КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 26 — Условные переменные

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2021

Оглавление

Переменные состояния (conditional variables)

Совокупность объекта синхронизации, предиката и мьютекса.

Типичный шаблон использования переменных состояния:

```
// безопасно проверим состояние/условие предиката SOME-CONDITION,
// препятствуя другим потокам его в этом момент изменить.
lock(mutex);
while (SOME-CONDITION is false) {
    wait(cond, mutex);          // блокируемся на переменной и освобождаем mutex
}
// делаем что нам надо, когда поток разблокируют
do_stuff();
unlock(mutex);
```

С другой стороны, поток, сигнализирующий о переменной условия, обычно выглядит как

```
// нам нужен эксклюзивный доступ, каким бы условие не было
lock(mutex);

ALTER-CONDITION

// Разбудим по крайней мере один поток из ожидающих изменения состояния
signal(cond);

// allow others to proceed
unlock(mutex)
```

Мьютекс используется для защиты предиката от изменения со стороны других потоков на время проверки. Поэтому прежде чем ждать, его необходимо заблокировать.

Переменные состояния следует использовать как место ожидания и получения уведомлений посредством предиката. Переменные состояния — это не само состояние и не событие. Условие содержится в окружающей логике программирования.

Ожидание «атомарно» разблокирует мьютекс, позволяя другим потокам получить доступ к переменной состояния, например, для сигнализации.

Затем, когда переменная состояния передается (signalled) или транслируется (broadcast to), один или несколько потоков из списка ожидания будут разбужены, а мьютекс волшебным образом будет заблокирован для этого разбуженного потока.

`wait` — атомарно освобождается мьютекс и поток уходит ждать.
`signal` — атомарно разблокирует поток возвращает ему мьютекс

<condition_variable> — типы условных переменных

Классы

`condition_variable` — класс условных переменных
`condition_variable_any` — условная переменная по любой блокировке

Классы перечислений

`cv_status` — состояние условной переменной

Функции

`notify_all_at_thread_exit()` — уведомить всех при выходе потока

std::condition_variable — класс условных переменных

```
class condition_variable;
```

Переменная условия — это объект, способный блокировать вызывающий поток до получения им уведомления о возобновлении.

Переменная условия использует **unique_lock** над мьютексом для блокировки потока при вызове одной из его функций ожидания.

Поток остается заблокированным до тех пор, пока его не разбудит другой поток, который вызывает функцию уведомления для того же объекта типа **condition_variable**.

Объекты типа **condition_variable** всегда используют **unique_lock <mutex>** для ожидания. С любым блокируемым типом работает **condition_variable_any**.

Открытые функции члены

`(constructor)` — создает переменную условия
`(destructor)` — разрушает переменную условия

Функции ожидания
`wait()` — ожидать уведомления
`wait_for()` — ожидать уведомления до истечения таймаута
`wait_until()` — ожидать уведомления до момента времени

Функции уведомления (сигнализирующие функции)
`notify_one()` — уведомить одного
`notify_all()` — уведомить всех

std::condition_variable::condition_variable — конструктор

(1) по умолчанию

```
condition_variable( );
```

(2) копирования [deleted]

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

Объекты типа **condition_variable** нельзя ни копировать, ни перемещать (для этого типа удаляются и конструктор копирования, и оператор присваивания).

std::condition_variable::~condition_variable — деструктор

```
~condition_variable( );
```

Любой поток, который был заблокирован по этой переменной, должен быть уведомлен перед вызовом деструктора. Ни один поток не должен начинать ожидание после вызова этого деструктора.

Пример condition_variable

```
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;               // мьютекс
std::condition_variable cv;   // переменная состояния (условия)
bool ready = false;           // предикат

void print_id(int id) {
    std::unique_lock<std::mutex> lck(mtx);
    while (!ready) {
        cv.wait(lck); // возможно "ложное пробуждение"
    }
    // ...
    std::cout << "thread " << id << '\n';
}

void go() {
    std::unique_lock<std::mutex> lck(mtx);
    ready = true;
    cv.notify_all();
}
```

```
int main () {

  std::thread threads[10];
  // spawn 10 threads:
  for (int i = 0; i < 10; ++i)
    threads[i] = std::thread(print_id, i);

  std::cout << "10 threads ready to race...\n";
  go();                       // go!

  for (auto& th : threads) th.join();

  return 0;
}
```

Вывод

```
10 threads ready to race...
thread 2
thread 0
thread 9
thread 4
thread 6
thread 8
thread 7
thread 5
thread 3
thread 1
```

std::condition_variable::wait — ожидать, пока не уведомят

(1) безусловная

```
void wait(unique_lock<mutex>& lck);
```

(2) предикативная

```
template <class Predicate>
void wait(unique_lock<mutex>& lck, Predicate pred);
```

Выполнение текущего потока (который должен перед этим обязательно заблокировать мьютекс **`lck`**) блокируется до получения уведомления.

В момент блокировки потока функция автоматически вызывает **`lck.unlock()`**, позволяя другим заблокированным на этом мьютексе потокам продолжить работу.

После уведомления (явно каким-либо другим потоком) функция разблокируется и вызывает **`lck.lock()`**, оставляя **`lck`** в том же состоянии, что и при вызове функции.

Затем функция возвращает управление.

Как правило, функция уведомляется о пробуждении посредством вызова в другом потоке либо функции-члена **`notify_one()`**, либо функции-члена **`notify_all()`**. Но некоторые реализации могут вызывать ложные пробуждающие вызовы без вызова какой-либо из этих функций.

Следовательно, пользователи этой функции должны убедиться, что их условие для возобновления выполнено.

Если указана предикативная форма, функция блокируется только в том случае, если **pred** возвращает **false**, а уведомления могут разблокировать поток только тогда, когда он становится истинным (что особенно полезно для проверки на предмет ложных пробуждающих вызовов).

Эта версия ведет себя так, как если бы она была реализована как:

```
while (!pred()) {
    wait(lck);
}
```

lck — объект типа **unique_lock**, чей объект мьютекса в данный момент заблокирован этим потоком.

Все одновременные вызовы функций-членов этого объекта должны использовать один и тот же базовый объект мьютекса, возвращаемый **lck.mutex()**.

pred — вызываемый объект или функция, которая не принимает аргументов и возвращает значение, которое может быть оценено как логическое значение. Он вызывается многократно, пока не станет истинным.

Пример condition_variable::wait (с предикатом)

```
#include <iostream>           // std::cout
#include <thread>             // std::thread, std::this_thread::yield
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex              mtx;
std::condition_variable cv;

int cargo = 0;
bool shipment_available() { return cargo != 0;} // предикат

void consume(int n) {

   for (int i = 0; i < n; ++i) {
      std::unique_lock<std::mutex> lck(mtx); // мьютекс
      cv.wait(lck, shipment_available);      // ожидаем
      // consume:                            // дождались
      std::cout << cargo << '\n';
      cargo = 0;                             // сбрасываем — мы же употребили
   }
}
```

```
int main () {

  std::thread consumer_thread(consume, 10);

  // произведем 10 штук при необходимости
  for (int i = 0; i < 10; ++i) {
     // пока есть чего им взять, спим
     while (shipment_available()) {
         std::this_thread::yield();
     }
     std::unique_lock<std::mutex> lck(mtx);
     cargo = i + 1;
     cv.notify_one();
  }

  consumer_thread.join();
  return 0;
}
```

std::condition_variable::notify_one — уведомить одного

```
void notify_one() noexcept;
```

Разблокирует один из потоков, ожидающих в данный момент данного условия.
Если нет ожидающих потоков, функция ничего не делает.
Если их больше одного, не указывается, какой из потоков будет выбран.

std::condition_variable::notify_all — уведомить всех

```
void notify_all() noexcept;
```

Разблокирует все потоки, ожидающие в данный момент данного условия.
Если нет ожидающих потоков, функция ничего не делает.

Пример condition_variable::notify_one

```
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;
std::condition_variable produce, consume;
int cargo = 0;     // используется совместно производителем и потребителем

void consumer () {
  std::unique_lock<std::mutex> lck(mtx);
  while (cargo == 0)
      consume.wait(lck);
  std::cout << cargo << '\n';
  cargo = 0;
  produce.notify_one();
}

void producer(int id) {
  std::unique_lock<std::mutex> lck(mtx);
  while (cargo != 0)
      produce.wait(lck);
  cargo = id;
  consume.notify_one();
}
```

```
int main () {

  std::thread consumers[10], producers[10];
  // spawn 10 consumers and 10 producers:
  for (int i = 0; i < 10; ++i) {
    consumers[i] = std::thread(consumer);
    producers[i] = std::thread(producer, i + 1);
  }

  // join them back:
  for (int i = 0; i < 10; ++i) {
    producers[i].join();
    consumers[i].join();
  }

  return 0;
}
```

Вывод

```
1
2
...
6
7
```

Пример condition_variable::notify_all

```
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex              mtx;
std::condition_variable cv;
bool ready =            false;

void print_id(int id) {
  std::unique_lock<std::mutex> lck(mtx);
  while (!ready)
      cv.wait(lck);
  // ...
  std::cout << "thread " << id << '\n';
}

void go() {
  std::unique_lock<std::mutex> lck(mtx);
  ready = true;
  cv.notify_all();
}
```

```
int main () {

  std::thread threads[10];
  // spawn 10 threads:
  for (int i = 0; i < 10; ++i)
    threads[i] = std::thread(print_id, i);

  std::cout << "10 threads ready to race...\n";
  go();                                                    // go!

  for (auto& th : threads) th.join();

  return 0;
}
```

Вывод

```
10 threads ready to race...
thread 6
thread 2
thread 5
thread 3
thread 4
thread 1
thread 7
thread 0
thread 9
thread 8
```

std::condition_variable::wait_for — ждать уведомления в течение таймаута

(1) безусловная

```
template <class Rep, class Period>
cv_status wait_for(unique_lock<mutex>& lck,
                   const chrono::duration<Rep, Period>& rel_time);
```

(2) Предикативная

```
template <class Rep, class Period, class Predicate>
bool wait_for(unique_lock<mutex>& lck,
              const chrono::duration<Rep, Period>& rel_time,
              Predicate pred);
```

Выполнение текущего потока (который перед этим должен обязательно заблокировать мьютекс **`lck`**) блокируется во время **`rel_time`** или до получения уведомления (если последнее произойдет первым).

В момент блокировки потока функция автоматически вызывает **`lck.unlock()`**, позволяя другим заблокированным потокам продолжить работу.

После уведомления или после прохождения **`rel_time`** функция разблокируется и вызывает **`lck.lock()`**, оставляя **`lck`** в том же состоянии, что и при вызове функции.

Затем функция возвращает управление.

Как правило, функция уведомляется о пробуждении посредством вызова в другом потоке либо члена **`notify_one()`**, либо члена **`notify_all()`**.

Некоторые реализации могут вызывать ложные пробуждающие вызовы без вызова какой-либо из этих функций. Следовательно, пользователи этой функции должны убедиться, что их условие для возобновления выполнено.

Если указано **`pred(2)`**, функция блокируется только в том случае, если **`pred`** возвращает **`false`**, а уведомления могут разблокировать поток только тогда, когда он становится истинным (что особенно полезно для проверки на предмет ложных пробуждающих вызовов).

Ведет себя так, как если бы она реализована как:

```
return wait_until(lck,
                  chrono::steady_clock::now() + rel_time,
                  std::move(pred));
```

Возвращаемое значение

Безусловная версия (1) возвращает **`cv_status::timeout`**, если функция возвратилась по прошествию **`rel_time`**, или **`cv_status::no_timeout`** в противном случае.

Версия с предикатом (2) возвращает **`pred()`** независимо от того, был ли активирован тайм-аут (хотя он может иметь значение **`false`** только при срабатывании).

Пример condition_variable::wait_for

```
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <chrono>             // std::chrono::seconds
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable, std::cv_status

std::condition_variable cv;
int value;

void read_value() {
  std::cin >> value;
  cv.notify_one();
}

int main () {
  std::cout << "Please, enter an integer (I'll be printing dots): \n";
  std::thread th(read_value);

  std::mutex mtx;
  std::unique_lock<std::mutex> lck(mtx);
  while (cv.wait_for(lck, std::chrono::seconds(1)) == std::cv_status::timeout){
    std::cout << '.' << std::endl;
  }
  std::cout << "You entered: " << value << '\n';
  th.join();
}
```

std::cv_status — состояние переменной состояния

```
enum class cv_status;
```

Тип, который указывает, вернулась ли функция из-за тайм-аута или нет.

Значения этого типа возвращаются членами **`wait_for()`** и **`wait_until()`** переменных состояния **`condition_variable`** и **`condition_variable_any`**.

Определена как:

```
enum class cv_status { no_timeout, timeout };
```

condition_variable_any — переменная состояния по любой блокировке

То же самое, что и **`condition_variable`**, за исключением того, что ее функции ожидания могут принимать любой блокируемый тип в качестве аргумента — объекты **`condition_variable`** могут принимать только **`unique_lock<mutex>`**).

В остальном они идентичны.

std::notify_all_at_thread_exit — уведомить всех при выходе из потока

```
void notify_all_at_thread_exit(condition_variable& cond,
                               unique_lock<mutex> lck);
```

Когда вызывающий поток завершается, все потоки, ожидающие **cond**, уведомляются о возобновлении выполнения.

Функция также получает право собственности на блокировку объекта мьютекса, управляемого **lck**, который хранится внутри функции и разблокируется при выходе из потока (непосредственно перед уведомлением всех потоков).

Она ведет себя ведя себя так, как если бы следующий код был бы вызван один раз ко всем объектам с продолжительностью хранения потока, которые должны быть уничтожены

```
lck.unlock();
cond.notify_all();
```

cond —
a condition_variable object to notify all at thread exit.
lck —
aunique_lock object whose mutex object is currently locked by this thread.
The object is acquired by the function (it shall be an rvalue).
All waiting threads on cond (if any) shall use the same underlying mutex object as lck.

Пример notify_all_at_thread_exit

```
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex              mtx;
std::condition_variable cv;
bool                    ready = false;

void print_id(int id) {
  std::unique_lock<std::mutex> lck(mtx);
  while (!ready) cv.wait(lck);
  // ...
  std::cout << "thread " << id << '\n';
}

void go() {
  std::unique_lock<std::mutex> lck(mtx);
  std::notify_all_at_thread_exit(cv, std::move(lck));
  ready = true;
}
```

```
int main () {

  std::thread threads[10];
  // spawn 10 threads:
  for (int i = 0; i < 10; ++i)
    threads[i] = std::thread(print_id, i);
  std::cout << "10 threads ready to race...\n";

  std::thread(go).detach();   // go!
  for (auto& th : threads) th.join();

  return 0;
}
```