CQ 20.

5 апреля 2023 г.

1 Что есть "состояние гонки"и к чему оно может привести?

Состояние гонки (Race Condition) – конкуренция (в нашем случае между потоками) за выполнение какой-либо операции первым.

Опасны состояния гонки при работе с данными, т.к. они могут привести к UB (неопределённому поведению). С другой стороны, в некоторых случаях состояние гонки безвредно (т.е. ни к чему особому не приводит). Например, при добавлении элемента в очередь (в общем случае).

2 Как используются мьютексы и условные переменные?

В соответствии с предыдущим вопросом становится понятно, что при работе с данными в многопоточном режиме необходим механимз безопасного модифицирования общих данных. Для этого существуют примитивы синхронизации, а также lockfree подход и атомарные единицы (альтернативные, но связанные подходы).

Mutex – mutual exclusion – средство для предоставления эксклюзивного доступа к данным для одного потока. В заголовке <mutex> существует три типа мьютексов:

- std::mutex;
- std::recursive_mutex;
- std::timed_mutex;
- std::scoped_lock.

Например, корректный доступ к данным (а точнее их изменение) может реализовываться с помощью методов std::mutex: lock() и unlock(). Более того, для удобства существует специальный класс std::lock_guard, конструктор которого автоматически будет вызывать lock при создании объекта класса, а деструктор автоматически будет вызывать unlock() (если этого не было сделано ранее). std::lock_guard является довольно прямолинейной реализацией идиомы RAII. Так, он не способен работать с std::timed_mutex. Однако более продвинутым/гибким является класс std::unique_lock (правда, объекты этого класса будут занимать больше памяти и работа с ними будет производиться чуть дольше), который способен:

- принимать незахваченный мьютекс в конструкторе;
- захватывать и освобождать мьютекс непосредственными версиями lock:
  - std::adopt_lock;
  - std::deffer_lock;
  - std::try_to_lock.

- выполнять временной захват;
- быть перемещённым в другой объект std::unique_lock.

Существенно, что защиту с использованием мьютекса можно обойти с помощью указателя или ссылки (поскольку может производиться "косвенное"изменение данных).

Условные переменные (соответствующий класс – condition_variable) тоже представляют собой примитив синхронизации. Он используется вместе с std::mutex для блокировки одного или нескольких потоков до тех пор, пока другой поток не изменит общую/разделяемую переменную (условие) и не уведомит об этом условную переменную (экземпляр класса condition_variable). Дело в том, что часто условие ожидается более одного раза, а потому не может быть использован класс future (get() позволит дождаться завершения потока, т.е. получим одноразовое ожидание условия). Использование же комбинации (флаг + цикл + задержка) не выглядит хорошим решением (хотя бы потому, что задержка может оказаться слишком маленьким или слишком большим). Поэтому и используют условные переменные:

- std::cv – для std::mutex;
- std::cv_any – для мьютексподобного класса.

Поток, намеревающийся изменить общую переменную, должен:

- Получить std::mutex (обычно через std::lock_guard).
- Изменить общую переменную, пока мьютекс захвачен потоком.
- Вызвать notify_one или notify_all на std::condition_variable (после снятия блокировки).

Кроме того, при работе с условными переменными бывают случайные пробуждения, фильтровать которые можно с использованием дополнительных проверок.

3 На что влияет выбор гранулярности блокировки?

Гранулярность – количество простых операций, охватываемых областью блокировки мьютекса (т.е. количество операций между lock() и unlock()). Соответственно, выбор гранулярности определяет, насколько сильно распространяется защита на данную операцию. Если несколько простых операций образуют единую составную, то может иметь смысл блокировать всю составную операцию. Типичный пример – проблема проектирования интерфейса, например, стека: функции top() и pop() могут привести к проблематичному состоянию гонки при слишком малой гранулярности.

С другой стороны, слишком большая гранулярность при большом количестве разделяемых потоками данных может свести на нет все преимущества параллелизма, т.к. потоки вынуждены работать по очереди, даже если обращаются к разным элементам данных. Известный пример – использование единственной глобальной блокировки ядра в первых системах Linux. Это решение работало, но получалось, что производительность одной системы с двумя процессорами гораздо ниже, чем двух однопроцессорных систем, а уж сравнивать производительность четырехпроцессорной системы с четырьмя однопроцессорными вообще не имело смысла – конкуренция за ядро оказывалась настолько высока, что потоки, исполняемые дополнительными процессорами, не могли выполнять полезную работу. В последующих версиях Linux гранулярность блокировок ядра уменьшилась, и в результате производительность четырехпроцессорной системы приблизилась к идеалу – четырехкратной производительности однопроцессорной системы, так как конкуренция за ядро значительно снизилась.

При использовании мелкогранулярных схем блокирования иногда для защиты всех данных, участвующих в операции, приходится захватывать более одного мьютекса. Бывают случаи, когда лучше повысить гранулярность защищаемых данных, чтобы для их защиты хватило одного мьютекса. Но это не всегда желательно, например, если мьютексы защищают отдельные экземпляры класса. В таком случае блокировка "на уровень выше"означает одно из двух: передать ответственность за блокировку пользователю или завести один мьютекс, который будет защищать все экземпляры класса. Ни одно из этих решений не вызывает восторга (далее см. следующий вопрос). К тому же когда для защиты одной операции приходится использовать два или более мьютексов, всплывает очередная проблема: взаимоблокировка (см. следующий вопрос).

4 Когда возникает взаимоблокировка и как её предотвратить?

По природе своей взаимоблокировка почти противоположна гонке: если в случае гонки два потока состязаются, кто придет первым, то теперь каждый поток ждет другого, и в результате ни тот, ни другой не могут продвинуться ни на шаг. Итак, при захвате мьютексов оба потока для выполнения некоторой операции должны захватить два мьютекса, но может сложиться так, что каждый поток захватил только один мьютекс и ждёт другого. Ни один поток не может продолжить, так как каждый ждёт, пока другой освободит нужный ему мьютекс. Такая ситуация и называется взаимоблокировкой; это самая трудная из проблем, возникающих, когда для выполнения операции требуется захватить более одного мьютекса.

Общая рекомендация, как избежать взаимоблокировок, заключается в том, чтобы всегда захватывать мьютексы в одном и том же порядке , – если мьютекс A всегда захватывается раньше мьютекса B, то взаимоблокировка не возникнет. Иногда это просто, потому что мьютексы служат разным целям, а иногда вовсе нет, например, если каждый мьютекс защищает отдельный объект одного и того же класса. Рассмотрим, к примеру, операцию сравнения двух объектов одного класса. Чтобы сравнению не мешала одновременная модификация, необходимо захватить мьютексы для обоих объектов. Однако, если выбрать какой-то определенный порядок (например, сначала захватывается мьютекс для объекта, переданного в первом параметре, а потом – для объекта, переданного во втором параметре), то легко можно получить результат, обратный желаемому: стоит двум потокам вызвать функцию сравнения, передав ей одни и те же объекты в разном порядке, как мы получим взаимоблокировку!

К счастью, в стандартной библиотеке есть на этот случай лекарство в виде функции std::lock, которая умеет захватывать сразу два и более мьютексов без риска получить взаимоблокировку.

Итак, предотвратить взаимоблокировку можно, если:

- Не делать вложенных блокировок. Идея состоит в том, чтобы не захватывать мьютекс, если уже захвачен какой-то другой. Если строго придерживаться этой рекомендации, то взаимоблокировка, обусловленная одними лишь захватами мьютексов, никогда не возникнет, потому что каждый поток в любой момент времени владеет не более чем одним мьютексом.

- Использовать мьютексы с приоритетами, т.к. при вызове пользовательского кода, если предварительно не освобождён мьютекс, а этот код захватывает какой-то мьютекс, то окажется нарушенной рекомендация избегать вложенных блокировок, и может возникнуть взаимоблокировка. (Правда, иногда избежать этого невозможно).

- Гарантировать одинаковый порядок блокировки мьютексов и выполнения программы (см. пример выше). Можно также воспользоваться иерархией блокировок.

- Использовать std::lock.

Стоит отметить, что применение данных рекомендаций не ограничивается блокировками (взаимоблокировка может возникать не только вследствие захвата мьютекса, а вообще в любой конструкции синхронизации, сопровождающейся циклом ожидания). Следует по возможности избегать вложенных блокировок, и точно так же не рекомендуется ждать поток, удерживая мьютекс, потому что этому потоку может потребоваться тот же самый мьютекс для продолжения работы. Аналогично, если вы собираетесь ждать завершения потока, то будет разумно определить иерархию потоков, так чтобы любой поток мог ждать только завершения потоков, находящихся ниже него в иерархии. Простой способ реализовать эту идею – сделать так, чтобы присоединение потоков происходило в той же функции, которая их запускала

5 Что есть атомарная операция и атомарный тип данных?

Под атомарными понимаются неделимые операции. Ни из одного потока в системе невозможно увидеть, что такая операция выполнена наполовину: она либо выполнена целиком, либо не выполнена вовсе. Если операция загрузки, которая читает значение объекта, атомарна, и все операции модификации этого объекта также атомарны, то в результате загрузки будет получено либо начальное значение объекта, либо значение, сохраненное в нем после одной из модификаций. И наоборот, если операция не атомарная, то другой поток может видеть, что она выполнена частично.

Атомарный объект – это такой объект, операции над которым можно считать неделимыми. Все стандартные атомарные типы определены в заголовке <atomic>. Любые операции над такими типами атомарны, и только операции над этими типами атомарны в смысле принятого в языке определения, хотя мьютексы позволяют реализовать кажущуюся атомарность других операций. На самом деле, и сами стандартные атомарные типы могут пользоваться такой эмуляцией: почти во всех имеется функция-член is_lock_free(), которая позволяет пользователю узнать, выполняются ли операции над данным типом с помощью действительно атомарных команд (x.is_lock_free() возвращает true) или с применением некоторой внутренней для компилятора и библиотеки блокировки (x.is_lock_free() возвращает false).

Единственный тип, в котором функция-член is_lock_free() отсутствует, – это std::atomic_flag. В действительности это простой булевский флаг, а операции над этим типом обязаны быть свободными от блокировок; если имеется простой свободный от блокировок булевский флаг, то на его основе можно реализовать простую блокировку и, значит, все остальные атомарные типы. После инициализации объект типа std::atomic_flag сброшен, и для него определены всего две операции: проверить и установить (функция-член test_and_set()) и очистить (функция-член clear()).

Доступ ко всем остальным атомарным типам производится с помощью специализаций шаблона класса std::atomic<>; их функциональность несколько богаче, но они необязательно свободны от блокировок (как было объяснено выше). Интерфейс каждой специализации отражает свойства типа.

Список литературы

[1] Конспект. И.С. Макаров.

[2] Параллельное программирование на C++ в действии [Практика разработки многопоточных программ]. Уильямс Энтони Д.

[3] https://en.cppreference.com/w/