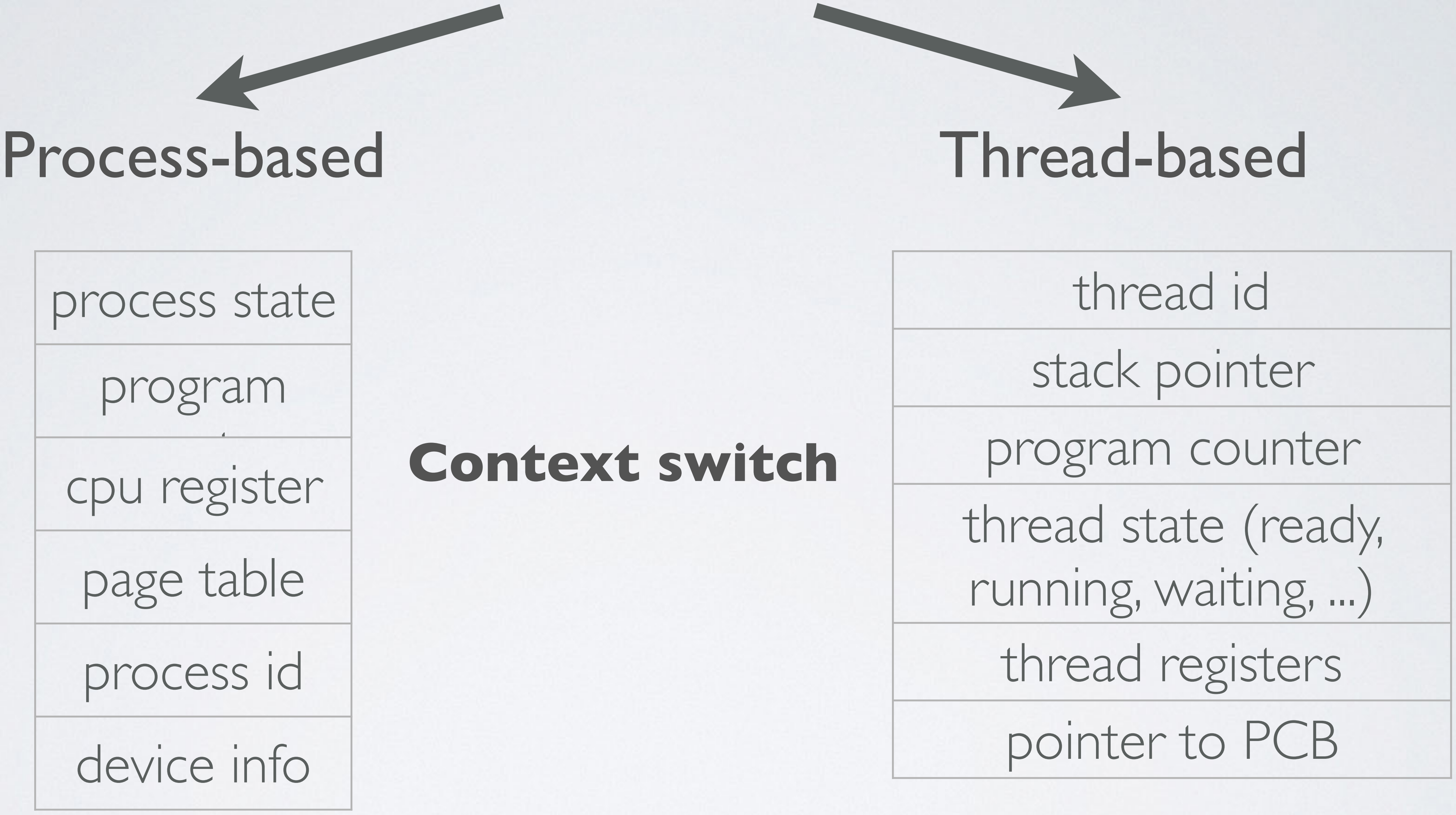
ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

MULTITASKING



Process-based

Thread-based

process state
program
cpu register
page table
process id
device info

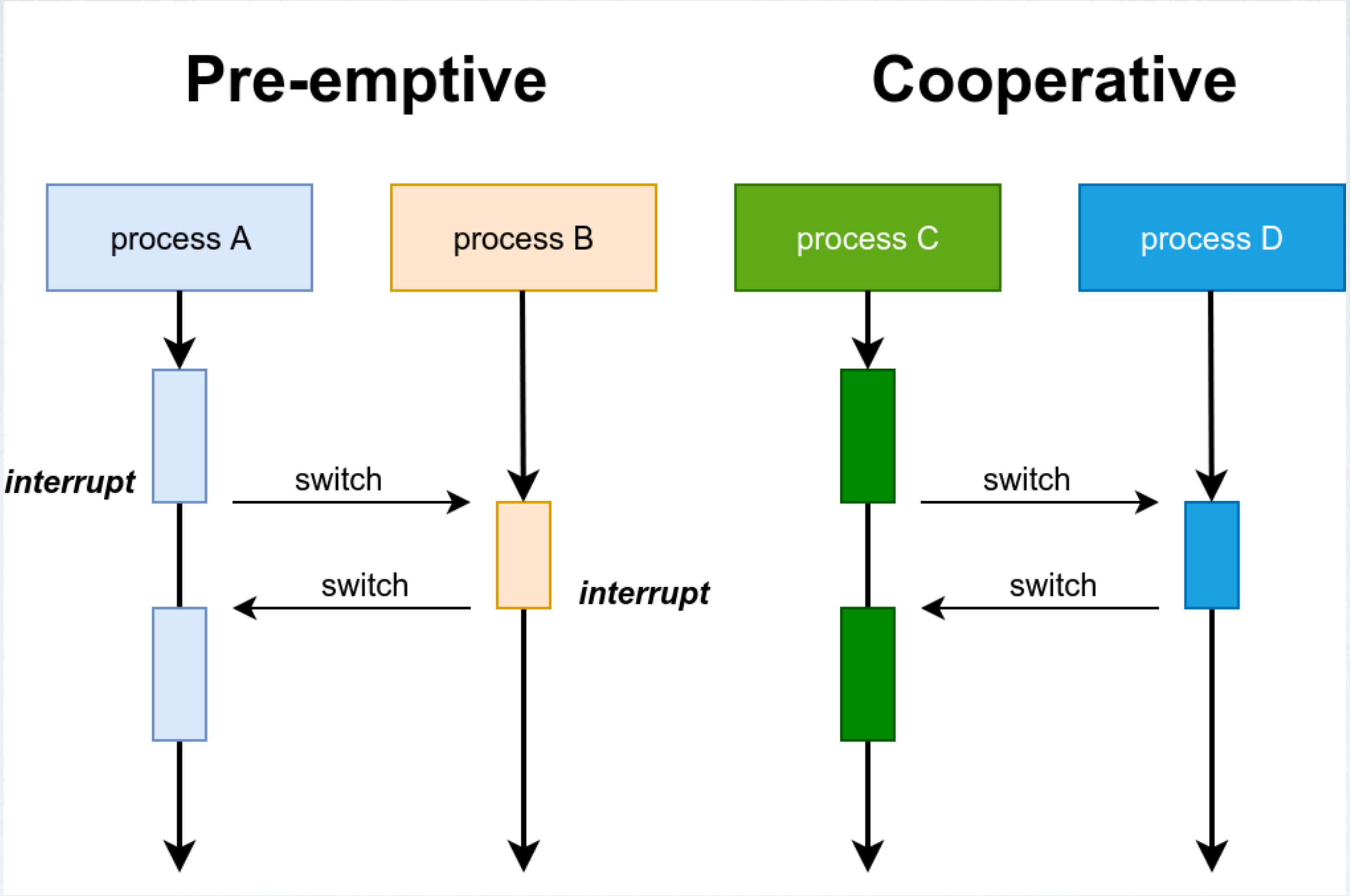
Context switch

thread id
stack pointer
program counter
thread state (ready, running, waiting, ...)
thread registers
pointer to PCB

Process Control Block (PCB)

Thread Control Block (TCB)

MULTITASKING



MULTITASKING

- Parallel.
- Asynchronous.
- Multithreading.

THREAD

```
thread() noexcept;

thread( thread&& other ) noexcept;

template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );

thread( const thread& ) = delete;
```

THREAD

```
thread() noexcept;

thread( thread&& other ) noexcept;

template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );

thread( const thread& ) = delete;



void do_some_work(){...}
...
std::thread thread_variable(do_some_work);
```

THREAD

```
bool thread::joinable() const noexcept;

void thread::join();    // sync
void thread::detach();  // async
```

```
int main()
{

   std::thread thread_variable([]{ std::this_thread::sleep_for(13ms);});

} // ~thread calls std::terminate() if joinable() == true
```

THREAD

```
struct func
{
    int& i;
    func(int& i): i(i){}

    void operator()()
    {
        do_something(i);
    }
};

void oops()
{
    int local_variable = 0;
    func func_variable{local_variable};

    std::thread thread{func_variable};
    do_something_in_current_thread();
    thread.detach();
}
```

THREAD

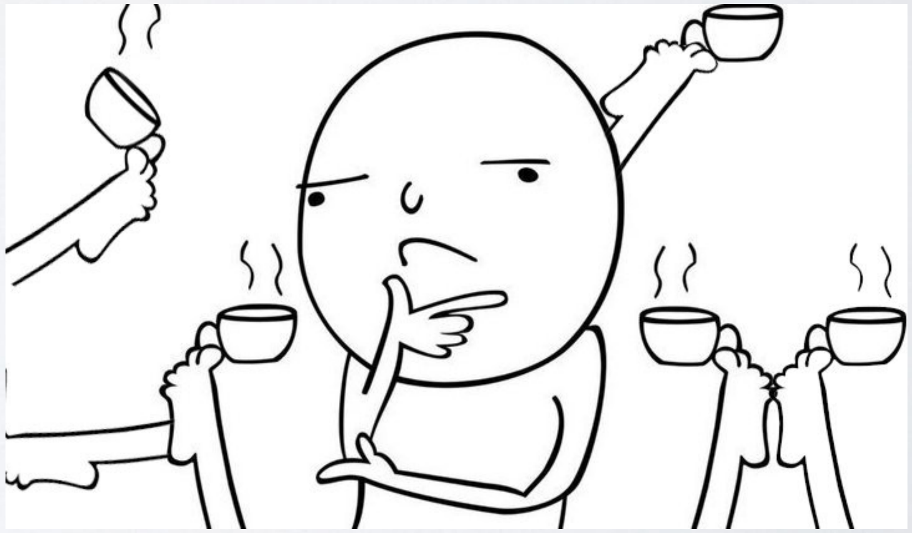
```
struct func
{
    int& i;
    func(int& i): i(i){}

    void operator()()
    {
        do_something(i);
    }
};

void oops()
{
    int local_variable = 0;
    func func_variable{local_variable};

    std::thread thread{func_variable};
    do_something_in_current_thread();
    thread.join();
}
```

Ok???

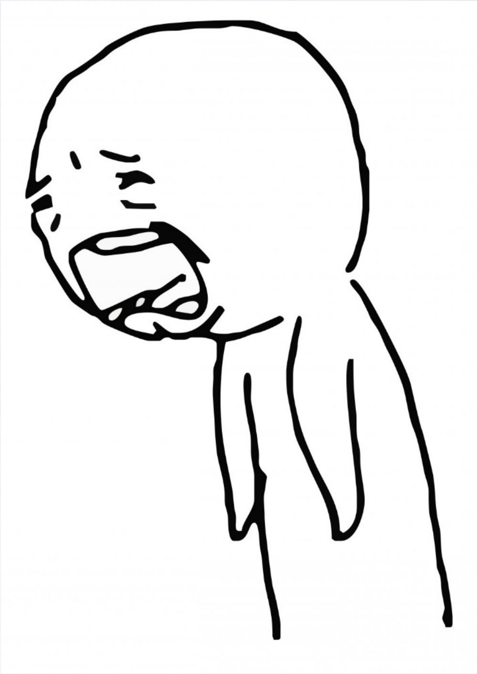


THREAD

```
struct func{...};

void oops()
{
   int local_variable = 0;
   func func_variable{local_variable};

   std::thread thread{func_variable};
   try
   {
      do_something_in_current_thread();
   }
   catch(...)
   {
      thread.join();
      throw;
   }
   thread.join();
}
```



THREAD

```
class thread_guard
{
   std::thread& t;
public:
   explicit thread_guard(std::thread& t): t(t){}
   ~thread_guard()
   {
      if(t.joinable())
      {
         t.join();
      }
   }
};
```

```
struct func{...};

void oops()
{
   int local_variable = 0;
   func func_variable{local_variable};

   std::thread thread{func_variable};
   thread_guard guard{thread};
   do_something_in_current_thread();
}
```

THREAD

```
class thread_guard
{
   std::thread& t;
public:
   explicit thread_guard(std::thread& t): t(t){}
   ~thread_guard()
   {
      if(t.joinable())
      {
         t.join();
      }
   }
};
```

```
struct func{...};

void oops()
{
   int local_variable = 0;
   func func_variable{local_variable};

   std::thread thread{func_variable};
   thread_guard guard{thread};
   do_something_in_current_thread();
}
```

THREAD

```
class thread_guard
{
   std::thread& t;
public:
   explicit thread_guard(std::thread& t): t(t){}
   ~thread_guard()
   {
      if(t.joinable())
      {
         t.join();  ???
      }
   }
};
```

```
struct func{...};

void oops()
{
   int local_variable = 0;
   func func_variable{local_variable};

   std::thread thread{func_variable};
   thread_guard guard{thread};
   do_something_in_current_thread();
}
```

JTHREAD

```
int main()
{
    std::jthread sleepy_worker([](std::stop_token token) {
        for(int i = 10; i; --i)
        {
            std::this_thread::sleep_for(300ms);
            if(token.stop_requested())
            {
                std::cout << "Sleepy worker is requested to stop\n";
                return;
            }
            std::cout << "Sleepy worker goes back to sleep\n";
        }
    });

    do_something_in_current();
    ...
    sleepy_worker.join();
}
//~jthread: if joinable() == true, calls request_stop() and then join();
```

PASS PARAMETERS TO THREAD

```
void update_data_for_widget(widget_id id, widget_data& data);

void oops_again(widget_id id)
{
   widget_data data;
   std::thread t(update_data_for_widget, id, data);
   display_status();
   t.join();
   process_widget_data(data);
}
```

PASS PARAMETERS TO THREAD

```
void update_data_for_widget(widget_id id, widget_data& data);

void oops_again(widget_id id)
{
   widget_data data;
   std::thread t(update_data_for_widget, id, data);
   display_status();
   t.join();
   process_widget_data(data);
}
```

Not Compiled!

PASS PARAMETERS TO THREAD

```
void update_data_for_widget(widget_id id, widget_data& data);

void not_oops_again(widget_id id)
{
   widget_data data;
   std::thread t(update_data_for_widget, id, std::ref(data));
   display_status();
   t.join();
   process_widget_data(data);
}
```

Compiled!

EXCEPTION INSIDE THREAD

```
int main()
{
   std::thread t([]{
      do_something();
      ...
      throw std::runtime_error();
      ...
   });

   t.join();
}
```

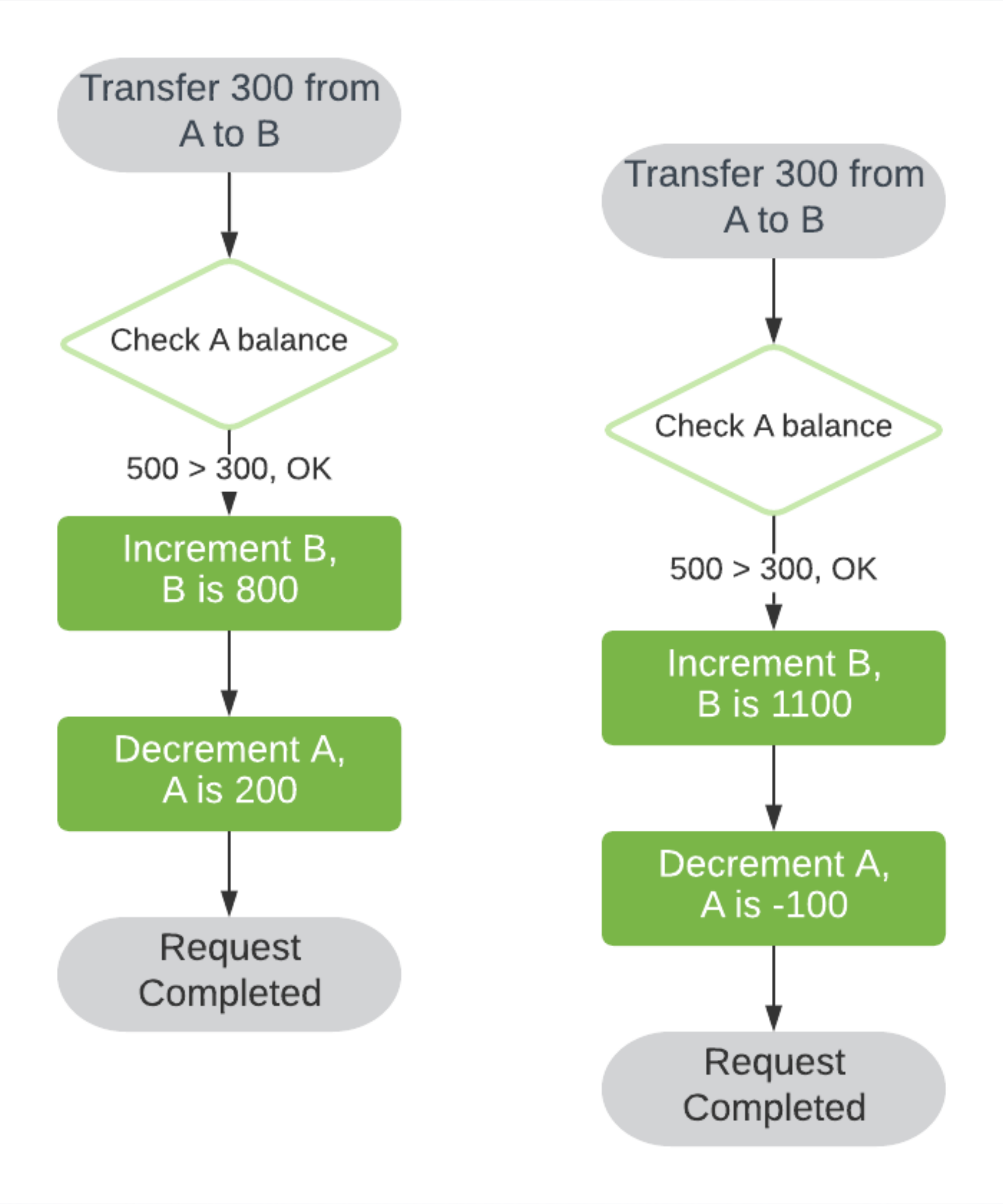
EXCEPTION INSIDE THREAD

```
int main()
{
   std::thread t([]{
      do_something();
      ...
      throw std::runtime_error();
      ...
   });

   t.join();
}
```

The return value of the top-level function is ignored and if it terminates by throwing an exception, **std::terminate** is called.

RACE CONDITIONS



RACE CONDITIONS

- Lock-free programming
- Synchronization primitives
- Software transactional memory

MUTUAL EXCLUSION (MUTEX)

```
std::mutex mutex;

mutex.lock();
//shared object modifications
...
mutex.unlock();

//RAII
std::lock_guard     //since C++11
std::scoped_lock    //since C++17
```

MUTUAL EXCLUSION (MUTEX)

```
std::list<int> some_list;
std::mutex mutex;

void add_to_list(int new_value)
{
   std::scoped_lock guard(mutex);
   some_list.push_back(new_value);
}

bool list_contains(int value_to_find)
{
   std::scoped_lock guard(mutex);
   return std::ranges::find(list, value_to_find) != some_list.end();
}
```

MUTUAL EXCLUSION (MUTEX)

```
class data_wrapper
{
   std::list<int> some_list;
   std::mutex mutex;

public:
   template <typename Function>
   void process_data(Function func)
   {
      std::scoped_lock guard(mutex);
      func(some_list);
   }
}
```

???

MUTUAL EXCLUSION (MUTEX)

```
class data_wrapper
{
   std::list<int> some_list;
   std::mutex mutex;

public:
   template <typename Function>
   void process_data(Function func)
   {
      std::scoped_lock guard(mutex);
      func(some_list);
   }
}
```

Bad code!

Don't pass pointers and references of protected data to outside lock

MUTUAL EXCLUSION (MUTEX)

```
threadsafe_stack<int> s; //usual stack with mutex inside

if(!s.empty())
{
   const int value = s.top();
   s.pop();
   do_something(value);
}
```

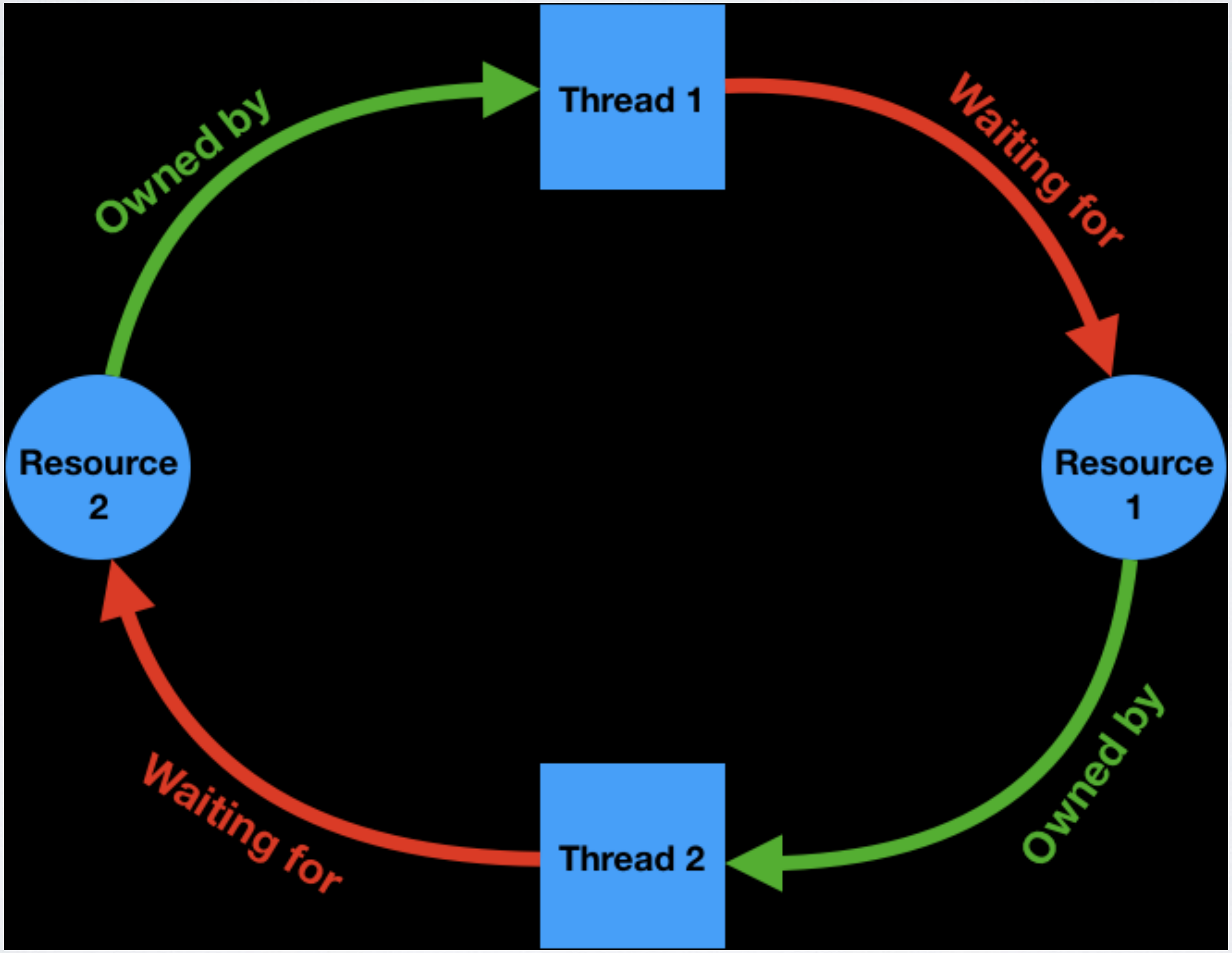
This code is not thread safe

MUTUAL EXCLUSION (MUTEX)

```
template <typename T>
class threadsafe_stack
{
public:
   threadsafe_stack();
   threadsafe_stack(const threadsafe_stack&);
   threadsafe_stack& operator=(const threadsafe_stack&) = delete;
   void push(T new_value);
   std::shared_ptr<T> pop();
   void pop(T& value);
   bool empty() const;
}
```

Now it is thread safe

DEADLOCK


Thread 1
Waiting for
Resource 1
Owned by
Thread 2
Waiting for
Resource 2
Owned by

DEADLOCK

```
template <typename T>
class Sample
{
   some_big_object value;
   std::mutex mutex;
public:
   ...
   friend void swap(Sample& lhs, Sample& rhs)
   {
      if(&lhs == &rhs)
      {
         return;
      }

      std::lock_guard lock_first(lhs.mutex);
      std::lock_guard lock_second(rhs.mutex);
      swap(lhs.value, rhs.value);
   }
};
```

DEADLOCK

```
template <typename T>
class Sample
{
   some_big_object value;
   std::mutex mutex;
public:
   ...
   friend void swap(Sample& lhs, Sample& rhs)
   {
      if(&lhs == &rhs)
      {
         return;
      }

      std::lock(lhs.mutex, rhs.mutex);
      std::lock_guard lock_first(lhs.mutex, std::adopt_lock);
      std::lock_guard lock_second(rhs.mutex, std::adopt_lock);
      swap(lhs.value, rhs.value);
   }
};
```

DEADLOCK

```cpp
template <typename T>
class Sample
{
   some_big_object value;
   std::mutex mutex;
public:
   ...
   friend void swap(Sample& lhs, Sample& rhs)
   {
      if(&lhs == &rhs)
      {
         return;
      }

      std::scoped_lock lock(lhs.mutex, rhs.mutex);
      swap(lhs.value, rhs.value);
   }
};
```

DEADLOCK

- Avoid nested locks.
- While locks, avoid user code.
- Set locks in fixed order.

RECURSIVE LOCK

```
template <typename T>
class Sample
{
    std::string shared;
    std::recursive_mutex mutex;
public:
    void func1()
    {
        std::lock_guard guard(mutex);
        shared = "fun1";
        std::cout << "in fun1, shared variable is now " << shared << '\n';
    }

    void func2()
    {
        std::lock_guard guard(mutex);
        shared = "fun2";
        std::cout << "in fun2, shared variable is now " << shared << '\n';
        func1();
        std::cout << "back in fun2, shared variable is " << shared << '\n';
    }
};
```

DEFER AND TRANSFER LOCK

```
struct Box
{
    explicit Box(int num) : num_things{num} {}

    int num_things;
    std::mutex mutex;
};

void func(Box &from, Box &to, int num)
{
    // don't actually take the locks yet
    std::unique_lock lock1{from.mutex, std::defer_lock};
    std::unique_lock lock2{to.mutex, std::defer_lock};

    // lock both unique_locks without deadlock
    std::lock(lock1, lock2);

    from.num_things -= num;
    to.num_things += num;

    // 'from.m' and 'to.m' mutexes unlocked in 'unique_lock' dtors
}
```

DEFER AND TRANSFER LOCK

```
std::unique_lock<std::mutex> get_lock()
{
    extern std::mutex mutex;
    std::unique_lock lock(mutex);
    prepare_data();
    return lock;
}

void process_data()
{
    std::unique_lock lock(get_lock);
    do_something();
}
```

Я: *указываю в резюме навык "многозадачность" *
Моя многозадачность: