Память в JVM, часть 2

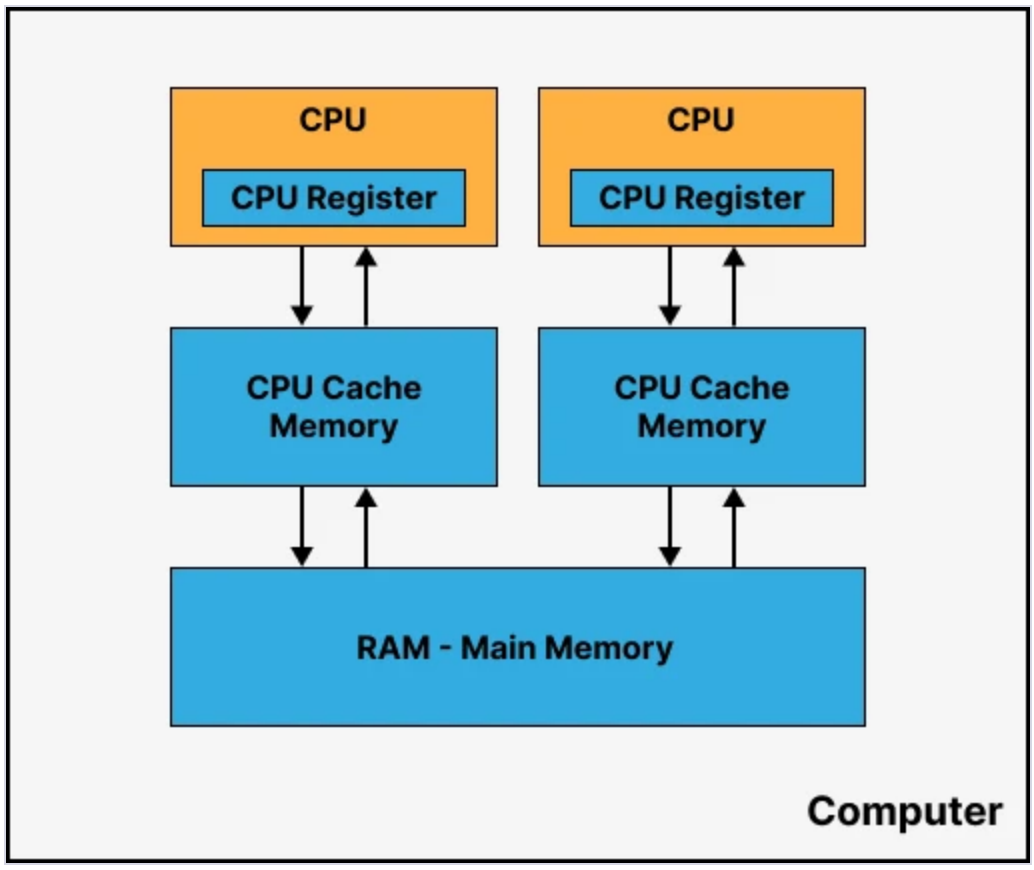
JSP & Servlets
18 уровень, 1 лекция

ОТКРЫТА

Аппаратная архитектура памяти

Современная аппаратная архитектура памяти отличается от внутренней Java-модели памяти. Поэтому нужно аппаратную архитектуру понимать, чтобы знать, как Java-модель работает с ней. В этом разделе описывается общая аппаратная архитектура памяти, а в следующем разделе описывается, как с ней работает Java.

Вот упрощенная схема аппаратной архитектуры современного компьютера:



В современном мире у компьютера 2 или более процессора и это уже норма. Некоторые из этих процессоров также могут иметь несколько ядер. На таких компьютерах возможно одновременное выполнение нескольких потоков. Каждое ядро процессора способно выполнять один поток в любой момент времени. Это значит, что любое Java-приложение является априори многопоточным и внутри вашей программы может быть запущено по одному потоку на ядро процессора одновременно.

Ядро процессора содержит набор регистров, которые находятся в его памяти (внутри ядра). Оно выполняет операции над данными регистра намного быстрее, чем над данными, которые находятся в основной памяти компьютера (ОЗУ). Это связано с тем, что процессор может получить доступ к этим регистрам гораздо быстрее.

Каждый ЦП тоже может иметь свой слой кэш-памяти. Большинство современных процессоров его имеют. К своей кэш-памяти процессор может получить доступ намного быстрее, чем к основной памяти, но не так быстро, как к своим внутренним регистрам. Значение скорости доступа к кэш-памяти примерно между скоростями доступа к основной памяти и к внутренним регистрам.

Более того, у процессоров имеет место быть многоуровневый кэш. Но это не так важно знать, чтобы понять, как Java-модель памяти взаимодействует с аппаратной памятью. Важно знать, что процессоры могут иметь некоторый уровень кэш-памяти.

Любой компьютер точно также содержит ОЗУ (область основной памяти). Все ядра могут получить доступ к основной памяти. Основная область памяти обычно намного больше, чем кэш-память ядер процессоров.

В момент, когда процессору нужен доступ к основной памяти, он считывает ее часть в свою кэш-память. Он может также считывать часть данных из кэша в свои внутренние регистры и затем выполнять операции над ними. Когда ЦПУ необходимо

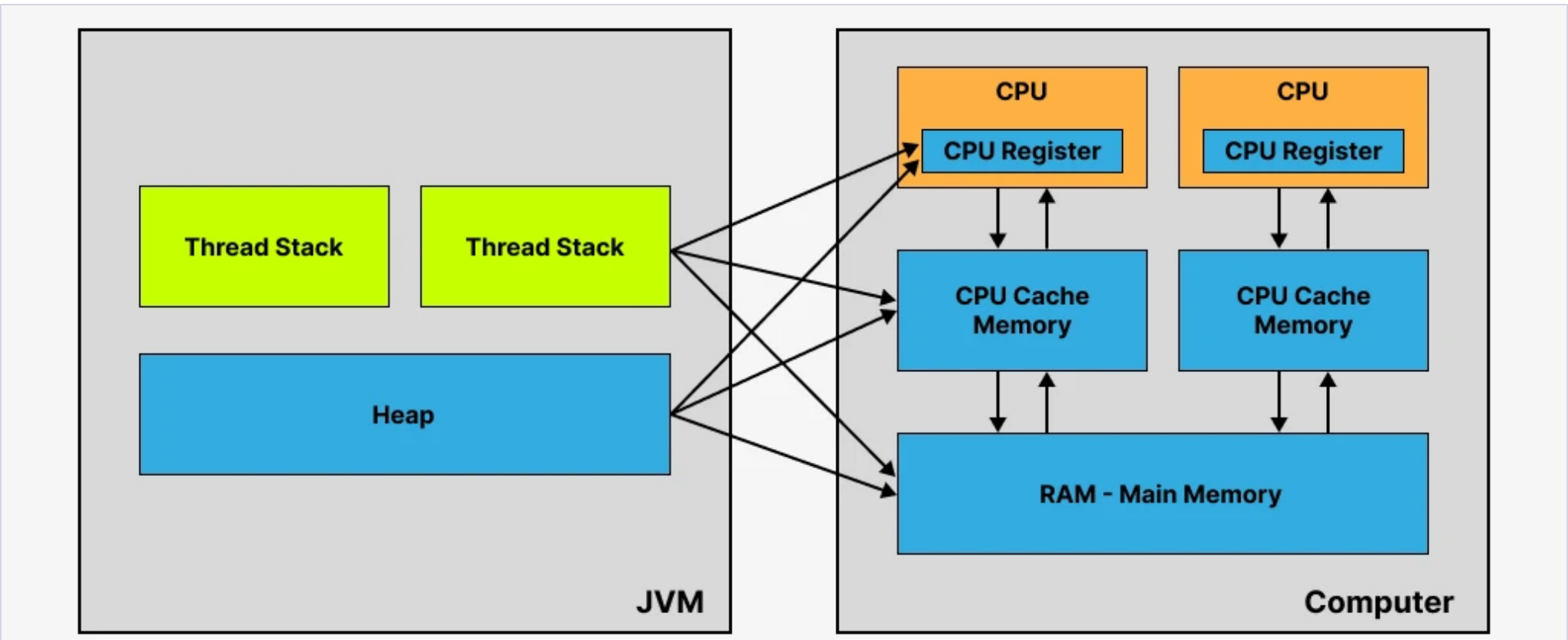
будет записать результат опять в основную память, он сбросит данные из своего внутреннего регистра в кэш-память, и в какой-то момент, в основную память.

Данные, хранящиеся в кэш-памяти, в обычном случае сбрасываются обратно в основную память, когда процессору необходимо сохранить в кэш-памяти что-то еще. Кэш имеет возможность очищать свою память и записывать данные одновременно. У процессора нет необходимости читать или записывать полный кэш каждый раз во время обновления. Обычно кэш обновляется небольшими блоками памяти, они называются “строка кэша”. Одна или несколько “строк кэша” могут быть считаны в кэш-память, и одна или более строк кэша могут быть сброшены назад в основную память.

Совмещение Java-модели памяти и аппаратной архитектуры памяти

Как уже упоминалось, Java-модель памяти и аппаратная архитектура памяти различны. Аппаратная архитектура не различает стеки потоков и кучу. На оборудовании стек потоков и HEAP (куча) находятся в основной памяти.

Части стеков и кучи потоков могут иногда присутствовать в кэшах и внутренних регистрах ЦП. Это показано на диаграмме:



Когда объекты и переменные могут храниться в различных областях памяти компьютера, могут возникнуть определенные проблемы. Вот две основные:

- Видимость изменений, которые произвел поток над общими переменными.
- Состояние гонки при чтении, проверке и записи общих переменных.

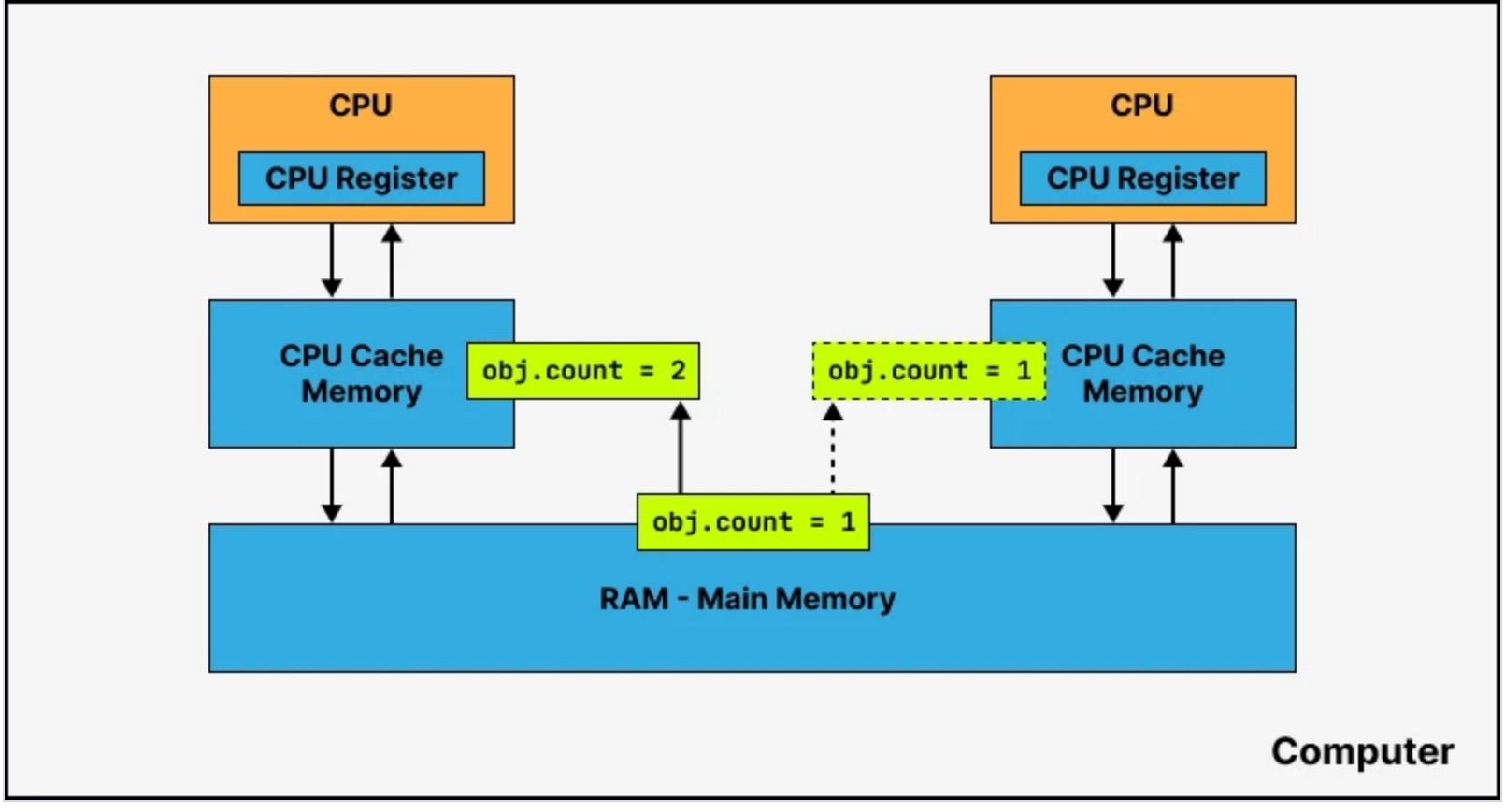
Обе эти проблемы я объясню далее.

Видимость общих объектов

Если два или более потока делят между собой объект без надлежащего использования volatile-объявления или синхронизации, то изменения общего объекта, сделанные одним потоком, могут быть невидимы для других потоков.

Представь, что общий объект изначально хранится в основной памяти. Поток, выполняющийся на ЦП, считывает общий объект в кэш этого же ЦП. Там он вносит изменения в объект. Пока кэш ЦП не был сброшен в основную память, измененная версия общего объекта не видна потокам, работающим на других ЦП. Таким образом, каждый поток может получить свою собственную копию общего объекта, каждая копия будет находиться в отдельном кэше ЦП.

Следующая диаграмма иллюстрирует набросок этой ситуации. Один поток, работающий на левом ЦП, копирует в его кэш общий объект и изменяет значение переменной count на 2. Это изменение невидимо для других потоков, работающих на правом ЦП, поскольку обновление для count еще не было сброшено обратно в основную память.



Для того, чтобы решить эту проблему, вы можете использовать ключевое слово volatile при объявлении переменной. Оно может гарантировать, что данная переменная считывается непосредственно из основной памяти и всегда записывается обратно в основную память, когда обновляется.

Состояние гонки (race condition)

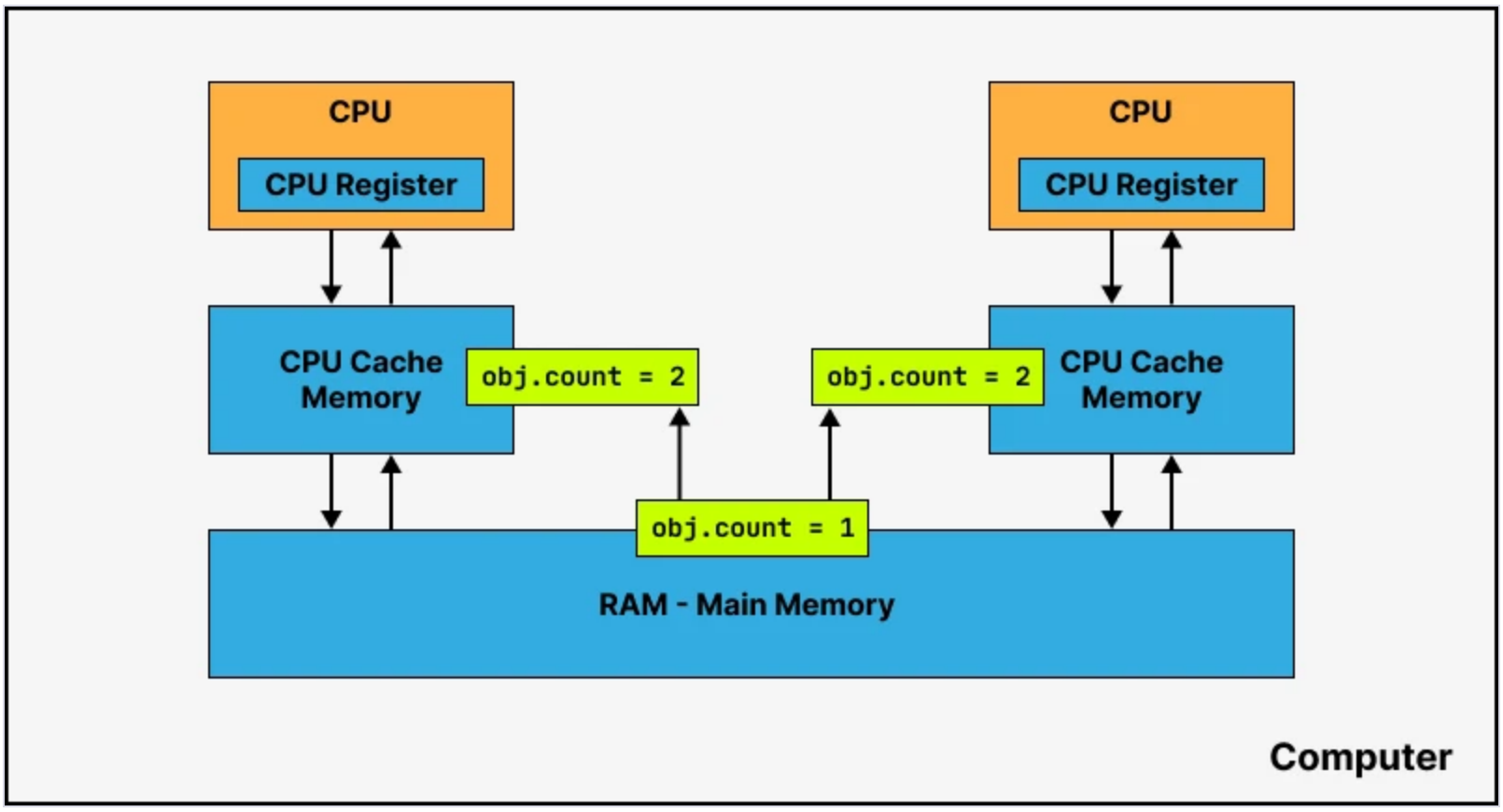
Если два или более потоков совместно используют один объект и более одного потока обновляют переменные в этом общем объекте, то может возникнуть состояние гонки.

Представьте, что поток A считывает переменную count общего объекта в кэш своего процессора. Представьте также, что поток B делает то же самое, но в кэш другого процессора. Теперь поток A прибавляет 1 к значению переменной count, и поток B делает то же самое. Теперь переменная была увеличена дважды — отдельно по +1 в кэше каждого процессора.

Если бы эти приращения были выполнены последовательно, переменная count была бы увеличена в два раза и обратно в основную память было бы записано (исходное значение + 2).

Тем не менее, два приращения были выполнены одновременно без надлежащей синхронизации. Независимо от того, какой из потоков (A или B), записывает свою обновленную версию count в основную память, новое значение будет только на 1 больше исходного значения, несмотря на два приращения.

Эта диаграмма иллюстрирует возникновение проблемы с состоянием гонки, которое описано выше:



Для решения этой проблемы вы можете использовать синхронизированный блок Java. Синхронизированный блок гарантирует, что только один поток может войти в данный критический раздел кода в любой момент времени.

Синхронизированные блоки также гарантируют, что все переменные, к которым обращаются внутри синхронизированного блока, будут считаны из основной памяти, и когда поток выйдет из синхронизированного блока, все обновленные переменные будут снова сброшены в основную память, независимо от того, объявлена ли переменная как volatile или нет.

‹ Предыдущая лекция

Следующая лекция ›

JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

Русский

СКАЧИВАЙТЕ НАШИ ПРИЛОЖЕНИЯ

ДОСТУПНО В Google Play

Загрузите в App Store