ЛЕКЦИЯ 16
ОЧЕРЕДИ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

МЕТНАЛЬНАЯ МОДЕЛЬ МЬЮТЕКСА

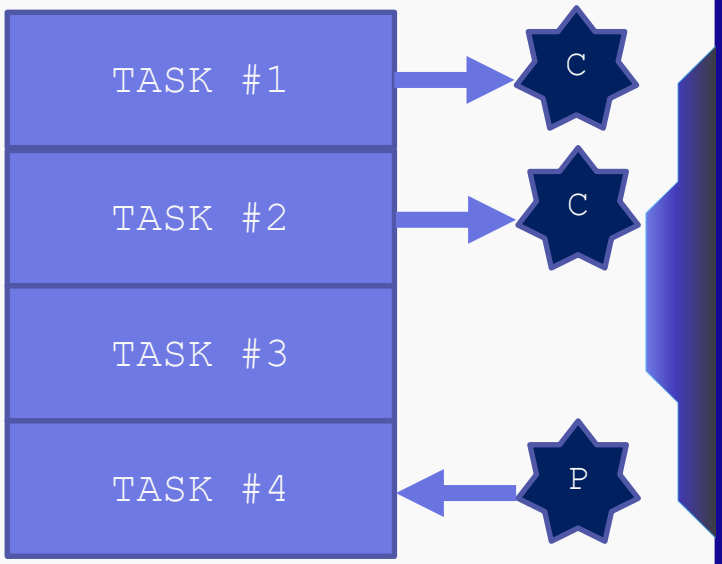
- Мьютекс – это очередь
- Внутри этой очереди потоки по очереди работают с ресурсом, который мьютексом защищен
- Подумайте вот над таким кодом

```
{
  std::lock_guard lk{m};
  res += 1;
}
```

- Если мы точно знаем, что у нас три потока, то что тут написано?

УБЕЖАТЬ ОТ МНОГОПОТОЧНОСТИ

- Проблемы проектирования многопоточного кода с классическими блокировками приводят к идее ограничить многопоточность
- В идеале хотелось бы оставить только один разделяемый ресурс – очередь задач и поставить потоки работать с очередью
- Исполнение каждой задачи из этой очереди может быть сколь угодно сложным, но уже однопоточным и может использовать небезопасные контейнеры



ПРОИЗВОДИТЕЛИ И ПОТРЕБИТЕЛИ

- Классическая формулировка проблемы касается всего двух потоков: производителя и потребителя, работающих с буфером фиксированного размера
- Нужно избегать как overflow так и underflow

- В этой проблеме много интересных граней

КЛАССИЧЕСКАЯ ОЧЕРЕДЬ: ИНТЕРФЕЙС

```
template <typename T> class ts_queue {
  mutable std::mutex Mut;
  std::condition_variable CondCons, ConProd;
  std::vector<T> Buffer; // ограниченный буфер
  int NCur = -1; // начальная позиция в буфере
public:
  ts_queue(int Size) : Buffer(Size) {}
  void push(T Data);
  void wait_and_pop(T &Data);

// возможно что-то еще?
```

PRODUCE-CONSUME

```
void push(T Data) {
  unique_lock<mutex> Lk{Mut};
  CondProd.wait(Lk,
  [this] {return !full();});
  NCur += 1;
  Buffer[NCur] = Data;
  CondCons.notify_one();
}
```

```
void wait_and_pop(T &Data) {
  unique_lock<mutex> Lk{Mut};
  CondCons.wait(Lk,
  [this] {return !empty();});
  Data = Buffer[NCur];
  NCur -= 1;
  CondProd.notify_one();
}
```

- Увы, есть небольшая проблема. Она иногда зависает .

ПРИМЕР ИСПОЛЬЗОВАНИЯ

```
for (;;) {
  int N;
  // critical section
  {
    lock_guard<mutex> Lk{M};
    if (NTasks < 0)
      break;
    N = NTasks; NTasks -= 1;
  }
  // produce then push
  tt::sleep_for(PTime);
  Q.push(N);
}
```

```
for (;;) {
  int N;
  // critical section
  {
    lock_guard<mutex> Lk{M};
    if (NTasks < 0)
      break;
  }
  // pop then consume
  Q.wait_and_pop(N);
  tt::sleep_for(CTime);
}
```

УБИРАЕМ ПРОБЛЕМУ

```
template <typename T> class ts_queue {
  bool Done = false;
....
  void push(T Data) {
    std::unique_lock<std::mutex> Lk{Mut};
    CondProd.wait(Lk, [this] {return !full() || done();});
    if (Done) return;
....
  void wake_and_done() {
    Done = true;
    CondCons.notify_all();
    CondProd.notify_all();
  }
```

ДОБАВЛЯЕМ НЕФИКСИРОВАННЫЙ БУФЕР

Допустим разрешен нефиксированный буфер
Это позволяет сэкономить одну из условных переменных

```
template <typename T> class ts_queue {
  mutable std::mutex Mut;
  std::condition_variable CondCons; // производители не ждут
  std::queue<T> Buffer; // неограниченный размер
```

ПРОИЗВОДСТВО ЗАДАЧИ

Для производительности важно, чтобы каждая обработка задачи потребителем проходила вне критической секции

```
{
  std::unique_lock<std::mutex> Lk{Mut};
  Q.push(std::move(Data)); // считаем, что это дешево
}
Cond.notify_one();
```

ПОТРЕБЛЕНИЕ ЗАДАЧИ

```
std::unique_lock<std::mutex> Lk{Mut};
Cond.wait(Lk, [this] {return !Q.empty();});

// взяли lock, началась критическая секция
T Task = std::move(Q.front());

// предполагаем наличие «лимитирующей» задачи
if (Task == Limiter<T>())
  return;

Data = std::move(Task);
Q.pop();
```

ОБСУЖДЕНИЕ

- Сложно ли расширить это с модельной проблемы с целыми на произвольные упакованные задачи?
- Прежде чем взяться за это, нам следует понять, как может выглядеть упакованная задача и, шире говоря, коммуникация с потоком

ВОЗВРАТ ДАННЫХ ИЗ ПОТОКА

Если поток используется для вычислений, всегда можно вернуть из него нечто по ссылке

```
auto divi = [](int &result, int a, int b) {result = a/b;}

int result;
std::thread t(divi, std::ref(result), 20, 5);
t.join();
std::cout << "result: " << result << std::endl;
```

НЕБОЛЬШАЯ ЗАДАЧА

Давайте попробуем сделать generic lambda вместо обычно как функцию потока

```
auto diva = [](auto &result, auto a, auto b) {result = a/b;}

int result;
std::thread t(diva, std::ref(result), 20, 5); // FAIL
t.join();
std::cout << "result: " << result << std::endl;
```

Удивительно, но в этом коде **две** ошибки. Кто укажет обе?

НЕБОЛЬШАЯ ЗАДАЧА: РЕШЕНИЕ?

Давайте попробуем сделать generic lambda вместо обычно как функцию потока

```
auto diva = [](auto &&result, auto a, auto b) {
  result.get() = a/b;
}
int result;
std::thread t(diva, std::ref(result), 20, 5);
t.join();
std::cout << "result: " << result << std::endl;
```

- Обе связаны с особенностями `reference wrapper`
- Во-первых, это правое значение. Во-вторых, вытащить левую ссылку можно только явно, вывести то ему неоткуда

https://godbolt.org/z/bj5TEnsxP

ОБСУЖДЕНИЕ

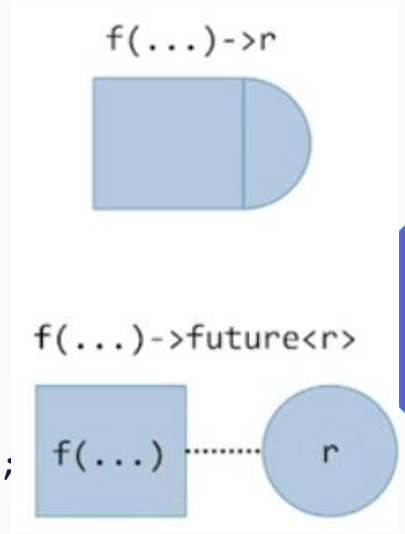
Канал связи когда в поток просто передается ссылка на некую переменную имеет свои недостатки

```
auto divide = [](auto &&result, auto a, auto b) {
  result.get() = a/b;
}
int result;
std::thread t(divi, std::ref(result), 20, 5); // FAIL
t.join();
std::cout << “result: “ << result << std::endl;
```

- Перечислите потенциальные проблемы, связанные с тем, что у нас нет контроля над тем, когда обновляется значение `result`

ЛУЧШИЙ КАНАЛ СВЯЗИ: FUTURES

```
std::promise<int> p;
std::future<int> f = p.get_future();
auto divi = [](auto &&result, auto a, auto b) {
  result.set_value(a / b);
};
std::thread t(divi, std::move(p), 30, 6);
t.detach(); // отправляем в полет
std::cout << "result: " << f.get() << std::endl;
```



- Здесь вызов `f.get()` заблокирует поток пока не получит сигнал от `p.set_value()`

https://godbolt.org/z/vGec9ev13

ПРАВИЛА ОСТОРОЖНОСТИ

- Обещание, которое никто не собирается выполнять – это **deadlock**

```
std::promise<int> pr; auto fut = pr.get_future();
fut.get(); // forever
```

- Обещание, которое уже выполнено – это исключение

```
std::promise<int> pr; pr.set_value(10);
pr.set_value(10); // Error: promise already satisfied
```

- Как и нарушенное обещание

```
std::promise<int> pr; auto fut = pr.get_future();
{ std::promise<int> pr2(std::move(pr)); } // Error: broken
promise
```

ОБСУЖДЕНИЕ

- Мы пока на этом не останавливались, но…
- Что если внутри потока выброшено исключение?

ИСПОЛЬЗОВАНИЕ EXCEPTION_PTR

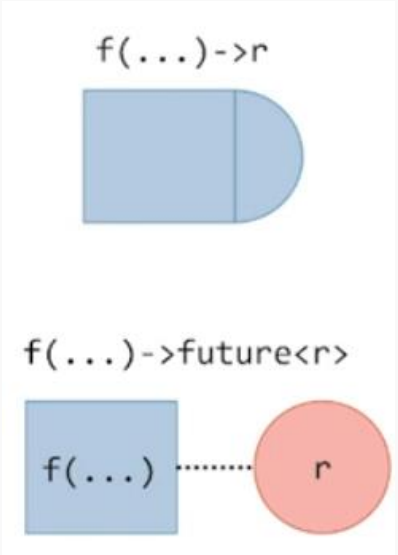
```
void do_raise {
  throw std::runtime_error(“Exception!”);
}
std::exception_ptr get_exception() {
  try { do_raise(); }
  catch(...) {return std::current_exception();}
  return nullptr;
}

// где-то далее в коде (в том числе в другом потоке)
std::exception_ptr e = std::get_exception();
std::rethrow_exception(e);
```

МАРШАЛИНГ ИСКЛЮЧЕНИЙ

- Механизм futures позволяет нам вытащить исключение наружу!

```
auto divi = [](auto&& result, auto a, auto b) {
  try {
    if(b == 0) throw “Divide by zero”;
    result.set_value(a / b);
  } catch(...) {
    result.set_exception(std::current_exception());
  }
};
// .... все тоже самое ....
std::cout << “result: “ << f.get() << std::endl;
```

https://godbolt.org/z/fWM99E885



ДОПОЛНЕНИЕ: ВКЛАДЫВАНИЕ ИСКЛЮЧЕНИЙ

```
try {
  open_file("nonexistent.file");
}  catch(...) {
  std::throw_with_nested(runtime_error("run() failed");
}
// где-то дальше
catch(std::runtime_error &e) {
  std::cout << e.what() << std::endl;
  std::rethrow_if_nested(e);
}
```

Разумеется, вложенные исключения точно также маршалятся в exception_ptr

ОБСУЖДЕНИЕ

- Лишний параметр promise – это все-таки бревно в глазу
- Мы бы хотели написать функцию деления в старом добром стиле

```
auto divi = [](auto a, auto b) {
  if (b == 0) throw std::overflow_error(“Divide by zero!”);
  result = a/b;}
```

- И дальше получить все остальное бесплатно
- Возможно ли это?

УПАКОВАННЫЕ ЗАДАЧИ

Задача, представляющая собой функцию в старом стиле, от которой оторван результат во future для отдачи на поток называется packaged

```
auto divi = [](auto a, auto b) {
  if (b == 0) throw std::overflow_error("Divide by zero!");
  result = a/b;}
std::packaged_task<int(int, int)> task {divi};
std::future<int> f = task.get_future(); // неявный promise
std::thread t(std::move(task), 30, 0);
```

https://godbolt.org/z/cvMKTojav

ОБСУЖДЕНИЕ

- А что, если бы мы могли отдавать потокам напрямую команду на завершение?

JTHREADS

- Класс `std::jthread` – это нововведение C++20, он делает `join()` в деструкторе

```
int foo() {
  std::jthread thread(thread_func, 5);
} // деструктор вызывает join()
```

- Кроме того, он может проверять, имеет ли смысл вызов `join()`

```
std::jthread t; EXPEQT_EQ(t.joinable(), false);
t = std::jthread(foo); EXPEQT_EQ(t.joinable(), true);
t.join(); EXPEQT_EQ(t.joinable(), false);
```

JTHREADS: ПРЕРЫВАЕМЫЕ ПОТОКИ

- Также он принимает `stop_token` для прерываемости

```
void bar(std::stop_token stop_token, int value) {
  while(!stop_token.stop_requested())
    std::cout << value++ << "\n";
}
int foo () {
  std::jthread t(bar, 5); // начинает печатать 5, 6, 7,....
  std::this_thread::sleep_for(1s);
  t.request_stop(); // попросили остановиться
}
```

ИНТЕРЕСНАЯ ЗАДАЧА

- Представим, мы хотим организовать барьер

```
void prepare(....) {
  // do something
  // -- hold here --
  // do something else when signaled
}
```

Функция `prepare` работает в отдельном потоке, ждет оповещения и продолжает работу

УСЛОВНЫЕ ПЕРЕМЕННЫЕ

- Первая идея – это условные переменные, которые мы уже рассматривали

```
void prepare(....) {
  // do something
  Prepared.notify_one();
  // -- hold here --
  std::unique_lock<std::mutex> Lk{Mut};
  Resumed.wait(Lk, [this] {return resumed();}
  // do something else when signaled
}
```

МЕХАНИЗМ SHARED_FUTURE

- В отличие от `std::future`, которое только `movable`, `std::shared_future` копируемо

- Используя его, много потоков могут ждать одного состояния

```
std::promise<void> ReadyP;
std::shared_future<void> ReadyF = ReadyP.get_future();
std::thread Fst([](auto SF) {SF.wait();}, ReadyF};
std::thread Snd([](auto SF) {SF.wait();}, ReadyF};
ReadyP.set_value(); // разблокируем оба
Fst.join(); Snd.join();
```

РАЗДЕЛЯЕМЫЕ БУДУЩИЕ

- Вторая идея это разделяемые будущие

```
void prepare(std::promise<void> &&Ready,
             std::shared_future<void> BackLink) {
  // do something
  Ready.set_value();
  BackLink.wait(); // -- hold here --
  // do something else when signaled
}
```

https://godbolt.org/z/zda8Wjsch

STD::LATCH

- Третья идея возвращает нам симметрию

```
void prepare(std::latch &L,
             std::latch BackL) {
  // do something
  L.count_down();
  BackL.wait(); // -- hold here --
  // do something else when signaled
}
```

- Особенно упрощается ожидание всех: мы используем один `std::latch` с обратным счетчиком и все

https://godbolt.org/z/M8n1vTc3o

ОЧЕРЕДЬ С ПРОИЗВОЛЬНЫМИ ЗАДАЧАМИ

- Пусть есть несколько разных по аргументам и типу результата функций

```
int fn1(int x, int y, int z) { return x + y + z; }
double fn2(std::vector<int> v) {return v.size() + 0.5;}
```

- Теперь хочется сделать очередь, в которой стояли бы экземпляры вызова этих функций (функция и упакованные аргументы) и отдать на пул

```
std::vector<std::thread> consumers;
for (int i = 0; i < nthreads; ++i)
  consumers.emplace_back(consumer_thread_func);
```

- Разумеется, `consumer_thread_func` хочется иметь одну

ПРОТОТИП THREAD_FUNC

- Мы хотели бы разработать `task_t`, чтобы функция-консьюмер могла быть устроена просто, анализируя только результат

```
for (;;) {
  if (safe_empty()) {yield(); continue;}
  task_t cur = safe_pop();
  int res = cur(); // или что-то вроде того
  if (res == -1) {
    safe_push([]{return -1;});
    break;
  }
}
```

- Но при этом под `task_t` должна иметь возможность жить любая функция

СТРУКТУРА TASK_T

- В сравнительно новых стандартах это совсем просто

```
// generic (type-erased) task to put on queue
// returns -1 if it is special signalling task
// (end of work for consumers)
// otherwise do what it shall and return 0
using task_t = std::move_only_function<int>();
```

- До C++23 этот класс приходилось писать руками

СОЗДАНИЕ ЗАДАЧ

- Для обёртки любой функции в task_t хочется простой синтаксис

```
auto &&[task, future_res] = create_task(func, args);
```

- Далее можно запушить task в очередь, а когда результат понадобится, ждать `future_res`
- Мы хотели бы, чтобы все операции с очередью сводились к легкому перемещению задач
- Проблема в том, что любая функция под `task_t` конечно будет требовать сохранения своих аргументов в замыкании, и мы хотели бы сохранить там нечто вроде `std::packaged_task`

СВОДИМ ВМЕСТЕ: СОЗДАНИЕ ЗАДАЧИ

```
template <typename F, typename... Args>
auto create_task(F f, Args &&... args) {
  std::packaged_task<std::remove_pointer_t<F>> tsk{f};
  auto fut = tsk.get_future();
  task_t t{???};
  return std::make_pair(std::move(t), std::move(fut));
}
```

- Что бы вы написали в создании задачи в таких условиях?
- Мы должны: захватить задачу, захватить аргументы и вернуть функцию, которая вызывает задачу (и возвращает 0, т.к. обычная задача не sentinel)

СВОДИМ ВМЕСТЕ: СОЗДАНИЕ ЗАДАЧИ

```
template <typename F, typename... Args>
auto create_task(F f, Args &&... args) {
  std::packaged_task<std::remove_pointer_t<F>> tsk{f};
  auto fut = tsk.get_future();
  task_t t{[ct = std::move(tsk),
    args = make_tuple(std::forward<Args>(args)...)]()
  mutable {std::apply(
    [ct = std::move(ct)](auto &&... args) mutable {
      ct(args...);
    }, std::move(args));
  return 0;}}; // обычная задача, не sentinel
  return std::make_pair(std::move(t), std::move(fut));
}
```

НЕБОЛЬШАЯ ПРОБЛЕМА

- Собранная вместе очередь иногда теряет задачи
- Что идёт не так?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Дональд Кнут Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М .: «Вильямс» , 2006. – 720 с.
3. Дональд Кнут Искусство программирования, том 2. Получисленные алгоритмы - The Art of Computer Programming, vol.2. Seminumerical Algorithms. — 3-е изд. — М .: «Вильямс» , 2007. — 832 с.
4. Sean Parent, Better Code – Concurrency, 2017
5. Geoffrey Romer, What do you mean “thread-safe”?, CppCon 2018
6. Скотт Мейерс, Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2016. – 298 с.
7. Rainer Grimm, Concurrency Patterns, Meeting C++ online, 2023