Асинхронность и параллелизм

Практическое объектно-ориентированное программирование

20.11.2024

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), 30, 6);
t.join();
std::cout « “result: “ « result « std::endl;
```

Небольшая задача

• Давайте попробуем сделать generic lambda вместо обычной как функцию потока

```
auto diva = [](auto& result, auto a, auto b) {
  result = a / b;
};
int result;
std::thread t(diva, std::ref(result), 30, 6); // FAIL
```

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

Обсуждение

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

```
auto divide = [](auto&& result, auto a, auto b) {
  result.get() = a / b;
}
int result;
std::thread t(divide, std::ref(result), 30, 6);
```

• 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 полёт
std::cout « “result: “ « f.get() « std::endl;
```

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

Правила осторожности

• Обещание которое никто не собирается выполнять это deadlock.

```
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.

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

Обсуждение

- Мы пока на этом не останавливались, но...
- Что если внутри потока произошло исключение?

Использование exception_ptr

```
void do_raise () {
  throw runtime_error("Exception!");
}
exception_ptr get_exception () {
  try { do_raise (); }
  catch (...) { return current_exception(); }
  return nullptr;
}
// где-то далее в коде (в том числе в другом потоке)
exception_ptr e = get_exception();
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());
  }
};
// .... всё то же самое ....
  std::cout « “result: “ « f.get() « std::endl;
```

Дополнение: вкладывание исключений

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

- Разумеется вложенные исключения точно также маршалятся в exception_ptr

Обсуждение

- Лишний параметр promise это всё-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");
  return a / b;
}
```

- И дальше получить всё 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");
  return 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);
```

Обсуждение

- А что если бы мы могли отдавать потокам прямую 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; EXPECT_EQ(t.joinable(), false);
t = std::jthread(foo); EXPECT_EQ(t.joinable(), true);
t.join(); EXPECT_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++ « std::endl;
}
int foo() {
  std::jthread t(bar, 5); // начинает печатать 5 6 7....
  std::this_thread::sleep_for(1s);
  t.request_stop(); // попросили остановиться.
}
```

Обсуждение

- Какой код лучше и почему?

```
const int *src;
int *dst;
// 1. явный цикл
for(i = 0; i < srclen; ++i)
  dst[i] = src[i];
// 2. функция стандартной библиотеки
memcpy(dst, src, srclen * sizeof(int));
```

- Считаем, что компилятор не может перевести одно в другое.

Обсуждение

• Итак, memcpy лучше. А если код обобщённый?

```
const T *src;
T *dst;
// 1. явный цикл
for(i = 0; i < srclen; ++i)
  dst[i] = src[i];
// 2. функция стандартной библиотеки
std::copy(src, src + srclen, dst);
```

• Перечислите как можно больше причин почему стандартный алгоритм лучше.

Оптимизации

• Ключевым аргументом за memcpy являются возможные низкоуровневые оптимизации, например чтение и запись большими блоками.
• Ключевым аргументом за std::copy является тот факт, что она может за счёт SFINAE понимать что для T можно вызвать memcpy.

```
template <class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last, OutputIt d_first) {
  если iterator contiguous и value_type тривиальный, то
    memcpy();
  иначе
    copy_impl(first, last, d_first);
}
```

Tricky case

- А что насчёт for_each?

```
// 1. явный цикл, у foo есть побочные эффекты
for(i = 0; i < srclen; ++i)
  foo(src[i]);
// 2. функция стандартной библиотеки
std::for_each(src, src + srclen, foo);
```

- Можно ли тут найти аргументы за функцию стандартной библиотеки кроме чисто религиозных?

Tricky case

- А что насчёт for$_e$*ach*?

```
// 1. явный цикл, у foo есть побочные эффекты
for(i = 0; i < srclen; ++i)
foo(src[i]);
// 2. функция стандартной библиотеки
auto&& policy = std::execution::par_unseq;
std::for_each(policy, src, src + srclen, foo);
```

- Новые дополнения в C++17 позволяют задавать функциям стандартной библиотеки политики для исполнения.

Стандартные политики

std::execution::seq

• Последовательное исполнение (линейный порядок).

std::execution::par

• Параллельное исполнение (неопределённый порядок).

std::execution::par_unseq

• Параллельное исполнение с возможной векторизацией (отсутствие порядка).

• Довольно много алгоритмов уже является параллельными, то есть имеют перегрузки для execution policy.

Напишем свой параллельный reduce?

• Основная идея проста и приятна: выделим каждому потоку аккумулировать свою часть, а потом соберём аккумулированные подрезультаты.

Напишем свой параллельный reduce?

```
template <typename Iterator, typename T>
T my_reduce(Iterator first, Iterator last, T init = 0) {
 std::vector<std::thread> threads(nthreads);
 std::vector<T> res(nthreads + 1);
 long bsize = length / nthreads;
 for (/* для каждой части i */)
  threads[tidx] = std::thread(accumulate_block, i, i + bsize, std::ref(res));
// собираем результаты
}
```

Но есть некие вопросы

- Как определить количество потоков?
- Как обработать “хвосты“ если размер данных не кратен количеству потоков?
- Всё это решаемо, конечно, но код получается довольно громоздким.
- Ссылка на пример внизу, обратите внимание на использование std::thread::hardware_concurrency() для определения оптимального количества потоков.
- Код довольно наивный так как потоки вынуждены постоянно запускаться и завершаться. Правильный пул потоков поддерживает живые потоки ждущими задач.

Используем packaged tasks?

• При написании на упакованных задачах можно хранить вектор futures.

```
template <typename FwdIt, typename T>
T my_reduce(FwdIt first, FwdIt last, T init = 0) {
 std::vector<std::future<T» results(nthreads);
 for (/* для каждой части i */) {
  packaged_task<T(FwdIt, FwdIt)> taskaccumulate_block;
 results[i] = task.get_future();
// помещаем task в поток
 }
// собираем результаты
}
```

Обсуждение

- Ещё более удобным кажется вообще не делать явное управление потоками, а просто отдать упакованную задачу на асинхронное исполнение.
- В этом случае, если операционная система поддерживает пулы потоков, задача может пойти в пул.

Асинхронные задачи

• Основной элемент синтаксического сахара это std::async.

```
auto divi = [](auto a, auto b) {
  if (b == 0)
    throw std::overflow_error("Divide by zero");
  return a / b;
};
std::future<int> f = std::async(divi, 30, 5);
auto x = f.get();
```

• Теперь вся механика скрыта внутри и маршалинг исключений мы получаем бесплатно.

Политики запуска async

• Асинхронное вычисление не обязательно означает “отданное на отдельный поток“, иногда мы можем выиграть просто от отложенного вычисления. async(policy, task, params);
• std::launch::async означает, что по возможности мы хотели бы распределить задачу для параллельного выполнения.
• std::launch::deferred означает, что мы хотели бы получить результат вычисления в точке, где он действительно нужен.
Использованный нами ранее async(task, params) ведёт себя так как будто оба флага выставлены.

Обсуждение

Удивительно, но нельзя позвать async без обоих флагов. Хотя бы один (launch::async или launch::deferred) должен присутствовать.

- Как вы думаете, почему?

Слишком много свободы компилятору?

- Всмотримся в следующую строчку надев волшебные очки.

std::future<int> f = std::async(divi, 30, 5);

- Мы видим что на самом деле она выглядит иначе.

std::future<int> f = std::async(async | deferred, divi, 30, 5);

- То есть даёт выбор: либо действительно запустить на отдельном потоке либо просто отложить вычисления до вызова get.
- Вообще-то это на удивление разные сценарии, так что хороший тон это явное указание политики (мы ещё вернёмся к этому позже).

Параллельный reduce снова

• При написании на явном async мы вообще нигде не видим потоков, но они есть.

```
template <typename FwdIt, typename T>
T my_reduce(FwdIt first, FwdIt last, T init = 0) {
  std::vector<std::future<T» results(ntasks);
  for (/* для каждой части i */)
    results[i] = std::async(std::launch::async, accblock, i, bsz);
// собираем результаты
  for (/* для каждой части i */)
    result += results[i].get();
```