Кирилл Волков @ MERA
github.com/vulko/Cpp_Basics_Lectures

C++

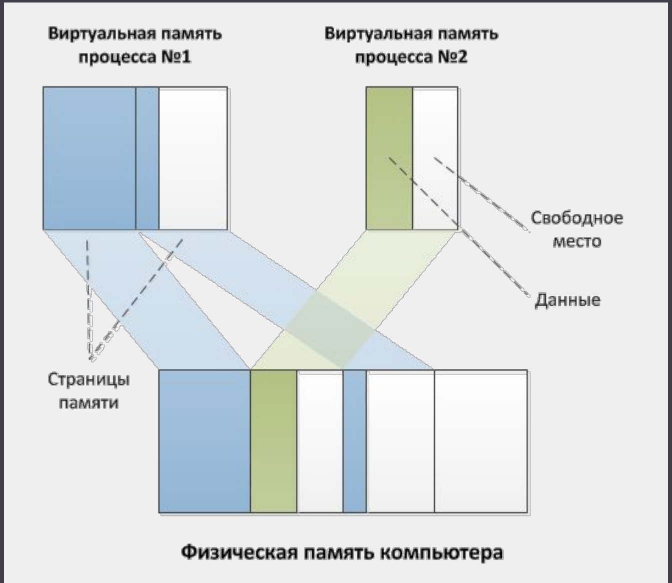
Парадигма исполняемого приложения в современных ОС. Процесс, поток и работа с памятью.

Однопоточность vs многопоточность. Создание и управление потоками. Основные проблемы и способы их решения. Принципы синхронизации и разделяемого доступа.

Парадигма работы ОС

- OS manages memory as pages
- Process is an instance of a program, being executed
- Process has a memory associated with it and one or multiple execution threads
- OS runs multiple independent processes to avoid system failures
- Process has MAIN thread of execution

- Thread is a sequence of commands
- Threads exist only as a part of a process
- Multiple threads share state, memory and other resources of a parent process
- Context switching btw threads is faster than context switching btw processes

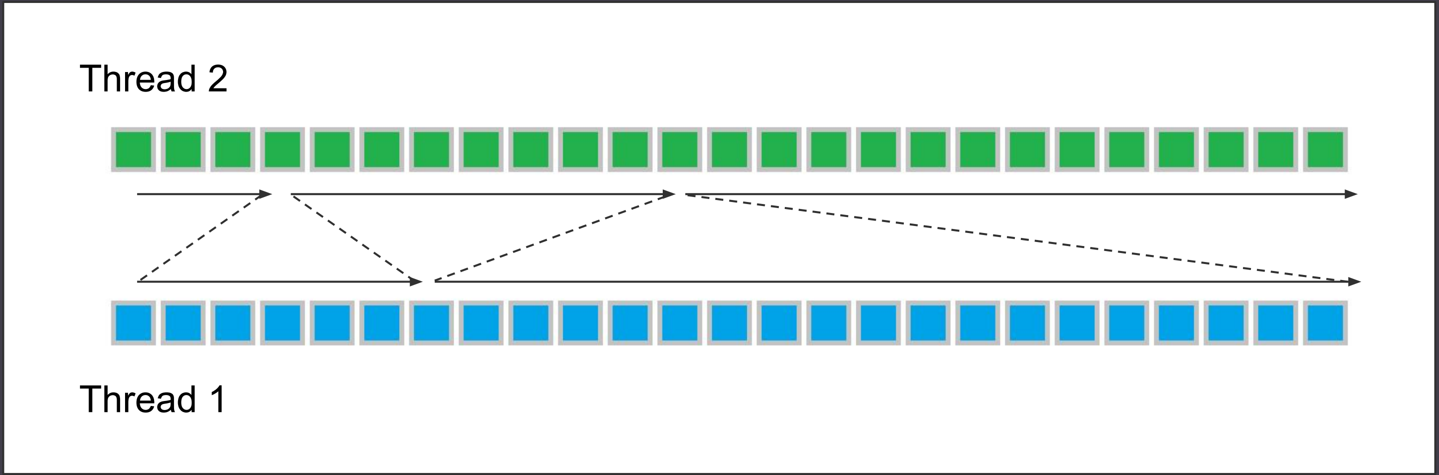

Виртуальная память процесса №1
Виртуальная память процесса №2
Свободное место
Данные
Страницы памяти
Физическая память компьютера

Процесс

- Memory (typically some region of virtual memory)
    - executable code
    - process-specific data (input and output)
    - call stack
    - heap to hold data generated during run time
- OS descriptors such as file descriptors or handles, data sources and sinks.
- Security attributes, such as the process owner and the set of permissions
- Processor state (context)
    - content of registers and physical memory addressing
    - state is typically stored in computer registers when the process is executing, and in memory otherwise
- OS holds most of this information about active processes in process control blocks
- Most OS have IPC mechanisms

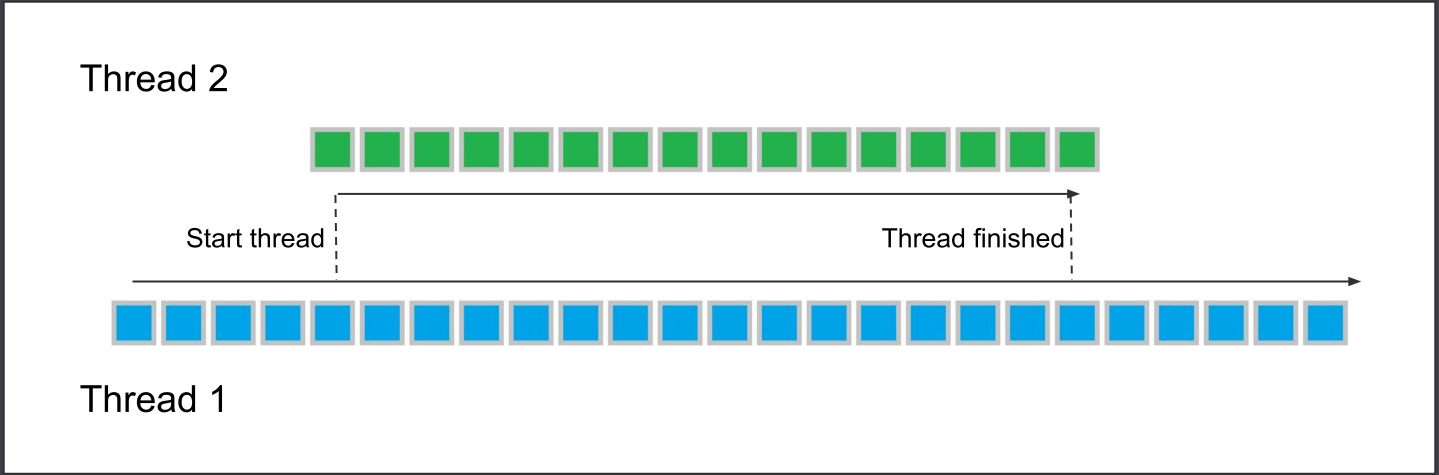
Поток

- Multiple threads can run on a single CPU core
- Process switches between threads, so they are executed consequently
- No need for synchronization
- No performance bonus, instead overhead when switching between threads



Поток

- Multiple threads can run on several CPU cores simultaneously
- Due to asynchronous behavior, might require synchronization (e.g. use of shared data)
- Potential deadlocks, race conditions, etc
- Performance bonus, no overhead when switching between threads



Поток

std::thread

Implements thread API

Runs a function in a separate thread

```
std::thread(FunctionPointer, Args...);
```

Start an std::thread

```
void f1(int n)
{
    for (int i = 0; i < 5; ++i) {
        std::cout << "Thread 1 executing\n";
        ++n;

        std::this_thread::sleep_for(
            std::chrono::milliseconds(10));
    }
}
```

```
// pass by value
std::thread t1(f1, n + 1);
```

Start an std::thread

```
void f2(int& n)
{
    for (int i = 0; i < 5; ++i) {
        std::cout << "Thread 2 executing\n";
        ++n;

        std::this_thread::sleep_for(
            std::chrono::milliseconds(10));
    }
}
```

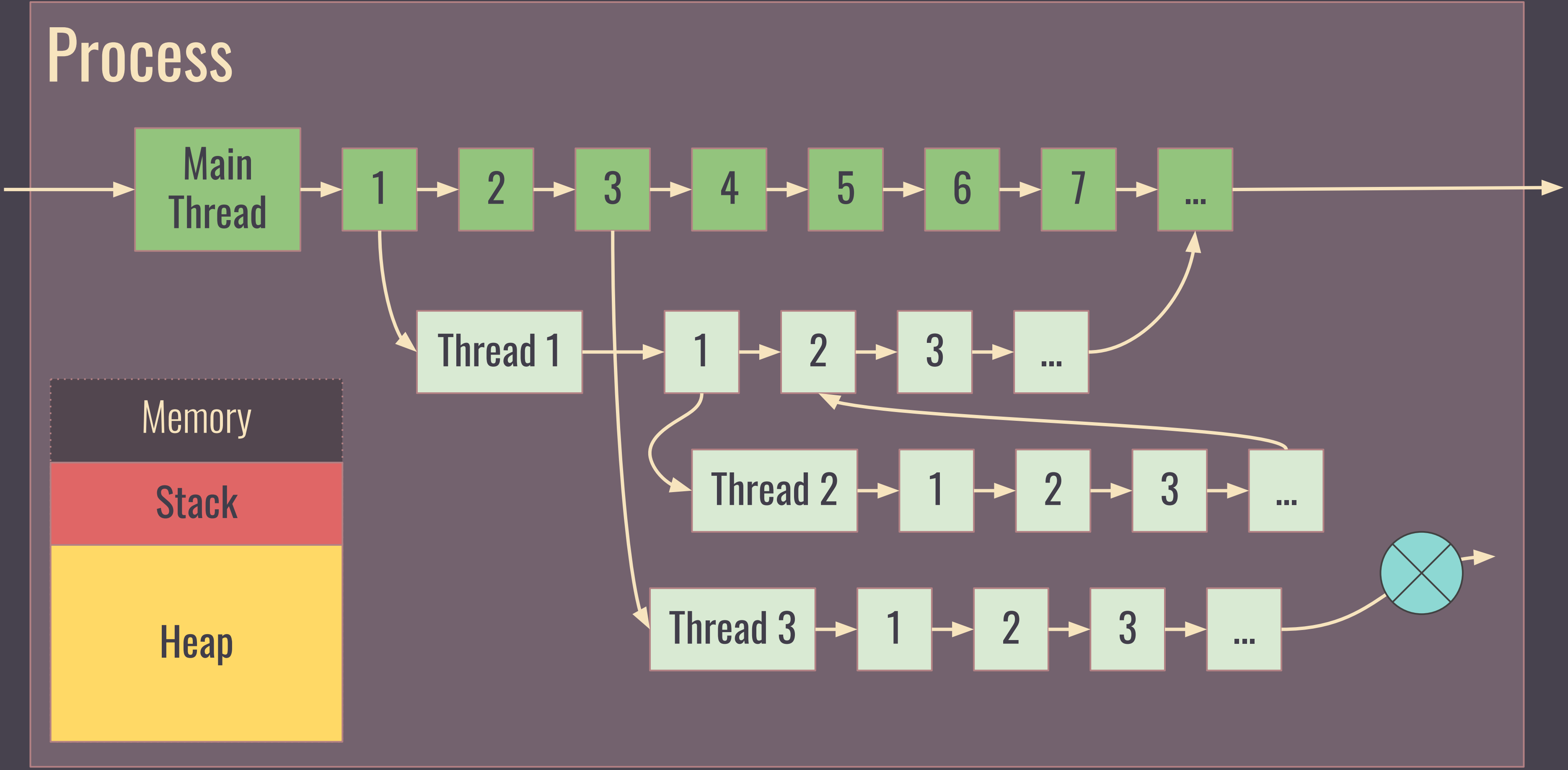
```
// pass by reference
std::thread t2(f2, std::ref(n));

// t4 is now running f2().
// t3 is no longer a thread
std::thread t3(std::move(t2));
```

Start an std::thread

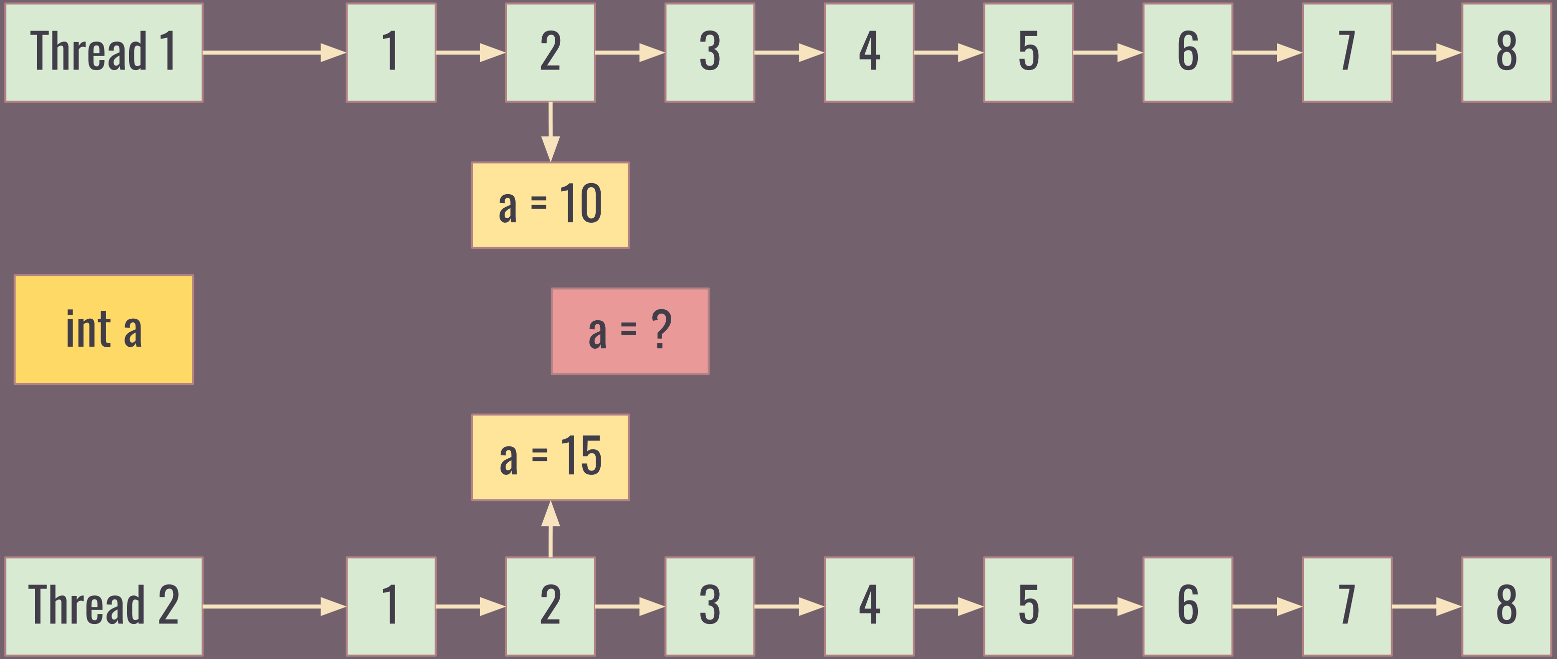
```
class foo {
public:
    void bar() {
        for (int i = 0; i < 5; ++i) {
            std::cout << "Thread 3 executing\n";
            ++n;
            std::this_thread::sleep_for(
                std::chrono::milliseconds(10));
        }
    }
    int n = 0;
};
```

```
foo f;
foo f1;

// t4 runs foo::bar() on object f
std::thread t4(&foo::bar, &f);

// t5 runs foo::bar() on object f1
std::thread t4(&foo::bar, &f1);
```

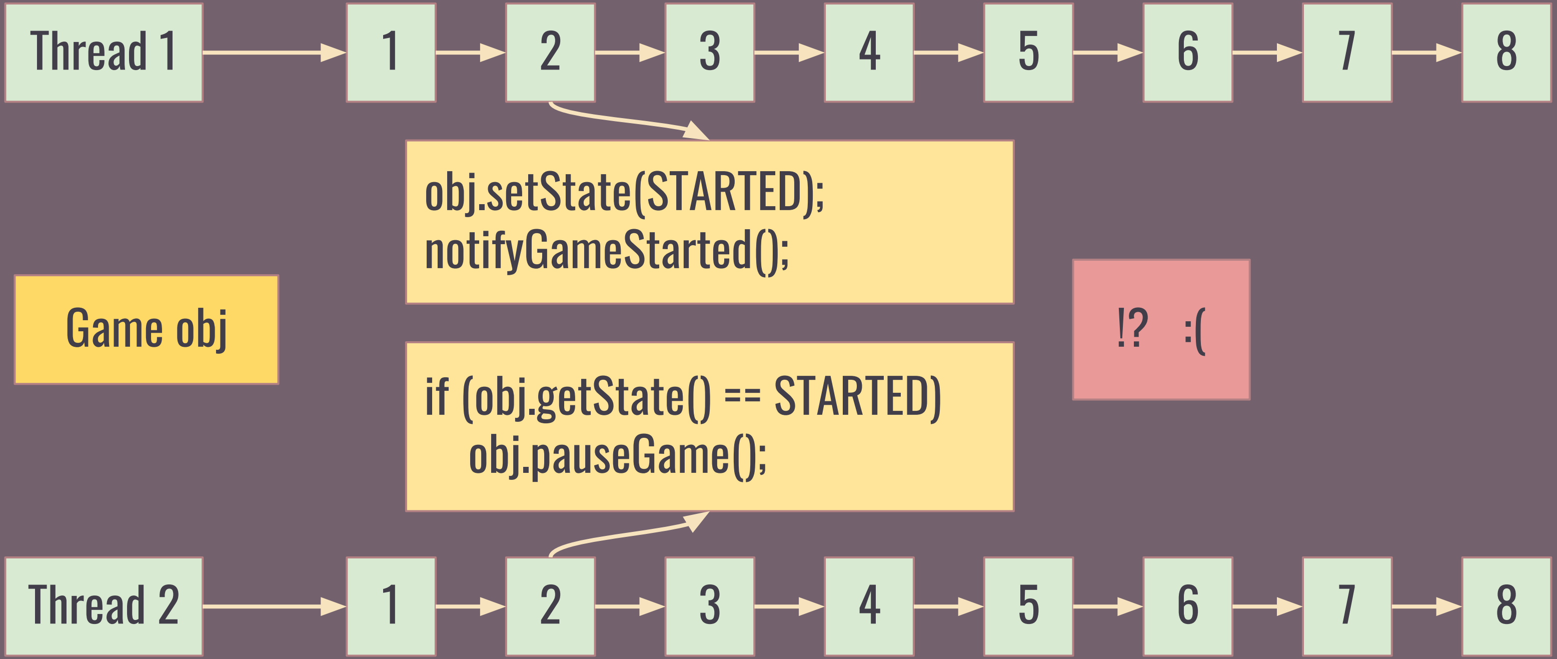
Парадигма процесса



Modify shared resource (race condition)



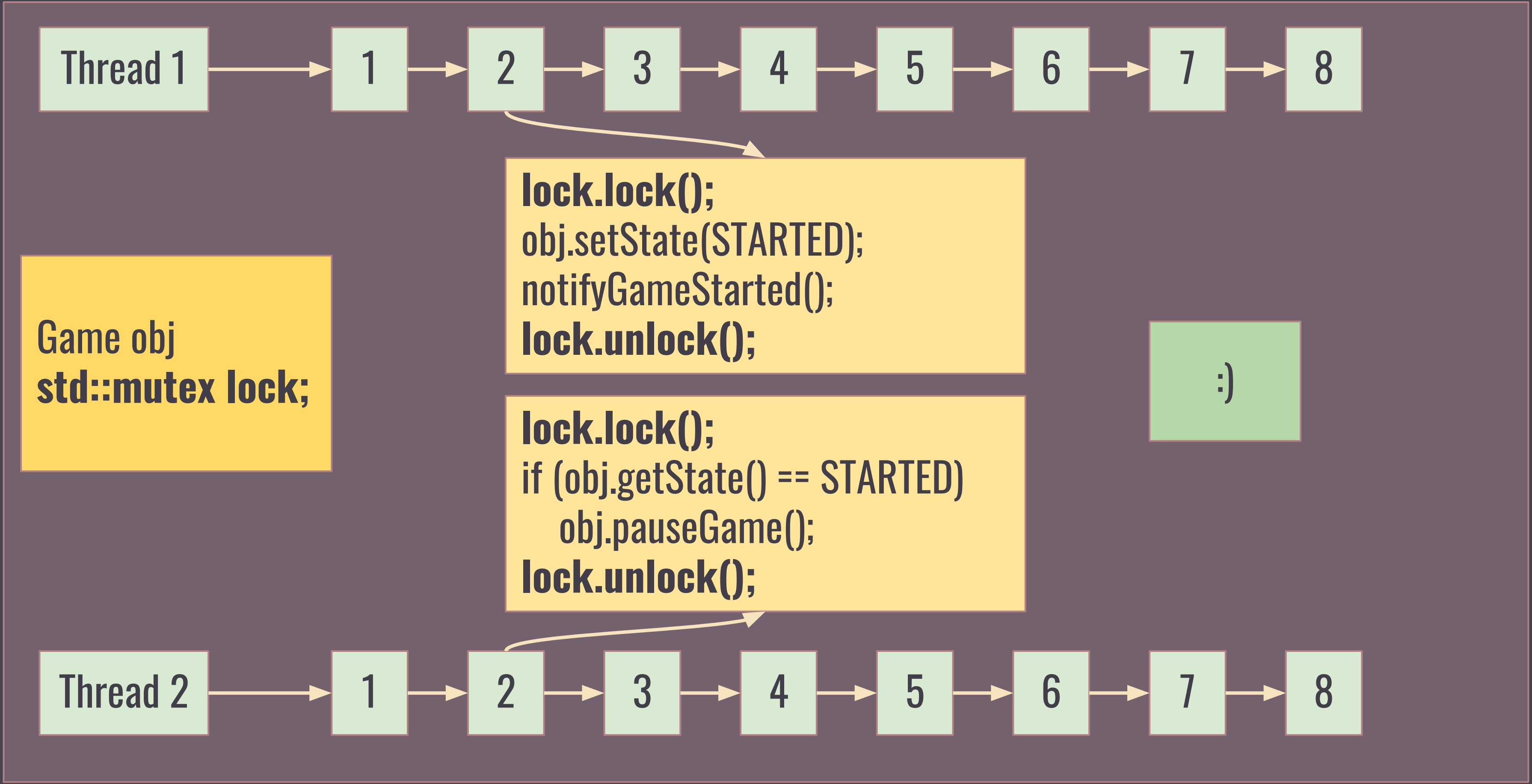
Modify shared resource (memory consistency issue)



```
obj.setState(STARTED);
notifyGameStarted();
```

```
if (obj.getState() == STARTED)
    obj.pauseGame();
```

Проблемы многопоточности: синхронизация



Thread 1 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

```
lock.lock();
obj.setState(STARTED);
notifyGameStarted();
lock.unlock();
```

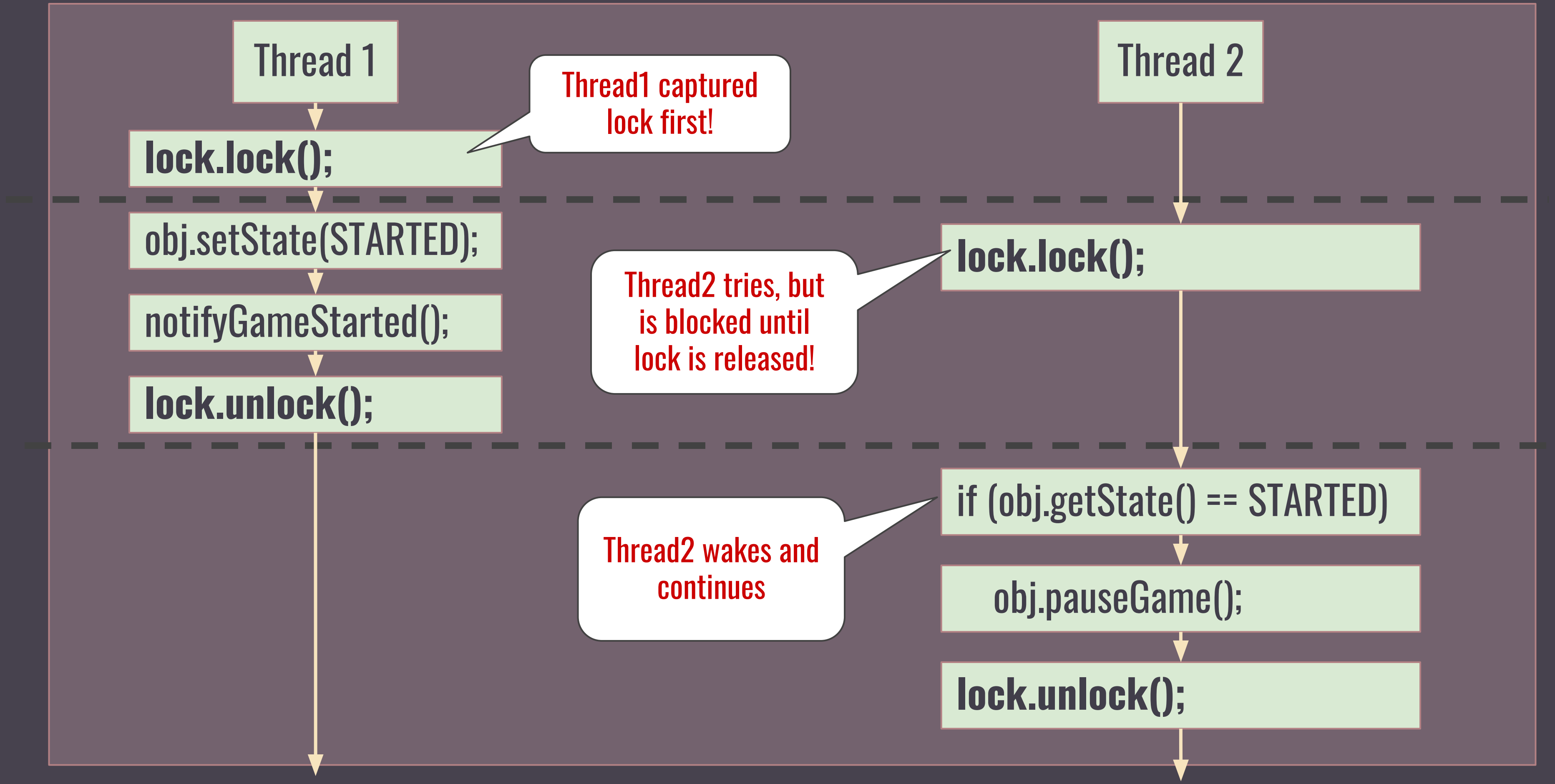
Game obj
std::mutex lock;

```
lock.lock();
if (obj.getState() == STARTED)
    obj.pauseGame();
lock.unlock();
```

:)

Thread 2 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

Проблемы многопоточности: синхронизация



Thread 1

```
lock.lock();
obj.setState(STARTED);
notifyGameStarted();
lock.unlock();
```

Thread1 captured lock first!

Thread 2

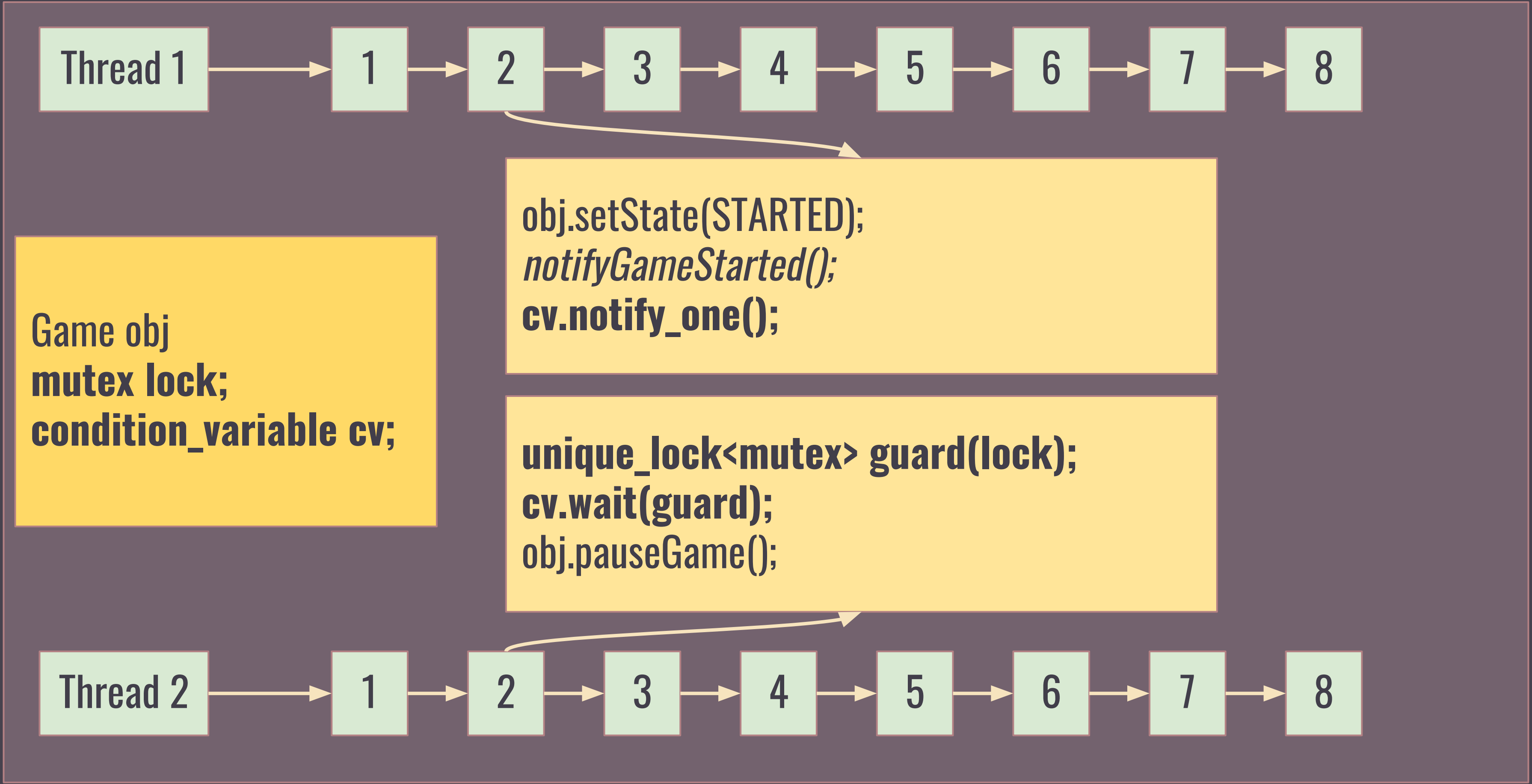
```
lock.lock();
```

Thread2 tries, but is blocked until lock is released!

```
if (obj.getState() == STARTED)
    obj.pauseGame();
lock.unlock();
```

Thread2 wakes and continues

Проблемы многопоточности: синхронизация

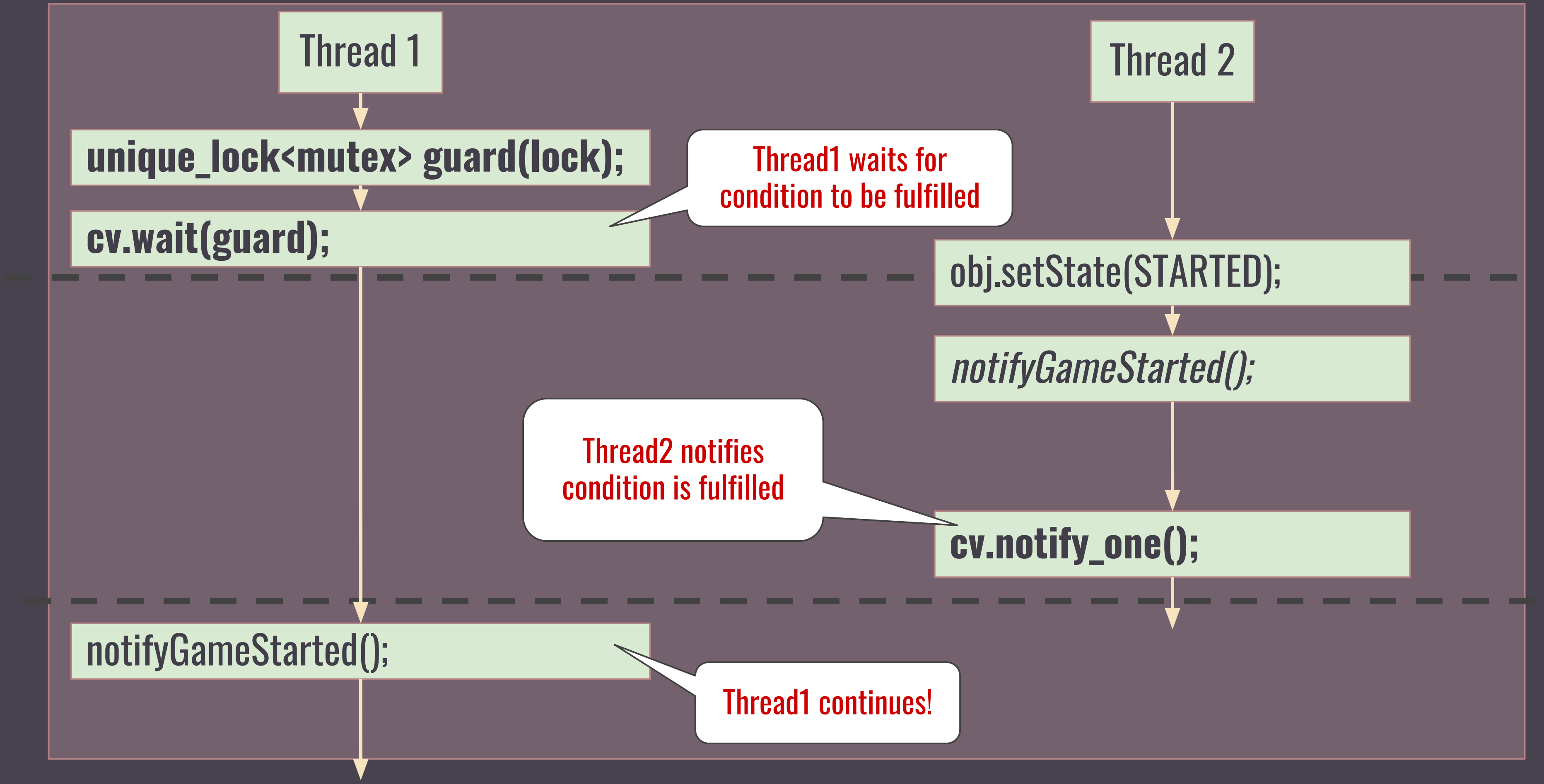


```
Game obj
mutex lock;
condition_variable cv;
```

```
obj.setState(STARTED);
notifyGameStarted();
cv.notify_one();
```

```
unique_lock<mutex> guard(lock);
cv.wait(guard);
obj.pauseGame();
```

Проблемы многопоточности: синхронизация



Thread 1

unique_lock<mutex> guard(lock);

cv.wait(guard);

Thread1 waits for condition to be fulfilled

notifyGameStarted();

Thread1 continues!

Thread 2

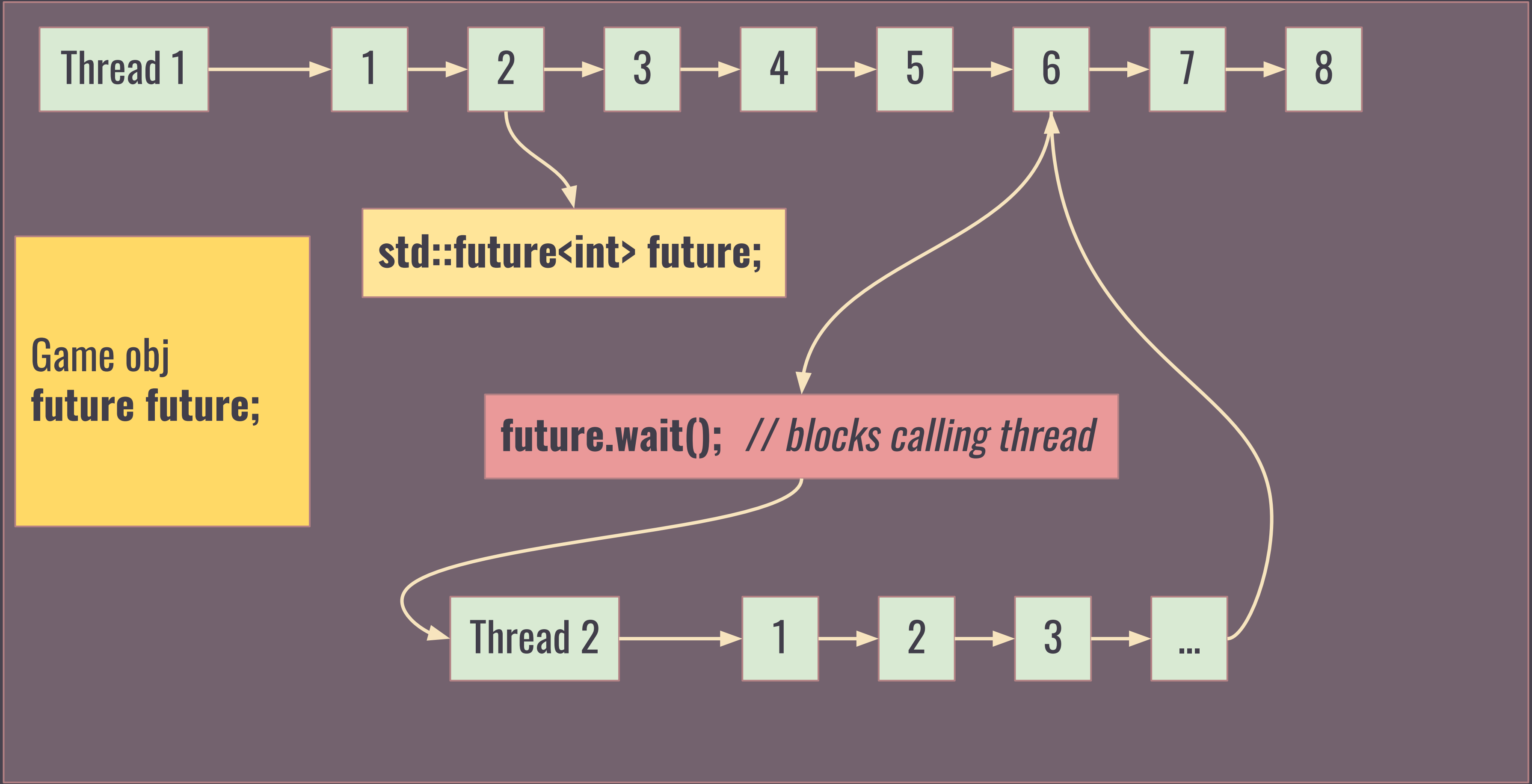
obj.setState(STARTED);

notifyGameStarted();

Thread2 notifies condition is fulfilled

cv.notify_one();

Проблемы многопоточности: синхронизация



Thread 1

1

2

3

4

5

6

7

8

std::future<int> future;

Game obj

future future;

future.wait(); *// blocks calling thread*

Thread 2

1

2

3

...

Future

```
// future from a packaged_task
std::packaged_task<int()> task( [] {                    // create packaged task
                                    return 7;
                                 } );
std::future<int> future = task.get_future();            // get a future
std::thread t(std::move(task));                         // launch on a thread

std::cout << "Waiting..." << std::flush;

future .wait();

std::cout << "Future result is available: " << future.get() << std::endl;

t.join();
```

Future

```
// future from an async()
std::future<int> future = std::async( std::launch::async,
                                      [] {
                                          return 8;
                                      } );

std::cout << "Waiting..." << std::flush;

future .wait();

std::cout << "Future result is available: " << future.get() << std::endl;
```

Future

```
void initiazer(std::promise<int>* promise)
{
    std::cout << "Async operation in thread running..." << std::endl;
    promise->set_value(35);
}

 ...

    // future from promise
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    std::thread th(initiazer, &promise);
    std::cout << future.get() << std::endl;
    th.join();
```