ОНЛАЙН-
ОБРАЗОВАНИЕ

OTUS

Разработчик C++

Базовый курс

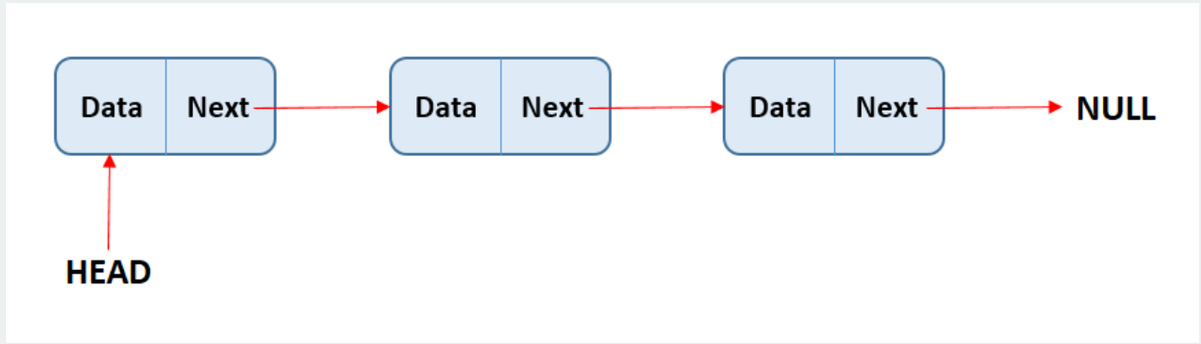
Структуры данных в многопоточной среде

Сергей Кольцов
профессиональный программист

Что сегодня?

- проблемы многопоточности
- наивный способ решения
- что можно улучшить
- паттерн Monitor object
- condition_variable

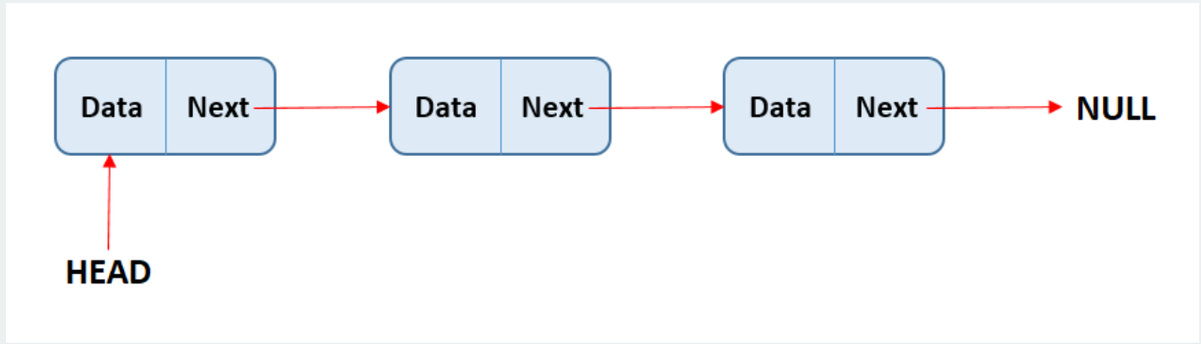
Проблемы?




```
1.  template <typename T>
2.  struct List {
3.      // public interface
4.  private:
5.      struct Node {
6.          T data;
7.          std::unique_ptr<Node> next;
8.          Node *prev = nullptr; // prev pointer is not owner
9.      };
10.     std::unique_ptr<Node> m_first = nullptr;
11.     Node *m_last = nullptr;
12.     size_t m_size = 0;
13. };
```

1 **Простой пример на структуре данных однонаправленный список.**

Проблемы?




```
1.  template <typename T>
2.  struct List {
3.      // public interface
4.  private:
5.      struct Node {
6.          T data;
7.          std::unique_ptr<Node> next;
8.          Node *prev = nullptr; // prev pointer is not owner
9.      };
10.     std::unique_ptr<Node> m_first = nullptr;
11.     Node *m_last = nullptr;
12.     size_t m_size = 0;
13. };
```

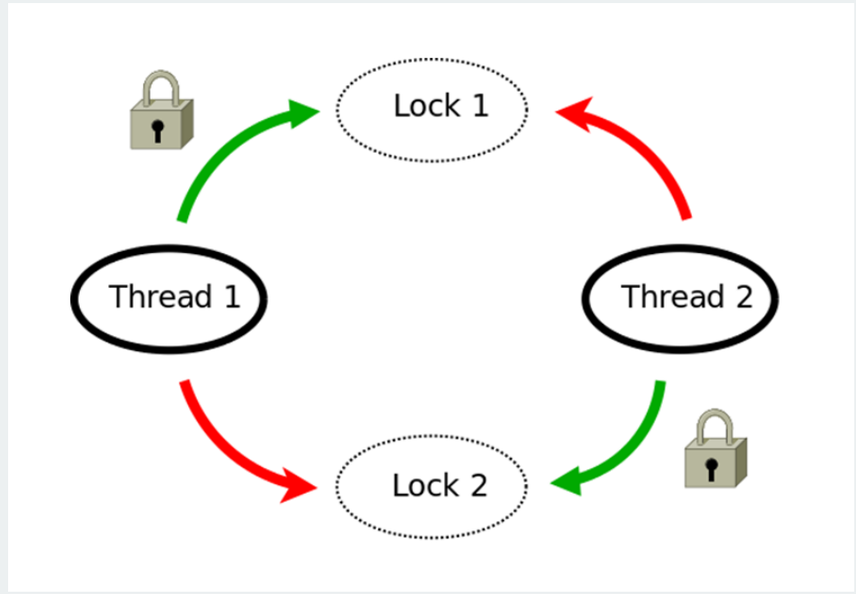
Мемберы класса

Любое многопоточное обращение на чтение/запись приведёт к проблемам.

Простой пример на структуре данных однонаправленный список.

Решение!

```
1.  template <typename T>
2.  struct List {
3.      // public interface
4.  private:
5.      struct Node {
6.          T data;
7.          std::unique_ptr<Node> next;
8.          Node *prev = nullptr; // prev pointer is not owner
9.      };
10.     std::unique_ptr<Node> m_first = nullptr;
11.     Node *m_last = nullptr;
12.     size_t m_size = 0;
13.     mutable std::recursive_mutex m_mutex;
14. };
```



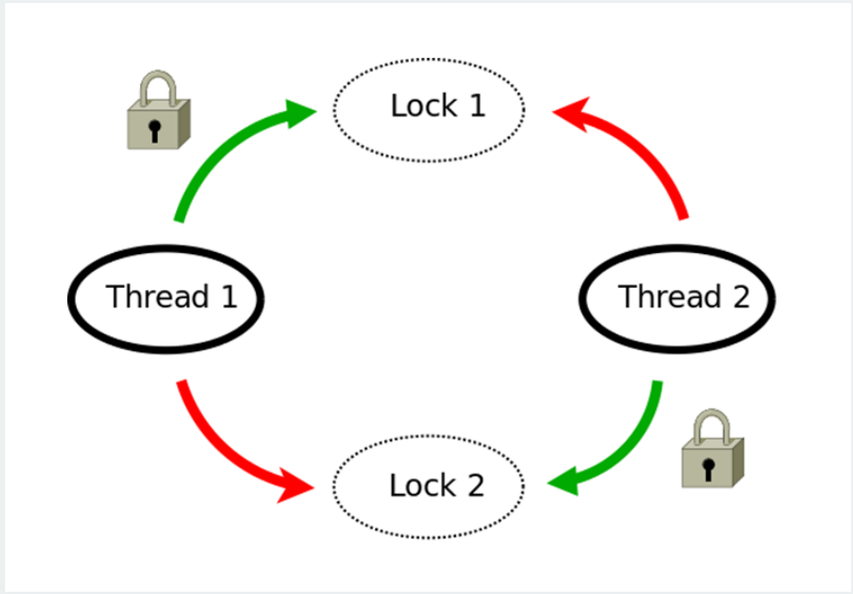
Добавим мьютекс

Можно ли решить проблему мьютексом? Конечно!

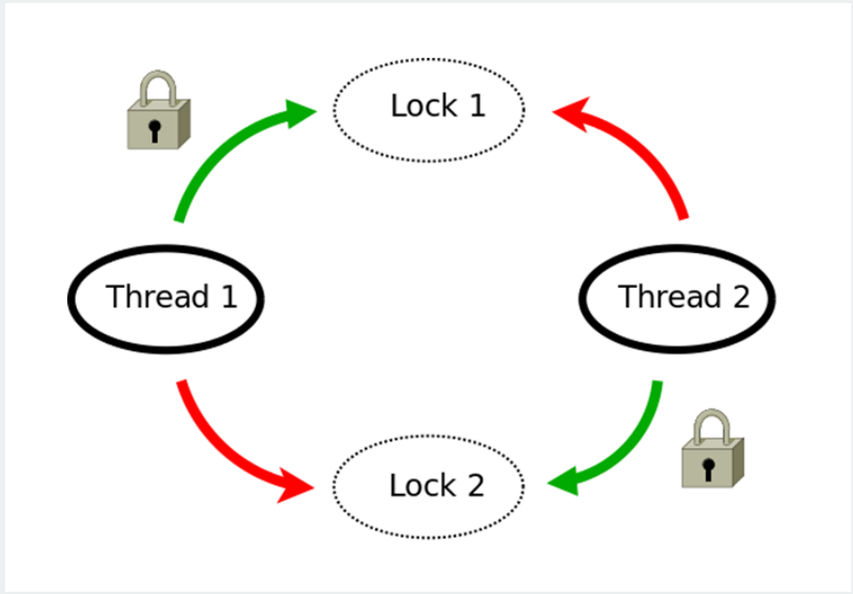
Решение!

Добавив мьютекс и его захват во все публичные методы:

- решим проблему
- получим потобезопасную реализацию
- и всё будет вполне себе хорошо
- но медленнее, чем хотелось бы
- особенно в однопоточном приложении
- и необходимость менять реализацию раздражает



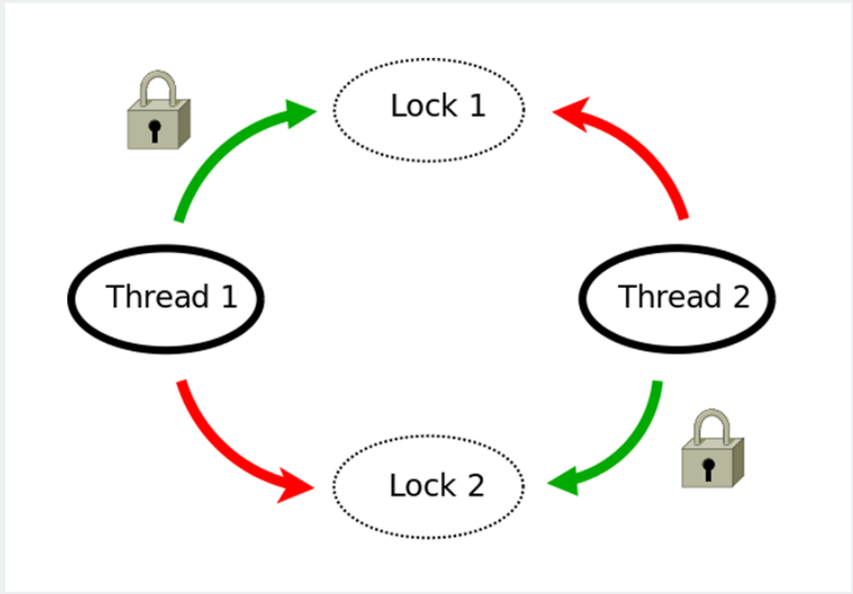
Улучшим?



Можно немного улучшить:

- обратим внимание на то, что:
- часть методов не работают со всем списком
  - при вставке в голову списка мы работаем только с головой
  - при вставке в конец - только с концом списка
  - можно разделить единую блокировку на две
  - первый мьютекс будет ответственен за голову списка
  - второй - за конец списка

Улучшим?



Можно немного улучшить:

- обратим внимание на то, что:
- часть методов не работают со всем списком
  - при вставке в голову списка мы работаем только с головой
  - при вставке в конец - только с концом списка
  - можно разделить единую блокировку на две
  - первый мьютекс будет ответственен за голову списка
  - второй - за конец списка

Но реализация будет сложнее

Ещё варианты?

Паттерн Монитор (Monitor Object)

Варианты есть всегда. Человечество изобрело множество.

Ещё варианты?

Паттерн Монитор (Monitor Object)

Варианты есть всегда. Человечество изобрело множество.

Ещё варианты?

Паттерн Монитор (Monitor Object)

Нет, не такой монитор ☺

Варианты есть всегда. Человечество изобрело множество.

Ещё варианты?

Паттерн Монитор (Monitor Object)

```
1.  template <class T>
2.  class monitor {
3.  public:
4.      template <typename... Args>
5.      monitor(Args &&...args) :
6.          m_data{std::forward<Args>(args)...}
7.      {}
8.      helper operator->() { return helper(this); }
9.  private:
10.     T m_data;
11.     std::mutex m_mutex;
12. };
```


Варианты есть всегда. Человечество изобрело множество.

Ещё варианты?

Паттерн Монитор (Monitor Object)

```
1.  template <class T>
2.  class monitor {
3.  public:
4.      template <typename... Args>
5.      monitor(Args &&...args) :
6.          m_data{std::forward<Args>(args)...}
7.      {}
8.      helper operator->() { return helper(this); }
9.  private:
10.     T m_data;
11.     std::mutex m_mutex;
12. };
```

Вся красота в классе `helper`

Варианты есть всегда. Человечество изобрело множество.

Ещё варианты?

Паттерн Монитор (Monitor Object)

```
1.  struct helper {
2.      helper(monitor *mon)
3.          : m_parent{mon}
4.          , m_lock{mon->m_mutex}
5.      {}
6.
7.      T *operator->() {
8.          return &m_parent->m_data;
9.      }
10. private:
11.     monitor *m_parent;
12.     std::lock_guard<std::mutex> m_lock;
13. };
```

А вот и красота

Варианты есть всегда. Человечество изобрело множество.

Ещё варианты?

Паттерн Монитор (Monitor Object)

Используем так:

```
1. monitor<std::vector<int>> thread_safe_vector{1, 2, 3, 4, 5};
2. auto value = thread_safe_vector->data()[3];
3. auto size = thread_safe_vector->size();
4. auto iter = thread_safe_vector->cbegin();
```


Варианты есть всегда. Человечество изобрело множество.

Ещё варианты?

Паттерн Монитор (Monitor Object)

Используем так:

```
1. monitor<std::vector<int>> thread_safe_vector{1, 2, 3, 4, 5};
2. auto value = thread_safe_vector->data()[3];
3. auto size = thread_safe_vector->size();
4. auto iter = thread_safe_vector->cbegin();
```

Каждая операция -> автоматически блокирует mutex

condition_variable

Суть проблемы

- есть ряд потоков
- некоторые помещают в структуру данные на обработку
- другие читают из структуры данные и обрабатывают
- назовём первую группу потоков производителями (Producer)
- вторую группу - потребителями (Consumer)
- вопрос - что делать потребителям, когда в структуре ещё нет данных?

condition_variable

Ответ - ждать!

Ожидание - хоть и скучно, но порой необходимо.

condition_variable

Но ждать можно по разному:

- обычный sleep_for
- активное ожидание (while(list.empty())
- ждать события «данные появились»

condition_variable

Но ждать можно по разному:

- обычный sleep_for
- активное ожидание (while(list.empty())
- ждать события «данные появились»

condition_variable - это про последний вариант

condition_variable

Поток-потребитель:

```
1. void consumer_thread() {
2.     std::unique_lock<std::mutex> lck{mutex};
3.     condition_variable.wait(lck);
4.     // wakeup here
5. }
```


Ожидание - хоть и скучно, но порой необходимо.

condition_variable

Поток-потребитель:

```
1. void consumer_thread() {
2.     std::unique_lock<std::mutex> lck{mutex};
3.     condition_variable.wait(lck);
4.     // wakeup here
5. }
```

Ждём тут

Работает на основе mutex-а

Который нужно предварительно захватить

Ожидание - хоть и скучно, но порой необходимо.

condition_variable

Поток-производитель:

```
1. void producer_thread() {
2.      // add data to list/queue/etc...
3.      // notify one waiting thread
4.      condition_variable.notify_one();
5. }
```


Ожидание - хоть и скучно, но порой необходимо.

condition_variable

Поток-производитель:

```
1. void producer_thread() {
2.      // add data to list/queue/etc...
3.      // notify one waiting thread
4.      condition_variable.notify_one();
5. }
```

Добавляем данные

Посылаем нотификацию одному потоку-потребителю

Спасибо
за внимание!

Заполните, пожалуйста
опрос о занятии.

Ответы на вопросы