Seminar 18.

26 марта 2023 г.

1 Что такое контекстное переключение задач?

Существует два основных типа параллелизма: мнимый и истинный. Мнимый параллелизм наиболее наглядно проявляется, когда у системы есть лишь один исполнитель. Если этот исполнитель задействован в решении нескольких задач, то под каждую задачу отводится некоторое время выполнения, после чего происходит переключение на иную задачу (с сохранением параметров предыдущей задачи, чтобы к ней можно было вернуться). Такое переключение задач называется контекстным.

Например, мнимый параллелизм может быть организован с помощью некой сущности, называемой планировщиком, который как раз выполняет (независимо, но, возможно, опираясь на рекомендации исполняемых программ) контекстное переключение задач:

- сохранение состояния задачи;
- сохранение счётчика команд;
- определение следующей задачи;
- загрузка состояния задачи.

Также мнимый параллелизм может быть организован без планировщика. Тогда контроль выполнения задач осуществляется самими программами, а переход к новой задаче производится с помощью специального метода yield().

2 Назовите основные подходы к организации параллелизма.

Параллелизм может быть организован с помощью процессов (выполнений инструкций программ) и потоков (части процессов). Использование процессов:

- безопасно;
- сложно;
- медленно;
- ресурсозатратно.

Это связано со свойствами взаимодействия между процессами (IPC). Дело в том, что ОС должна обеспечить защиту процессов, так чтобы ни один не мог случайно изменить данные, принадлежащие другому. Кроме того, существуют неустранимые накладные расходы на запуск

нескольких процессов: для запуска процесса требуется время, ОС должна выделить внутренние ресурсы для управления процессом и т.д. Т.о., IPC реализуется посредством специальных механизмов ОС (и является основой для разграничения адресного пространства между процессами). В связи со свойствами процессов и организации их работы последних достаточно мало.

Потоки являются частью какого-либо процесса (поэтому их может быть много). Потоки можно считать облегчёнными процессами – каждый из них работает независимо от всех остальных, и все они могут выполнять разные последовательности команд. Однако все принадлежащие процессу потоки разделяют общее адресное пространство и имеют прямой доступ к большей части данных, а потому должны бытьь синхронизированы. Зато благодаря общему адресному пространству и отсутствию защиты данных от доступа со стороны разных потоков накладные расходы, связанные с наличием нескольких потоков, существенно меньше, так как на долю операционной системы выпадает гораздо меньше учетной работы, чем в случае нескольких процессов. Использование потоков является:

- опасным;
- простым;
- быстрым;
- незатратным.

В целом, параллелизм используется для:

- разделения обязанностей и организации ситемы;
- повышения производительности путём:
  - разбиения по задачам;
  - разбиения по данным.

3 Что может влиять на производительность параллельных алгоритмов?

На производительность параллельных алгоритмов могут влиять:

- Количество исполнителей и контекстное переключение. Они определяют число потоков, которое может быть эффективно использовано программой.
- Конкуренция за данные (перебрасывание кэша). Если два потока, одновременно выполняющиеся на разных процессорах, читают одни и те же данные, то обычно проблемы не возникает – данные просто копируются в кэши каждого процессора. Но если один поток модифицирует данные, то изменение должно попасть в кэш другого процессора, а на это требуется время. В зависимости от характера операций в двух потоках и от упорядочения доступа к памяти, модификация может привести к тому, что один процессор должен будет остановиться и подождать, пока аппаратура распространит изменение.
- Ложное разделение (строки кэша). Поскольку аппаратный кэш оперирует блоками памяти, небольшие элементы данных, находящиеся в смежных ячейках, часто оказываются в одной строке кэша. Иногда это хорошо: с точки зрения производительности лучше, когда данные, к которым обращается поток, находятся в одной, а не в нескольких строках кэша. Однако, если данные, оказавшиеся в одной строке кэша, логически не связаны

между собой и к ним обращаются разные потоки, то возможно возникновение неприятной проблемы. Например, если в строке кэша помещается несколько переменных. Если эти переменные принадлежат разным потокам, то поскольку каждый из них обращается лишь к своему элементу, перебрасывание кэша будет иметь место. Строка кэша разделяется, хотя каждый элемент данных принадлежит только одному потоку. Отсюда и название ложное разделение. Решение заключается в том, чтобы структурировать данные таким образом, чтобы элементы, к которым обращается один поток, находились в памяти рядом (и, следовательно, с большей вероятностью попали в одну строку кэша), а элементы, к которым обращаются разные потоки, отстояли далеко друг от друга и попадали в разные строки кэша.

- Локальность данных. В противоположность ложному разделению, локальность данных – расположение данных, к которым обращается поток, так, что они разбросаны по памяти. Из-за этого велика вероятность, что данные находятся в разных строках кэша (т.е. разбросаны). Поэтому из памяти в кэш процессора приходится загружать больше строк кэша, что увеличивает задержку памяти и снижает общую производительность.

Вероятно, стоит также отметить закон Амдала, отражающий возможность увеличения производительности программы с использованием параллельных алгоритмов. В принципе это касается количества исполнителей и контекстного переключения, но также и структуры самой программы (алгоритма).

4 Как в стандартной библиотеке реализована концепция асинхронного исполнения?

Предположим, у нас есть некоторая задача. С помощью концепции асинхронного исполнения мы можем запустить её исполнение в рамках программы и далее продолжить работу над оставшимися задачами, условно говоря, забыв о данной. Результат мы получим когда-то позже (можно проконтролировать когда). Итак, начало выполнения асинхронных операций требует достаточно мало времени по сравнению с временем выполнения самих задач, отданных потокам, что позволяет выполнять множество асинхронных операций одновременно.

В C++ класс std::thread не позволяет вернуть результат работы потока (а также отловить ошибку вне созданного потока), а потому в дело вступает класс std::future, обеспечивающий механизм доступа к результату асинхронных операций. Используя метод get(), можно получить результат работы функции, которая была запущена в отдельном потоке с помощью std::future. Сам же future object может быть создан несколькими способами:

- С помощью функции std::async(). Она имеет две политики запуска std::launch::async (тут же будет запускать выполнение задачи в отдельном потоке) и std::launch::deferred (выполнение задачи откладывается до момента запроса результата).

- С помощью класса std::packaged_task, из объектов которого можно получать future object, используя метод get_future.

- С помощью класса std::promise, из объектов которого можно получать future object, используя метод get_future. Это способ является наиболее общим (или наиболее низкоуровневым) из представленных. Именно он позволяет сохранять значения рассчитанные в рамках потока (set_value), или сохранять исключение, полученное при выполнении задачи в потоке (set_exception). Причём это исключение будет получено только в результате применения метода get() к future object.

Также с для future object существуют методы wait() и valid(), определяющие состояние последнего. Первая приостанавливает основно поток, пока не будет выполнена задача потока, запущенного с помощью future object. Вторая проверяет запущен ли данным future object какой-либо поток. Хороший тон, проверять future object с помощью valid(), чтобы не поймать UB.

5 Что нужно учитывать при замене последовательной реализации алгоритма на параллельную?

При замене последовательной реализации алгоритма на параллельную нужно учитывать объем вычислений, т.к. при простых операциях ожидание основной памяти очень долгое и последовательная версия может обгонять параллельную, поэтому следует проводить непосредственные замеры времени работы. Кроме того, хорошо писать параллельный алгоритм безопасно, организовав корректную обработку ошибок (или исключений) в программе. Если уточнять, то важным являются:

- время подготовки;
- количество команд;
- количество элементов;
- характер задачи;
- работа с памятью.

Для определения наилучших параметров проще всего провести замеры для последовательной и параллельной имплементации.

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/