
# ВВЕДЕНИЕ В МУЛЬТИПРОГРАММИРОВАНИЕ

ЛЕКЦИЯ 14

# dead_lock.cpp

```
std::lock_guard<std::mutex> lock(a);
std::lock_guard<std::mutex> lock(b);
```

```
std::lock_guard<std::mutex> lock(b);
std::lock_guard<std::mutex> lock(a);
```

Возникает когда несколько потоков пытаются получить доступ к нескольким ресурсам в разной последовательности.

# Локальное хранилище потока `tlc.cpp`

- Набор статических/глобальных переменных, локальных по отношению к данному потоку.

- Отличие от реентерабельности в том, что используются не только «локальные переменные потока» и его параметры,, но и специальный менеджер ресурсов.

Вообще говоря, в данном примере TLS не совсем потокобезопасный.

# Потоко-безопасный Stack `stack.cpp`

Классы «обертки» позволяют непротиворечиво использовать мьютекс в RAII-стиле с автоматической блокировкой и разблокировкой в рамках одного блока. Эти классы:

**lock_guard**
когда объект создан, он пытается получить мьютекс (вызывая lock()), а когда объект уничтожен, он автоматически освобождает мьютекс (вызывая unlock())

Печать на экран это то же разделяемый ресурс – так что печать то же защищаем мьютексом.

# Пример опасной ситуации
## pass_out.cpp

Передача данных из за границу lock.

**Правило**:

Ни когда не передавайте во вне указатели или ссылки на защищаемые данные. Это касается сохранение данных в видимой области памяти или передачи данных как параметра в пользовательскую функцию.

# Exceptions в многопоточной среде exception.cpp

1. Исключения между потоками не передаются!
2. Нужно устроить хранилище исключений, для того что бы их потом обработать!

# future + async **async_1.cpp**

future – служит для получения результата вычислений из другого потока.

Т.е. Функция выполняемая thread теперь может возвращать значение.

– это шаблон (параметр – тип возвращаемого значения)

– конструируется с помощью std::async

— результат выполнения получается методом get()

— async запускает поток и синхронизирует результат с возвращаемым future

```
T function( ) {
 return T( );
}
int main ( ){
std::future<T> fut = std::async (function);
T x = fut.get( );
}
```

# std::future<T>
## ограничения

1. Нельзя вызывать сразу несколько get() из разных std::thread. Если нужно синхронизировать сразу несколько потоков – то лучше использовать условные переменные (будет расмотренно далее).
2. Нельзя копировать (только передача по ссылке).

# future работает на promise
## promise.cpp

**template <class T> promise** – шаблон который сохраняет значение типа T, которое может быть получено с помощью future

**get_future** – запрос future, связанного со значением внутри promise

**set_value** – устанавливает значение (и передает его в вызов get_future)

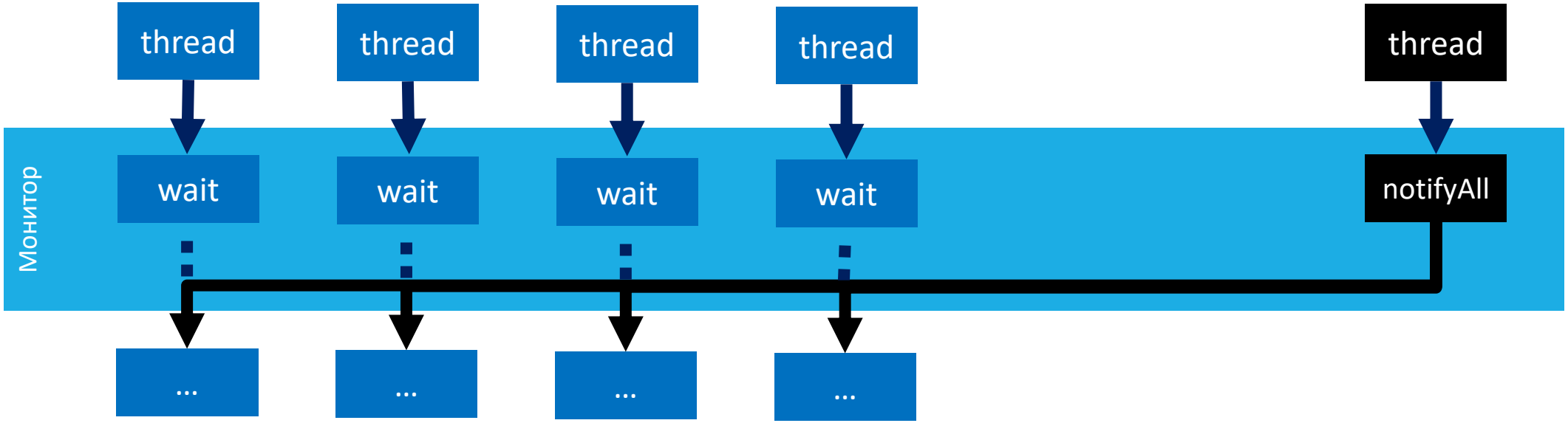
**set_exception** – передача исключения

# Пример
## async_2.cpp

**Параллельные вычисления – это легко!**

# Условная переменная

примитив синхронизации, обеспечивающий блокирование одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания. Условные переменные используются вместе с ассоциированным мьютексом и являются элементом некоторых видов мониторов.



# Условные переменные <condition_variable>
## `conditional_1.cpp`

### condition_variable

требует от любого потока перед ожиданием сначала выполнить std::unique_lock

1. Должен быть хотя бы один поток, ожидающий, пока какое-то условие станет истинным. Ожидающий поток должен сначала выполнить unique_lock.
2. Должен быть хотя бы один поток, сигнализирующий о том, что условие стало истинным. Сигнал может быть послан с помощью notify_one(), при этом будет разблокирован один (любой) поток из ожидающих, или notify_all(), что разблокирует все ожидающие потоки.
3. В виду некоторых сложностей при создании пробуждающего условия, которое может быть предсказуемых в многопроцессорных системах, могут происходить ложные пробуждения (spurious wakeup). Это означает, что поток может быть пробужден, даже если никто не сигнализировал условной переменной. Поэтому необходимо еще проверять, верно ли условие пробуждение уже после то, как поток был пробужден.

# multi_producer_consumer.cpp

```
static void consumer(size_t id)
{
while (!production_stopped || !q.empty()) {
     std::unique_lock<std::mutex> lock(q_mutex);

if (go_consume.wait_for(lock, 1s, [] { return !q.empty(); })) {
     pcout{} << " item "
               << std::setw(3) << q.front() << " --> Consumer "
               << id << '\n';
     q.pop();
     go_produce.notify_all();
     std::this_thread::sleep_for(130ms);
     }
     }

pcout{} << "EXIT: Consumer " << id << '\n';
}
```

# std::condition_variable

- ожидание уведомления
- механизм синхронизации потоков
- довольно сложно использовать (мьютекс, цикл, условие)
- есть проблемы spurious wakeup / lost wakeup

# Проблема блокировок

1. Взаимоблокировки (Deadlocks)
2. Надежность — вдруг владелец блокировки помрет?
3. Performance
   - Параллелизма в критической секции нет!
   - Владелец блокировки может быть вытеснен планировщиком

# Закон Амдала

Джин Амдал (Gene Amdahl) - один из разработчиков всемирно известной системы IBM 360 в 1967 году предложил формулу, отражающую зависимость ускорения вычислений, достигаемого на многопроцессорной ВС, как от числа процессоров, так и от соотношения между последовательной и распараллеливаемой частями программы. Проблема рассматривалась Амдалом исходя из положения, что объем решаемой задачи (рабочая нагрузка - число выполняемых операций) с изменением числа процессоров, участвующих в ее решении, остается неизменным.

Пусть
*f* - доля операций, которые должны выполняться последовательно одним из процессоров и
*1-f* - доля, приходящаяся на распараллеливаемую часть программы.
Тогда ускорение, которое может быть получено на ВС из *n* процессоров, по сравнению с однопроцессорным решением не будет превышать величины:

***S(n)=T(1)/T(n)=1/[f+(1-f)/n].***

Например, если половина операций подлежит распараллеливанию на 4 машинах, то ускорение равно:

S(4) = 1/(0.5+0.5/4)=1.6 т.е. Только в полтора раза!

# Итого

- ✓ Многопоточность
- ✓ Параллельность
- ■ Асинхронность

# Итого: Многопоточность

возможность операционной системы (системы поддержки  выполнения программ) создать несколько дополнительных потоков выполнения в рамках одного процесса.

# Итого: Параллельность

выполнение некоторых участков кода программы одновременно  на разных вычислительных мощностях системы.

# Итого: Асинхронность

Отделение вызывающего потока от потока исполнения запроса.

# Что когда применять?

- **Многопоточность** – в любой программе, где есть несколько потоков.
- **Параллельность** – где важна вычислительная скорость И задачи можно эффективно распараллеливать.
- **Асинхронность** – где можем найти занятие, пока ждём результата.

# asynchron_1.cpp

```
auto main() -> int {
 PrintHandler printHandler;
 EventLoop eventLoop;

 eventLoop.addHandler(&printHandler);
 eventLoop.send({EventCode::start,"starting"});
 std::thread workerThread{userThread, std::ref(eventLoop)};

 eventLoop.exec();
 workerThread.join();

return 0;
}
```

# asynchron_2.cpp

```
int sum{};
bool done{false};

eventLoop.send({
EventCode::add,
 std::make_shared<Data>(2,3),
 std::make_shared<ContextAdd>(sum,done),
[](std::shared_ptr<Context> cnt){
 if(auto ptr=std::static_pointer_cast<ContextAdd>(cnt))
  ptr->done = true;
 }
});
```

# Проблемы асинхронности

Асинхронный код требует:

1. хранение контекста операции
2. умение этот контекст восстанавливать
3. специальный функционал завершения
4. механизм возврата результата


# Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ