Задача №1. «Проблема обедающих философов»

**1. Краткие теоретические сведения**

В 1965 году Дейкстра сформулировал и решил проблему синхронизации, названную им **проблемой обедающих философов**. Проблема обедающих философов моделирует процессы, соревнующиеся за монопольный доступ к ограниченному количеству ресурсов.  В классической формулировке она звучит следующим образом:

**Пять философов сидят за круглым столом, и у каждого есть тарелка со спагетти. Спагетти настолько скользкие, что каждому философу нужно две вилки, чтобы с ними управиться. Между каждыми двумя тарелками лежит одна вилка. Жизнь философа состоит из чередующихся периодов поглощения пищи и размышлений. Когда философ голоден, он пытается получить две вилки – левую и правую, в любом порядке. Если ему удалось получить вилки, он ест, а затем кладет вилки обратно и продолжает размышления. Вопрос состоит в следующем: можно ли написать алгоритм, который моделирует действия для каждого философа и никогда не застревает (отсутствует взаимоблокировка). Мы будем решать эту проблему, переформулировав ее так: философы – китайцы, едят рис палочками. Естественно, что в таком случае для принятия пищи необходимо ровно две палочки (две вилки для принятия пищи – совершенно неестественны).**

**2. Решение задачи**

Практически любая операционная система, в том числе и ОС Windows,  позволяет создавать необходимые нам примитивы синхронизации, запускать и останавливать процессы и потоки. Меняются при этом только вызовы функций операционной системы. Поэтому возможно решение этой задачи с использованием операционной системы Windows. А языком программирования можно выбрать Delphi.

Модуль *main* предназначим для организации графического интерфейса, запуска и останова потоков, имитирующих действия философов.  Сами действия философов опишем в модуле *Philosopher,*жизненный цикл философа - в процедуре *Execute*. Понятно, что философ может принимать пищу только тогда, когда философы слева и справа от него пищу не принимают. Максимум два философа одновременно могут есть. Для отслеживания состояния каждого философа используем массив *state*. Синхронизацию действий философов будем производить, используя семафоры, по одному на каждого философа. Если философ голоден, он проверяет наличие палочек слева и справа. Если палочки свободны, семафор открывается (выполняется операция *ReleaseSemaphore*в процедуре *test*, вызванной из процедуры *take\_stick*). Дальнейшая попытка опустить семафор (*WaitForSingleObject* в процедуре *take\_stick*) при этом не имеет значения. Если же семафор не был открыт, семафор опускается и происходит блокировка потока до тех пор, пока соседний поток не попытается разблокировать его (вызов *test* из процедуры*put\_stick*для левого и правого соседа).

Так как в процессе работы потоков происходит обращение к общему массиву *state*, необходимо также синхронизировать доступ к нему. Существует два способа синхронизации – с помощью критической секции и с помощью мьютекса. Наличие критической секции гарантирует, что только один процесс в каждый момент времени может читать и изменять информацию в массиве *state*. Размеры критической секции при этом ограничиваются вызовами процедур *EnterCriticalSection* и *LeaveCriticalSection*. Если используется мьютекс, тот же блок кода ограничивается вызовами функций *WaitForSingleObject*и *ReleaseMutex*.

Классические задачи взаимодействия процессов

В литературе по операционным системам можно встретить множество

интересных проблем использования различных методов синхронизации,

ставших предметом широких дискуссий и анализа. В данном разделе мы рас-

смотрим наиболее известные проблемы.

Задача обедающих философов

В 1965 году Дейкстра сформулировал и решил проблему синхрониза-

ции, названную им задачей обедающих философов. С тех пор все изобрета-

тели очередного примитива синхронизации считали своим долгом продемон-

стрировать его наилучшие качества, показав, насколько элегантно с его по-

мощью решается задача обедающих философов. Суть задачи довольно про-

ста. Пять философов сидят за круглым столом и у каждого из них есть тарел-

ка спагетти. Эти спагетти настолько скользкие, что есть их можно только

двумя вилками. Между каждыми двумя тарелками лежит одна вилка. Внеш-

ний вид стола показан на рис.

Жизнь философа состоит из чередующихся периодов приема пищи и

размышлений. (Это положение из разряда абстракций даже в отношении фи-

лософов, но вся их остальная деятельность к задаче не относится.) Когда фи-

лософ становится голоден, он старается поочередно в любом порядке завла-

деть правой и левой вилкой. Если ему удастся взять две вилки, он на некото-

рое 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 нико-

гда при этом не попадает в состояние зависания? (Можно заметить, что по-

требность в двух вилках выглядит несколько искусственно. Может быть, 2

стоит переключиться с итальянской на китайскую кухню, заменив спагетти

рисом, а вилки палочками для еды?)

Существует самое простое решение этой задачи. Процедура доступа к

спагетти ждет, пока левая вилка не освободится, и берет ее. Затем ждет осво-

бождения правой вилки. После еды возвращает обе вилки на место. К сожа-

лению, это решение ошибочно. Допустим, что все пять философов одновре-

менно берут левую от себя вилку. Тогда никто из них не сможет взять пра-

вую вилку, что приведет к взаимной блокировке.

Можно 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 по-

вторяет весь процесс. Это предложение также ошибочно, но уже по другой

причине. При некоторой доле невезения все философы могут приступить к

выполнению алгоритма одновременно, взяв левые вилки и увидев, что пра-

вые вилки недоступны, опять одновременно положить на место левые вилки,

и так до бесконечности. Подобная ситуация, при которой все программы бес-

конечно работают, но не могут добиться никакого прогресса, называется го-

лоданием, или зависанием процесса. (Эта ситуация называется голоданием

даже в том случае, если проблема возникает вне стен итальянского или ки-

тайского ресторана.)

Можно подумать, что стоит только заменить одинаковое для всех фи-

лософов время ожидания после неудачной попытки взять правую вилку на

случайное, шансы, что все они будут топтаться на месте хотя бы в течение

часа, будут очень малы. 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 ком-

пьютера 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 можно внести улучшение, позволяющее избежать взаим-

ной блокировки и зависания. Для этого нужно защитить пять вилок на столе

одним двоичным семафором. Пред тем как брать вилки, философ должен

выполнить в отношении этого семафора операцию down. А после того, как он

положит вилки на место, он должен выполнить в отношении семафора опе-

рацию up. С теоретической точки зрения это вполне достаточное решение.

Но с практической в нем не учтен вопрос производительности: в каждый мо-

мент времени сможет есть спагетти только один философ. А при наличии пя-

ти вилок, вообще то, одновременно могут есть спагетти два философа.

Более рациональное решение не вызывает взаимной блокировки и до-

пускает максимум параллелизма для произвольного числа философов. В нем

используется массив, в котором отслеживается, чем занимается философ: ест,

размышляет или пытается поесть (пытается взять вилки). Перейти в состоя-3

ние приема пищи философ может, только если в этом состоянии не находит-

ся ни один из его соседей. Соседи философа N определяются макросами

LEFT и RIGHT. Иными словами, если N равен 2, LEFT равен 1, a RIGHT ра-

вен 3. В программе используется массив семафоров, по одному семафору на

каждого философа: голодный философ может блокироваться, если нужная

ему вилка занята.

Задача читателей и писателей4

Задача обедающих философов хороша для моделирования процессов,

которые соревнуются за исключительный доступ к ограниченному количест-

ву ресурсов, например к устройствам ввода-вывода. Другая общеизвестная

задача касается читателей и писателей. В ней моделируется доступ к базе

данных. Представим, к примеру, систему бронирования авиабилетов, в кото-

рой есть множество соревнующихся процессов, желающих обратиться к ней

по чтению и записи. Вполне допустимо наличие нескольких процессов, од-

новременно считывающих информацию из базы данных, но если один про-

цесс обновляет базу данных (проводит операцию записи), никакой другой

процесс не может получить доступ к базе данных даже для чтения информа-

ции. Вопрос в том, как создать программу для читателей и писателей?

В одном из решении первый читатель для получения доступа к базе

данных выполняет в отношении семафора db операцию down. А все следую-

щие читатели просто увеличивают значение счетчика rc. Как только читатели

прекращают свою работу, они уменьшают значение счетчика, а последний из

них выполняет в отношении семафора операцию up, позволяя заблокирован-

ному писателю, если таковой имеется, приступить к работе.

В представленном здесь решении есть одна достойная упоминания не-

достаточно очевидная особенность. Допустим, что какой-то читатель уже ис-

пользует базу данных, и тут появляется еще один читатель. Поскольку одно-

временная работа двух читателей разрешена, второй читатель допускается к

базе данных. По мере появления к ней могут быть допущены и другие до-

полнительные читатели.

Теперь допустим, что появился писатель. Он может не получить доступ

к базе данных, поскольку писатели должны иметь к базе данных исключи-

тельный доступ, поэтому писатель приостанавливает свою работу. Позже по-

являются и другие читатели. Доступ дополнительным читателям будет от-

крыт до тех пор, пока будет активен хотя бы один читатель. Вследствие этой

стратегии, пока продолжается наплыв читателей, они все будут получать

доступ к базе по мере своего прибытия. Писатель будет приостановлен до тех

пор, пока не останется ни одного читателя. Если новый читатель будет при-

бывать, скажем, каждые 2 с, и каждый читатель затрачивает на свою работу

по 5 с, писатель доступа никогда не дождется.

Чтобы предотвратить такую ситуацию, программу можно слегка изме-

нить: когда писатель находится в состоянии ожидания, то вновь прибываю-

щий читатель не получает немедленного доступа, а приостанавливает свою

работу и встает в очередь за писателем. При этом писатель должен переждать

окончания работы тех читателей, которые были активны при его прибытии, и

не должен пережидать тех читателей, которые прибывают уже после него.

Недостаток этого решения заключается в снижении производительности за

счет меньших показателей параллельности в работе.

Проблема спящего брадобрея

Действие еще одной классической проблемной ситуации межпроцесс-

ного взаимодействия разворачивается в парикмахерской. В парикмахерской

есть один брадобрей, его кресло и n стульев для посетителей. Если желаю-

щих воспользоваться его услугами нет, брадобрей сидит в своем кресле и

спит. Если в парикмахерскую приходит клиент, он должен разбудить брадо-

брея. Если клиент приходит и видит, что брадобрей занят, он либо садится на

стул (если есть место), либо уходит (если места нет). Необходимо запро-

граммировать брадобрея и посетителей так, чтобы избежать состояния состя-

зания. У этой задачи существует много аналогов в сфере массового обслужи-

вания, например информационная служба, обрабатывающая одновременно

ограниченное количество запросов, с компьютеризированной системой ожи-

дания для запросов.

В решении можно использовать три семафора: customers, для подсчета

ожидающих посетителей (клиент, сидящий в кресле брадобрея, не учитыва-

ется — он уже не ждет); barbers, количество брадобреев 0 или 1), простаи-

вающих в ожидании клиента, и mutex для реализации взаимного исключения.

Также используется переменная waiting, предназначенная для подсчета ожи-

дающих посетителей. Она является копией переменной customers. Присутст-

вие в программе этой переменной связано с тем фактом, что прочитать теку-

щее значение семафора невозможно.

В этом решении посетитель, заглядывающий в парикмахерскую, дол-

жен сосчитать количество ожидающих посетителей. Если посетителей мень-

ше, чем стульев, новый посетитель остается, в противном случае он уходит.

Когда брадобрей приходит утром на работу, он выполняет процедуру

barber, блокируясь на семафоре customers, поскольку значение семафора

равно 0. Затем брадобрей засыпает, как показано на рис., и спит, пока не при-

дет первый клиент. Приходя в парикмахерскую, посетитель выполняет про-

цедуру customer, запрашивая доступ к mutex для входа в критическую об-

ласть. Если вслед за ним появится еще один посетитель, ему не удастся что-

либо сделать, пока первый посетитель не освободит доступ к mutex. Затем

посетитель проверяет наличие свободных стульев, в случае неудачи освобо-

ждает доступ к mutex и уходит. Если свободный стул есть, посетитель увели-

чивает значение целочисленной переменной waiting. Затем он выполняет

процедуру up на семафоре customers, тем самым активизируя поток брадо-

брея. В этот момент оба — посетитель и брадобрей — активны. Когда посе-

титель освобождает доступ к mutex, брадобрей захватывает его, проделывает

некоторые служебные операции и начинает стричь клиента. По окончании 7

стрижки посетитель выходит из процедуры и покидает парикмахерскую. В

отличие от предыдущих программ, цикла посетителя нет, поскольку каждого

посетителя стригут только один раз. Цикл брадобрея существует, и брадо-

брей пытается найти следующего посетителя. Если ему это удается, он стри-

жет следующего посетителя, в противном случае брадобрей засыпает. Стоит

отметить, что, несмотря на отсутствие передачи данных в проблеме читате-

лей и писателей и в проблеме спящего брадобрея, обе эти проблемы относят-

ся к проблемам межпроцессного взаимодействия, поскольку требуют син-

хронизации нескольких процессов.