Методические указания к семинару
по теме «Моделирование конкурентных процессов»

Проблему синхронизации в ситуациях, когда несколько задач конкурируют за блокировки, можно проиллюстрировать с помощью «задачи об обедающих философах», которую сформулировал Эдсгер Дейкстра в 1965 году. Этот пример часто служит стандартным тестовым сценарием, с помощью которого оцениваются методы синхронизации.

Сценарий: пять молчаливых философов сидят за круглым столом перед блюдом с роллами и суши. Между каждой парой ближайших философов лежит палочка для азиатской еды. Философы занимаются тем, что у них обычно получается лучше всего: они размышляют и едят. Каждую палочку может держать только один философ, так что никто из них не может использовать палочку, которую в данный момент держит кто-то другой. Философ может брать палочки, которые лежат справа и слева от него, может делать это только тогда, когда палочки доступны, и не может начать есть, пока не возьмет обе палочки. После того как философ закончит есть, он должен положить обе палочки, чтобы их могли взять соседи.

Требуется разработать алгоритм, чтобы каждый философ мог переключаться между двумя состояниями: он либо ест, либо размышляет. Предполагается, что ни один философ не знает, когда другие захотят есть или размышлять, таким образом, получается конкурентная система.

Взятие суши с блюда является критической секцией, и чтобы ее защитить, можно разработать процесс взаимоисключения, используя две палочки как мьютексы. Таким образом, когда философ хочет взять суши, он сначала берет палочку слева, если она доступна, и устанавливает для нее блокировку. Затем он берет палочку справа, если она доступна, и тоже устанавливает для нее блокировку. Теперь философ держит две палочки: значит, он находится в критической секции и может съесть еду. После этого он кладет на стол правую палочку, чтобы снять блокировку, а затем кладет левую палочку. Наконец, поев, он возвращается к философским размышлениям.

В коде примера поток Philosopher представляет одного философа. Он содержит его имя и два мьютекса: forkOnLeft и forkOnRight (left_chopstick и right_chopstick), которые философ захватывает в указанном порядке.

Общая переменная sushi представляет оставшиеся суши на общем блюде. В цикле while философы берут суши, пока они остаются на блюде. В цикле философ устанавливает блокировку сначала для левой, а потом для правой палочки. Затем, если на блюде еще остались суши, философ берет один из них, уменьшая переменную sushi.

Философы переключаются между едой и размышлениями. Но поскольку они ведут себя как конкурентные задачи, ни один из них не знает, когда другие захотят есть или размышлять, что может вызвать затруднения. Какие проблемы могут возникнуть при выполнении этого кода и как их можно решить?

Программа может не завершаться: она застревает на одном месте, а на блюде все еще остаются суши. Это возможная ситуация, если, допустим, первый философ проголодался, и он берет палочку A. В то же время второй философ тоже хочет перекусить и берет палочку B. Каждый поток удерживает одну из двух блокировок, которые ему нужны, но оба бесконечно ожидают, пока другой поток освободит оставшуюся блокировку.

Это пример так называемой взаимной блокировки (deadlock). Эта ситуация заключается в том, что несколько задач ожидают ресурсов, которые заняты другими задачами, и ни одна из них не может продолжить выполнение. Программа навечно застревает в этом состоянии, поэтому ее приходится прерывать вручную.

Если запустить программу еще раз, она приведет к взаимной блокировке после другого количества еды - точное значение каждый раз зависит от того, как система планирует задачи.

Может повезти, и такая ситуация никогда не проявится, но даже если взаимная блокировка может возникнуть хотя бы теоретически, ее стоит предотвратить. Опасность взаимной блокировки возникает каждый раз, когда

задача пытается заполучить сразу несколько блокировок. Как избежать взаимных блокировок - типичная проблема в конкурентных программах, где критические секции кода защищаются с помощью мьютексов.

В реальных программах никогда не рассчитывайте на конкретный порядок выполнения. Если в программе выполняются несколько потоков, то порядок их выполнения не детерминирован. Если важно, в каком порядке выполняются потоки относительно друг друга, необходимо применять синхронизацию. Но если нужна оптимальная производительность, следует избегать синхронизации, насколько это возможно. В частности, стоит создавать задачи с высокой детализацией вычислений, которым не нужна синхронизация; это позволит ядрам обрабатывать каждую назначенную им задачу как можно быстрее.

Вернемся к задаче с философами. Чтобы предотвратить взаимную блокировку, можно предусмотреть, чтобы каждый философ мог взять либо две палочки, либо ни одной. Для этого нужно добавить в систему арбитра, который отвечает за палочки (назовем его официант). Чтобы взять палочку, философ сначала спрашивает разрешения у официанта. Официант дает разрешение только одному философу, пока тот не возьмет обе палочки. В любой момент философ может вернуть палочки обратно.

Функции официанта можно реализовать с помощью еще одной блокировки. Поскольку это решение вводит новую центральную сущность (официанта), оно может ограничить конкурентность: если философ ест, а один из его соседей запрашивает палочки, то всем остальным философам придется ждать, пока этот запрос будет выполнен, даже если палочки им доступны. В реальной компьютерной системе арбитр делает практически то же самое, упорядочивая доступ со стороны потоков-исполнителей. Такой подход снижает уровень конкурентности, и это надежный, но не самый эффективный вариант.

Можно установить приоритеты для блокировок, чтобы философы сначала пытались брать одну и ту же палочку. Тогда проблема взаимной блокировки не возникнет, потому что философы будут конкурировать за одну первую блокировку.

В этом случае оба философа должны договориться, что из двух палочек, которые они собираются использовать, палочку с бóльшим приоритетом всегда нужно брать первой. В нашем случае оба философа одновременно конкурируют за высокоприоритетную палочку. Когда один философ побеждает и берет ее, на столе остается только палочка с меньшим приоритетом. Поскольку философы согласились сначала брать высокоприоритетную палочку, второй философ не сможет взять оставшуюся. И у философа, который взял первую палочку, теперь есть доступ к палочке с меньшим приоритетом, так что он может начать есть обеими палочками.

Назначим приоритеты для палочек. Допустим, палочка A имеет больший приоритет, а палочка B – приоритет ниже. Каждый философ всегда должен сначала брать палочку с бóльшим приоритетом.

В коде примера философ 2 создавал проблему, когда брал палочку B до палочки A. Чтобы решить проблему, надо изменить порядок захвата палочек, не изменяя остальной код. Сначала берется палочка A, и только после этого палочка B.

Еще один способ предотвратить взаимные блокировки — установить таймаут для попыток блокирования. Если в течение определенного периода времени задаче не удается успешно установить все необходимые блокировки, мы заставляем поток освободить все блокировки, которые он удерживает в данный момент. Однако такой подход может создать другую проблему - активную блокировку.

Активная блокировка (livelock) отчасти похожа на взаимную блокировку: она возникает, когда две задачи конкурируют за один набор ресурсов. Однако при активной блокировке задача уступает свою первую блокировку в попытке установить вторую. Установив вторую блокировку, она снова пытается установить первую. Задача оказывается в том же заблокированном состоянии, потому что она тратит все время не на реальную работу, а на то, чтобы постоянно освобождать и пытаться установить то одну, то другую блокировку.

Представьте, что вы звоните по телефону, но вызываемый абонент в это же самое время пытается позвонить вам. Вы оба кладете трубку и одновременно повторяете попытку, что создает такую же ситуацию. В итоге никому из вас не удается дозвониться до другого.

Активная блокировка возникает тогда, когда задачи активно выполняют конкурентные операции, но выполнение программы от этого не продвигается. Эта ситуация похожа на взаимную блокировку, но отличается тем, что задачи ведут себя «вежливо», позволяя другим выполнить свою работу в первую очередь.

Кроме того, что при этом не выполняется никакой полезной работы, такая ситуация может привести к тому, что система будет перегружена из-за частых переключений контекста, что снизит ее производительность. К тому же планировщик ОС не сможет справедливо распределять время, потому что не знает, какая задача дольше всего ожидала общего ресурса.

Чтобы избежать блокировок этого типа, можно иерархически упорядочить последовательность блокировок, как мы это делали, чтобы избавиться от взаимных блокировок. В таком случае только один процесс сможет успешно установить обе блокировки.

Выявлять и устранять активные блокировки часто оказывается сложнее, чем взаимные блокировки. Дело в том, что в сценариях активной блокировки участвуют сложные динамические взаимодействия между несколькими сущностями, отчего бороться с ними становится непросто.

Активные блокировки составляют подмножество более обширной категории проблем нехватки ресурсов (starvation).

Некоторым философам могут вообще никогда не достаться обе палочки. Если так происходит относительно редко, с этим можно смириться, но когда палочек постоянно не хватает, философы останутся голодными.

Нехватка ресурсов не зря называется ресурсным голодом: поток буквально голодает, не получая доступа к запрашиваемым ресурсам, и никакого прогресса не происходит. Если другая «жадная» задача часто удерживает блокировку по общему ресурсу, то «голодная» задача так и не сможет выполниться.

Нехватка ресурсов обычно происходит из-за упрощенного алгоритма планирования, который является частью системы выполнения. Он должен равномерно распределять ресурсы между всеми задачами и, в частности, заботиться о том, чтобы ни одна задача не сталкивалась с постоянным блокированием доступа к ресурсам, которые ей необходимы. Обработка задач с разными приоритетами зависит от ОС, но планировщик обычно чаще запускает задачи с более высоким приоритетом, а это может вызвать нехватку ресурсов для низкоприоритетных задач. Другой фактор, который может привести к ресурсному голоду - слишком много задач в системе, когда задачам приходится чересчур долго ожидать выполнения.

С нехваткой ресурсов можно бороться, если применять алгоритм планирования с очередями приоритетов, где используется метод старения (aging): если поток долго ожидает в системе, его приоритет постепенно повышается. Со временем поток повышает приоритет настолько, чтобы система могла запланировать его для обращения к ресурсам и полноценно выполнить. Если вам захочется узнать больше про этот принцип, почитайте книгу Эндрю Таненбаума «Современные операционные системы» (Andrew Tanenbaum «Modern Operating Systems»).

Задание:

Дополните код модели регистрацией длительности интервалов еды каждого философа как для варианта модели управления потоками (thread), так и для варианта управления сопроцессами (async).

После завершения выполнения модели выведите на экран суммарную длительность еды каждого философа. Понаблюдайте за результатами при разных количествах философов (три варианта -от 3 до 10).