Процессы и потоки. Межпроцессное и межпоточное взаимодействие. Часть 3

Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

Дата последней модификации: 15.10.2014 22:30:00

История изменений

20.07.2014 – Версия 0.1. Первичный документ. Ковтун В.Ю.

Содержание

Лекция 3. Процессы и потоки. Межпроцессное и межпоточное взаимодействие. Часть 3

Вопросы

1. Особенности синхронизации в распределённых системах.

2. Классические проблемы межпроцессорного взаимодействия.

Особенности синхронизации в распределенных системах

К вопросам связи процессов, реализуемой путем передачи сообщений или вызовов RPC, тесно примыкают и вопросы синхронизации процессов. Синхронизация необходима процессам для организации совместного использования ресурсов, таких как файлы или устройства, а также для обмена данными.

В однопроцессорных системах решение задач взаимного исключения, критических областей и других проблем синхронизации осуществлялось с использованием общих методов, таких как семафоры и мониторы. Однако эти методы не совсем подходят для распределенных систем, так как все они базируются на использовании разделяемой оперативной памяти. Например, два процесса, которые взаимодействуют, используя семафор, должны иметь доступ к нему. Если оба процесса выполняются на одной и той же машине, они могут иметь совместный доступ к семафору, хранящемуся, например, в ядре, делая системные вызовы. Однако если процессы выполняются на разных машинах, то этот метод не применим, для распределенных систем нужны новые подходы.

Передача сообщений

В роли чего-то другого выступает передача сообщений. Этот метод межпроцессного взаимодействия использует два примитива: `send` и `receive`, которые скорее являются системными вызовами, чем структурными компонентами языка (что отличает их от мониторов и делает похожим на семафоры). Поэтому, их легко можно поместить в библиотечные процедуры, например:

send(destination. &message);

receive(source, &message);

Первый запрос посылает сообщение заданному адресату, а второй - получает сообщение от указанного источника (или от любого источника, если это не имеет значения). Если сообщения нет, второй запрос блокируется до поступления сообщения либо немедленно возвращает код ошибки.

Разработка систем передачи сообщений

С системами передачи сообщений связано большое количество сложных проблем и конструктивных вопросов, которых не возникает в случае семафоров и мониторов. Особенно много сложностей появляется в случае взаимодействия процессов, происходящих на различных компьютерах, соединенных сетью. Так, сообщение может потеряться в сети. Чтобы избежать потери сообщений, отправитель и получатель договариваются, что при получении сообщения получатель посылает обратно **подтверждение** приема сообщения. Если отправитель не получает подтверждения через некоторое время, он отсылает сообщение еще раз.

Теперь представим, что сообщение получено, но подтверждение до отправителя не дошло. Отправитель пошлет сообщение еще раз, и до получателя оно дойдет дважды. Крайне важно, чтобы получатель мог отличить копию предыдущего сообщения от нового. Обычно проблема решается с помощью помещения порядкового номера сообщения в тело самого сообщения. Если к получателю приходит письмо с номером, совпадающим с номером предыдущего письма, письмо классифицируется как копия и игнорируется. Решение проблемы успешного обмена информации в условиях ненадежной передачи сообщений составляет основу изучения компьютерных сетей.

Для систем обмена сообщениями также важен вопрос названий процессов. Необходимо однозначно определять процесс, указанный в запросе **send** или **receive.** Кроме того, встает вопрос **аутентификации:** каким образом клиент может определить, что он взаимодействует с настоящим файловым сервером, а не с самозванцем?

Помимо этого, существуют конструктивные проблемы, существенные при расположении отправителя и получателя на одном компьютере. Одной из таких проблем является производительность. Копирование сообщений из одного процесса в другой происходит гораздо медленнее, чем операция на семафоре или вход в монитор. Было проведено множество исследований с целью увеличения эффективности передачи сообщений.

Решение проблемы производителя и потребителя с передачей сообщений

Теперь рассмотрим решение проблемы производителя и потребителя с передачей сообщений и без использования разделенной памяти. Решение представлено в Листинг 2. 1. Мы предполагаем, что все сообщения имеют одинаковый размер и сообщения, которые посланы, но еще не получены, автоматически помещаются ОС в буфер. В этом решении используются N сообщений, по аналогии с N сегментами в буфере. Потребитель начинает с того, что посылает производителю N пустых сообщений. Как только у производителя оказывается элемент данных, который он может предоставить потребителю, он берет пустое сообщение и отсылает назад полное. Таким образом, общее число сообщений в системе постоянно и их можно хранить в заранее заданном участке памяти.

Если производитель работает быстрее, чем потребитель, все сообщения будут ожидать потребителя в заполненном виде. При этом производитель блокируется в ожидании пустого сообщения. Если потребитель работает быстрее, ситуация инвертируется: все сообщения будут пустыми, а потребитель будет блокирован в ожидании полного сообщения.

Листинг 2. 1. Решение проблемы производителя и потребителя с использованием N сообщений

```
#define N 100          //количество сегментов в буфере
void producer(void)
{
        int item;
        message m; //буфер для сообщений
        while (TRUE) {
              item = produce_item(); //сформировать нечто, что заполнит буфер
              receive(consumer, &m); //ожидание прибытия пустого сообщения
              build_message(&m, item); //сформировать сообщение для отправки
              send(consumer, &m); //отослать элемент потребителю
         }
}
void consumer(void)
{
              int item, i;
              message m;
              for (i = 0; i < N; i++) send(producer, &m); //отослать N пустых сообщений
              while (TRUE) {
                   receive(producer, &m); //получить сообщение с элементом
                   item = extract_item(&m) ; //извлечь элемент из сообщения
                   send(producer, &m); //отослать пустое сообщение
                   consume_item(item); //обработка элемента
              }
}
```

Передача сообщений может быть реализована по-разному. Рассмотрим способ адресации сообщений. Можно присвоить каждому из процессов уникальный адрес и

адресовать сообщение непосредственно процессам. Другой подход состоит в использовании новой структуры данных, называемой **почтовым ящиком.** Почтовый ящик — это буфер для определенного количества сообщений, тип которых задается при создании ящика. При использовании почтовых ящиков в качестве параметров адреса `send` и `receive` задаются почтовые ящики, а не процессы. Если процесс пытается послать сообщение в полный почтовый ящик, ему приходится подождать, пока хотя бы одно сообщение не будет удалено из ящика.

В задаче производителя и потребителя оба они создадут почтовые ящики, достаточно большие, чтобы хранить N сообщений. Производитель будет посылать сообщения с данными в почтовый ящик потребителя, а потребитель будет посылать пустые сообщения в почтовый ящик производителя. С использованием почтовых ящиков метод буферизации очевиден: в почтовом ящике получателя хранятся сообщения, которые были посланы процессу-получателю, но еще не получены.

Другим предельным случаем использования почтовых ящиков является принципиальное отсутствие буферизации. При таком подходе, если `send` выполняется раньше, чем `receive`, посылающий процесс блокируется до выполнения `receive`, когда сообщение может быть напрямую скопировано от отправителя к получателю без промежуточной буферизации. Если `receive` выполняется раньше, чем `send`, получающий процесс блокируется до выполнения `send`. Этот метод часто называется **рандеву,** он легче реализуется, чем схема буферизации сообщений, но менее гибок, поскольку отправитель и получатель должны работать в режиме жесткой синхронизации.

Передача сообщений часто используется в системах с параллельным программированием. Характерным примером системы передачи сообщений является MPI (Message-Passing Interface – интерфейс передачи сообщений).

Барьеры

Последний из рассмотренных нами механизмов синхронизации предназначался скорее для групп процессов, нежели для ситуаций с двумя процессами типа производитель-потребитель. Некоторые приложения делятся на фазы, и существует правило, что процесс не может перейти в следующую фазу, пока к этому не готовы все остальные процессы. Этого можно добиться, разместив в конце каждой фазы барьер. Когда процесс доходит до барьера, он блокируется, пока все процессы не дойдут до барьера. Действие барьера представлено на Рис. 1.

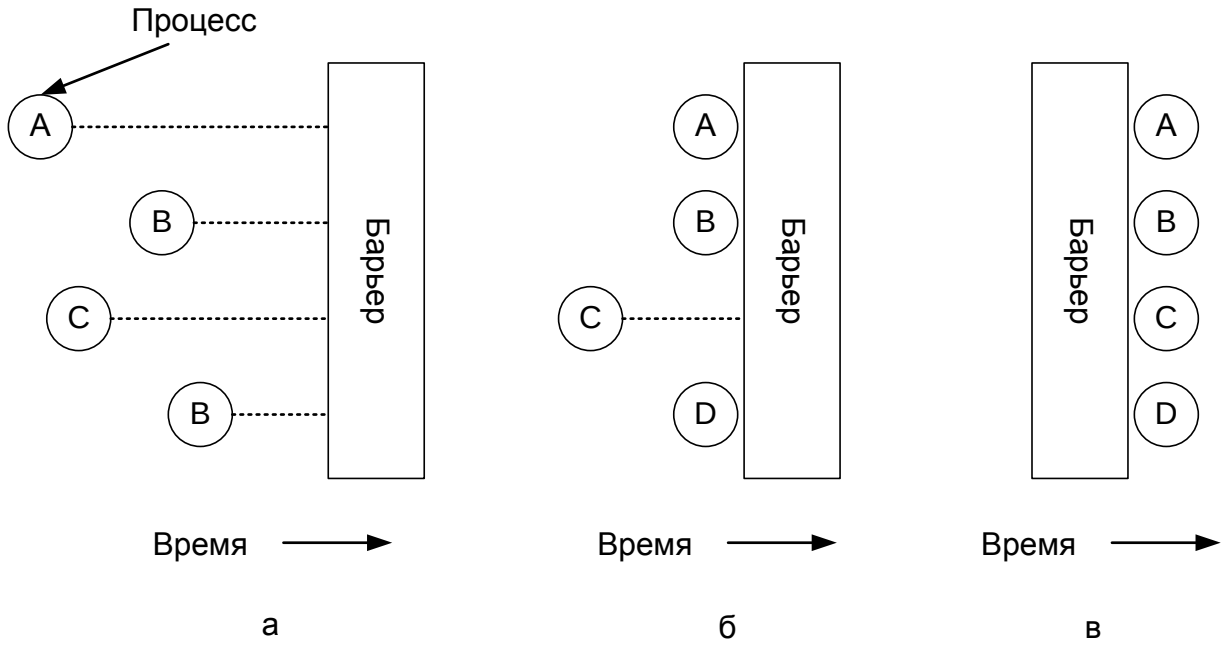

Рис. 1. Применение барьеров: процессы, приближающиеся к барьеру (а); все процессы, кроме одного, блокированы барьером (б); как только последний процесс достигает барьера все процессы переходят в следующую фазу (в)

На Рис. 1(а) представлены четыре процесса, приближающиеся к барьеру. Это означает, что они заняты вычислениями и еще не дошли до конца фазы. Через некоторое время первый процесс завершает вычисления, предусмотренные в этой фазе. Он выполняет примитив `barrier`, чаще всего вызывая библиотечную процедуру. Затем процесс приостанавливается. Через некоторое время второй и третий процессы заканчивают первую фазу и выполняют примитив `barrier` (Рис. 1(б)). Наконец, когда последний процесс достигает барьера, все процессы переходят в следующую фазу, как показано на Рис. 1(в).

Рассмотрим типичную задачу релаксации в физике или машиностроении в качестве примера ситуации, требующей наличия барьеров. Обычно задача представляется в виде матрицы с некоторыми начальными значениями. Эти значения могут соответствовать температуре в разных точках металлической пластины. Необходимо рассчитать, через какое время установится постоянное распределение температуры в случае нагрева одной стороны пластины.

К матрице применяется некоторое преобразование, например соответствующее законам термодинамики, чтобы получить матрицу значений температуры через некоторое время. Преобразование применяется снова и снова, чтобы получить зависимость температуры в каждой точке от времени. Результатом будет серия матриц, соответствующих различным моментам времени.

Теперь представим, что матрица очень большая (скажем, миллион на миллион) и для ускорения расчетов необходимы параллельные процессы, возможно, на мультипроцессоре. Различные процессы обрабатывают различные части матрицы, рассчитывая новые элементы на основе старых по законами физики. Очевидно, что ни один процесс не должен начинать итерацию n+1, пока все процессы не закончили свою текущую работу. Этой цели можно достичь, если каждый процесс будет выполнять операцию `barrier`, закончив свою часть итерации. Когда все процессы закончили работу и новая матрица, являющаяся входными данными для следующей итерации, составлена, все процессы одновременно начнут следующую итерацию.

Классические проблемы межпроцессного взаимодействия

В литературе по операционным системам можно встретить множество интересных проблем использования различных методов синхронизации, ставших предметом широких дискуссий и анализа. В данном разделе мы рассмотрим наиболее известные проблемы.

Проблема обедающих философов

В 1965 году Дейкстра сформулировал и решил проблему синхронизации, названную им **проблемой обедающих философов**. С тех пор каждый, кто изобретал еще один новый примитив синхронизации, считал своим долгом продемонстрировать достоинства нового примитива на примере проблемы обедающих философов. Проблему можно сформулировать 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 (Рис. 2).

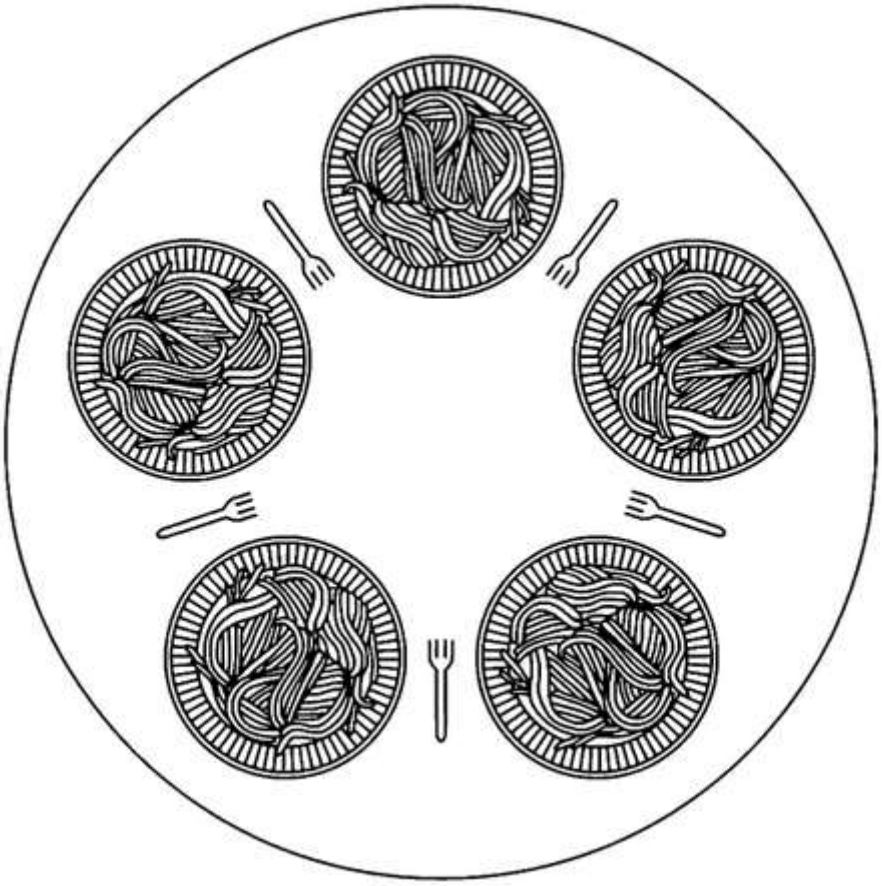

Рис. 2. Время обеда на факультете философии

Жизнь философа состоит из чередующихся периодов поглощения пищи и размышлений. (Разумеется, это абстракция, даже применительно к философам, но остальные процессы жизнедеятельности для нашей задачи несущественны.) Когда философ голоден, он пытается получить две вилки, левую и правую, в любом порядке. Если ему удалось получить две вилки, он некоторое время ест, затем кладет вилки обратно и продолжает размышления. Вопрос состоит в следующем: можно ли написать алгоритм, который моделирует эти действия для каждого философа и никогда не застревает? (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?).

Существует самое простое решение этой задачи. Процедура доступа к спагетти ждет, пока левая вилка не освободится, и берет ее. Затем ждет освобождения правой вилки. После еды возвращает обе вилки на место. 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 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, 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.

Можно подумать, что стоит только заменить одинаковое для всех философов 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.

В программу можно внести улучшение, позволяющее избежать взаимной 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 операцию `up`. С теоретической точки зрения это вполне достаточное решение. Но с практической в нем не учтен вопрос производительности: в каждый момент времени сможет есть спагетти только один философ. А при наличии пяти вилок, вообще то, одновременно могут есть спагетти два философа.

Более рациональное решение, представленное в Листинг 2. 2 не вызывает взаимной блокировки и допускает 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 состояние приема пищи философ может, только если в этом состоянии не находится ни один из его соседей. Соседи философа N определяются макросами `LEFT` и `RIGHT`. Иными словами, если N равен 2, `LEFT` равен 1, а `RIGHT` равен 3. В программе используется массив семафоров, по одному семафору на каждого философа: голодный философ может блокироваться, если нужная ему вилка занята.

Листинг 2. 2. Решение задачи обедающих философов

```
#define N 5                     //Количество философов
#define LEFT (i+N.1)%N   //Номер левого соседа философа с номером 1
#define RIGHT (i+1)%N    //Номер правого соседа философа с номером 1
#define THINKING 0         //Философ размышляет
#define HUNGRY   1         //Философ пытается получить вилки
#define EATING    2         //Философ ест
typedef int semaphore;    //Семафоры - особый вид целочисленных переменных
int state[N];                 //Массив для отслеживания состояния каждого философа
semaphore mutex = 1;   //Взаимное исключение для критических областей
semaphore s[N];           //Каждому философу по семафору
void philosopher(int i)    //i - номер философа, от 0 до N-1
{
        while (TRUE) {    //Повторять до бесконечности
              think();      //Философ размышляет
              take forks(i);//Получает две вилки или блокируется
              eat();          //Спагетти, ням-ням
              put_forks(i);  //Кладет на стол обе вилки
         }
}
void take_forks(int 1) //i - номер философа, от 0 до N-1
{
        down(&mutex); //Вход в критическую область
        state[i] - HUNGRY; //Фиксация наличия голодного философа
        test(i); //Попытка получить две вилки
        up(&mutex); //Выход из критической области
        down(&s[i]); //Блокировка, если вилок не досталось
}
void put_forks(i) //i-номер философа, от 0 до N-1
{
        down(&mutex); //Вход в критическую область
        state[i] = THINKING; //Философ перестал есть
        test(LEFT); //Проверить, может ли есть сосед слева
```

```
        test(RIGHT); //Проверить, может ли есть сосед справа
        up(&mutex); //Выход из критической области
}
void test(i) //i – номер философа, от 0 до N-1
{
        if (state[i] == HUNGRY && state[LEFT]! = EATING && state[RIGHT]! = EATING)  {
        state[i] = EATING;
        up(&s[i]);
        }
}
```

Обратите внимание, что каждый процесс запускает процедуру `philosopher` в качестве своей основной программы, но остальные процедуры `take_forks`, `put_forks` и `test` являются обычными процедурами, а не отдельными процессами.

Проблема читателей и писателей

Задача обедающих философов хороша для моделирования процессов, которые соревнуются за исключительный доступ к ограниченному количеству 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 которой есть множество соревнующихся процессов, желающих обратиться к ней по чтению и записи. Вполне допустимо наличие нескольких процессов, одновременно считывающих информацию из базы данных, но если один процесс обновляет базу данных (проводит операцию записи), никакой другой процесс не может получить доступ к базе данных даже для чтения информации. 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 следующие читатели просто увеличивают значение счётчика `rc`. Как только читатели прекращают свою работу, они уменьшают значение счётчика, а последний из них выполняет в отношении семафора операцию `up`, позволяя заблокированному писателю, если таковой имеется, приступить к работе.

В представленном здесь решении есть одна достойная упоминания недостаточно очевидная особенность. Допустим, что какой-то читатель уже использует базу данных, и тут появляется еще один читатель. Поскольку одновременная работа двух читателей разрешена, второй читатель допускается к базе данных. По мере появления к ней могут быть допущены и другие дополнительные читатели.

Теперь допустим, что появился писатель. Он может не получить доступ к базе данных, поскольку писатели должны иметь к базе данных исключительный доступ, поэтому писатель приостанавливает свою работу. Позже появляются и другие читатели. Доступ дополнительным читателям будет открыт 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 прибывать, скажем, каждые 2 с, и каждый читатель затрачивает на свою работу по 5 с, писатель доступа никогда не дождётся.

Чтобы предотвратить такую ситуацию, программу можно слегка изменить: когда писатель находится в состоянии ожидания, то вновь прибывающий 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.

Решение проблемы представлено в Листинг 2. 3.

Листинг 2. 3. Решение проблемы читателей и писателей

```
typedef int semaphore;          //Воспользуйтесь своим воображением
semaphore mutex =1;             //Контроль доступа к rc
semaphore db = 1;               //Контроль доступа к базе данных
int rc = 0:                     //Количество процессов, читающих или желающих читать
void reader(void)
{
     while (TRUE) {       //Повторять до бесконечности
          down(&mutex);      //Получение монопольного доступа к rc
          rc =» rc+1;         //Одним читающим процессом больше
          if (rc == 1) down(&db); //Если этот читающий процесс - первый...
          up(&mutex);              //Отказ от монопольного доступа к rc
          read_data_base();        //Доступ к данным
          down(&mutex);           //Получение монопольного доступа к rc
          rc = rc-1;              //Одним читающим процессом меньше
          if (rc == 0) up(&db);   //Если этот читающий процесс - последний...
          up(&mutex);             //Отказ от монопольного доступа к rc
          use_data_read();        //Вне критической области
     }
}
void writer(void)
{
     while (TRUE) {               //Повторять до бесконечности
          think_up_data();        // Вне критической области
          down(&db);              //Получение монопольного доступа
          write_data_base();      //Запись данных
          up(&db);                //Отказ от монопольного доступа
     }
}
```

Проблема спящего брадобрея

Действие еще одной классической проблемной ситуации межпроцессного взаимодействия разворачивается в парикмахерской. В парикмахерской есть один брадобрей, его кресло и n стульев для посетителей. Если желающих воспользоваться его услугами нет, брадобрей сидит в своем кресле и спит. Если в парикмахерскую приходит клиент, он должен разбудить брадобрея. Если клиент приходит и видит, что брадобрей занят, он либо садится на стул (если есть место), либо уходит (если места нет). Необходимо запрограммировать брадобрея и посетителей так, чтобы избежать состояния состязания. У этой задачи существует много аналогов в сфере массового обслуживания, например информационная служба, обрабатывающая одновременно ограниченное количество запросов, с компьютеризированной системой ожидания для запросов.

В решении можно использовать три семафора: `customers`, для подсчета ожидающих посетителей (клиент, сидящий в кресле брадобрея, не учитывается — он уже не ждет); `barbers`, количество брадобреев 0 или 1, простаивающих в ожидании клиента, и `mutex` для реализации взаимного исключения. Также используется переменная `waiting`, предназначенная для подсчета ожидающих посетителей. Она является копией переменной `customers`. Присутствие в программе этой переменной связано с тем фактом, что прочитать текущее значение семафора невозможно.

В этом решении посетитель, заглядывающий в парикмахерскую, должен сосчитать количество ожидающих посетителей. Если посетителей меньше, 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 и спит, пока не придет первый клиент. Приходя в парикмахерскую, посетитель выполняет процедуру `customer`, запрашивая доступ к mutex для входа в критическую область. 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 освобождает доступ к `mutex` и уходит. Если свободный стул есть, посетитель увеличивает значение целочисленной переменной `waiting`. Затем он выполняет процедуру up на семафоре `customers`, тем самым активизируя поток брадобрея. В этот момент оба — посетитель и брадобрей — активны. Когда посетитель освобождает доступ к `mutex`, брадобрей захватывает его, проделывает некоторые служебные операции и начинает стричь клиента. По окончании 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 брадобрей пытается найти следующего посетителя. Если ему это удается, он стрижет следующего посетителя, в противном случае брадобрей засыпает. Стоит отметить, что, несмотря на отсутствие передачи данных в проблеме читателей и писателей и в проблеме спящего брадобрея, обе эти проблемы относятся к проблемам межпроцессного взаимодействия, поскольку требуют синхронизации нескольких процессов. Решение представлено в Листинг 2. 4.

Листинг 2. 4. Решение спящего брадобрея

```
#define CHAIRS 5                //Количество стульев для посетителей
typedef int semaphore;          //Догадайтесь сами
semaphore customers = 0;        //Количество ожидающих посетителей
semaphore barbers = 0;          //Количество брадобреев, ждущих клиентов
semaphore mutex = 1;            //Для взаимного исключения
int waiting = 0;                //Ожидающие (не обслуживаемые) посетители

void barber(void)
{
        while (TRUE) {
            down(&customers);  //Если посетителей нет, уйти в состояние ожидания
            down(&mutex);      //Запрос доступа к waiting
            waiting = waiting-1;  //Уменьшение числа ожидающих посетителей
            up(&barbers);      //Один брадобрей готов к работе
            up(&mutex);        //Отказ от доступа к waiting
            cut hair();        //Клиента обслуживают (вне критической области)
         }
}

void customer(void)
{
        down(&mutex);          //Вход в критическую область
        if (waiting < CHAIRS) {    //Если свободных стульев нет, придётся уйти
        waiting = waiting + 1      //Увеличения числа ожидающих посетителей
        up(&Customers);            //При необходимости, разбудить брадобрея
```

```
            up(&mutex);                 //Отказ от доступа к waiting
            down(&barbers);             //Если брадобрей занят, уйти в состояние ожидания
            get_haircut();              //Клиента усаживают и обслуживают
        } else {
            up(&mutex);             //Много посетителей, из парикмахерской придётся уйти
        }
}
```

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

3. Microsoft Development Network. URL: http://msdn.com