20.10.2020 ИС2к Операционные системы

Задача обедающих философов

В 1965 году Дейкстра сформулировал, а затем решил проблему синхронизации, названную им задачей обедающих философов. С тех пор все изобретатели очередного примитива синхронизации считали своим долгом продемонстрировать его наилучшие качества, показав, насколько элегантно с его помощью решается задача обедающих философов. Суть задачи довольно проста. Пять философов сидят за круглым столом, и у каждого из них есть тарелка спагетти. Эти спагетти настолько скользкие, что есть их можно только двумя вилками. Между каждыми двумя тарелками лежит одна вилка. Внешний вид стола показан на рис.

Жизнь философа состоит из чередующихся периодов приема пищи и размышлений. (Это положение из разряда абстракций даже в отношении философов, но вся их остальная деятельность к задаче не относится.) Когда философ становится голоден, он старается поочередно в любом порядке завладеть правой и левой вилками. Если ему удастся взять две вилки, он на некоторое время приступает к еде, затем кладет обе вилки на стол и продолжает размышления. Основной вопрос состоит в следующем: можно ли написать программу для каждого философа, который действует предполагаемым образом и никогда при этом не попадает в состояние зависания? (Можно заметить, что потребность в двух вилках выглядит несколько искусственно. Может быть, стоит переключиться с итальянской на китайскую кухню, заменив спагетти рисом, а вилки — палочками для еды?)

В листинге показано самое простое решение этой задачи. Процедура 1акв\_/отк ждет, пока вилка не освободится, и берет ее. К сожалению, это решение ошибочно.

Допустим, что все пять философов одновременно берут левую от себя вилку. Тогда никто из них не сможет взять правую вилку, что приведет к взаимной блокировке.

Можно без особого труда изменить программу так, чтобы после получения левой вилки программа проверяла доступность правой вилки. Если эта вилка недоступна, философ кладет на место левую вилку, ожидает какое-то время, а затем повторяет весь процесс. Это предложение также ошибочно, но уже по другой причине. При некоторой доле невезения все философы могут приступить к выполнению алгоритма одновременно и, взяв левые вилки и увидев, что правые вилки недоступны, опять одновременно положить на место левые вилки, и так до бесконечности. Подобная ситуация, при которой все программы бесконечно работают, но не могут добиться никакого прогресса, называется **голоданием**, или **зависанием процесса**. (Эта ситуация называется голоданием даже в том случае, если проблема возникает вне стен итальянского или китайского ресторана.)

Можно подумать, что стоит только заменить одинаковое для всех философов время ожидания после неудачной попытки взять правую вилку на случайное, и вероятность, что все они будут топтаться на месте хотя бы в течение часа, будет очень мала. Это правильное рассуждение, и практически для всех приложений повторная попытка через некоторое время не вызывает проблемы. К примеру, если в популярной локальной сети Ethernet два компьютера одновременно отправляют пакет данных, каждый из них выжидает случайный отрезок времени, а затем повторяет попытку. И на практике это решение неплохо работает. Но в некоторых приложениях может отдаваться предпочтение решению, которое срабатывает всегда и не может привести к отказу по причине неудачной череды случайных чисел. Вспомним о системе управления безопасностью атомной электростанции.

В программу, показанную в листинге можно внести улучшение, позволяющее избежать взаимной блокировки и зависания. Для этого нужно защитить пять операторов, следующих за вызовом *think,* двоичным семафором. Перед тем как брать вилки, философ должен выполнить в отношении переменной *mutex* операцию *down.* А после того как он положит вилки на место, он должен выполнить в отношении переменной *mutex* операцию *up.* С теоретической точки зрения это вполне достаточное решение. Но с практической — в нем не учтен вопрос производительности: в каждый момент времени может есть спагетти только один философ. А при наличии пяти вилок одновременно могут есть спагетти два философа.

Решение, представленное в листинге 2.14, не вызывает взаимной блокировки и допускает максимум параллелизма для произвольного числа философов. В нем используется массив state, в котором отслеживается, чем занимается философ: ест, размышляет или пытается поесть (пытается взять вилки). Перейти в состояние приема пищи философ может, только если в этом состоянии не находится ни один из его соседей. Соседи философа с номером i определяются макросами LEFT и RIGHT. Иными словами, если i равен 2, то LEFT равен 1, а RIGHT равен 3.

В программе используется массив семафоров, по одному семафору на каждого философа, поэтому голодный философ может блокироваться, если нужная ему вилка занята. Обратите внимание на то, что каждый процесс в качестве своей основной программы запускает процедуру philosopher, но остальные процедуры: take\_forks, put\_ forks и test — это обычные процедуры, а не отдельные процессы.

Относительно простое решение задачи достигается путём добавления официанта возле стола. Философы должны дожидаться разрешения официанта перед тем, как взять вилку. Поскольку официант знает, сколько вилок используется в данный момент, он может принимать решения относительно распределения вилок и тем самым предотвратить взаимную блокировку философов. Если четыре вилки из пяти уже используются, то следующий философ, запросивший вилку, вынужден будет ждать разрешения официанта — которое не будет получено, пока вилка не будет освобождена. Предполагается, что философ всегда пытается сначала взять левую вилку, а потом — правую (или наоборот), что упрощает логику. Официант работает, как [семафор](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) — понятие, введённое Дейкстрой в 1965 году.

Чтобы показать, как это решение работает, предположим, что философы обозначены от А до Д по часовой стрелке. Если философы А и В едят, то заняты четыре вилки. Философ Б сидит между А и В, так что ему недоступна ни одна из вилок. В то же время, философы Г и Д имеют доступ к одной неиспользуемой вилке между ними. Предположим, что философ Г хочет есть. Если он тут же берёт свободную вилку, то становится возможна взаимная блокировка философов. Если вместо этого он спрашивает разрешения у официанта, то тот просит его подождать — и можно быть уверенным в том, что как только пара вилок освободится, то по крайней мере один философ сможет взять две вилки. Таким образом, взаимная блокировка становится невозможной.

Другое простое решение достигается путём присвоения [частичного порядка](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D1%82%D0%B8%D1%87%D0%BD%D0%BE_%D1%83%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D1%87%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE) ресурсам (в данном случае вилкам) и установления соглашения, что ресурсы запрашиваются в указанном порядке, а возвращаются в обратном порядке. Кроме того, не должно быть двух ресурсов, не связанных порядком, используемых одной рабочей единицей.

Пусть ресурсы (вилки) будут пронумерованы от 1 до 5, и каждая рабочая единица (философ) всегда берёт сначала вилку с наименьшим номером, а потом вилку с наибольшим номером из двух доступных. Далее, философ кладёт сначала вилку с бо́льшим номером, потом — с меньшим. В этом случае, если четыре из пяти философов одновременно возьмут вилку с наименьшим номером, на столе останется вилка с наибольшим возможным номером. Таким образом, пятый философ не сможет взять ни одной вилки. Более того, только один философ будет иметь доступ к вилке с наибольшим номером, так что он сможет есть двумя вилками. Когда он закончит использовать вилки, он в первую очередь положит на стол вилку с бо́льшим номером, потом — с меньшим, тем самым позволив другому философу взять недостающую вилку и приступить к еде.

Данное решение было предложено Дейкстрой.

В то время, как иерархия ресурсов позволяет избежать взаимных блокировок, данное решение не всегда является практичным, в особенности когда список необходимых ресурсов неизвестен заранее. Например, если рабочая единица удерживает ресурс 3 и 5 и решает, что ей необходим ресурс 2, то она должна выпустить ресурс 5, затем 3, после этого завладеть ресурсом 2 и снова взять ресурс 3 и 5. Компьютерные программы, которые работают с большим количеством записей в [базе данных](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), не смогут работать эффективно, если им потребуется выпускать все записи с верхними индексами прежде, чем завладеть новой записью. Это делает данный метод непрактичным.

Пример ниже показывает решение, где вилки не представляются явно. Философы могут есть, если ни один из их соседей не ест. Аналогично системе, где философы, которые не могут взять вторую вилку, должны положить первую вилку до того, как они попробуют снова.

В отсутствие блокировок, связанных с вилками, философы должны обеспечивать то, что начало принятия пищи не основывается на старой информации о состоянии соседей. Например: Если философ Б видит, что A не ест в данный момент времени, а потом поворачивается и смотрит на В, A мог начать есть, пока философ Б смотрит на В. Используя одну [взаимоисключающую блокировку (Мьютекс)](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81), можно избежать этой проблемы. Эта блокировка не связана с вилками, но она связана с решением процедур, которые могут изменить состояние философов. Это обеспечивается монитором.

Алгоритм монитора реализует схему «проверить, взять и положить» и совместно использует взаимоисключающую блокировку. Заметьте, что философы, желающие есть, не будут иметь вилок.

Если монитор разрешает философу, желающему есть, действовать, то философ снова завладевает первой вилкой, прежде чем взять уже свободную вторую.

По окончании текущего приёма пищи философ оповещает монитора о том, что обе вилки свободны.

Стоит заметить, что этот алгоритм монитора не решает проблемы голодания. Например, философ Б может бесконечно ждать своей очереди, если у философов A и В периоды приёма пищи всё время пересекаются. Чтобы гарантировать также, что ни один философ не будет голодать, можно отслеживать, сколько раз голодный философ не ел, когда его соседи положили вилки на стол. Если количество раз превысит некий предел, такой философ перейдёт в состояние Голодания и алгоритм монитора форсирует процедуру завладения вилками, выполняя условие недопущения голодания ни одного из соседей.

Философ, не имеющий возможности взять вилки из-за того, что его сосед голодает, находится в режиме полезного ожидания окончания приёма пищи соседом его соседа. Эта дополнительная зависимость снижает параллелизм. Увеличение значения порога перехода в состояние Голодание уменьшает этот эффект.

Выводы

Чтобы сгладить влияние прерываний, операционные системы предоставляют концептуальную модель, состоящую из параллельно запускаемых последовательных процессов. Процессы могут создаваться и уничтожаться в динамическом режиме. У каждого процесса есть собственное адресное пространство.

Некоторым приложениям выгодно в рамках одного процесса иметь несколько потоков управления. Эти потоки имеют независимое планирование, и каждый из них имеет собственный стек, но все потоки, принадлежащие одному процессу, используют общее адресное пространство. Потоки могут быть реализованы в пространстве пользователя или в пространстве ядра.

Процессы могут взаимодействовать друг с другом, используя соответствующие примитивы, к которым относятся семафоры, мониторы или сообщения. Эти примитивы используют, чтобы исключить одновременное пребывание двух процессов в их критических областях, то есть для предотвращения ситуации, приводящей к хаосу. Процесс может находиться в работающем, готовом к работе или заблокированном состоянии и может изменять состояние, когда он сам или другие процессы задействуют один из примитивов взаимодействия процессов. Взаимодействие потоков имеет сходный характер.

Примитивы взаимодействия процессов могут использоваться для решения таких задач, как «производитель — потребитель», «обедающие философы» и «читатель — писатель». Но даже при использовании этих примитивов нужно соблюдать особую осторожность во избежание ошибок и взаимных блокировок.

Было изучено множество алгоритмов планирования. Некоторые из них, например алгоритм первоочередного выполнения самого короткого задания, используются преимущественно в пакетных системах. Другие же получили распространение как в пакетных, так и в интерактивных системах. В числе этих алгоритмов циклическое и приоритетное планирование, многоуровневые очереди, гарантированное, лотерейное и справедливое планирование. Некоторые системы проводят четкую грань между механизмом и политикой планирования, что позволяет пользователям управлять алгоритмом планирования.

Уважаемые студенты напишите все ли понятно и переходим к другой теме или еще разбираемся с этой Процессы и потоки.