Параллелизм

#Потоки #scheduling #Атомарность #Синхронизация

Конкурентность - не по порядку

Потоки

Потоки — абстракция операционной системы, позволяющая выполнять некие куски кода параллельно. Каждый поток имеет «контекст», в который входит кусок памяти под стек и копию регистров процессора. Очевидно, что потоков в системе может быть запущено куда больше, чем у нас имеется процессоров и ядер, поэтому операционная система занимается тем, что schedule'ит (занимается диспетчеризацией, планированием) выполнение процессов, восстанавливая регистры в ядрах процессора для работы некоторого потока некоторое время, затем сохраняет обратно и даёт выполняться следующему.

Планирование должно быть хорошим, чтобы каждый поток работал примерно одинаковое время (но при этом достаточно малое, т.е. нас не удовлетворит, если каждый поток будет работать по несколько секунд), а так же само планирование занимало как можно меньше времени. Тут мы, традиционно, приходим к балансу latency/throughput: если давать потокам работать очень маленькие промежутки времени — задержки будут минимальны, но отношение времени полезной работы к времени на переключение контекстов уменьшится, и в итоге полезной работы будет выполнено меньше, а если слишком много — задержки станут заметны, что может проявиться на пользовательском интерфейсе/IO.

Тут же хотелось бы отметить довольно важную вещь, к которой будем отсылаться позже. Потоки можно погружать в сон. Мы можем попросить операционную систему усыпить поток на некоторое время. Тогда она может учитывать это при планировании, исключив этот поток из очереди на получение процессорного времени, что даст больше ресурсов другим потокам, либо снизит энергопотребление. Кроме того, в теории можно настраивать приоритеты, чтобы некоторые потоки работали дольше, либо чаще, чем другие.

Синхронизация

Ядер у нас стало много, а память как была одна сплошная, так и осталась (хотя в современной организации памяти можно ногу сломать, там и физическая, и виртуальная, и страницы туда-сюда ходят, что-то где-то кешируется, проверяется на ошибки, странно, что всё это до сих пор работает вообще).

И когда мы из нескольких потоков начинаем менять одну и ту же память — всё может пойти совершенно не так, как мы этого хотели. Могут возникать «гонки», когда в зависимости от того, в каком порядке выполнятся операции, вас ожидает разный результат, что может приводить к потере данных. И, что ещё хуже — данные могут совсем портиться в некоторых случаях. К примеру, если мы записываем некоторое значение на невыровненный адрес памяти, нам нужно записать в две ячейки памяти, и может получиться так, что один поток запишет в одну ячейку, а другой в другую. И если вы записывали указатель, или индекс массива — то скорее всего вы получите падение программы через какое-то время, когда попытаетесь обратиться к чужой памяти.

Чтобы этого не происходило, нужно использовать примитивы синхронизации, чтобы получать гарантии того, что некоторые блоки кода будут выполняться в строгом порядке относительно каких-то других блоков кода.

Spinlock

Самый простой примитив: у нас есть boolean'овая переменная, если *true* — значит блокировка кем-то была получена, *false* — свободна. И два метода: **Lock**, **Unlock**. **Unlock** устанавливает значение в *false*. **Lock** в цикле делает либо **TAS**, либо **CAS** (об этом будет далее), чтобы атомарно сменить *false* на *true*, и пытаться до тех пор, пока не получится.

Имея такую штуку мы можем делать блоки кода, которые будут выполняться эксклюзивно (т.е. пока один выполняется, другой стартовать не может). В начале блока он выполняет метод **Lock**, в конце **Unlock**. Важно, чтобы он не попытался сделать **Lock** второй раз, иначе некому будет сделать **Unlock** и получится **deadlock**.

Возможно навешивание плюшек. Например, сохранение идентификатора потока, кто захватил блокировку, чтобы никто кроме захватившего потока, не мог сделать **Unlock**, и если мы сделали **Lock** второй раз — он не зациклился, такая реализация это называется **«мьютексом»**. А если вместо boolean'а у нас беззнаковое число, а **Lock** ждёт числа больше нуля, уменьшая его на единицу, получается **«семафор»**, позволяющий работать некоторому заданному числу блоков одновременно, но не более.

Поскольку блокировка работает в бесконечном цикле она просто «жгёт» ресурсы, не делая ничего полезного. Существуют реализации, которые при неудачах говорят

планировщику *«я всё, передай остаток моего времени другим потокам»*. Или, например, есть **«адаптивные мьютексы»**, которые при попытке получить блокировку, которую удерживает поток, который в данный момент выполняется — использует spinlock, а если не выполняется — передаёт выполнение другим потокам. Это логично, т.к. если поток, который может освободить ресурс сейчас работает, то у нас есть шансы, что он вот-вот освободится. А если он не выполняется — есть смысл подождать чуть больше и передать выполнение другому потоку, возможно даже тому, кого мы ждём.

По возможности стоит избегать использования спинлоков, их можно использовать только если блоки кода, которые они защищают, выполняются невероятно быстро, что шансы коллизии крайне малы. Иначе мы можем затратить довольно большое количество ресурсов просто на обогрев комнаты.

Лучшая альтернатива блокировкам — усыпление потока, чтобы планировщик исключил его из претендентов на процессорное время, а затем просьба из другого потока разбудить первый поток обратно, чтобы он обработал новые данные, которые уже на месте. Но это справедливо только для очень длительных ожиданий, если же скорости освобождения блокировок соразмерны с одним «тактом» выполнения потока — производительнее использовать спинлоки.

Барьер памяти

В погоне за скоростью работы, архитекторы процессоров создали очень много тёмной магии, которая пытается предсказывать будущее и может переставлять процессорные инструкции местами, если они не зависят друг от друга. Если у вас каждый процессор работает только со своей памятью — вы никогда не заметите, что инструкции *a = 5* и *b = 8* выполнились в другом порядке, не так, как вы написали в коде.

Но при параллельном исполнении это может привести к интересным случаям. Пример из википедии:

```
// Processor #1:
while (f == 0);
// Memory fence required here
print(x);
```

```
// Processor #2:
x = 42;
```

```
// Memory fence required here
f = 1;
```

Казалось бы, что здесь может пойти не так? Цикл завершится, когда **f** будет присвоена единица, на момент чего переменная **x** вроде как уже будет равна **42**. Но нет, процессор может произвольно менять местами такие инструкции и в итоге сначала может выполниться `f = 1`, а затем `x = 42` и может вывестись не **42**, а что-нибудь другое. И даже более того. Если с порядком записи всё нормально, может быть изменён порядок чтения, где значение **x** прочитается перед тем, как мы войдём в цикл.

Атомарность

Поскольку некоторые операции с памятью могут быть потоконебезопасны (например, прибавить к числу другое число и сохранить его обратно), существуют операции для проведения атомарных операций, для которых есть специальные процессорные инструкции.

Наиболее часто используемые:

- **Test-and-Set** — записывает значение, возвращает предыдущее.
- **Compare-and-Swap** — записывает значение только в том случае, если текущее значение переменной совпадает с неким ожидаемым, возвращает успех записи. Возвращаясь к нашему спинлоку, операция получения блокировки может быть реализована как при помощи **TAS**:

```
void lock() {
// Мы будем записывать 1 до тех пор, пока между нашими выполнениям кто-то не запишет в неё 0
while (test_and_set(&isLocked, 1) == 0);
}
```

Так и при помощи **CAS**:

```
void lock() {
// Пытаемся записать 1, ожидая 0
while (!compare_and_swap(&isLocked, 0, 1))
}
```