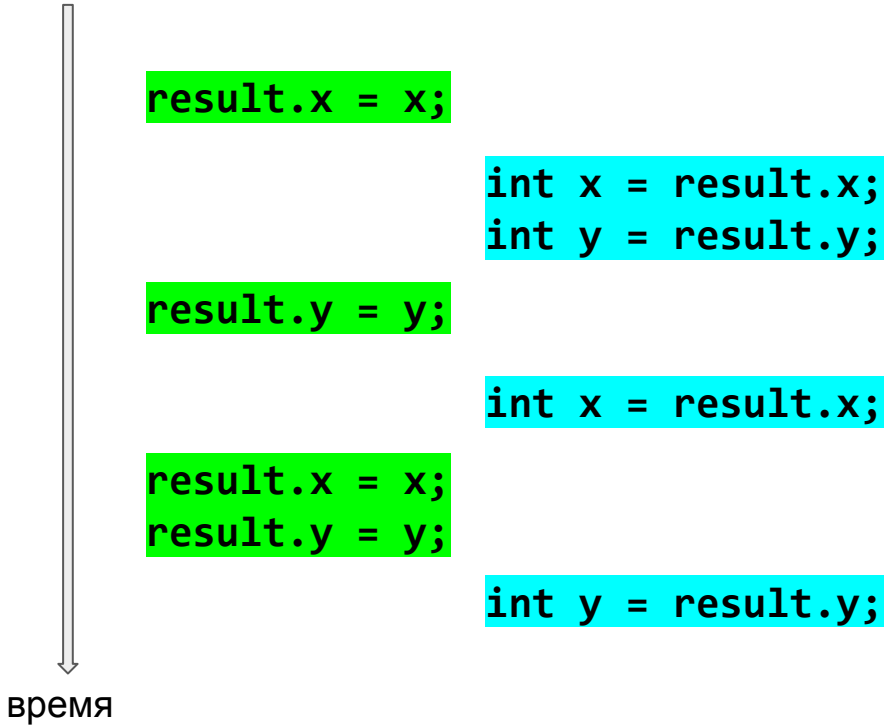
```
void printer() {
    while (...) {
        int x = result.x;
        int y = result.y;

        cout & sleep
    }
}
```

worker & printer

Всё хорошо


result.x = x;
result.y = y;
int x = result.x;
int y = result.y;
result.x = x;
result.y = y;
int x = result.x;
int y = result.y;
время

Полезли демоны



Критические секции

Участки кода, где происходит чтение или запись разделяемых переменных, обращение к разделяемым ресурсам (например, cout) и прочие действия, которые нужно защищать друг от друга

Критические секции

Участки кода, где происходит чтение или запись разделяемых переменных, обращение к разделяемым ресурсам (например, cout) и прочие действия, которые нужно защищать друг от друга

worker

```
result.x = x;
result.y = y;
```

критические секции

printer

```
int x = result.x;
int y = result.y;
```

Критические секции

Факторизуются по разделяемым переменным

Секции, модифицирующие разные переменные, чаще всего не нужно защищать друг от друга

Определение, какие участки - критические, лежит на программисте

Простейший способ защиты - в один момент времени (реального, а не ядерного) только один поток может находится в критической секции

Критические секции

Факторизуются по разделяемым переменным

Секции, модифицирующие разные переменные, чаще всего не нужно защищать друг от друга

Определение, какие участки - критические, лежит на программисте

Простейший способ защиты - в один момент времени (**реального**, а не ядерного) только один поток может находится в критической секции

Мьютексы (mutex, mutually exclusive)

Объект, находящийся постоянно в одном из двух состояний - свободный (начальное состояние) или заблокированный

Любой поток может узнать, заблокирован мьютекс, или нет

Любой поток может заблокировать мьютекс, если тот свободен, и освободить, если он заблокирован

Мьютексы (mutex, mutually exclusive)

С каждой группой критических секций ассоциируется мьютекс

Перед входом в критическую секцию поток ждёт, пока мьютекс не станет свободным, дождавшись, блокирует его и заходит в критическую секцию

Перед выходом из критической секции поток освобождает мьютекс

Давайте попробуем!

Давайте попробуем!

На первый взгляд мьютексом может
работать обычная логическая переменная

```
Point result;
```

worker

```
result.x = x;
result.y = y;
```

printer

```
int x = result.x;
int y = result.y;
```

Мьютекс

```
Point result;
boolean resultLock;
```

worker

```
result.x = x;
result.y = y;
```

printer

```
int x = result.x;
int y = result.y;
```

Входы в критические секции

```
Point result;
boolean resultLock;
```

worker

```
while (resultLock);
resultLock = true;

result.x = x;
result.y = y;
```

printer

```
while (resultLock);
resultLock = true;

int x = result.x;
int y = result.y;
```

Выходы из критических секций

```
Point result;
boolean resultLock;
```

worker

```
while (resultLock);
resultLock = true;

result.x = x;
result.y = y;

resultLock = false;
```

printer

```
while (resultLock);
resultLock = true;

int x = result.x;
int y = result.y;

resultLock = false;
```

Ну что, вроде нормально?

```
Point result;
boolean resultLock;
```

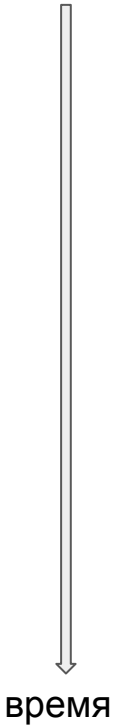
worker

```
while (resultLock);
resultLock = true;

result.x = x;
result.y = y;

resultLock = false;
```

printer

```
while (resultLock);
resultLock = true;

int x = result.x;
int y = result.y;

resultLock = false;
```

Всё хорошо

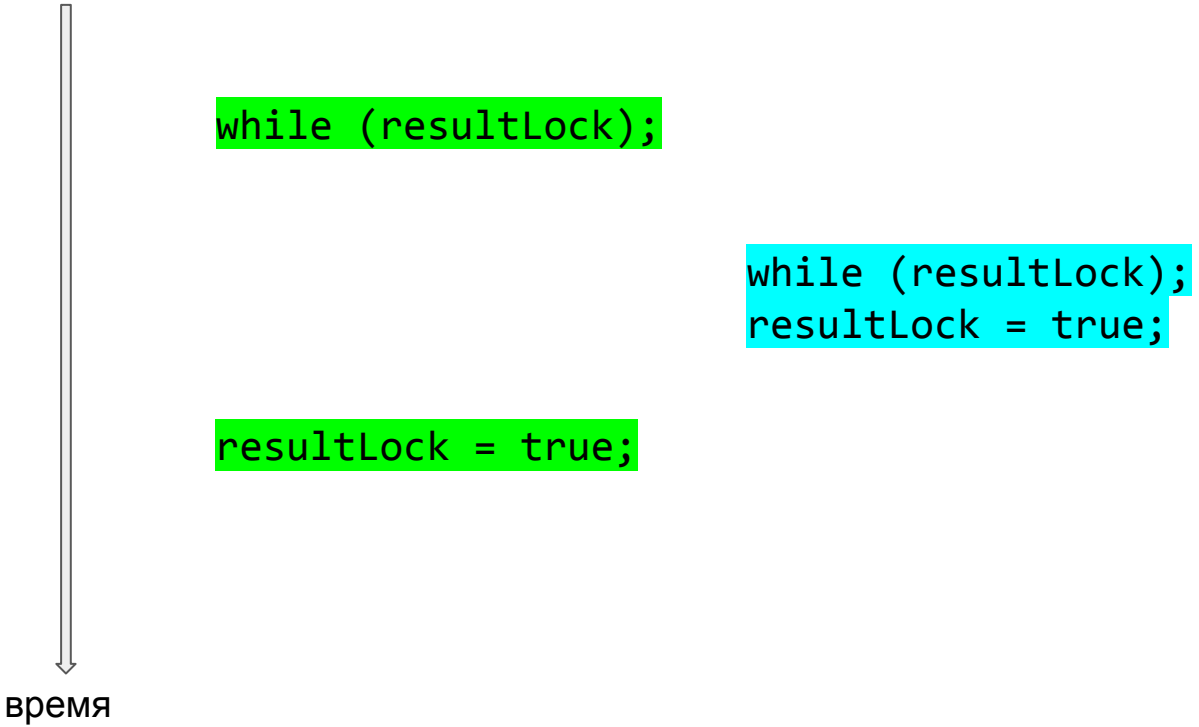


```
while (resultLock);
resultLock = true;
```

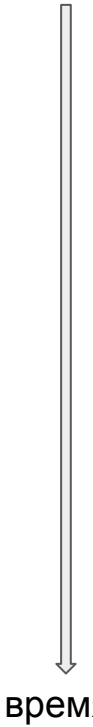
```
while (resultLock);
resultLock = true;
```

время

Hea


while (resultLock);
while (resultLock);
resultLock = true;
resultLock = true;
время

Hea



время

```
while (resultLock);
```

```
while (resultLock);
resultLock = true;
```

```
resultLock = true;
```

оба потока зайдут в
критическую секцию


Ah [redacted], here we go again.

Что делать?

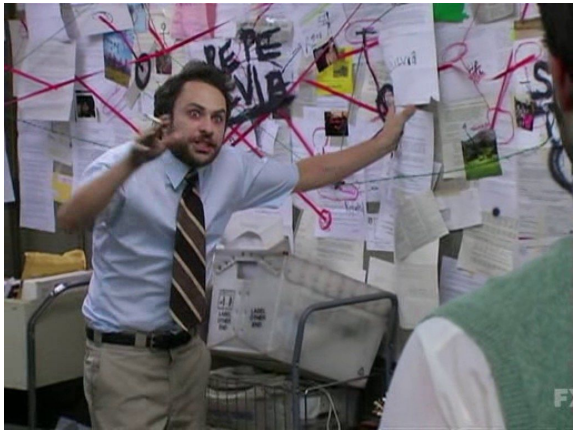
Защитить мьютекс ещё одним мьютексом?

Подсыпать каких-нибудь sleep’ов?

Что делать?

Защитить мьютекс ещё одним мьютексом?

Подсыпать каких-нибудь sleep’ов?



Хватит это терпеть

Атомарная операция - операция, которая выполняется целиком или не выполняется вовсе

Хватит это терпеть

Атомарная операция - операция, которая выполняется целиком или не выполняется вовсе

Все современные процессоры общего назначения предоставляют некоторый набор атомарных операций

compare & swap (CAS)

Атомарное сравнение переменной с предполагаемым значением и (в случае равенства) замена её значения на новое

Распространённый вариант (напрямую поддержан intel x86 & x64)

Другие процессоры могут предоставлять эквивалентные операции

Не работает

```
Point result;
boolean resultLock;
```

worker

```
while (resultLock);
resultLock = true;

result.x = x;
result.y = y;

resultLock = false;
```

printer

```
while (resultLock);
resultLock = true;

int x = result.x;
int y = result.y;

resultLock = false;
```

Проверка и вход в критическую секцию атомарен

```
Point result;
boolean resultLock;
```

worker

```
while (!CAS(resultLock, false, true));

result.x = x;
result.y = y;

resultLock = false;
```

printer

```
while (!CAS(...));

int x = result.x;
int y = result.y;

resultLock = false;
```

Есть в стандартной библиотеке C++11

Defined in header <atomic>

```
template< class T >
bool atomic_compare_exchange_weak( std::atomic<T>* obj,
                                   typename std::atomic<T>::value_type* expected,
                                   typename std::atomic<T>::value_type desired ) noexcept;
template< class T >
bool atomic_compare_exchange_weak( volatile std::atomic<T>* obj,
                                   typename std::atomic<T>::value_type* expected,
                                   typename std::atomic<T>::value_type desired ) noexcept;
```

(1) (since C++11)

```
template< class T >
bool atomic_compare_exchange_strong( std::atomic<T>* obj,
                                     typename std::atomic<T>::value_type* expected,
                                     typename std::atomic<T>::value_type desired ) noexcept;
template< class T >
bool atomic_compare_exchange_strong( volatile std::atomic<T>* obj,
                                     typename std::atomic<T>::value_type* expected,
                                     typename std::atomic<T>::value_type desired ) noexcept;
```

(2) (since C++11)

```
template< class T >
bool atomic_compare_exchange_weak_explicit( std::atomic<T>* obj,
                                            typename std::atomic<T>::value_type* expected,
                                            typename std::atomic<T>::value_type desired,
                                            std::memory_order succ,
                                            std::memory_order fail ) noexcept;
template< class T >
bool atomic_compare_exchange_weak_explicit( volatile std::atomic<T>* obj,
                                            typename std::atomic<T>::value_type* expected,
                                            typename std::atomic<T>::value_type desired,
                                            std::memory_order succ,
                                            std::memory_order fail ) noexcept;
```

(3) (since C++11)

```
template< class T >
bool atomic_compare_exchange_strong_explicit( std::atomic<T>* obj,
                                              typename std::atomic<T>::value_type* expected,
                                              typename std::atomic<T>::value_type desired,
                                              std::memory_order succ,
```

Но проще воспользоваться встроенными mutex

```
Point result;
mutex resultLock;
```

worker

```
resultLock.lock();

result.x = x;
result.y = y;

resultLock.unlock();
```

printer

```
resultLock.lock();

int x = result.x;
int y = result.y;

resultLock.unlock();
```

Интерфейс mutex

void lock() - атомарно проверяет, свободен ли мьютекс, и, если да, блокирует его, иначе ждёт, пока мьютекс разблокируется другим потоком

bool try_lock() - атомарно проверяет, свободен ли мьютекс, и, если да, блокирует его и возвращает true, иначе возвращает false

void unlock() - разблокирует мьютекс

Интерфейс mutex

void lock() - атомарно проверяет, свободен ли мьютекс, и, если да, блокирует его, иначе **ждёт**, пока мьютекс разблокируется другим потоком

Это ожидание называется **блокировкой** потока, вызвавшего lock()

Интерфейс mutex

Если один и тот же поток попробует заблокировать один и тот же мьютекс дважды:

1) Мьютекс заблокируется на первой попытке
2) На второй попытке заблокируется сам поток
3) И мьютекс никто не освободит

recursive_mutex

Хранит в себе идентификатор потока, заблокировавшего его, и количество входов в критические секции

Очевидно дороже обычного mutex

Использовать при необходимости

mutex + инкапсуляция

Разделяемый объект включает в себя полем мьютекс

Критические секции оформляются методами, вызываемыми от объекта

mutex + инкапсуляция

Разделяемый объект включает в себя полем мьютекс

Критические секции оформляются методами, вызываемыми от объекта

Не работает, если в критической секции затрагиваются несколько разных разделяемых объектов

Взаимная блокировка (deadlock)

```
mutex lock1;
mutex lock2;
```

thread 1

```
lock1.lock();
...
lock2.lock();
...
lock2.unlock();
lock1.unlock();
```

thread 2

```
lock2.lock();
...
lock1.lock();
...
lock1.unlock();
lock2.unlock();
```

Взаимная блокировка (deadlock)

Может быть не такой явной

Например, быть размазанной по стеку вызовов

Удачной отладки!

Резюме

1) Минимизируйте разделяемую память, давайте доступ потокам к объектам сознательно и с чётко определённой целью
2) Определяйте критические секции, по возможности инкапсулируйте их с мьютексами
3) Избегайте лишних блокировок - это дорого и ведёт к взаимным блокировкам
4) Не используйте sleep и самопальные мьютексы для синхронизации

Резюме

Многозадачность - мощный и опасный инструмент

И огромная тема в IT (на порядок больше рассмотренного в лекции)

Необходимость его использовать двигает прогресс в языках и библиотеках

Так, например, следствие - популярность функциональных языков

Не моргай!

Q & A