ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение

высшего профессионального образования

ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

М.Л. Долженкова

О.В. Караваева

Синхронизация процессов

Учебное пособие

Киров 2009

Печатается по решению редакционно-издательского совета Вятского государственного университета

УДК 004.451(07)
Д640

Долженкова М.Л., Караваева О.В. Синхронизация процессов: учебное пособие. – Киров: Изд-во ВятГУ, 2009. – 82 с.

Рецензент: кафедра «Информатики и математики» Московского гуманитарно-экономического института (Кировский филиал)

В учебном пособии описаны основные средства синхронизации процессов и потоков. Рассмотрено решение классических задач синхронизации и даны варианты заданий для выполнения лабораторных работ.

Пособие рассчитано на студентов, обучающихся по специальности 230101 (220100) «Вычислительные машины, комплексы, системы и сети» и изучающих дисциплины «Параллельное программирование» и «Операционные системы». Оно может быть полезным студентам других специальностей при знакомстве с использованием средств операционных систем для синхронизации процессов и потоков.

Редактор Н.Ю. Целищева

Оглавление

Введение

Двумя центральными темами, связанными с современными операционными системами (ОС), являются многозадачность и распределенные вычисления. Их основой, как и основой технологий разработки операционных систем, являются параллельные вычисления. Необходимо рассмотреть два важных аспекта параллельных вычислений: взаимоисключения и синхронизацию.

Взаимное исключение означает такое совместное использование кода, ресурсов или данных несколькими процессами, при котором в каждый момент времени доступ к совместно используемым объектам имеет только один процесс.

Синхронизация тесно связана с взаимным исключением, и представляет собой способность нескольких процессов координировать свою деятельность путем обмена информацией.

Анализ вопросов, связанных с параллельными вычислениями, необходимо начать с рассмотрения вопросов архитектуры операционных систем, имеющих отношение к этому понятию. Здесь важную роль играет аппаратная поддержка параллельных вычислений, а также важнейшие механизмы программной поддержки: семафоры, мониторы и передача сообщений.

Все многозадачные операционные системы используют концепцию процесса, начиная с однопользовательских операционных систем, например, Windows 98, и заканчивая операционными системами для мейнфреймов, таких, как OS/390, которые способны поддерживать работу тысяч пользователей. Основные требования, которым должны удовлетворять операционные системы, могут быть сформулированы с использованием понятия процесса.

Процесс (***задача***) – абстракция, описывающая выполняющуюся программу. Для операционной системы процесс представляет единицу работы, заявку на потребление системных ресурсов.

Архитектура операционной системы должна удовлетворять определенным требованиям:

- чередовать выполнение нескольких процессов, чтобы повысить степень использования процессора при обеспечении разумного времени отклика;
- распределять ресурсы между процессами в соответствии с заданной стратегией (т. е. предоставляя определенным функциям или приложениям более высокий приоритет), избегая в то же время взаимоблокировок;
- поддерживать обмен информацией между процессами, а также обеспечивать возможности создания процессов пользователями. Обе эти возможности могут помочь в структурировании приложений.

Основная работа традиционной операционной системы связана с управлением процессами. В любой момент времени этапы исполнения каждого процесса характеризуются одним из нескольких состояний, в число которых входят состояние *готовности, выполнения и заблокированности (ожидания)*. Операционная система отслеживает состояния процессов и управляет их изменением, для чего ей приходится поддерживать довольно подробные структуры данных, описывающие каждый процесс. Операционная система должна выполнять функции планирования и предоставлять инструментарий для совместного выполнения процессов и их синхронизации.

В многозадачной однопроцессорной системе несколько различных процессов могут выполняться, чередуясь один с другим. В многопроцессорной системе несколько процессов могут не только чередоваться, но и выполняться одновременно. Эти типы параллелизма вызывают массу сложных проблем, с которыми сталкивается как создающий приложение программист, так и операционная система.

Во многих операционных системах традиционная концепция процесса разделена на две части: первая из них связана с принадлежностью ресурсов (процесс), а вторая – с выполнением машинных команд (поток). Процесс может содержать несколько потоков.

Поток (нить) – это легковесный процесс, выполняемый в виртуальной памяти, которая может использоваться другими потоками процесс.

В многопоточной системе принадлежность ресурсов остается атрибутом процесса, в то время как сам процесс представляет собой множество параллельно выполняющихся потоков. Потоки возникли в операционных системах как средство распараллеливания вычислений. Многопоточная организация помогает лучше структурировать приложения, а также повысить производительность их работы.

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами. Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами. Для этого в памяти поддерживаются специальные информационные структуры, в которые система записывает, какие ресурсы выделены каждому процессу. Можно назначить процессу ресурсы в единоличное пользование или в совместное пользование с другими процессами. Некоторые из ресурсов выделяются процессу при его создании, а некоторые – динамически, по запросам во время выполнения. Ресурсы могут быть приписаны процессу на все время его жизни или только на определенный период. При выполнении этих функций подсистема управления процессами взаимодействует с другими подсистемами операционной системы, ответственными за управление ресурсами, такими как управление памятью, подсистема ввода-вывода, файловая система.

В операционных системах, где существуют процессы и потоки, процесс рассматривается операционной системой как заявка на потребление всех видов ресурсов, кроме процессорного времени. Процесс – это контейнер для набора ресурсов, используемых потоками. Процессорное время распределяется между потоками, которые представляют собой последовательности команд (потоки выполнения).

Например, в Windows 2000, на самом высоком уровне абстракции, процесс включает:

- закрытое виртуальное адресное пространство – диапазон адресов виртуальной памяти, которым может пользоваться процесс;
- исполняемую программу – начальный код и данные, проецируемые на виртуальное адресное пространство процесса;
- список открытых описателей различных системных ресурсов – семафоров, коммуникационных портов, файлов и других объектов, доступных всем потокам в данном процессе;
- контекст защиты (security context), называемый *маркером доступа* (access token) и идентифицирующий пользователя, группы безопасности и привилегии, сопоставленные с процессом;
- идентификатор процесса – уникальное значение, которое идентифицирует процесс в рамках операционной системы;
- минимум один поток. *Поток* (thread) – некая сущность внутри процесса, получающая процессорное время для выполнения. Без потока программа процесса не может выполняться. Поток включает следующие наиболее важные элементы:

1) содержимое набора регистров процессора, отражающих состояние процессора;

2) два стека, один из которых используется потоком при выполнении в режиме ядра, а другой – в пользовательском режиме;

3) закрытую область памяти, называемую локальной памятью потока (thread-local storage, TLS) и используемую подсистемами, библиотеками исполняющих систем (run-time libraries) и DLL;

4) уникальный идентификатор потока (во внутрисистемной терминологии также называемый идентификатором клиента: идентификаторы процессов и потоков генерируются из одного пространства имен и никогда не перекрываются).

Кроме закрытого адресного пространства и одного или нескольких потоков у каждого процесса имеются идентификация защиты и список открытых описателей таких объектов, как файлы и разделы общей памяти, или синхронизирующих объектов вроде мьютексов, событий и семафоров (рис. 1).

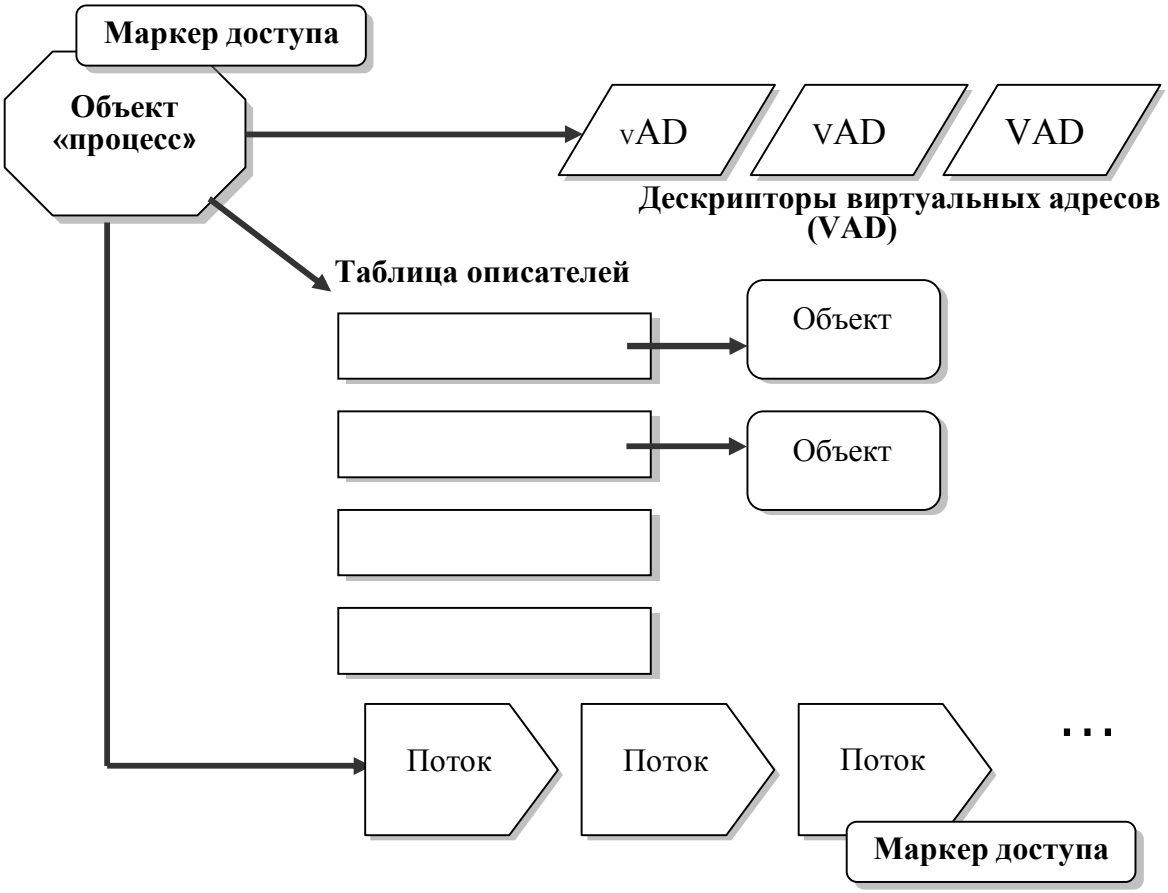

Рис. 1. Процесс и его ресурсы

1. Цели и средства синхронизации

Процессам часто нужно взаимодействовать друг с другом. Например, один процесс может передавать данные другому процессу или несколько процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает проблема синхронизации процессов, которая может решаться приостановкой и активизацией процессов, организацией очередей, блокированием и освобождением ресурсов. Существует достаточно обширный класс средств операционной системы, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Если операционная система не поддерживает потоки, то все, о чем будет сказано в дальнейшем, будет относиться к синхронизации процессов. Потребность в синхронизации потоков возникает только в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Во многих операционных системах эти средства называются средствами межпроцессного взаимодействия – Inter Process Commucations (IPC), что отражает историческую первичность понятия «процесс» по отношению к понятию «поток». Обычно к средствам IPC относят не только средства межпроцессной синхронизации, но и средства межпроцессного обмена данными.

1.1. Проблемы синхронизации

Выполнение процесса в мультипрограммной среде всегда имеет асинхронный характер. Очень сложно с полной определенностью сказать на каком этапе выполнения будет находиться процесс в определенный момент времени. Даже в однопрограммном режиме не всегда можно точно оценить время выполнения задачи. Это время во многих случаях существенно зависит от значения исходных данных, которые влияют на количество циклов, направления разветвления программы, время выполнения операций ввода-вывода и т. п.

Так как исходные данные в разные моменты запуска задачи могут быть разными, то и время выполнения отдельных этапов и задачи в целом, является весьма неопределенной величиной.

Еще более неопределенным является время выполнения программы в мультипрограммной системе. Моменты прерывания потоков, время нахождения их в очередях к разделяемым ресурсам, порядок выбора потоков для выполнения – все эти события являются результатом стечения многих обстоятельств и могут быть интерпретированы как случайные.

В лучшем случае можно оценить вероятностные характеристики вычислительного процесса, например вероятность его завершения за данный период времени.

Пренебрежение вопросами синхронизации процессов может привести к неправильной работе этих процессов или даже к краху системы. На рис. 2 изображена программа печати файлов (принт-сервер). Эта программа печатает по очереди все файлы, имена которых последовательно в порядке поступления записывают в специальный общедоступный файл «заказов» другие программы.

Особая переменная *next*, также доступная всем процессам-клиентам, содержит номер первой свободной для записи имени файла позиции файла «заказов». Процессы-клиенты читают эту переменную, записывают в соответствующую позицию файла «заказов» имя своего файла и наращивают значение *next* на единицу. Если в какой-то момент процесс *R* решил распечатать свой файл, для этого он прочитал значение переменной *next*, которое, например, будет равно *4*. Процесс запомнил это значение, но поместить имя файла не успел, так как его выполнение было прервано (например, в следствии исчерпания выделенного кванта времени). Очередной процесс *S*, желающий распечатать файл, прочитал то же самое значение переменной *next*, поместил в четвертую позицию имя своего файла и нарастил значение переменной на единицу.

Когда в очередной раз управление будет передано процессу *R*, то он, продолжая выполняться в полном соответствии со значением текущей свободной позиции, полученным во время предыдущей итерации, запишет имя файла также в позицию 4, поверх имени файла процесса *S*. Таким образом, процесс *S* никогда не увидит свой файл распечатанным.

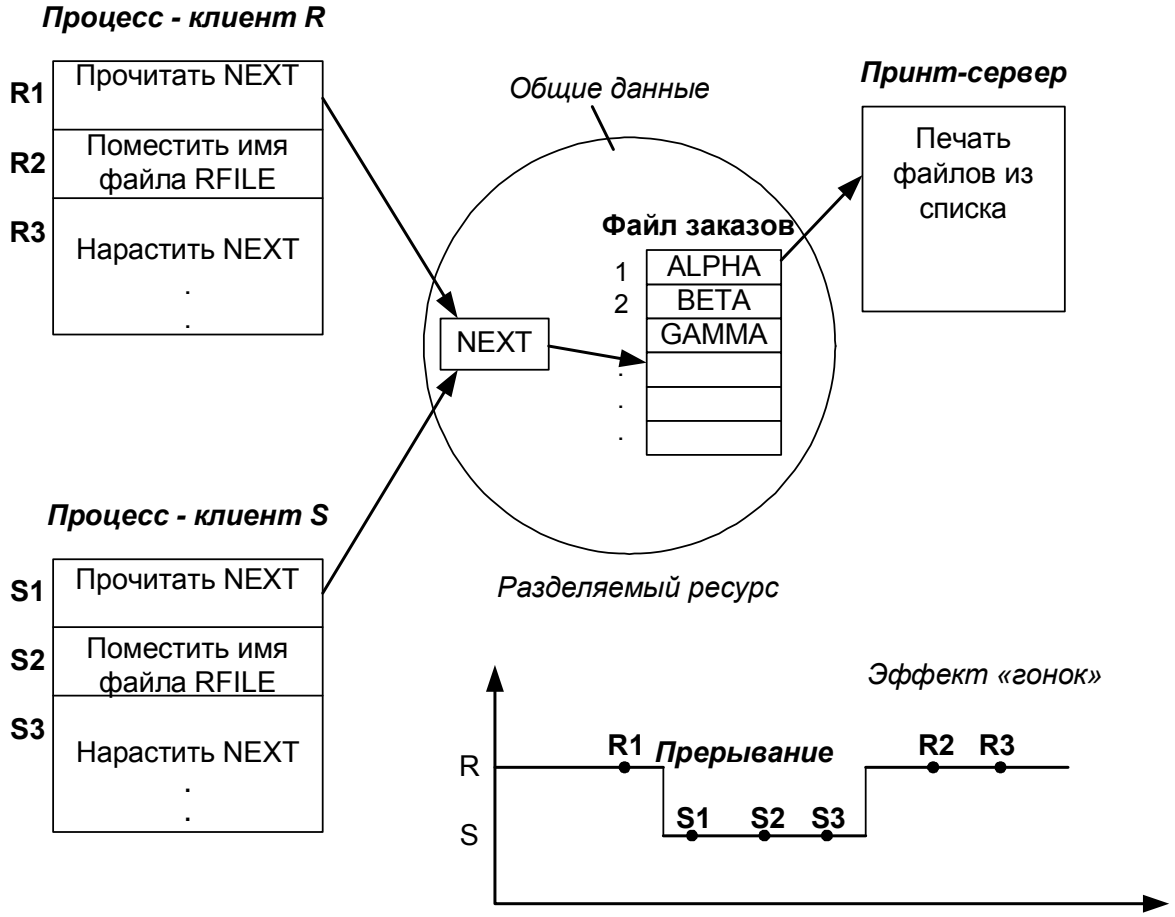

Рис. 2. Пример необходимости синхронизации

Сложность проблемы синхронизации состоит в нерегулярности возникающих ситуаций. В предыдущем примере можно представить и другое развитие событий: были потеряны файлы нескольких процессов или, напротив, не был потерян ни один файл. В данном случае все определяется взаимными скоростями процессов и моментами их прерывания. Поэтому отладка взаимодействующих процессов является сложной задачей. Ситуации, когда два или более процесса обрабатывают разделяемые данные, и конечный результат зависит от соотношения скоростей процессов, называются ***гонками***. Для исключения эффекта гонок по отношению к некоторому ресурсу, необходимо обеспечить, чтобы в каждый момент в критической секции, т. е. части программы, в которой осуществляется доступ к разделяемым данным, связанной с этим ресурсом, находился максимум один процесс. Этот прием называют ***взаимным исключением***.

Рис. 3. Пример тупика

Еще одна проблема требующая синхронизации – взаимная блокировка процессов, называемая так же дедлоками (deadlocks), клинчами (clinch) или тупиками. Пусть двум процессам *A* и *B* для выполнения работы требуется ресурс *R1* и *R2* (рис. 3). Пусть каждому ресурсу соответствует блокирующая переменная доступная процессам *A* и *B*, т. е. если один процесс занял ресурс, то он становится заблокированным для другого ресурса до момента освобождения предыдущим. Может возникнуть ситуация, при которой после того как процесс *A* занял ресурс *R1* и установил блокирующую переменную, он был прерван. Управление получил процесс *B*, который занял ресурс *R2*, но при попытке обращения к ресурсу R1 был заблокирован (ресурс *R1* занят процессом *A*).

Управление снова получил процесс *A*, который в соответствии с программой попытался получить доступ к ресурсу *R2*, но так же был заблокирован (ресурс *R2* занят процессом *B*). Такое положение и получило название ***тупик***.

2. Проектирование параллельных вычислительных процессов

Все известные средства для решения проблемы взаимного исключения основаны на специально введенных программных и аппаратных возможностях, к которым относятся блокировка памяти, специальные команды типа «Проверка и установка» и управление системой прерываний, позволяющих организовать такие механизмы, как семафорные операции, мониторы, почтовые ящики и другие.

Любая возможность обеспечения поддержки взаимных исключений должна соответствовать следующим требованиям.

1) Взаимные исключения должны осуществляться в принудительном порядке. В любой момент времени из всех процессов, имеющих критический раздел для одного и того же ресурса или разделяемого объекта, в этом разделе может находиться лишь только один процесс.

2) Процесс, завершающий работу в некритическом разделе, не должен влиять на другие процессы.

3) Не должна возникать ситуация бесконечного ожидания доступа к критическому разделу (т. е. не должны появляться взаимоблокировки).

4) Когда в критическом разделе нет ни одного процесса, любой процесс, запросивший возможность входа в него, должен немедленно его получить.

5) Не предусматривается никаких предположений о количестве процессов или их относительных скоростях работы.

6) Процесс остается в критическом разделе только в течение ограниченного времени.

Простейший способ обеспечить взаимное исключение – позволить процессу, находящемуся в критической секции, запрещать все прерывания. Однако этот способ непригоден, так как опасно доверять управление системой пользовательскому процессу. Он может надолго занять процессор, а при крахе процесса в критической области крах потерпит вся система, потому что прерывания никогда не будут разрешены.

Другим способом является использование блокирующих переменных. С каждым разделяемым ресурсом связывается двоичная переменная, которая принимает значение *1*, если ресурс свободен (т. е. ни один процесс не находится в данный момент в критической секции, связанной с данным процессом), и значение *0*, если ресурс занят. На рис. 4 показан фрагмент алгоритма процесса, использующего для реализации взаимного исключения доступа к разделяемому ресурсу *D* блокирующую переменную *F(D)*. Перед входом в критическую секцию процесс проверяет, свободен ли ресурс *D*. Если он занят, то проверка циклически повторяется, если свободен, то значение переменной *F(D)* устанавливается в *0*, и процесс входит в критическую секцию. После того, как процесс выполнит все действия с разделяемым ресурсом *D*, значение переменной *F(D)* снова устанавливается равным *1*.

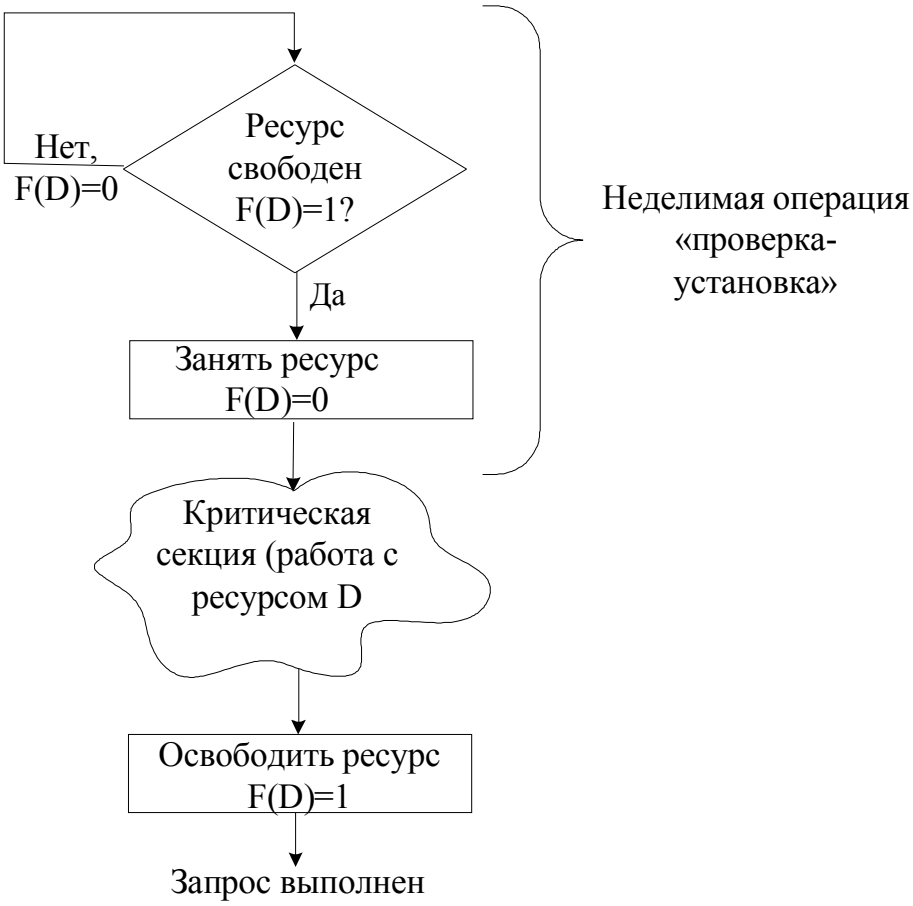

Рис. 4. Реализация критических секций с использованием блокирующих переменных

Если все процессы написаны с использованием вышеописанных соглашений, то взаимное исключение гарантируется. Следует заметить, что операция проверки и установки блокирующей переменной должна быть неделимой. Пусть в результате проверки переменной процесс определил, что ресурс свободен, но сразу после этого, не успев установить переменную в *0*, был прерван. За время его приостановки другой процесс занял ресурс, вошел в свою критическую секцию, но также был прерван, не завершив работы с разделяемым ресурсом. Когда управление было возвращено первому процессу, он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом, был нарушен принцип взаимного исключения, что потенциально может привести к нежелательным последствиям. Во избежание таких ситуаций в системе команд машины желательно иметь единую команду «проверка-установка», или же реализовывать системными средствами соответствующие программные примитивы, которые бы запрещали прерывания на протяжении выполнения операции «Проверка и установка».

Реализация критических секций с использованием блокирующих переменных имеет существенный недостаток: в течение времени, когда один процесс находится в критической секции, другой процесс, которому требуется тот же ресурс, будет выполнять рутинные действия по опросу блокирующей переменной. Для устранения таких ситуаций может быть использован так называемый аппарат событий. С помощью этого средства могут решаться не только проблемы взаимного исключения, но и более общие задачи синхронизации процессов. В разных операционных системах аппарат событий реализуется по-своему, но в любом случае используются системные функции аналогичного назначения, которые условно назовем *wait(x)* и *post(x)*, где x – идентификатор некоторого события.

На рис. 5 показан фрагмент алгоритма процесса, использующего эти функции. Если ресурс занят, то процесс не выполняет циклический опрос, а вызывает системную функцию *wait(D)*, здесь *D* – событие, заключающееся в освобождении ресурса *D*. Функция *wait(D)* переводит активный процесс в состояние ОЖИДАНИЕ и делает отметку в его дескрипторе, что процесс ожидает события *D*. Процесс, который в это время использует ресурс *D*, после выхода из критической секции выполняет системную функцию *post(D)*, в результате чего операционная система просматривает очередь ожидающих процессов и переводит процесс, ожидающий события *D*, в состояние ГОТОВНОСТЬ.

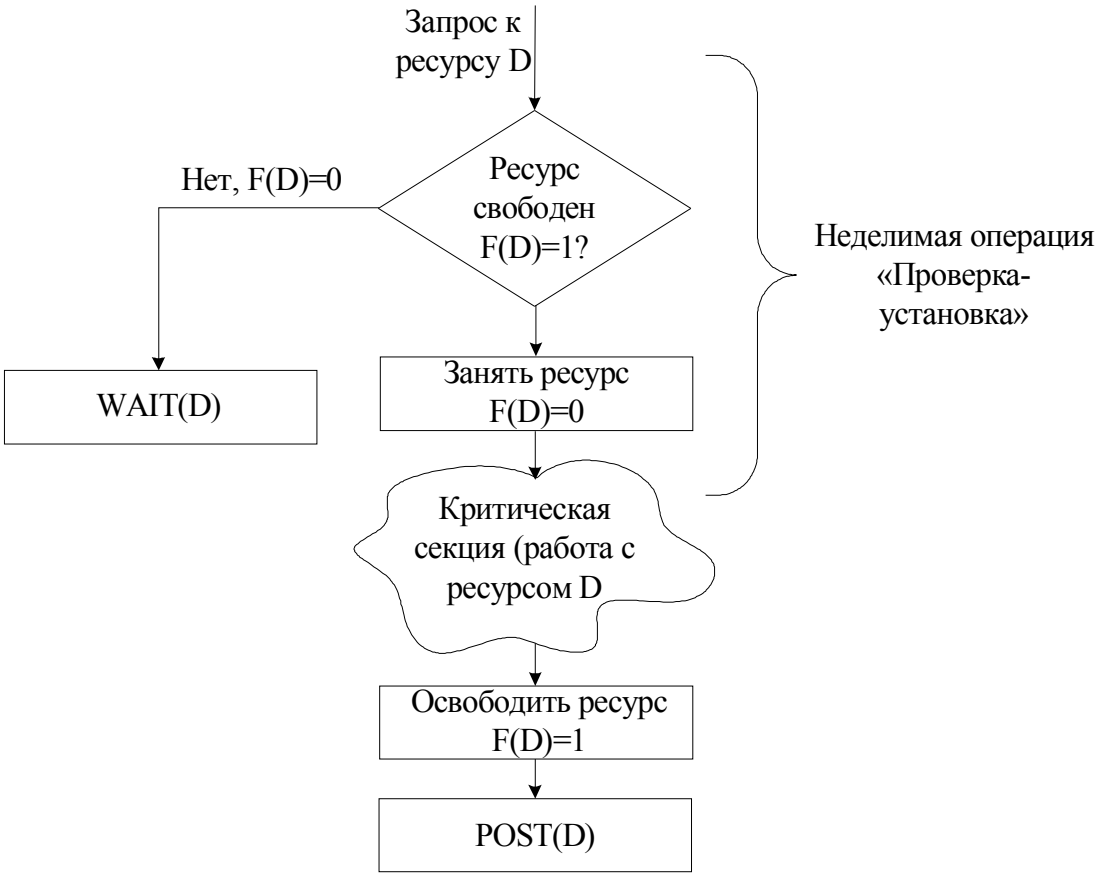

Рис. 5. Реализация критической секции с использованием системных функций *wait(d)* и *post(D)*

В вычислительных системах с одним процессором параллельные процессы не могут перекрываться, а способны только чередоваться. Следовательно, для того чтобы гарантировать взаимное исключение, достаточно защитить процесс от прерывания. Эта возможность может быть обеспечена в форме примитивов, определенных системным ядром для запрета и разрешения прерываний. Процесс в таком случае может обеспечить взаимоисключение следующим образом:

```
while (true)
{/* Запрет прерываний */;
/* Критический раздел */;
/* Разрешение прерываний */;
/* Остальной код */;
}
```

Поскольку критический раздел не может быть прерван, выполнение взаимоисключения гарантируется. Однако цена такого подхода высока. Эффективность работы может заметно снизиться, поскольку при этом ограничена возможность процессора по чередованию программ.

Если вычислительная система включает несколько процессоров, то вполне возможно (обычно так и бывает), что одновременно выполняются несколько процессов. В этом случае запрет прерываний не гарантирует выполнения взаимоисключений. Здесь отсутствует отношение «ведущий/ведомый (master/slave)» – процессоры работают независимо, «на равных», и не имеется механизма прерывания, на котором могли бы основываться взаимоисключения.

На уровне аппаратного обеспечения обращение к ячейке памяти исключает любые другие обращения к той же ячейке. Основываясь на этом принципе, разработчики процессоров предлагают ряд машинных команд, которые за один цикл выборки команды атомарно выполняют над ячейкой памяти два действия: чтение и запись, или чтение и проверка значения. Поскольку эти действия выполняются в одном цикле, на них не в состоянии повлиять никакие другие инструкции.

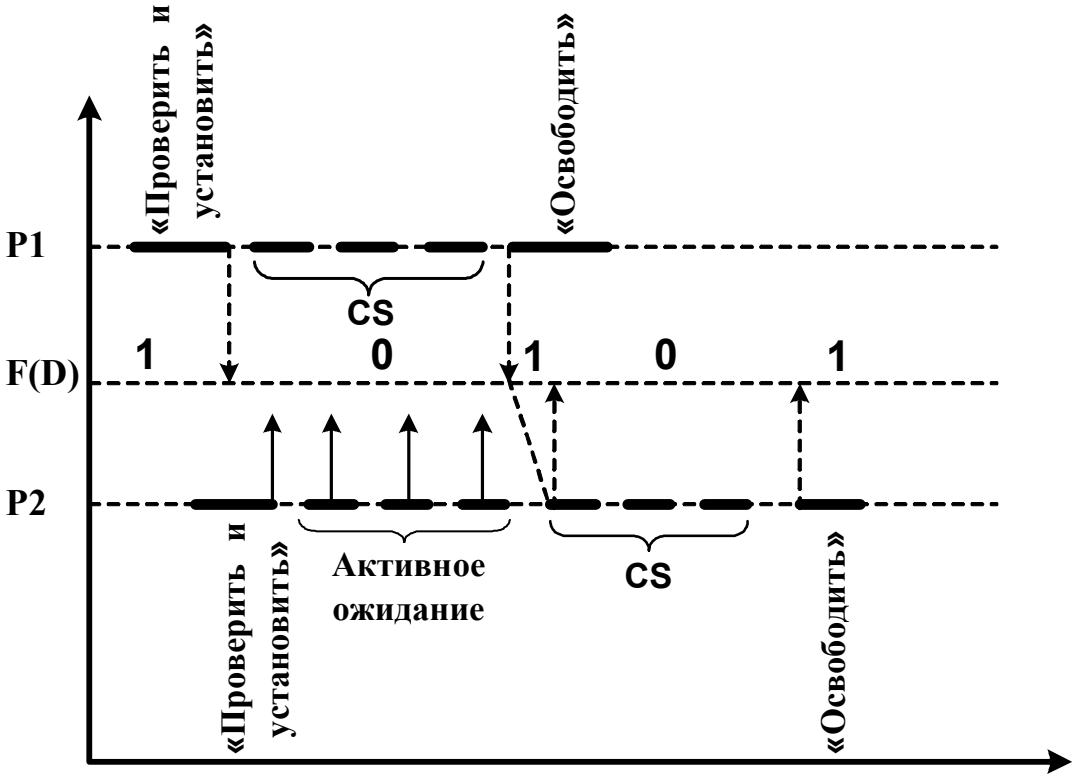

Рис. 6. Временная диаграмма команды «Проверка и установка»

Инструкцию «проверка и установка» значения можно определить следующим образом: инструкция проверяет значение своего аргумента *i,* если его значение равно *0*, функция заменяет его на *1* и возвращает *true*. В противном случае значение переменной не изменяется и возвращается значение *false*.

На рис. 6 приведена временная диаграмма исполнения команды «Проверка и установка». Функция «Проверка и установка» выполняется атомарно, т. е. ее выполнение не может быть прервано.

Листинг: аппаратная поддержка взаимных исключений

```
/*  Инструкция проверка и установка */
const int n = /* Количество процессов */;
int bolt;
 void P(int i)
```

```
{while(true)
{while(testset(bolt))
/* Ничего не делать */;
 /* Критический раздел */;
bolt = 0;
/* Остальная часть кода */; }
}
void main()
{bolt =0;
parbegin(P(l),P(2), ..,P(n)};
}
```

В листинге показан протокол взаимных исключений, основанный на использовании описанной инструкции. Разделяемая переменная *bolt* инициализируется нулевым значением. Только процесс, который может войти в критический раздел, находит, что значение переменной *bolt* равно *0*. Все остальные процессы при попытке входа в критический раздел переходят в режим ожидания. Выйдя из критического раздела, процесс устанавливает значение переменной *bolt* равным *0*, после чего один и только один процесс из множества, ожидающих входа в критический раздел, получает требуемый ему доступ. Выбор этого процесса зависит от того, какому из процессов удалось выполнить инструкцию *testset* первым.

Подход, основанный на использовании специальной машинной инструкции для осуществления взаимных исключений, имеет ряд преимуществ. Он применим к любому количеству процессов при наличии как одного, так и нескольких процессоров, разделяющих основную память. Этот подход очень прост, а потому легко проверяем. Он может использоваться для поддержки множества критических разделов, каждый из которых может быть определен при помощи своей собственной переменной.

Однако у такого подхода имеются и серьезные недостатки.

- Используется активное ожидание. Следовательно, в то время как процесс находится в ожидании доступа к критическому разделу, он продолжает потреблять процессорное время.

- Возможно голодание. Если процесс покидает критический раздел, а входа в него ожидают несколько других процессов, то выбор ожидающего процесса произволен. Следовательно, может оказаться, что какой-то из процессов будет ожидать входа в критический раздел бесконечно.

- Возможна взаимоблокировка. Рассмотрим следующий сценарий в однопроцессорной системе. Процесс *P1* выполняет инструкцию *testset* и входит в критический раздел. После этого процесс *P1* прерывается процессом *P2* с более высоким приоритетом. Если *P2* попытается обратиться к тому же ресурсу, что и *P1*, ему будет отказано в доступе в соответствии с механизмом взаимоисключений, и он войдет в цикл пережидания занятости. Однако в силу того что процесс *P1* имеет более низкий приоритет, он не получит возможности продолжить работу, так как в наличии имеется активный процесс с высоким приоритетом.

2.1. Блокировка памяти

Программный подход может быть реализован для параллельных процессов, которые выполняются как в однопроцессорной, так и в многопроцессорной системе с разделяемой основной памятью. Обычно такие подходы предполагают элементарные взаимоисключения на уровне доступа к памяти, т. е. одновременный доступ (чтение и/или запись) к одной и той же ячейке основной памяти упорядочивается при помощи некоторого механизма. При этом порядок предоставления доступа не определяется порядком обращения процессов за доступом к памяти. Никакой иной поддержки со стороны аппаратного обеспечения, операционной системы или языка программирования не предполагается.

2.1.1. Алгоритм Деккера

Алгоритм взаимных исключений для процессов был предложен голландским математиком Деккером (Dekker). Разработаем этот алгоритм по стадиям, демонстрируя узкие места и возможные ошибки при создании параллельно работающих программ.

Первая попытка

Как упоминалось ранее, любая попытка взаимного исключения должна опираться на некий фундаментальный механизм исключений аппаратного обеспечения. Наиболее общим механизмом может служить ограничение, согласно которому к некоторой ячейке памяти в определенный момент времени может осуществляться только одно обращение. Воспользовавшись этим ограничением, зарезервируем глобальную ячейку памяти, которую назовем *turn*. Процесс (*P0* или *P1*), который намерен выполнить критический раздел, сначала проверяет содержимое ячейки памяти *turn*. Если значение *turn* равно номеру процесса, то процесс может войти в критический раздел. В противном случае он должен ждать, постоянно опрашивая значение *turn* до тех пор, пока оно не позволит процессу войти в критический раздел. Такая процедура известна как пережидание занятости (*busy waiting*), поскольку процесс вынужден, по сути, не делать ничего полезного до тех пор, пока не получит разрешение на вход в критический раздел. Более того, он постоянно опрашивает значение переменной и тем самым потребляет процессорное время.

После того как процесс, получивший право на вход в критический раздел, выходит из него по завершении работы, он должен обновить значение *turn*, присвоив ему номер другого процесса.

Говоря формально, имеется глобальная переменная *int turn = 0;*.

Это решение гарантирует корректную работу взаимоисключения, однако имеет два недостатка. Во-первых, при входе в критический раздел процессы должны строго чередоваться, и скорость работы будет диктоваться более медленным из двух процессов. Если процессу *P1* вход в критический раздел

требуется раз в час, а процессору *P2* – 1000 раз в час, то темп работы *P1* будет таким же, как и у процесса *P2*. Во-вторых, гораздо более серьезная проблема возникает в случае сбоя одного из процессов – второй процесс оказывается заблокирован (при этом неважно, происходит ли сбой процесса в критическом разделе или нет).

Процесс 0	*Процесс 1*
while (turn != 0)	*while (turn != 1)*
/ ожидание */;*	*/* ожидание */;*
/ Критический раздел */;*	*/* Критический раздел */;*
turn =1;	*turn = 0;*

Вторая попытка

Проблема при первой попытке заключается в том, что при ее реализации хранилось имя процесса, который имел право входа в критический раздел, в то время как в действительности требуется информация об обоих процессах. По сути, каждый процесс должен иметь собственный ключ к критическому разделу, так что если даже произойдет сбой одного процесса, второй все равно сможет получить доступ к критическому разделу. Для удовлетворения этого условия определен логический вектор *flag* (флаг), в котором *flag[0]* соответствует процессу *P0*, а *flag[1]* — процессу *P1*. Каждый процесс может ознакомиться с флагом другого процесса, но не может его изменить. Когда процессу требуется войти в критический раздел, он периодически проверяет состояние флага другого процесса до тех пор, пока тот не примет значение *false*, указывающее, что другой процесс покинул критический раздел. Процесс немедленно устанавливает значение своего собственного флага равным *true* и входит в критический раздел. По выходе из критического раздела процесс сбрасывает свой флаг.

Сейчас разделяемые переменные выглядят следующим образом:

enum boolean { false = 0, true = 1; }; boolean flag [2] = { false, false };

Даже если произойдет сбой одного из процессов вне критического раздела (включая код установки значения флага), то второй процесс заблокирован не будет. Этот второй процесс в таком случае сможет входить в критический раздел всякий раз, как только это потребуется, поскольку флаг другого процесса всегда будет иметь значение *false*. Однако если сбой произойдет в критическом разделе (или перед входом в критический раздел, но после установки значения флага равным *true*), то другой процесс окажется навсегда заблокированным.

Процесс 0	*Процесс 1*
while (flag[1])	*while (flag[0])*
/ ожидание */;*	*/* ожидание */;*
flag[0] = true;	*flag[1] = true;*
/ Критический раздел */;*	*/* Критический раздел */;*
flag[0] = false;	*flag[1] = false;*

Описанное решение, по сути, оказывается еще хуже предложенного ранее, поскольку даже не гарантирует взаимного исключения. Проблема заключается в том, что предложенное решение не является независимым от относительной скорости выполнения процессов.

Можно рассмотреть такую последовательность действий.

1) *P0* выполняет инструкцию *while* и находит, что значение *flag[1]* равно *false*.

2) *P1* выполняет инструкцию *while* и находит, что значение *flag[0]* равно *false*.

3) *P0* устанавливает значение *flag[0]* равным *true* и входит в критический раздел.

4) *P1* устанавливает значение *flag[1]* равным *true* и входит в критический раздел.

Поскольку после этого оба процесса одновременно оказываются в критическом разделе, программа некорректна.

Третья попытка

В код вносится небольшое изменение.

Процесс 0	*Процесс 1*
flag[0] = true;	*flag[1] = true;*
while (flag[1])	*while (flag[0])*
/ Ничего не делаем */;*	*/* Ничего не делаем */;*
*/*Критический раздел */;*	*/* Критический раздел */;*
flag[0] = false;	*flag[1] = false;*

Как и ранее, если происходит сбой одного процесса в критическом разделе, включая код установки значения флага, то второй процесс оказывается заблокированным (и, соответственно, если сбой произойдет вне критического раздела, то второй процесс блокирован не будет).

Далее проверяется гарантированность взаимоисключения, прослеживая процесс *P0*. После того как процесс *P0* установит *flag[0]* равным *true*, *P1* не может войти в критический раздел до тех пор, пока туда не войдет и затем не покинет его процесс *P0*. Может оказаться так, что процесс *P1* уже находится в критическом разделе в тот момент, когда *P0* устанавливает свой флаг. В этом случае процесс P0 будет заблокирован инструкцией *while* до тех пор, пока *P1* не покинет критический раздел. Аналогичные действия происходят при рассмотрении процесса *P1*.

Тем самым гарантируется взаимное исключение, однако третья попытка порождает еще одну проблему. Если оба процесса установят значения флагов равными *true* до того, как один из них выполнит инструкцию *while*, то каждый из процессов будет считать, что другой находится в критическом разделе, и тем самым осуществится взаимоблокировка.

Четвертая попытка

В третьей попытке установка процессом флага состояния выполнялась без учета информации о состоянии другого процесса. Взаимоблокировка возникала по причине того, что каждый процесс мог добиваться своих прав на вход в критический раздел, и отсутствовала возможность отступить назад из имеющегося положения. Можно попытаться исправить ситуацию, делая процессы более «уступчивыми»: каждый процесс, устанавливая свой флаг равным *true*, сообщает о своем желании войти в критический раздел, но готов отложить свой вход, уступая другому процессу.

```
Процесс 0                          Процесс 1
flag[0] = true;                    flag[1] = true;
while (flag[1])                     while (flag[0])
flag[0] = false;                   flag[1] = false;
 /* Задержка */                    /* Задержка */
 flag[0] = true;                   flag[1] = true,
/* Критический раздел */;          /*Критический раздел*/;
 flag[0] = false;                  flag[1]=false;
```

Взаимоисключение гарантируется, однако нужно рассмотреть возможную последовательность событий:

1) *P0* устанавливает значение *flag[0]* равным *true*;
2) *P1* устанавливает значение *flag[1]* равным *true*;
3) *P0* проверяет *flag [1];*
4) *P1* проверяет *flag [0];*
5) *P0* устанавливает значение *flag[0]* равным *false*;
6) *P1* устанавливает значение *flag[1]* равным *false*;
7) *P0* устанавливает значение *flag[0]* равным *true*
8) *P1* устанавливает значение *flag[l]* равным *true.*

Эту последовательность можно продолжать до бесконечности, и ни один из процессов так и не сможет войти в критический раздел. Это не взаи-

моблокировка, так как любое изменение относительной скорости двух процессов разорвет замкнутый круг и позволит одному из процессов войти в критический раздел. Такую ситуацию можно назвать неустойчивой взаимоблокировкой (livelock). Здесь необходимо отметить, что обычная взаимоблокировка осуществляется, когда несколько процессов желают войти в критический раздел, но ни одному из них это не удается. В случае неустойчивой взаимоблокировки существует приводящая к успеху последовательность действий, но, вместе с тем, возможна и такая, при которой ни один из процессов не сможет войти в критический раздел.

Хотя описанный сценарий маловероятен, и вряд ли такая последовательность продлится долго, тем не менее теоретически такая возможность имеется. Поэтому и четвертую попытку следует отвергнуть как неудачную.

Правильное решение

Необходимо следить за состоянием обоих процессов, что обеспечивается массивом *flag*. Но, как показала четвертая попытка, этого недостаточно. Нужно навязать определенный порядок действий двум процессам, чтобы избежать проблемы «взаимной вежливости», которая была описана выше. С этой целью можно использовать переменную *turn* из первой попытки. В данном случае эта переменная указывает, какой из процессов имеет право на вход в критический раздел.

Можно описать это решение следующим образом. Когда процесс *P0* намерен войти в критический раздел, он устанавливает свой флаг равным true, а затем проверяет состояние флага процесса *P1*. Если он равен *false*, *P0* может немедленно входить в критический раздел, в противном случае *P0* обращается к переменной *turn*. Если *turn = 0*, это означает, что сейчас – очередь процесса *P0* на вход в критический раздел, и *P0* периодически проверяет состояние флага процесса *P1*. Этот процесс, в какой-то момент времени обнаруживает, что сейчас не его очередь для входа в критический раздел, и устанавливает

свой флаг равным *false*, давая возможность процессу *P0* войти в критический раздел. После того как *P0* выйдет из критического раздела, он установит свой флаг равным *false* для освобождения критического раздела и присвоит переменной *turn* значение *1* для передачи прав на вход в критический раздел процессу *P1*.

Листинг: алгоритм Деккера

```
boolean  flag[2];
int  turn;
Процесс 0                                   Процесс 1
{ while(true)                               {while (true)
    {flag[0]   =  true;                         {flag [1] = true;
    while(flag[l])                              while(flag[0])
    if   {turn ==  1)                           if (turn == 0)
    {flag[0] = false;                           {flag[l]=false;
     while(turn == 1)                            while(turn==0)
    /* Ничего не делать */;                      /*Ничего не делать */;
    flag[0] = true; }                           flag[l] = true;}
    /* Критический раздел */;                   /* Критический раздел */;
    turn = 1;                                   turn=0;
    flag[0] = false;                            flag[1] = false;
     /* Остальной код */;                    /* Остальной код */;
}                                           }
     main()
    {flag[0] = false;   flag[l] = false;
    turn =1;
    parbegin(PO,P1);
    }
```

2.1.2. Алгоритм Питерсона

Алгоритм Деккера решает задачу взаимных исключений, но достаточно сложным путем, корректность которого не так легко доказать. Питерсон (Peterson) предложил простое и элегантное решение. Как и в алгоритме Деккера глобальная переменная *flag* указывает положение каждого процесса по отношению к взаимоисключению, а глобальная переменная *turn* разрешает конфликты одновременности.

Листинг: алгоритм Питерсона для двух процессов

```
boolean flag [2];
int turn;
Процесс 0                          Процесс 1
 {while(true)                      {while(true)
{ flag[0] = true;                  { flag[l] = true;
 turn = 1;                         turn = 0;
while(flag[1] && turn ==1)         while(flag[0] && turn == 0)
 /* Ничего не делать */;           /* Ничего не делать */;
/* Критический раздел */;          /* Критический раздел */;
 flag[0] = false;                  flag[l] = false;
 /* Остальной код */; }            /* Остальной код */; }
}                                  }
main()
{flag[0] = false;
flag[l] = false;
 parbegin (PO,P1); }
```

Выполнение условий взаимоисключения легко показать. После того как *flag[0]* установлен процессом *P0* равным *true*, процесс *P1* не может войти в критический раздел. Если же процесс *P1* уже находится в критическом разделе, то *flag[1] = true,* и для процесса *P0* вход в критический раздел заблокирован. Однако взаимная блокировка в данном алгоритме предотвращена. Пред-

положим, что процесс *P0* заблокирован в своем цикле *while*. Это означает, что *flag[1]* равен *true*, а *turn = 1*. Процесс *P0* может войти в критический раздел, когда либо *flag[1]* становится равным *false*, либо *turn* становится равным 0. Рассмотрим три исчерпывающих случая.

1) *P1* не намерен входить в критический раздел. Такой случай невозможен, поскольку при этом выполнялось бы условие *flag [1] = false*.

2) *P1* ожидает вход в критический раздел. Такой случай также невозможен, поскольку если *turn = 1*, то *P1* способен войти в критический раздел.

3) *P1* циклически использует критический раздел, монополизировав доступ к нему. Этого не может произойти, поскольку *P1* вынужден перед каждой попыткой входа в критический раздел давать такую возможность процессу *P0*, устанавливая значение *turn* равным *0*.

Следовательно, имеется простое решение проблемы взаимных исключений для двух процессов.

Из-за наличия недостатков в случае использования как программных, так и аппаратных решений следует рассмотреть и другие механизмы обеспечения взаимоблокировок.

2.2. Семафорные примитивы Дейкстры

В 1968 году Эдсгер Дейкстра опубликовал работу, посвященную архитектуре маленькой ОС, основанной на работе небольшой постоянной группы параллельных процессов, которые синхронизировались с помощью семафоров.

Фундаментальный принцип заключается в том, что два или более процесса могут сотрудничать посредством простых сигналов, так что в определенном месте процесс может приостановить работу до тех пор, пока не дождется соответствующего сигнала. Требования кооперации любой степени сложности могут быть удовлетворены соответствующей структурой сигналов. Для сигнализации используются специальные переменные, называющиеся семафорами. ***Семафор –*** переменная специального типа, доступная параллель-

ным процессам для проведения над ней только двух операций: «закрытия» и «открытия».

Для выполнения семафорных операций Дейкстра ввел два новых примитива. Для передачи сигнала через семафор *s* процесс выполняет примитив *signal(s)*, а для получения сигнала – примитив *wait(s)*. В последнем случае процесс приостанавливается до тех пор, пока не осуществится передача соответствующего сигнала. В статье Дейкстры и многих других источниках вместо *wait* используется *P* (от голландского *proberen – проверка*), а вместо *signal – V* (от голландского *verhogen – увеличение*). Предполагается, что примитивы *wait* и *signal* атомарны, т. е. они не могут быть прерваны, и каждая из подпрограмм может рассматриваться как единый шаг.

Существует несколько типов семафоров: двоичные, считающие, счетные, множественные и др.

Двоичный семафор может принимать два значения: 0 и 1. Допустимыми значениями семафора являются целые числа. При этом возможны два варианта. Первый вариант – допустимы только целые положительные числа, включая нуль. Второй вариант – допустимы любые целые числа.

Семафор играет роль вспомогательного критического ресурса. Семафорный механизм работает по схеме, в которой сначала исследуется состояние критического ресурса, идентифицируемое значением семафора, а затем осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. При отказе доступа к критическому ресурсу используется режим «пассивного ожидания». Поэтому в состав механизма включаются средства формирования и обслуживания очереди ожидающих процессов. Эти средства реализуются супервизором операционной системы. Основным достоинством использования семафорных операций является отсутствие состояния «активного ожидания», что может существенно повысить эффективность работы мультипрограммной вычислительной системы.

Пусть *S* – семафор, операции над которым определяются следующим образом:

Wait (S): уменьшение значения семафора *S* на единицу, если это возможно. Если это значение становится отрицательным, процесс, выполняющий операцию *wait*, блокируется. Если значение *S* больше нуля, процесс продолжает работу.

Signal (S): если ни один процесс не ждет у данного семафора, *S* увеличивается на единицу. В противном случае – освобождается один процесс и его работа продолжается с инструкции, следующей за инструкцией *wait*. Переменная *S* увеличивается на единицу одним неделимым действием. Выборка, инкремент и запоминание не могут быть прерваны, и другим процессам отказано в доступе к *S* во время выполнения этой операции.

В частном случае, когда семафор *S* может принимать значения только нуль и единица, он превращается в блокирующую переменную, называемую двоичным семафором. Операция *wait*, заключает в себе потенциальную возможность перехода процесса, который ее выполняет, в состояние ожидания, в то время как *signal* может при некоторых обстоятельствах активизировать другой процесс, приостановленный операцией *wait*.

Семафоры могут применяться для разных целей. Их можно использовать для взаимного исключения, синхронизации взаимодействующих процессов и управления выделением ресурсов. В зависимости от конкретного способа применения используются разные начальные значения семафора и различные последовательности вызова операций *signal(S)* и *wait(S)*.

С помощью семафора, инициализированного значением *1*, можно обеспечить монопольный доступ к общему ресурсу, например к совместно используемой структуре данных.

Одна из возможных схем доступа каждого из трех процессов, обозначенных как *А*, *Б* и *В*, к общему ресурсу показана на рис. 7. Этот ресурс защищен семафором, который инициализирован значением *1*. Процесс *А* выполняет вызов *wait(S)* и входит в свою критическую секцию, где производится доступ к общему ресурсу. Пока *А* находится в критической секции, *Б* пытается войти в свою критическую секцию, связанную с тем же ресурсом, для чего также осуществляет вызов *wait(S)*. Затем *А* выходит из своей критической секции, выполнив вызов *signal(S)*. После этого *Б* входит в критическую секцию. Большую часть времени процессы *А* и *Б* могут функционировать параллельно, однако их критические секции, связанные с одним и тем же ресурсом, должны выполняться строго последовательно, чтобы в каждый конкретный момент времени к ресурсу обращался только один процесс.

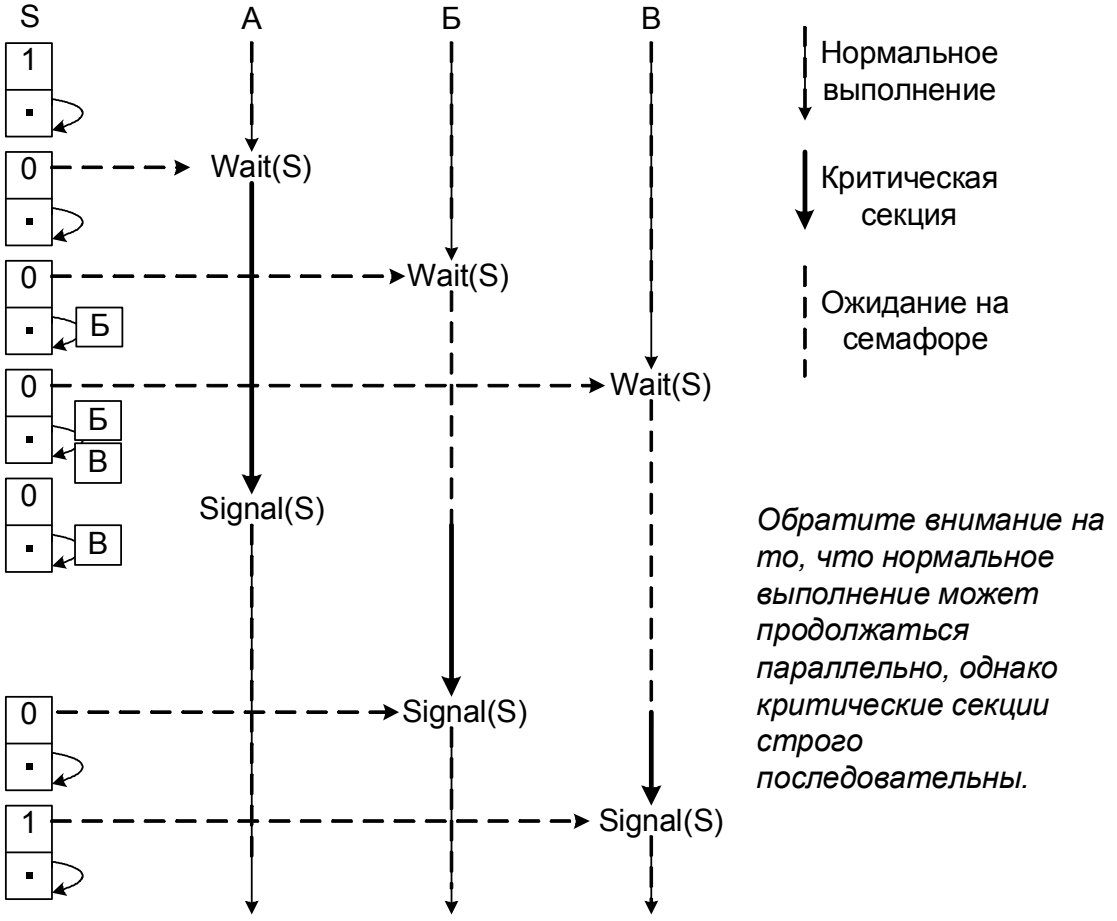

Рис. 7. Обращение процессов к совместно используемым данным, защищенным семафором

Необходимо обратить внимание на то, что описанный алгоритм основан на определенной схеме планирования процессов, ожидающих у семафора: первым в критическую секцию входит процесс, который первым запросил ее использование. Способ планирования процессов зависит от конкретной реализации и не является частью концепции семафора. Например, при выполнении одним из процессов вызова *signal(S)* можно освобождать все ожидающие процессы, и тогда в критическую секцию войдет тот из них, который в очередной раз первым выполнит вызов *wait(S)*. В этом случае выбор процесса будет определяться алгоритмом планирования, используемым в операционной системе. Еще один вариант заключается в организации планирования для очереди семафора на основе приоритетов процессов, которые в этом случае должны быть известны семафору. Временные диаграммы процессов приведены на рис. 8.

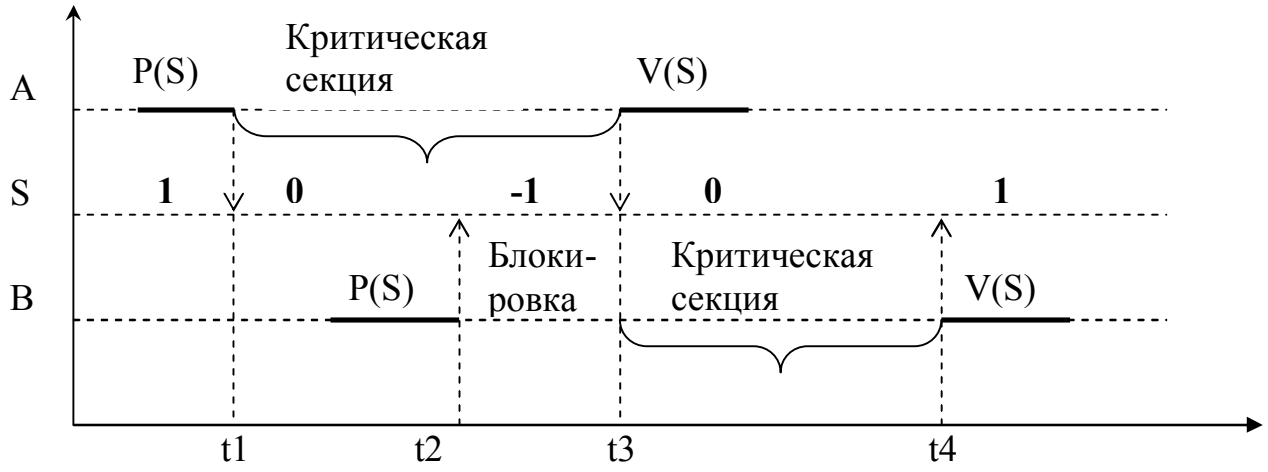

Рис. 8. Временная диаграмма для двоичного семафора

С помощью семафоров можно синхронизировать действия двух процессов. Пусть два процесса связаны следующим образом: при достижении процессом *А* определенной точки он не может продолжать работу, пока процесс *Б* не выполнит некоторую задачу. Для их синхронизации можно применить инициализированный значением *0* семафор, у которого процесс *А* должен ждать в точке синхронизации (выполнив вызов *wait(S)*), пока процесс *Б* не вызовет метод *signal(S)*.

Описанную схему можно обобщить для нескольких сигнализирующих процессов. Семафор, используемый таким образом, называется *приватным:* ждать у него может только один процесс, а сигнализировать с его помощью о программных событиях – произвольное количество процессов.

С каждым семафором может связываться список процессов ожидающих разрешение пройти семафор. ОС может назначить процесс на выполнение, заблокировать исполняющийся процесс и поместить его в семафорную очередь, деблокировать процесс, возобновив его исполнение. Находясь в списке заблокированных, ожидающий у семафора процесс находится в состоянии неактивного ожидания.

До сих пор рассматривался доступ нескольких процессов к одному ресурсу, осуществляемый в режиме взаимного исключения. Но также возможна и другая ситуация, когда несколько процессов имеют возможность обращаться к ресурсу одновременно, но количество таких обращений должно быть ограничено. Например, если в состав компьютерной системы входят два одинаковых принтера, для процессов не имеет значения, на каком из них печатать. Эти принтеры фактически представляют собой один ресурс, которым одновременно могут пользоваться два процесса. Соответственно семафор этого ресурса можно инициализировать значением *2*. Для взаимного исключения (т. е. использования ресурса в каждый конкретный момент времени не более чем одним процессом) семафор инициализируется значением *1*.

Каждый раз, когда некоторый процесс запрашивает доступ к ресурсу и значение семафора этого ресурса больше нуля, данное значение уменьшается на единицу. По достижении им значения нуль последующие попытки доступа к ресурсу должны быть блокированы, а предпринявшие их процессы установлены в очередь. После того как процесс освобождает ресурс, семафор проверяет, имеются ли в очереди другие процессы. При их отсутствии значение семафора увеличивается на единицу, если же таковые находятся в очереди, одному из них предоставляется доступ к ресурсу. Для того чтобы семафор можно было использовать описанным способом, его примитив *wait(S)* устанавливает вызывающий процесс в очередь. Освобождая ресурс, процесс должен вызвать примитив его семафора *signal(S)*. При этом каждый ресурс следует защитить собственным семафором.

При выполнении *wait(S)* и *signal(S)* примитивов для счетного семафора допускается изменение семафора не на *1*, а на любое определенное число *R*. Такое изменение обозначается: *wait(S,R)* и *signal(S,R)*. Если в результате выполнения примитива *wait(S)* полученное значение семафора остается положительным, то процесс продолжается. В противном случае процесс «засыпает» на семафоре.

Особенностью механизма множественных семафоров является возможность проверки в одном примитиве не одного, а нескольких семафоров и обработки их по определенным правилам. Очередь ожидания в таком случае может быть связана с составным условием, которое проверяется в соответствующих примитивах.

Семафоры являются универсальным и гибким средством, обеспечивающим взаимодействие между процессами только в режиме «один-к-одному» (между двумя именованными процессами). Операции *wait(S)* и *signal(S)* могут использоваться многими процессами, первая – для ожидания сигналов о событиях, поступающих от более чем одного процесса, а вторая – для подачи сигнала одному ожидающему процессу.

Если единственным средством взаимодействия между процессами, которое предоставляет некоторый язык программирования, является класс семафора, то в этом случае возникает ряд важных для определенных типов систем проблем, перечисленных ниже.

Применение семафоров не является обязательным, и их используют только для удобства. При программировании с их использованием возможны ошибки. Достаточно сложно запомнить, какой семафор связан с каждой из структур данных, процессов или ресурсов. Программист может также забыть вызвать операцию *wait* и сразу обратиться к незащищенным общим данным, или забыть осуществить вызов *signal* и оставить структуру данных заблокированной на неопределенное время.

Проверить, занят ли семафор, можно только с помощью блокирующего вызова *wait*. В некоторых ситуациях, когда семафор занят, предпочтительно не ждать его освобождения, а перейти к другим действиям. Указанное ограничение можно обойти путем создания нового процесса, чтобы один процесс продолжал работу, а второй ждал у семафора.

Операция *wait* не позволяет задать в качестве аргумента список семафоров (например, чтобы запрограммировать альтернативный порядок действий в соответствии с порядком поступления сигналов). Реализация такой возможности отличается трудоемкостью и требует больших затрат ресурсов. Данную проблему тоже можно решить путем создания дополнительных процессов.

Время блокирования процесса у семафора неограниченно. Процесс ждет до тех пор, пока не поступит заданный сигнал.

Семафор не является средством, с помощью которого один процесс может контролировать другой без участия самого процесса.

Если единственным механизмом взаимодействия между процессами является семафор и необходимо, чтобы они обменивались информацией, у них должно быть некоторое общее адресное пространство. Обмен данными может осуществляться по схеме «производитель – потребитель». Для передачи минимального объема информации могло бы использоваться значение семафора, но оно процессам недоступно.

2.3. Мониторы

Монитор – это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для распределения одного или группы общих ресурсов. Чтобы обеспечить выделение нужного ресурса процесс должен обратиться к процедуре монитора. Причем в фиксированный момент времени только один процесс может войти в монитор. Процессам, которые хотят войти в монитор, когда он уже занят, приходится ждать, причем режимом ожидания управляет сам монитор. Структура работы с монитором приведена на рис. 10.

Внутренние данные монитора могут быть либо глобальными (относящимися ко всем процедурам монитора), либо локальными. Ко всем данным можно обращаться только изнутри монитора.

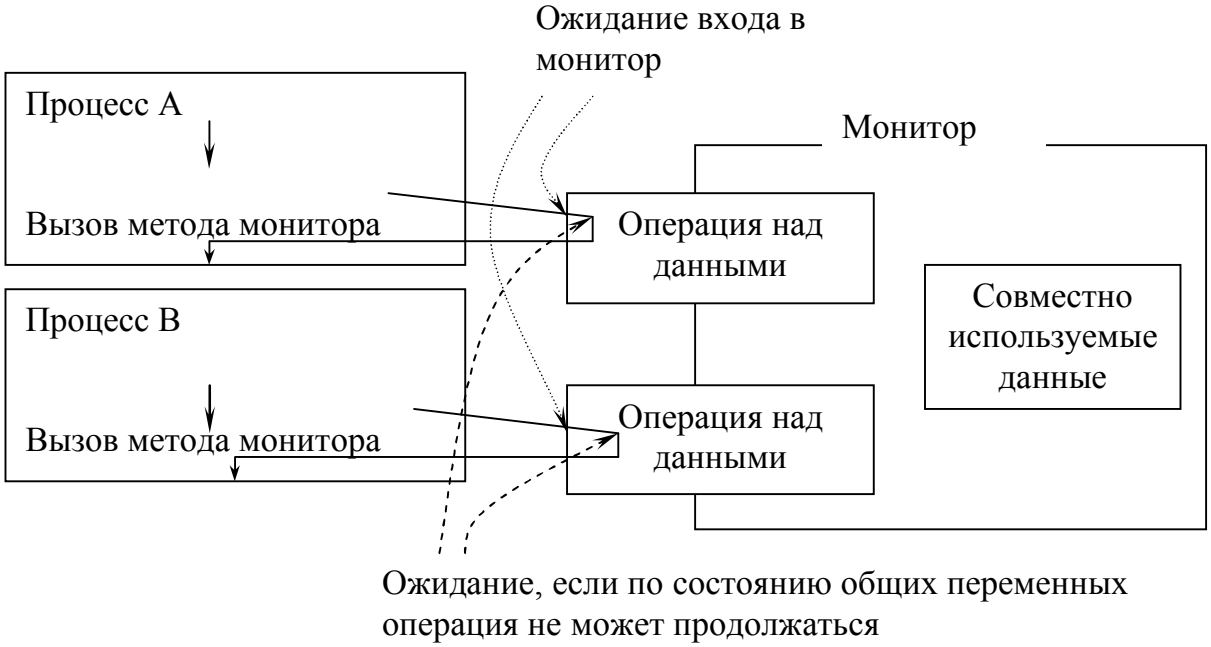

Рис. 10. Структура программы с мониторами

Монитор не является процессом, это пассивный объект, который приходит в активное состояние только тогда, когда какой-то объект обращается к нему за услугами. Часто монитор сравнивают с запертой комнатой, от которой

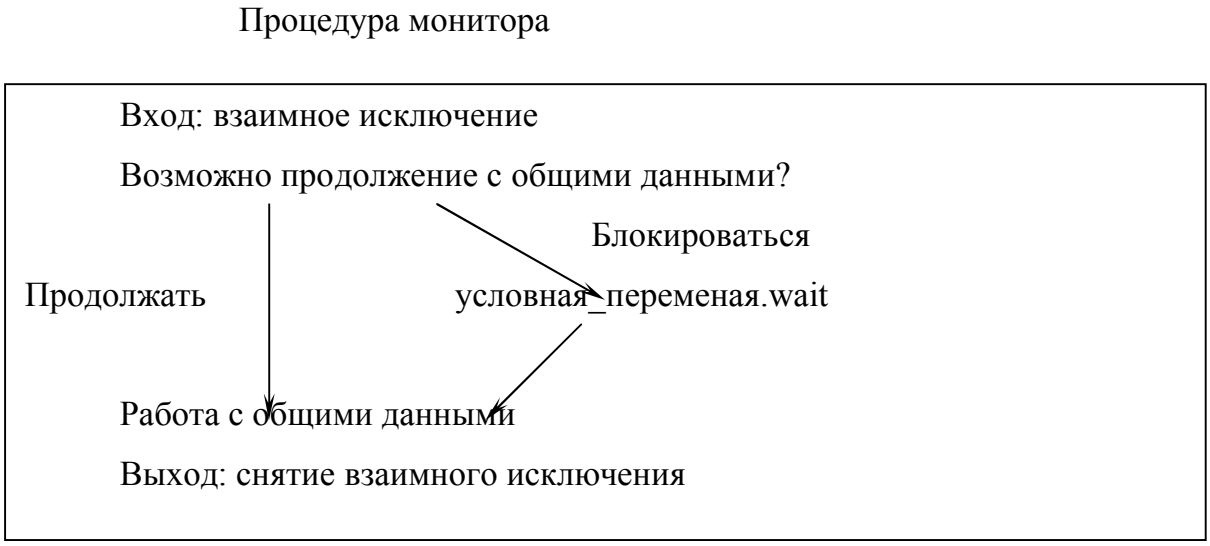

Рис. 11. Взаимное исключение в процедуре монитора

имеется только один ключ. Услугами этой комнаты может воспользоваться только тот, у кого есть ключ. При этом процессу запрещается оставаться там сколь угодно долго.

В качестве примера программы-монитора может выступать планировщик ресурсов. Действительно, каждому процессу когда-нибудь понадобятся ресурсы, и он будет обращаться к планировщику. Но планировщик одновременно может обслуживать только один ресурс.

Иногда монитор задерживает обратившийся к нему процесс. Это происходит, например, в случае обращения за занятым ресурсом. Монитор блокирует процесс с помощью команды *wait*, а в случае освобождения ресурса выдает команду *signal*. При этом освободившийся ресурс предоставляется одному из ожидавших его процессов вместе с разрешением на продолжение работы. Управление передается команде монитора, непосредственно следующей за операцией *wait* (рис. 11).

Для гарантии того, что процесс, ожидающий некоторого ресурса, со временем обязательно его получит, необходимо, чтобы ожидающий процесс имел более высокий приоритет, чем новый процесс, пытающийся войти в монитор.

Для решения большинства задач не достаточно только механизма взаимного исключения. Требуется еще и условная синхронизация. Например, если ресурс занят, либо полон буфер для записи данных, процесс должен остановиться и ждать условий необходимых для продолжения работы. Это и есть *условная синхронизация*. В большинстве систем на базе мониторов она реализуется с помощью объектов специального вида, которые называются *условными переменными*.

Если предположить, что нескольким процессам необходим доступ к определенному критическому ресурсу, тогда можно использовать простой монитор – распределитель ресурсов.

```
Monitor распределитель;
Var ресурс_занят: логический;
Ресурс_свободен: условная переменная;
Procedure  захват_ресурса
{
If  ресурс_занят
{ ресурс_свободен.Wait();
ресурс_занят:=истина;}
}
Procedure  возврат_ресурса
{
ресурс_занят:=ложь;
ресурс_свободен. Signal();
}
main()
```

{

ресурс_занят:=ложь;

}

В отличие от семафора, который реализуется как целочисленная переменная плюс очередь, условная переменная содержит только очередь. У нее нет значения или состояния. Операция *wait* всегда устанавливает вызывающий процесс в очередь, операция *signal* пробуждает один из ожидающих процессов. Если ожидающих процессов в системе нет, то ресурс возвращается в множество свободных ресурсов.

Мониторы более гибки, чем семафоры. В форме мониторов сравнительно легко можно реализовать различные синхронизирующие примитивы, в частности семафоры и почтовые ящики. Кроме того, мониторы позволяют нескольким процессам совместно использовать программу, представляющую собой критический участок.

Тесное взаимодействие между процессами предполагает не только синхронизацию – обмен временными сигналами, но и передачу, и получение произвольных данных – обмен сообщениями.

2.4. Очереди сообщений

Метод межпроцессного взаимодействия на основе сообщений позволяет нескольким процессам, выполняемым на одной машине, взаимодействовать между собой путем передачи и приема сообщений. Система организует специальную структуру, куда любой процесс может поместить свою корреспонденцию или взять предназначенное ему послание. У любого сообщения есть целочисленный идентификатор (тип), присваиваемый сообщению процессом-отправителем, благодаря чему процесс-получатель может избирательно принимать сообщения только нужных ему типов.

Метод, основанный на использовании сообщений, устраняет некоторые недостатки каналов. В частности, к обеим конечным точкам канала с целью чтения и записи данных может подключаться множество процессов. Однако

механизма, который обеспечивал бы избирательную связь между читающим и записывающим процессами, нет.

Пусть, например, процессы *A* и *B* подсоединены к конечной точке канала для записи, а процессы *C* и *D* – для чтения. Если *A* и *B* записывают данные в канал в расчете на то, что данные процесса *A* будет читать процесс *C*, а данные процесса *B* – процесс *D*, то *C* и *D* не смогут избирательно читать данные, предназначенные конкретно каждому из них. Другая проблема состоит в том, что для успешной передачи сообщения к обоим концам канала должны быть подсоединены процессы, иначе данные теряются. Таким образом, каналы и находящиеся в них данные являются нерезидентными объектами, и, если процессы выполняются не одновременно, надежный обмен данными не обеспечивается.

Механизм сообщений позволяют осуществлять доступ к центральной очереди сообщений сразу множеству процессов. При этом каждый процесс, который помещает сообщение в очередь, должен указать для своего сообщения целочисленный тип. Процесс-получатель сможет выбрать это сообщение, указав тот же тип. Кроме того, сообщения не удаляются из очереди, даже если к ней не обращается ни один процесс. Сообщения удаляются из очереди только тогда, когда процессы обращаются к ним явно. Следовательно, механизм использования сообщений позволяет осуществить более гибкое многопроцессное взаимодействие.

2.5. Почтовые ящики

Почтовый ящик – это информационная структура с заданными правилами, описывающими ее работу. Она состоит из главного элемента, где располагается описание почтового ящика, и нескольких гнезд заданных размеров для размещения сообщений.

Если процесс P_1 желает послать процессу P_2 некоторое сообщение, он записывает его в одно из гнезд почтового ящика, откуда P_2 в требуемый момент времени может его извлечь. Иногда оказывается необходимым процессу

P_1 получить подтверждение, что P_2 принял переданное сообщение. Тогда образуются почтовые ящики с двусторонней связью. Известны и другие модификации почтовых ящиков, в частности порты, многовходовые почтовые ящики и т. д. Для установления связей между процессами широко используются почтовые ящики. Примером такого применения может служить операционная система IPMX86 BC фирмы Intel для вычислительных комплексов на основе микропроцессоров IAPX86 или IAPX88. Для целей синхронизации разрабатываются специальные примитивы создания и уничтожения почтовых ящиков, отправки и запроса сообщений.

Можно выделить три типа имеющихся в системе запросов: с ограниченным ожиданием, неограниченным ожиданием или без ожидания. В случае ограниченного ожидания процесс, подавший запрос, если почтовый ящик пуст, ожидает некоторое время поступления сообщения. Если по истечении заданного кванта времени сообщение в почтовый ящик не поступило, процесс активизируется и выдает уведомление, что запрос не обслужен. В случае запроса без ожидания, если почтовый ящик пуст, указанные действия выполняются немедленно. Если время ожидания не ограничено, процесс не активизируется до поступления сообщения в почтовый ящик.

Очевидно, что почтовые ящики позволяют осуществлять как обмен информацией между процессами, так и их синхронизацию.

Почтовый ящик может быть связан с парой процессов, только с отправителем, только с получателем, или его можно получить из множества почтовых ящиков, которые используют все или несколько процессов. Почтовый ящик, связанный с процессом-получателем, облегчает посылку сообщений от нескольких процессов в фиксированный пункт назначения. Если почтовый ящик не связан жестко с процессами, то сообщение должно содержать идентификаторы и процесса-отправителя, и процесса-получателя.

Правила работы почтового ящика могут быть различными в зависимости от его сложности. В простейшем случае сообщения передаются только в

одном направлении. Процесс P_1 может посылать сообщения до тех пор, пока имеются свободные гнезда. Если все гнезда заполнены, то P_1 может либо ждать, либо заняться другими делами и попытаться послать сообщение позже. Аналогично процесс P_2 может получать сообщения до тех пор, пока имеются заполненные гнезда. Если сообщений нет, то он может либо ждать сообщений, либо продолжать свою работу. Эту простую схему работы почтового ящика можно усложнять и получать более сложные системы общения – двунаправленные и многовходовые почтовые ящики.

Двунаправленный почтовый ящик, связанный с парой процессов, позволяет подтверждать прием сообщений. Если используется множество гнезд, то каждое из них хранит либо сообщение, либо подтверждение. Чтобы гарантировать передачу подтверждений, когда все гнезда заняты, подтверждение на сообщение помещается в то же гнездо, которое было использовано для сообщения. Это гнездо уже не используется для другого сообщения до тех пор, пока подтверждение не будет получено.

Из-за того, что некоторые процессы не забрали свои сообщения, связь может быть приостановлена. Если каждое сообщение снабдить пометкой времени, то управляющая программа может периодически уничтожать старые сообщения.

Процессы могут быть также остановлены в связи с тем, что другие процессы не смогли послать им сообщения. Если время поступления каждого остановленного процесса в очередь блокированных процессов регистрируется, то управляющая программа может периодически посылать процессам пустые сообщения, чтобы они не ждали очень долго. Реализация почтовых ящиков требует использования примитивных операторов низкого уровня, таких как *P*- и *V*-операции, или каких-либо других средств, но пользователи могут использовать средства более высокого уровня, к которым относятся следующие операции:

1) *send-message* (*Получатель, Сообщение, Буфер*) - переписывает сообщение в некоторый буфер, помещает его адрес в переменную «Буфер» и до-

бавляет буфер к очереди «Получатель». Процесс, выдавший операцию *send-message*, продолжит свое исполнение;

2) *wait-message* (*Отправитель, Сообщение, Буфер*) – блокирует процесс, выдавший операцию, до тех пор, пока в его очереди не появится какое-либо сообщение. Когда процесс устанавливается на процессор, он получает имя отправителя в переменной «Отправитель», текст сообщения – в «Сообщение» и адрес буфера – в «Буфер». Затем буфер удаляется из очереди, и процесс может записать в него ответ отправителю;

3) *send-answer* (*Результат, Ответ, Буфер*) – записывает «Ответ» в тот «Буфер», из которого было получено сообщение, и добавляет буфер к очереди отправителя. Если отправитель ждет ответ, он деблокируется;

4) *wait-answer* (*Результат, Ответ, Буфер*) – блокирует процесс, выдавший операцию, до тех пор, пока в «Буфер» не поступит «Ответ». После того, как «Ответ» поступил и процесс установлен на процессор, «Ответ» переписывается в память процессу, а буфер освобождается. Результат указывает, является ли ответ пустым, то есть выданным операционной системой.

Основные достоинства почтовых ящиков:

1) процессу не нужно знать о существовании других процессов до тех пор, пока он не получит сообщения от них;

2) два процесса могут обмениваться несколькими сообщениями за один раз;

3) ОС может гарантировать, что никакой процесс не вмешается в «беседу» других процессов;

4) очереди буферов позволяют процессу-отправителю продолжать работу, не обращая внимания на получателя.

Основным недостатком буферизации сообщений является появление еще одного ресурса, которым нужно управлять. Другим недостатком можно считать статический характер ресурса: количество буферов для передачи сообщений через почтовый ящик фиксировано. Поэтому появились механизмы, подобные почтовым ящикам, но реализованные на принципах динамического выделения памяти под передаваемые сообщения.

3. Классические задачи синхронизации процессов

Взаимодействие между двумя или более параллельными процессами определяется задачей синхронизации. Количество различных задач синхронизации не ограничено. Однако некоторые из них являются типовыми. Большинство задач по согласованию параллельных процессов в реальных ОС можно решить либо с помощью типовых задач, либо с помощью их модификаций.

3.1. Задача «Поставщик – потребитель»

Взаимодействие двух процессов, выполняющихся в режиме мультипрограммирования, показано на рис. 12.

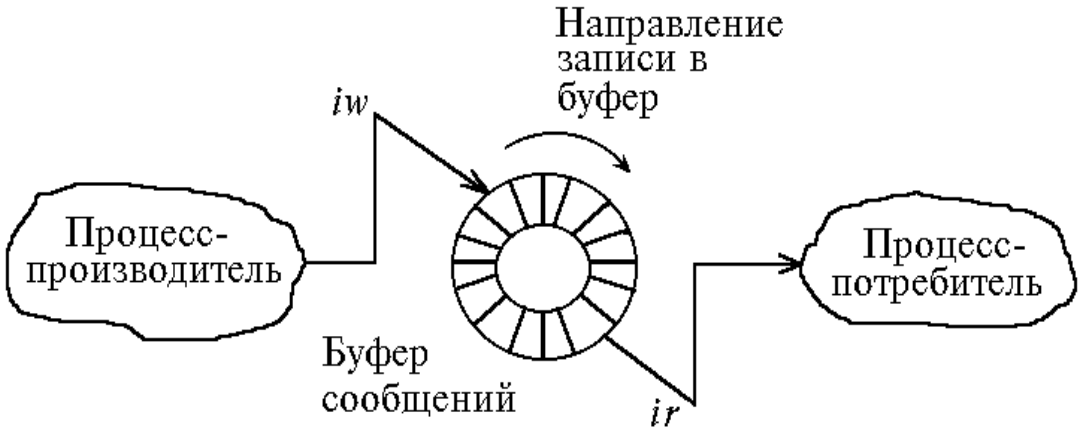

Рис. 12. Задача «Поставщик-потребитель»

Процессы взаимодействуют через некоторую обобщенную область памяти – буфер сообщений. В эту область процесс-поставщик помещает очередное сообщение, а процесс-потребитель – считывает сообщение из этой области. В общем случае буфер способен хранить несколько сообщений. Необходимо согласовать работу процессов при одностороннем обмене сообщениями, чтобы удовлетворять следующие требования:

1) выполнять взаимоисключение по отношению к критическому ресурсу (буферу сообщений);

2) учитывать состояние буфера, характеризующее возможность записи или чтения очередного сообщения. Процесс-поставщик при попытке записи в полный буфер должен быть заблокирован. Попытка процесса-потребителя чтения из пустого буфера также должна быть блокирована.

3.2. Задача «Читатель – писатель»

Другой классической задачей синхронизации доступа к ресурсам является задача «Читатель – писатель», иллюстрирующая широко распространенную модель совместного доступа к данным (рис. 13). Например, в системе, резервирующей билеты, множество конкурирующих процессов читают одни и те же данные. Однако, когда один из процессов начинает модифицировать данные, ни какой другой процесс не должен иметь доступа к этим данным, даже для чтения.

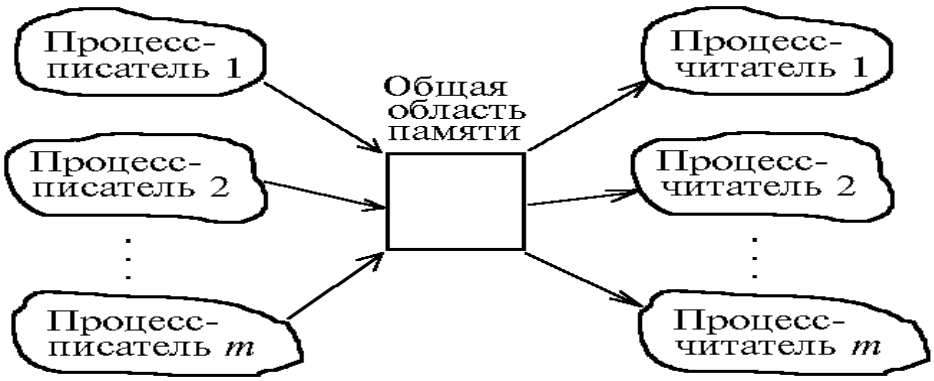

Рис. 13. Задача «Читатель – писатель»

Задачу «Читатель-писатель» можно решать, исходя из следующих предположений.

1) Наибольший приоритет имеет читатель: в этом случае любому из процессов – писателей запрещается вход в критическую секцию до тех пор, пока в ней находится хотя бы один читающий процесс.

```
взаимн_искл= 1;/* контроль за доступом к «rc» (разделямый ресурс) */
доступ = 1;          /* контроль за доступом к базе данных */
число_процессов = 0;    /* кол-во процессов читающих или пишущих */

Procedure читатель ;
{
    while (TRUE)                    /* бесконечный цикл */
    {wait (взаим_искл);       /* получить эксклюзивный доступ к «rc»*/
    число_процессов  = число_процессов + 1;/* еще одним читателем
                                                    больше */
    if (число_процессов == 1)  wait (доступ);          /* если это первый
                читатель, нужно заблокировать эксклюзивный
                доступ к базе */
    signal(взаим_искл);              /*освободить ресурс  */
    read_data_base(); /* доступ к данным */
    wait (взаим_искл);         /*получить эксклюзивный доступ к
                                        ресурсу*/
    число_процессов = число_процессов - 1:  /* теперь одним
                                            читателем меньше */
    if (число_процессов == 0) signal (доступ);         /*если это был
        последний читатель, разблокировать эксклюзивный доступ
        к базе данных */
    signal (взаим_искл);       /*освободить разделяемый ресурс  */
    use_data_read();  /* некритическая секция */
```

```
        }
}
```

```
Procedure  писатель ;
{
        while(TRUE)                     /* бесконечный цикл */
        {think_up_data();  /* некритическая секция */
        wait (доступ);      /* получить эксклюзивный доступ к данным*/
        write_data_base();          /* записать данные */
        signal (доступ);   /* отдать эксклюзивный доступ */
        }
}
```

Процесс, ожидающий доступа по записи, будет ждать до тех пор, пока все читающие процессы не окончат работу, и если в это время появляется новый читающий процесс, он тоже беспрепятственно получит доступ. Чтобы этого избежать, можно модифицировать алгоритм.

2) Наибольший приоритет имеет писатель. Тогда в случае, если имеется хотя бы один ожидающий процесс-писатель, новые процессы-читатели не получают доступ к ресурсу, а ожидают, когда процесс-писатель обновит данные.

Отрицательная сторона данного решения заключается в том, что оно несколько снижает производительность процессов-читателей, так как вынуждает их ждать в тот момент, когда ресурс не занят в эксклюзивном режиме.

3.3 Задача «Обедающие философы»

За круглым столом расставлены пять стульев, на каждом из которых сидит определенный философ. В центре стола – большое блюдо спагетти, на столе лежат пять вилок – каждая между двумя соседними тарелками. Каждый философ может находиться только в двух состояниях: либо он размышляет, либо ест спагетти. Начать думать философу ничего не мешает. Чтобы начать есть, нужны две вилки (одна в правой и одна в левой руке). Закончив еду, философ кладет вилки справа и слева от своей тарелки и начинает размышлять до тех пор, пока снова не проголодается (рис. 14).

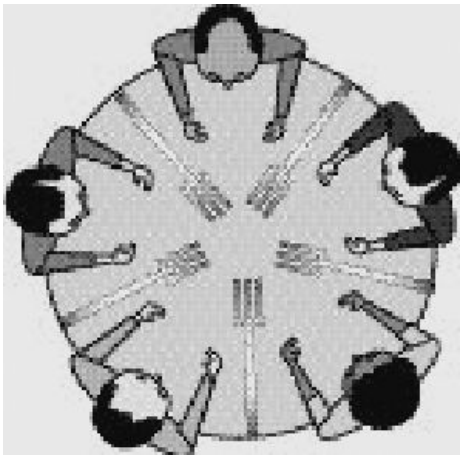

Рис. 14. Задача «Обедающие философы»

В представленной задаче имеются две опасные ситуации: ситуация голодной смерти и ситуация голодания отдельного философа. Ситуация голодной смерти возникает в случае, когда философы одновременно проголодаются и одновременно попытаются взять, например, свою левую вилку. В данном случае возникает тупиковая ситуация, так как никто из них не может начать есть, не имея второй вилки. Для исключения ситуации голодной смерти были предложены различные стратегии распределения вилок.

Ситуация голодания возникает в случае заговора двух соседей слева и справа против определенного философа. Заговорщики поочередно забирают вилки то слева, то справа от него. Такие согласованные действия злоумышленников приводят жертву к вынужденному голоданию, так как он никогда не сможет воспользоваться обеими вилками.

Существует простейшее решение с использованием семафоров: когда один из философов захочет есть, он берет вилку слева от себя, если она в наличии, а затем – вилку справа от себя. Закончив есть, он возвращает обе вилки на свои места. На первый взгляд все просто, однако данное решение может привести к тупиковой ситуации. Что произойдет, если все философы захотят есть в одно и то же время? Каждый из них получит доступ к своей левой вилке и будет находиться в состоянии ожидания второй вилки до бесконечности.

Другим решением может быть алгоритм, который обеспечивает доступ к вилкам только четырем из пяти философов. Тогда всегда среди четырех философов по крайней мере один будет иметь доступ к двум вилкам. Данное решение не имеет тупиковой ситуации. Алгоритм решения может быть представлен следующим образом:

```
# define N 5                    /* количество философов */
# define LEFT (i-1)%N   /* номер левого соседа для i-ого философа */
# define RIGHT (i+1)%N       /* номер правого соседа для i-ого
                               философа*/
# define THINKING 0           /* философ думает */
# define HUNGRY 1             /* философ голоден */
# define EATING 2             /* философ ест */

int state[N]={0,0,0,0,0};     /* массив состояний философов */
semaphore mutex=1;            /* семафор для критической секции */
semaphore s[N];               /* по одному семафору на философа */

void philosopher (int i)      /* i : номер философа от 0 до N-1 */
{
      while (TRUE)                  /* бесконечный цикл */
            {
                  think();          /* философ думает */
```

```
            take_forks(i);        /*философ берет обе вилки  или
        блокируется */
            eat();                /* философ ест */
            put_forks(i);       /* философ освобождает обе вилки */
        }
}
void take_forks(int i)      /* i : номер философа от 0 до N-1 */
{
        down(mutex);            /* вход в критическую секцию */
        state[i] = HUNGRY;      /*записываем, что i-ый  философ
                                голоден */
        test(i);            /* попытка взять обе вилки */
        up(mutex);          /* выход из критической секции */
        down(s[i]);         /* блокируемся, если вилок нет */
}

void put_forks(i)           /* i : номер философа от 0 до N-1 */
{
    down(mutex);            /* вход в критическую секцию */
    state[i] = THINKING;    /* философ закончил есть */
    test(LEFT);/* проверить может ли левый сосед сейчас есть */
    test(RIGHT);/* проверить может ли правый сосед сейчас есть*/
    up(mutex);   /* выход из критической секции */
}
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]);
        }
    }
```

3.4. Задача «Спящий парикмахер»

Эта задача имеет несколько вариантов. В простейшем случае в задаче рассматривается парикмахерская, в которой работает один парикмахер, имеется одно кресло для стрижки и несколько кресел в приемной для посетителей, ожидающих своей очереди. Если в парикмахерской нет посетителей, парикмахер засыпает прямо на своем рабочем месте. Появившийся посетитель должен его разбудить, в результате чего парикмахер приступает к работе. Если в процессе стрижки появляются новые посетители, они должны либо подождать своей очереди, либо покинуть парикмахерскую, если в приемной нет свободного кресла для ожидания. Задача состоит в том, чтобы корректно запрограммировать поведение парикмахера и посетителей.

Понадобится 3 семафора: *customers* – подсчитывает количество посетителей, ожидающих в очереди, *barbers* – обозначает количество свободных парикмахеров (в случае одного парикмахера его значения либо 0, либо 1) и *mutex* – используется для синхронизации доступа к разделяемой переменной *waiting*. Переменная *waiting*, как и семафор *customers*, содержит количество посетителей, ожидающих в очереди. Она используется в программе для того, чтобы иметь возможность проверить, есть ли свободное кресло для ожидания, и при этом не заблокировать процесс, если кресла не окажется (в этом случае посетитель должен покинуть парикмахерскую). Как и в предыдущем примере, эта переменная является разделяемым ресурсом, и доступ к ней охраняется семафором *mutex*.

```
semaphore customers = 0;        /* посетители, ожидающие в очереди */
semaphore barbers = 0;          /* парикмахеры, ожидающие посетителей */
semaphore mutex = 1;    /* контроль за доступом к переменной waiting */
int waiting = 0;
```

```
void barber()
{
        while (true) {
                        down(customers); /* если customers == 0, т.е.
                        посетителей нет, то заблокируемся до появления
                        посетителя */
                        down(mutex);        /* получаем доступ к waiting */
                        waiting = wating – 1;      /* уменьшаем кол-во
                        ожидающих клиентов */
                        up(barbers);                /* парикмахер готов к работе
                        */
                        up(mutex);          /* освобождаем ресурс waiting */
                        cut_hair();         /* процесс стрижки */
                }
}
void customer()
{
        down(mutex);        /* получаем доступ к waiting */
        if (waiting < CHAIRS)    /* есть место для ожидания */
        {
                waiting = waiting + 1; /* увеличиваем кол-во ожидающих клиентов
        */
                up(customers);              /* если парикмахер спит, это его разбудит
        */
                up(mutex);   /* освобождаем ресурс waiting */
                down(barbers);      /* если парикмахер занят, переходим в
        состояние ожидания, иначе – занимаем парикмахера*/
```

```
            get_haircut();              /* процесс стрижки */
        }
        else
            up(mutex);  /* нет свободного кресла для ожидания – придется
            уйти */
}
```

Сейчас необходимо рассмотреть более сложный пример работы парикмахерской. Пусть в парикмахерской три кресла, три парикмахера, зал ожидания, в котором четыре клиента могут разместиться на диване, а остальные – стоя. Правила пожарной безопасности ограничивают общее количество клиентов внутри помещения двадцатью людьми. В данном примере предполагается, что всего мастерская должна обслужить пятьдесят клиентов.

Клиент не может войти в парикмахерскую, если она полностью заполнена другими клиентами. Оказавшись внутри, клиент либо присаживается на диван, либо стоит, если диван занят. Когда парикмахер освобождается, к нему отправляется клиент с дивана, который дольше всех ждал своей очереди; если имеются стоящие клиенты, то тот из них, кто ожидает дольше других, присаживается на диван. По окончании стрижки принять плату может любой парикмахер, но, так как кассир в парикмахерской лишь один, плата принимается в один момент времени только от одного клиента. Рабочее время парикмахера разделяется на стрижку, принятие оплаты от клиента и сон в своем кресле в ожидании очередного клиента.

В листинге показана реализация парикмахерской с использованием семафоров; предполагается, что все очереди семафоров обрабатываются по принципу «первым вошел – первым вышел».

Основное тело программы активизирует процессы пятидесяти клиентов, трех парикмахеров и одного кассира. Назначение различных синхронизирующих операторов следующее.

1) Вместимость парикмахерской и дивана управляется семафорами *max_capacity* и *sofa* соответственно. Каждый раз при попытке клиента войти в парикмахерскую значение семафора *max_capacity* уменьшается; когда клиент покидает парикмахерскую, оно увеличивается. Если парикмахерская заполнена, то процесс клиента приостанавливается функцией *wait (max_capacity)* . Аналогично обрабатывается и попытка присесть на диван.

2) В наличии имеются три парикмахерских кресла, и следует обеспечить их корректное использование. Семафор *barber_chair* гарантирует одновременное обслуживание не более трех клиентов. Клиент не поднимется с дивана до тех пор, пока не окажется свободным хотя бы одно кресло (вызов *wait (barber_chair)),* а каждый парикмахер сообщает о том, что его кресло освободилось (вызов *signal (barber_chair)*). Справедливый доступ к парикмахерским креслам гарантируется организацией очередей семафоров: клиент, который первым блокирован в очереди, первым же и приглашается на стрижку. Если в процедуре клиента вызов *wait (barber_chair)* разместить после *signal (sofa),* то каждый клиент будет только присаживаться на диван, после чего немедленно вставать и занимать стартовую позицию у кресла.

3) Семафор *cust_ready* обеспечивает подъем спящего парикмахера, сообщая ему о новом клиенте.

4) Если клиент оказался в кресле, он должен высидеть там до окончания стрижки, о котором просигнализирует семафор *finished*.

5) Семафор *leave_b_chair* не позволяет парикмахеру пригласить нового клиента до тех пор, пока предыдущий не покинет кресло.

6) Кассир должен быть уверен, что каждый клиент, покидая парикмахерскую, расплатится, а каждый клиент, оплатив стрижку, должен получить чек. Каждый клиент, покинув кресло, оплачивает услуги парикмахера, после

чего дает знать об этом кассиру (вызов *signal (payment)*) и дожидается получения кассового чека (вызов *wait (receipt)*).

7) В целях экономии средств парикмахерская не нанимает отдельного кассира. Это действие выполняет парикмахер, когда не стрижет клиента. Для того чтобы обеспечить выполнение парикмахером в один момент времени только одной функции, используется семафор *coord*.

3.5. Задача о курильщиках

Еще одной задачей, на примере которой можно наиболее полно проиллюстрировать применение различных аппаратов синхронизации, является задача о курильщиках.

Три курильщика сидят за столом. У одного есть табак, у другого – бумага, у третьего – спички. На столе могут появиться извне два из трех упомянутых предмета. Появившиеся предметы забирает тот курильщик, у которого будет полный набор предметов. Он сворачивает папиросу, раскуривает ее и курит. Новые два предмета могут появиться на столе только после того, как он закончит курить. Другие курильщики в это время не могут начать курение.

Данную задачу можно решить, используя различные семафорные примитивы. В данном пособии не приводится примера решения описанной выше задачи. Студентам предлагается выполнить ее самостоятельно в соответствии с заданием на лабораторную работу.

3.6. Алгоритм банкира

ОС (банкир) обладает конечным числом страниц (конечным капиталом в $). Она (он) обслуживает пользовательские процессы (клиентов), которые могут занимать у нее (у него) страницы ($) на следующих условиях.

1) Пользовательский процесс (клиент) делает запрос (заем) для выполнения некоторой работы (для совершения сделки), которая будет завершена за конечный промежуток времени.

2) Пользовательский процесс (клиент) должен заранее указать максимальную потребность в страницах (в $) для выполнения работы (для совершения сделки).

3) Пока запрос (заем) не превышает заранее установленную потребность, пользовательский процесс (клиент) может увеличивать или уменьшать свой заем страницу за страницей ($ за $).

4) Пользовательский процесс (клиент), который просит увеличить его текущий заем, обязуется принимать без недовольства следующий ответ: «Если я дал вам запрашиваемую страницу ($) и вы еще не превысили бы свою установленную потребность, и поэтому вы имеете право на следующую страницу ($). Однако в настоящий момент, по некоторым причинам, мне неудобно ее (его) дать, но я обещаю вам страницу ($) в должное время».

5) Гарантия для пользовательского процесса (клиента) того, что этот момент действительно наступит, основана на предусмотрительности ОС (банкира) и на том факте, что остальные пользовательские процессы (клиенты) подчиняются тем же условиям, что и он сам. Как только пользовательский процесс (клиент) получает страницу ($), он приступает к своей работе (к своей сделке) с ненулевой скоростью, т. е. в конечный промежуток времени он или запросит новую страницу (новый $), или возвратит страницу ($), или закончит работу (сделку), что означает возвращение всех страниц (всего займа).

Основными вопросами являются:

1) при каких условиях ОС (банкир) может приступить к обслуживанию нового пользовательского процесса (заключить контракт с новым клиентом)?

2) при каких условиях ОС (банкир) может выдать следующую страницу ($) пользовательскому процессу (клиенту), не опасаясь попасть в тупиковую ситуацию?

Кроме этого имеется ряд предположений для разрабатываемой программы.

Например, пользовательские процессы пронумерованы от 1 до N и страницы пронумерованы от 1 до M. С каждым процессом связана переменная *номер_страницы*, значение которой после очередной выдачи страницы определяет номер только что выданной страницы. В свою очередь с каждой страницей связана переменная *номер_процесса*, значение которой указывает процесс, которому она была выдана.

Каждый пользовательский процесс имеет переменную состояния *процесс_состояние* – переменная клиента; при этом *процесс_состояние=1* означает: «Мне нужна страница» (иначе *процесс_состояние=0*). Каждая страница имеет переменную состояния *страница_состояние*; при этом *страница_состояние=1* означает, что она находится среди буферного пула свободных страниц. С каждым процессом и каждой страницей связаны двоичные семафоры: *процесс_семафор* и *страница_семафор.*

Предполагается, что каждая страница выдается и возвращается на основании указаний пользовательского процесса, но процесс не имеет возможности возвратить взятую страницу мгновенно. После того, как процесс заявляет, что страница ему больше не нужна, последняя не становится немедленно доступной для последующего использования. Фактически возврат страницы заканчивается после того, как та действительно присоединится к буферному пулу свободных страниц. Об этом страница будет сообщать процессу с помощью общего семафора процесса *возвращенные_страницы*. Операция *wait* над этим семафором должна предостеречь пользовательский процесс от неумышленного превышения своей максимальной потребности. Перед каждым запросом страницы процесс выполнит *wait*-операцию над семафором *возвращенные_страницы*, а начальное значение переменной *возвращенные_страницы* полагается равным потребности.

Процедура попытки выделить страницу процессу j:

{int i, свободные_страницы;

```
boolean int, завершение_под_сомнением[N];
if процесс_пер[j]=1
        {свободные_страницы=Объем_буфера-1;
        требование[j]=требование[j]-1;
        заем[j]=заем[j]+1;
        for(i=1;i<=N;i++)
        завершение_под_сомнением[i]=true;
        LO: for(i=1;i<=N;i++)
                {if завершение_под_сомнением[i] and
                требование[j]<=свободные_деньги
                    {if i!=j
                        {завершение_под_сомнением[i]=false;
                        свободные_деньги=свободные_деньги+заем[i];
                        goto LO;}
                    else
                        {i=0;
                        L1: i=i+1;
                        if страница_пер[i]=0 goto L1;
                        номер_страницы[j]=i;
                        номер_процесса[i]=j;
                        процес_пер[j]=0;
                        страница_пер[i]=0;
                        Объем_буфера=Объем_буфера-1;
                        попытка_выделить_страницу_процессу=true;
                        V(процесс_сем[j];
                        V(страница_сем[j]; return;
                        }
                    }
                }
```

требование[j]=требование[j]+1;

заем[j]=заем[j]-1;

}

попытка_выделить_страницу_процессу=false;

}

взаимоискл=1; Объем_буфера=M;

for (k=1;k<=N;k++)

{заем[k]=0; процесс_сем[k]=0;процесс_пер[k]=0;

требование[k]=потребность[k];

возвращенные_страницы[k]=потребность[k];}

У процесса запрос новой страницы (часть программы процесса) описывается следующей последовательностью операторов:

P(возвращенные_страницы[n]);

P(взаимоискл);

процесс_пер[n]=1;

попытка_выделить_страницу_процессу[n];

V(взаимоискл);

P(процесс_сем[n]);

После завершения последнего оператора значение переменной <номер страницы[n]> определяет только что выданную страницу; процесс может использовать этот номер, но обязан возвратить его в надлежащее время ОС.

Структура программы для страницы:

Страница m:

{ int h;

L: P(страница_сем[m]); // разблокирует процесс

/ теперь номер_процесса[m] указывает процесс, которому выдана страница. Страница обслуживает процесс до тех пор, пока она необходима.*

*Для возвращения в буферный пул страница поступает следующим образом: */*

```
P(взаимоискл);
требование[номер_процесса[m]]=
                требование[номер_процесса[m]]+1;
заем[номер_процесса[m]]= заем[номер_процесса[m]]-1;
страница_пер[m]=1;
V_буфера= V_буфера+1
/* Сообщение процессу*/
V(возвращенные_страницы [номер_процесса[m]]);
J: V(взаимоискл);
goto L;
                }
```

4. Задания для лабораторных работ

Указания. Программа должна быть реализована на языке высокого уровня. Право выбора языка предоставляется студенту. Для реализации взаимодействующих процессов использовать класс TThread. Реализовать корректную передачу данных. Протестировать программу при различных взаимных скоростях работы процессов. Для организации различных скоростей работы использовать временные задержки.

В интерфейс программы нужно включить следующие визуальные компоненты:

1) наглядное отображение текущего содержимого буфера;

2) для каждого потока:

- компонент для регулирования скорости работы (любой по выбору студента);
- кнопки «Запуск», «Останов» и «Пауза».

Вариант 1

1. С помощью блокирующих переменных организовать работу параллельных вычислительных потоков. Первый поток записывает в файл числа (до 100), не делящиеся на 2. Второй поток считывает из файла числа, проверяет, являются ли они простыми, если нет – удаляет их из файла.
2. Задачу о производителе и потребителе с буфером на одну запись решить с помощью семафорных примитивов.

Вариант 2

1. С помощью блокирующих переменных организовать работу параллельных вычислительных потоков. Первый поток из заранее заготовленного файла считывает координаты точек и рисует окружности с центром в этих точках. Второй поток из этого же файла считывает координаты точек и рисует квадраты с левой верхней вершиной в считанной точке.
2. Задачу о писателе и читателе решить с помощью мьютекса (два читателя, приоритет писателя).

Вариант 3

1. С помощью блокирующих переменных организовать работу параллельных вычислительных потоков. Первый поток добавляет в таблицу пользователей случайным выбором имени из уже имеющегося списка. Второй поток проверяет, не дублируются ли пользователи, и заменяет имена повторяющихся словом «повтор».
2. Задачу об обедающих философах решить с помощью монитора Хоара.

Вариант 4

1. С помощью блокирующих переменных организовать работу параллельных вычислительных потоков. Первый поток записывает в файл подсчитанные значения функции Y= (ln x)/3 на интервале $x \in$ [1;15] с шагом h=1. Второй поток считывает из файла очередное значение и подсчитывает сумму всех считанных значений в переменной Z.
2. Задачу о спящем парикмахере решить с помощью семафорных примитивов.

Вариант 5

1. С помощью блокирующих переменных организовать работу параллельных вычислительных потоков. Первый поток подсчитывает сумму текущего времени ч+м+с и записывает в таблицу. Второй поток проверяет четное ли число записано в таблице, если да – устанавливает в соответствующем поле таблицы значение true, если нет – false.
2. Задачу о производителе и потребителе с буфером на одну запись решить с помощью мьютекса.

Вариант 6

1. С помощью блокирующих переменных организовать работу параллельных вычислительных потоков. Первый поток записывает в файл случайные символы. Второй поток считывает из файла символ и удаляет его, если это буква латинского алфавита.
2. Задачу о писателе и читателе решить с помощью монитора Хоара (два читателя, приоритет читателя).

Вариант 7

1. С помощью блокирующих переменных организовать работу параллельных вычислительных потоков. Первый поток записывает в файл user.txt введенные пользователем символы с инвертированным порядком их следования. Второй поток записывает в файл user.txt текущее время.
2. Задачу об обедающих философах решить с помощью семафорных примитивов.

Вариант 8

1. С помощью блокирующих переменных организовать работу параллельных вычислительных потоков. Первый поток считает площадь треугольника S, координаты вершин которого (3; 8), (9; 0), (i; j), где $i, j \in [1; 6]$, шаг 1, затем считает площадь поверхности пирамиды G с высотой равной (i+j). Второй поток считает координаты третьей вершины треугольника с координатами (0; 1), (–2; –4), (x; y) по координатам двух известных вершин и его площади S (определенной первым потоком), далее считает объем пирамиды с высотой $h \in [1; 8]$, шаг 1.
2. Задачу о спящем парикмахере решить с помощью мьютекса.

Вариант 9

1. С помощью блокирующих переменных организовать работу параллельных вычислительных потоков. Первый поток считает значение функции $y = 3x - 1,\ x \in [3; 6],$ шаг 1, и значение функции z = 2x – 1, затем создает окно с количеством строк ввода равным (x+z). Второй поток создает окно с количеством кнопок равным y.
2. Задачу о производителе и потребителе с буфером на одну запись решить с помощью монитора Хоара.

Вариант 10

1. С помощью блокирующих переменных организовать работу параллельных вычислительных потоков. Первый поток создает строку L произвольной длины с произвольным количеством букв «A» и букв «C», остальное – символы «_». Второй поток подсчитывает количество символов «_» в строке L и выводит различные сообщения в зависимости от того, четное это количество или нет.
2. Задачу о писателе и читателе решить с помощью семафорных примитивов (два читателя, приоритет писателя).

Вариант 11

1. С помощью алгоритма Деккера организовать работу параллельных вычислительных потоков. Первый поток записывает в файл числа (до 100), делящиеся на 2. Второй поток считывает из файла числа, проверяет, делятся ли они на 4 или 3, если нет – удаляет их из файла.
2. Задачу об обедающих философах решить с помощью мьютекса.

Вариант 12

1. С помощью алгоритма Деккера организовать работу параллельных вычислительных потоков. Первый поток генерирует случайным образом две координаты двух точек, записывает их в файл и рисует прямые влево от точки. Второй поток из этого файла считывает координаты точек и рисует квадраты с левой верхней вершиной в считанной точке.
2. Задачу о курильщиках решить с помощью монитора Хоара.

Вариант 13

1. С помощью алгоритма Деккера организовать работу параллельных вычислительных потоков. Первый поток добавляет в таблицу строки, содержащие случайный набор букв. Второй поток проверяет, является ли первая буква строки гласной, и заменяет такие строки словом «гласная».
2. Задачу о производителе и потребителе с буфером на одну запись решить с помощью системных функций EnterCriticalSection() и LeaveCritica1Section().

Вариант 14

1. С помощью алгоритма Деккера организовать работу параллельных вычислительных потоков. Первый поток записывает в файл подсчитанные значения функции $Y=x^2/18$ на интервале $x \in [1; 12]$ с шагом h=1. Второй поток считывает из файла очередное значение и подсчитывает сумму всех считанных значений в переменной Z.
2. Задачу о писателе и читателе решить с помощью системных функций EnterCriticalSection() и LeaveCritica1Section() (три читателя, приоритет писателя).

Вариант 15

1. С помощью алгоритма Деккера организовать работу параллельных вычислительных потоков. Первый поток подсчитывает сумму сегодняшней даты и времени (г+м+д+ч+мин+с) и записывает в таблицу. Второй поток проверяет, четное ли число записано в таблице, если да – устанавливает соответствующее поле таблицы в true, если нет – в false.
2. Задачу об обедающих философах решить с помощью системных функций EnterCriticalSection() и LeaveCritica1Section().

Вариант 16

1. С помощью алгоритма Деккера организовать работу параллельных вычислительных потоков. Первый поток записывает в файл случайные символы. Второй поток считывает из файла символ и удаляет его, если это цифра.
2. Задачу о спящем парикмахере решить с помощью монитора Хоара.

Вариант 17

1. С помощью алгоритма Деккера организовать работу параллельных вычислительных потоков. Первый поток записывает в файл user.txt введенные пользователем символы. Второй поток записывает в файл user.txt текущее время.
2. Задачу о производителе и потребителе с бесконечным буфером решить с помощью монитора Хоара.

Вариант 18

1. С помощью алгоритма Деккера организовать работу параллельных вычислительных потоков. Первый поток считает площадь треугольника S, координаты вершин которого (4; 3), (8; 12), (i, j), где i, j $\in$ [9; 14] затем считает площадь поверхности пирамиды G с высотой равной (i + j). Второй поток считает координаты третьей вершины треугольника с координатами (–2; 6), (2; –4), (x; y) по координатам двух известных вершин и его площади S (определенной первым потоком), далее считает объем пирамиды с высотой h $\in$ [4; 10] шаг 1.
2. Задачу о курильщиках решить с помощью системных функций EnterCriticalSection() и LeaveCritica1Section().

Вариант 19

1. С помощью алгоритма Деккера организовать работу параллельных вычислительных потоков. Первый поток считает значение функции $y = 2x + 3$, $x \in [1; 4]$, шаг 1, и значение функции $z = 3x - 1$, затем создает окно с количеством строк ввода равным (x+z). Второй поток создает окно с количеством кнопок равным y.
2. Задачу о производителе и потребителе с ограниченным буфером решить с помощью системных функций EnterCriticalSection() и LeaveCritica1Section().

Вариант 20

1. С помощью алгоритма Деккера организовать работу параллельных вычислительных потоков. Первый поток создает строку L произвольной длины с произвольным количеством букв «м» и «н», остальное – случайные цифры. Второй поток подсчитывает количество цифр в строке L, значение которых меньше 6, и выводит различные сообщения в зависимости от того, четное это количество или нет.
2. Задачу о производителе и потребителе с кольцевым буфером решить с помощью системных функций EnterCriticalSection() и LeaveCritica1Section().

Вариант 21

1. С помощью алгоритма Петерсона организовать работу параллельных вычислительных потоков. Первый поток записывает в файл числа (до 100), не делящиеся на 3. Второй поток считывает из файла числа, проверяет, четные они или нет, если четные – удаляет их из файла.
2. Задачу о курильщиках решить с помощью семафорных примитивов.

Вариант 22

1. С помощью алгоритма Петерсона организовать работу параллельных вычислительных потоков. Первый поток из заранее заготовленного файла считывает координаты точек и рисует треугольник, одна из вершин которого находится в считанной точке. Второй поток из этого же файла считывает координаты точек и рисует квадраты с левой нижней вершиной в считанной точке.
2. Задачу о производителе и потребителе с бесконечным буфером решить с помощью мьютекса.

Вариант 23

1. С помощью алгоритма Петерсона организовать работу параллельных вычислительных потоков. Первый поток добавляет в таблицу случайные строки, состоящие из букв и цифр. Второй поток проверяет, четное ли количество цифр, если да, то заменяет цифры символом «*».
2. Задачу о производителе и потребителе с ограниченным буфером решить с помощью семафорных примитивов.

Вариант 24

1. С помощью алгоритма Петерсона организовать работу параллельных вычислительных потоков. Первый поток записывает в файл подсчитанные значения функции $Y = (x^2/3 \ln x)$ на интервале $x \in [2; 7]$ с шагом h=1. Второй поток считывает из файла очередное значение и подсчитывает сумму всех считанных значений в переменной *Z*.
2. Задачу о производителе и потребителе с ограниченным буфером решить с помощью мьютекса.

Вариант 25

1. С помощью алгоритма Петерсона организовать работу параллельных вычислительных потоков. Первый поток подсчитывает сумму текущего времени и даты (д+мин+с) и записывает в таблицу. Второй поток проверяет, четное ли число записано в таблице, если да – устанавливает соответствующее поле таблицы в true, если нет – в false.
2. Задачу о производителе и потребителе с кольцевым буфером решить с помощью семафорных примитивов.

Вариант 26

1. С помощью алгоритма Петерсона организовать работу параллельных вычислительных потоков. Первый поток записывает в файл случайные символы. Второй поток считывает из файла символ и удаляет его, если это не буква.
2. Задачу о производителе и потребителе с кольцевым буфером решить с помощью мьютекса.

Вариант 27

1. С помощью алгоритма Питерсона организовать работу параллельных вычислительных потоков. Первый поток записывает в файл введенные пользователем символы, кроме первого и последнего. Второй поток записывает в файл текущее время и количество символов в файле.
2. Задачу о курильщиках решить с помощью мьютекса.

Вариант 28

1. С помощью алгоритма Петерсона организовать работу параллельных вычислительных потоков. Первый поток считает площадь треугольника S, координаты вершин которого (–1; –1), (2; 3), (i, j), где i, j $\in$ [4; 7], шаг 1, затем считает площадь поверхности пирамиды G с высотой равной (i+j). Второй поток считает координаты третьей вершины треугольника с координатами (3; –2), (2; 7), (x; y) по координатам двух известных вершин и его площади S (определенной первым потоком), далее считает объем пирамиды с высотой (i, j), где h $\in$ [3; 9], шаг 1.

2. Задачу о спящем парикмахере решить с помощью системных функций EnterCriticalSection() и LeaveCritica1Section().

Вариант 29

1. С помощью алгоритма Петерсона организовать работу параллельных вычислительных потоков. Первый поток считает значение функции $y = 4x - 2$, $x \in [2; 8]$, шаг 1 и значение функции $z = 2x + 1$, затем создает окно с количеством строк ввода равным (x+z). Второй поток создает окно с количеством кнопок равным y.
2. Задачу о производителе и потребителе с кольцевым буфером решить с помощью монитора Хоара.

Вариант 30

1. С помощью алгоритма Петерсона организовать работу параллельных вычислительных потоков. Первый поток создает строку L произвольной длины с произвольным количеством цифр и букв в ней. Второй поток подсчитывает количество букв и цифр в строке L и выводит различные сообщения в зависимости от того, больше букв или больше цифр.
2. Задачу о производителе и потребителе с ограниченным буфером решить с помощью монитора Хоара.

Вариант 31

1. С помощью семафорных примитивов организовать работу параллельных вычислительных потоков. Первый поток записывает в файл числа (до 100), не делящиеся на 2. Второй поток считывает из файла числа, проверяет, являются ли они простыми, если нет – удаляет их из файла.
2. Задачу об обедающих философах решить с помощью алгоритма Петерсона.

Вариант 32

1. С помощью семафорных примитивов организовать работу параллельных вычислительных потоков. Первый поток из заранее заготовленного файла считывает координаты точек и рисует окружности с центром в этих точках. Второй поток из этого же файла считывает координаты точек и рисует квадраты с левой верхней вершиной в считанной точке.
2. Задачу о спящем парикмахере решить с помощью алгоритма Деккера.

Вариант 33

1. С помощью семафорных примитивов организовать работу параллельных вычислительных потоков. Первый поток добавляет в таблицу пользователей случайным выбором имени из уже имеющегося списка. Второй поток проверяет, не дублируются ли пользователи, и заменяет имена повторяющихся словом «повтор».
2. Задачу о производителе и потребителе с буфером на одну запись решить с помощью блокирующих переменных.

Вариант 34

1. С помощью семафорных примитивов организовать работу параллельных вычислительных потоков. Первый поток записывает в файл подсчитанные значения функции $Y=(\ln x)/3$ на интервале $x \in [1; 15]$ с шагом $h=1$. Второй поток считывает из файла очередное значение и подсчитывает сумму всех считанных значений в переменной *Z*.
2. Задачу о писателе и читателе решить с помощью алгоритма Петерсона (два читателя, приоритет писателя).

Вариант 35

1. С помощью семафорных примитивов организовать работу параллельных вычислительных потоков. Первый поток подсчитывает сумму текущего времени (ч+м+с) и записывает в таблицу. Второй поток проверяет, четное ли число записано в таблице, если да – устанавливает соответствующее поле таблицы в true, если нет – в false.
2. Задачу о курильщиках решить с помощью алгоритма Деккера.

Вариант 36

1. С помощью семафорных примитивов организовать работу параллельных вычислительных потоков. Первый поток записывает в файл случайные символы. Второй поток считывает из файла символ и удаляет его, если это буква латинского алфавита.
2. Задаче о спящем парикмахере решить с помощью блокирующих переменных.

Вариант 37

1. С помощью семафорных примитивов организовать работу параллельных вычислительных потоков. Первый поток записывает в файл user.txt введенные пользователем символы с инвертированным порядком их следования. Второй поток записывает в файл user.txt текущее время.
2. Задачу о производителе и потребителе с ограниченным буфером решить с помощью блокирующих переменных.

Вариант 38

1. С помощью семафорных примитивов организовать работу параллельных вычислительных потоков. Первый поток считает площадь треугольника S, координаты вершин которого (3; 8), (9; 0), (i; j), где i, j $\in$ [1; 6], шаг 1, затем считает площадь поверхности пирамиды G с высотой равной (i+j). Второй поток считает координаты третьей вершины треугольника с координатами (0; 1), (–2; –4), (x; y) по координатам двух известных вершин и его площади S (определенной первым потоком), далее считает объем пирамиды с высотой h $\in$ [1; 8], шаг 1.
2. Задачу о писателе и читателе решить с помощью алгоритма Деккера (два читателя, приоритет читателя).

Вариант 39

1. С помощью семафорных примитивов организовать работу параллельных вычислительных потоков. Первый поток считает значение функции $y = 3x - 1$, $x \in [3; 6]$, шаг 1 и значение функции $z = 2x - 1$, затем создает окно с количеством строк ввода равным (x+z). Второй поток создает окно с количеством кнопок равным y.
2. Задачу о производителе и потребителе с ограниченным буфером решить с помощью алгоритма Деккера.

Вариант 40

1. С помощью семафорных примитивов организовать работу параллельных вычислительных потоков. Первый поток создает строку L произвольной длины с произвольным количеством букв «A» и «C», остальное – символы «_». Второй поток подсчитывает количество символов «_» в строке L и выводит различные сообщения в зависимости от того, четное это количество или нет.
2. Задачу о производителе и потребителе с неограниченным буфером решить с помощью блокирующих переменных.

Вариант 41

1. С помощью мьютексов организовать работу параллельных вычислительных потоков. Первый поток записывает в файл числа (до 100), делящиеся на 2. Второй поток считывает из файла числа, проверяет, делятся ли они на 4 или 3, если нет – удаляет их из файла.
2. Задачу о производителе и потребителе с неограниченным буфером решить с помощью алгоритма Петерсона.

Вариант 42

1. С помощью мьютексов организовать работу параллельных вычислительных потоков. Первый поток генерирует случайным образом две координаты двух точек, записывает их в файл и рисует прямые влево от точки. Второй поток из этого файла считывает координаты точек и рисует квадраты с левой верхней вершиной в считанной точке.
2. Задачу о производителе и потребителе с кольцевым буфером решить с помощью алгоритма Петерсона.

Вариант 43

1. С помощью мьютексов организовать работу параллельных вычислительных потоков. Первый поток добавляет в таблицу строки, содержащие случайный набор букв. Второй поток проверяет, является ли первая буква строки гласной, и заменяет такие строки словом «гласная».
2. Задачу о производителе и потребителе с ограниченным буфером решить с помощью алгоритма Питерсона.

Вариант 44

1. С помощью мьютексов организовать работу параллельных вычислительных потоков. Первый поток записывает в файл подсчитанные значения функции $Y= x^2/18$ на интервале $x \in [1; 12]$ с шагом h=1. Второй поток считывает из файла очередное значение и подсчитывает сумму всех считанных значений в переменной *Z*.
2. Задачу о спящем парикмахере решить с помощью алгоритма Петерсона.

Вариант 45

1. С помощью мьютексов организовать работу параллельных вычислительных потоков. Первый поток подсчитывает сумму сегодняшней даты и времени (г+м+д+ч+мин+с) и записывает в таблицу. Второй поток проверяет, четное ли число записано в таблице, если да – устанавливает соответствующее поле таблицы в true, если нет – в false.
2. Задачу о производителе и потребителе с буфером на одну запись решить с помощью алгоритма Питерсона.

Вариант 46

1. С помощью мьютексов организовать работу параллельных вычислительных потоков. Первый поток записывает в файл случайные символы. Второй поток считывает из файла символ и удаляет его, если это цифра.
2. Задачу о производителе и потребителе с кольцевым буфером решить с помощью блокирующих переменных.

Вариант 47

1. С помощью мьютексов организовать работу параллельных вычислительных потоков. Первый поток записывает в файл user.txt введенные пользователем символы. Второй поток записывает в файл user.txt текущее время.
2. Задачу о производителе и потребителе с неограниченным буфером решить с помощью алгоритма Деккера.

Вариант 48

1. С помощью мьютексов организовать работу параллельных вычислительных потоков. Первый поток считает площадь треугольника S, координаты вершин которого (4; 3), (8; 12), (i; j), где i, j ∈ [9; 14], шаг 1, затем считает площадь поверхности пирамиды G с высотой равной (i+j). Второй поток считает координаты третьей вершины треугольника с координатами (–2; 6), (2; –4), (x; y) по координатам двух известных вершин и его площади S (определенной первым потоком), далее считает объем пирамиды с высотой h ∈ [4; 10], шаг 1.
2. Задачу о производителе и потребителе с кольцевым буфером решить с помощью алгоритма Деккера.

Вариант 49

1. С помощью мьютексов организовать работу параллельных вычислительных потоков. Первый поток считает значение функции y= 2x + 3, x ∈ [1; 4] шаг 1 и значение функции z =3x – 1, затем создает окно с количеством строк ввода равным (x+z). Второй поток создает окно с количеством кнопок равным y.
2. Задачу о производителе и потребителе с буфером на 1 запись решить с помощью алгоритма Деккера.

Вариант 50

1. С помощью мьютексов организовать работу параллельных вычислительных потоков. Первый поток создает строку L произвольной длины с произвольным количеством букв «м» и «н», остальное – случайные цифры. Второй поток подсчитывает количество цифр в строке L, значение которых меньше 6, и выводит различные сообщения в зависимости от того, четное это количество или нет.
2. Задачу о писателе и читателе решить с помощью блокирующих переменных (три читателя, приоритет писателя).

Вариант 51

1. С помощью мониторов Хоара организовать работу параллельных вычислительных потоков. Первый поток записывает в файл числа (до 100), не делящиеся на 3. Второй поток считывает из файла числа, проверяет, четные они или нет, если четные – удаляет их из файла.
2. Задачу об обедающих философах решить с помощью блокирующих переменных.

Вариант 52

1. С помощью мониторов Хоара организовать работу параллельных вычислительных потоков. Первый поток из заранее заготовленного файла считывает координаты точек и рисует треугольник, одна из вершин которого находится в считанной точке. Второй поток из этого же файла считывает координаты точек и рисует квадраты с левой нижней вершиной в считанной точке.
2. Задачу о курильщиках решить с помощью блокирующих переменных.

Вариант 53

1. С помощью мониторов Хоара организовать работу параллельных вычислительных потоков. Первый поток добавляет в таблицу случайные строки, состоящие из букв и цифр. Второй поток проверяет, четное ли количество цифр, если да, то заменяет цифры символом «*».
2. Задачу о курильщиках решить с помощью алгоритма Петерсона.

Вариант 54

1. С помощью мониторов Хоара организовать работу параллельных вычислительных потоков. Первый поток записывает в файл подсчитанные значения функции $Y= x^2/3 \ln x$ на интервале $x \in [2; 7]$ с шагом $h = 1$. Второй поток считывает из файла очередное значение и подсчитывает сумму всех считанных значений в переменной *Z*.
2. Задачу о производителе и потребителе с кольцевым буфером решить с помощью алгоритма Деккера.

Вариант 55

1. С помощью мониторов Хоара организовать работу параллельных вычислительных потоков. Первый поток подсчитывает сумму текущего времени и даты (д+мин+с) и записывает в таблицу. Второй поток проверяет, четное ли число записано в таблице, если да – устанавливает соответствующее поле таблицы в true, если нет – в false.
2. Задачу об обедающих философах решить с помощью блокирующих переменных.

Вариант 56

1. С помощью мониторов Хоара организовать работу параллельных вычислительных потоков. Первый поток записывает в файл случайные символы. Второй поток считывает из файла символ и удаляет его, если это не буква.
2. Задачу о курильщиках решить с помощью алгоритма Питерсона.

Вариант 57

1. С помощью мониторов Хоара организовать работу параллельных вычислительных потоков. Первый поток записывает в файл введенные пользователем символы, кроме первого и последнего. Второй поток записывает в файл текущее время и количество символов в файле.
2. Задачу о курильщиках решить с помощью алгоритма Деккера.

Вариант 58

1. С помощью мониторов Хоара организовать работу параллельных вычислительных потоков. Первый поток считает площадь треугольника S, координаты вершин которого (–1; –1), (2; 3), (i; j), где i, j $\in$ [4; 7], шаг 1, затем считает площадь поверхности пирамиды G с высотой равной (i+j). Второй поток считает координаты третьей вершины треугольника (3; -2),

(2; 7), (x; y) по координатам двух известных вершин и его площади S (из первого потока), и считает объем пирамиды с высотой h ∈ [3; 9], шаг 1.

2. Задачу о производителе и потребителе с буфером на одну запись решить с помощью алгоритма Деккера.

Вариант 59

1. С помощью мониторов Хоара организовать работу параллельных вычислительных потоков. Первый поток считает значение функции Y = 4x – 2, x ∈ [2; 8] шаг 1 и значение функции z = 2x + 1, затем создает окно с количеством строк ввода равным (x+z). Второй поток создает окно с количеством кнопок равным y.
2. Задачу об обедающих философах решить с помощью алгоритма Деккера.

Вариант 60

1. С помощью мониторов Хоара организовать работу параллельных вычислительных потоков. Первый поток создает строку L произвольной длины с произвольным количеством цифр и букв в ней. Второй поток подсчитывает количество букв и цифр в строке L и выводит различные сообщения в зависимости от того, больше букв или больше цифр.
2. Задачу о курильщиках решить с помощью алгоритма Деккера.

Заключение

Любое взаимодействие процессов или потоков связано с их *синхронизацией,* которая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступлении этого события. Синхронизация лежит в основе любого взаимодействия потоков, связано ли это взаимодействие с разделением ресурсов или с обменом данными.

Ежесекундно в системе происходят сотни событий, связанных с распределением и освобождением ресурсов, и ОС должна иметь надежные и производительные средства, которые бы позволяли ей синхронизировать потоки с происходящими в системе событиями.

Для синхронизации потоков прикладных программ программист может использовать как собственные средства и приемы синхронизации, так и средства операционной системы. Два потока одного прикладного процесса могут координировать свою работу с помощью доступной для них обоих глобальной логической переменной, которая устанавливается в единицу при осуществлении некоторого события, например выработки одним потоком данных, нужных для продолжения работы другого. Однако во многих случаях более эффективными или даже единственно возможными являются средства синхронизации, предоставляемые ОС в форме системных вызовов. Так, потоки, принадлежащие разным процессам, не имеют возможности вмешиваться каким-либо образом в работу друг друга. Без посредничества ОС они не могут приостановить друг друга или оповестить о произошедшем событии. Средства синхронизации используются операционной системой не только для синхронизации прикладных процессов, но и для ее внутренних нужд.

Обычно разработчики операционных систем предоставляют в распоряжение прикладных и системных программистов широкий спектр средств синхронизации. Эти средства могут образовывать иерархию, когда на основе более простых средств строятся более сложные, а также быть функционально

специализированными. Часто функциональные возможности разных системных вызовов синхронизации перекрываются, так что для решения одной задачи программист может воспользоваться несколькими вызовами в зависимости от своих личных предпочтений.

Для синхронизации процессов и потоков, решающих общие задачи и совместно использующих ресурсы, в операционных системах существуют специальные средства: критические секции, семафоры, мьютексы, события, таймеры. Отсутствие синхронизации может приводить к таким нежелательным последствиям, как гонки и тупики.

Литература

1. Богачев, К.Ю. Основы параллельного программирования [Текст] / К.Ю. Богачев.– М.: БИНОМ. Лаборатория знаний, 2003.

2. Бэкон, Д. Операционные системы. Параллельные и распределенные системы [Текст] / Д. Бэкон, Т. Харрис. – М.: Питер, 2004.

3. Воеводин, В.В. Параллельные вычисления [Текст] / В.В. Воеводин, Вл.В. Воеводин. – СПб.: БХВ-Петербург, 2002.

4. Гергель, В.П. Основы параллельных вычислений для многопроцессорных вычислительных систем [Текст] / В.П. Гергель, Р.Г. Стронгин. – Н. Новгород: ННГУ, 2001.

5. Гордеев, А.В. Операционные системы [Текст]: учеб. / А. В. Гордеев. – 2-е изд. – СПб.: Питер, 2004.

6. Гордеев, А.В. Системное программное обеспечение [Текст]: учеб. / А.В. Гордеев, А.Ю. Молчанов. – СПб.: Питер, 2001, 2002.

7. Корнеев, В.В. Параллельные вычислительные системы [Текст] / В.В. Корнеев. – М.: Нолидж, 1999.

8. Корнеев, В.В. Параллельное программирование в MPI [текст] / В.В. Корнеев. – М.; Ижевск: Институт компьютерных исследований, 2003.

9. Немнюгин, С. Параллельное программирование для многопроцессорных вычислительных систем [текст] / С. Немнюгин, О. Стесик. – СПб.: БХВ-Петербург, 2002.

10. Олифер, В.Г. Сетевые операционные системы: учеб. [Текст] / В.Г. Олифер, Н.А. Олифер. – СПб.; М.; Харьков; Минск: Питер, 2001, 2003.

11. Столлингс, В. Операционные системы [Текст] / В. Столлингс. – 4-е изд. – М.: Вильямс, 2004.

12. Таненбаум, Э. Современные операционные системы [Текст] / Э. Таненбаум. – 2-е изд. – СПб.: Питер, 2002.