Оглавление

[Взаимодействующие процессы 1](#_Toc164076302)

[**Категории средств обмена информацией** 2](#_Toc164076303)

[Логическая организация механизма передачи информации 3](#_Toc164076304)

[Нити исполнения 5](#_Toc164076305)

[Механизмы синхронизации 8](#_Toc164076306)

[Критическая секция 9](#_Toc164076307)

[**Алгоритмы синхронизации** 12](#_Toc164076308)

[**Interleaving, race condition и взаимоисключения** 12](#_Toc164076309)

[Программные алгоритмы организации взаимодействия процессов 15](#_Toc164076310)

[Требования, предъявляемые к алгоритмам 15](#_Toc164076311)

[Семафоры 23](#_Toc164076312)

[Мониторы 25](#_Toc164076313)

[Сообщения 29](#_Toc164076314)

[Эквивалентность семафоров, мониторов и сообщений 29](#_Toc164076315)

[Заключение 33](#_Toc164076316)

## Взаимодействующие процессы

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.

Для чего процессам нужно заниматься совместной деятельностью? Какие существуют причины для их *кооперации*?

* Повышение скорости работы. Пока один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разбивается на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.
* Совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.
* Модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда различные ее части представляют собой отдельные *процессы*, взаимодействующие путем передачи *сообщений* через микроядро.
* Наконец, это может быть необходимо просто для удобства работы пользователя, желающего, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Процессы не могут взаимодействовать, не общаясь, то есть не обмениваясь информацией. "Общение" процессов обычно приводит к изменению их поведения в зависимости от полученной информации. Если *деятельность* процессов остается неизменной при любой принятой ими информации, то это означает, что они на самом деле в "общении" не нуждаются. Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть **кооперативными** или *взаимодействующими процессами* , в отличие от *независимых процессов*, не оказывающих друг на друга никакого воздействия.

Различные процессы в вычислительной системе изначально представляют собой обособленные сущности. Работа одного процесса не должна приводить к нарушению работы другого процесса. Для этого, в частности, разделены их адресные пространства и системные ресурсы, и для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Нельзя просто поместить *значение*, вычисленное в одном процессе, в область памяти, соответствующую переменной в другом процессе, не предприняв каких-либо дополнительных усилий. Давайте рассмотрим основные аспекты организации совместной работы процессов.

**Категории средств обмена информацией**

Процессы могут взаимодействовать друг с другом, только обмениваясь информацией. *По* объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории.

* ***Сигнальные***. Передается минимальное количество информации – один бит, "да" или "нет". Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям. Вспомним профессора Плейшнера из кинофильма "Семнадцать мгновений весны". Сигнал тревоги – цветочный горшок на подоконнике – был ему передан, но профессор проигнорировал его. И к чему это привело?
* ***Канальные***. "Общение" процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.
* ***Разделяемая память***. Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием *разделяемой памяти* занимается операционная система (если, конечно, ее об этом попросят). "Общение" процессов напоминает совместное проживание студентов в одной комнате общежития. Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности (если вы переложили на другое место вещи вашего соседа по комнате, а часть из них еще и выбросили). Использование *разделяемой памяти* для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как *сигнальным* и *канальным средствам коммуникации* для этого необходимы специальные системные вызовы. *Разделяемая память* представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

## Логическая организация механизма передачи информации

При рассмотрении любого из средств коммуникации нас будет интересовать не их физическая реализация (общая *шина* данных, прерывания, аппаратно *разделяемая память* и т. д.), а логическая, определяющая в конечном счете механизм их использования. Некоторые важные аспекты логической реализации являются общими для всех категорий средств связи, некоторые относятся к отдельным категориям. Давайте кратко охарактеризуем основные вопросы, требующие разъяснения при изучении того или иного способа обмена информацией.

#### Как устанавливается связь?

Могу ли я использовать средство связи непосредственно для обмена информацией сразу после создания процесса или первоначально необходимо предпринять определенные действия для инициализации обмена? Например, для использования общей памяти различными процессами потребуется специальное обращение к операционной системе, которая выделит необходимую область адресного пространства. Но для передачи сигнала от одного процесса к другому никакая инициализация не нужна. В то же время передача информации по линиям связи может потребовать первоначального резервирования такой линии для процессов, желающих обменяться информацией.

К этому же вопросу тесно примыкает вопрос о способе адресации при использовании средства связи. Если я передаю некоторую информацию, я должен указать, куда я ее передаю. Если я желаю получить некоторую информацию, то мне нужно знать, откуда я могу ее получить.

Различают два способа адресации: *прямую* и *непрямую*. В случае *прямой адресации* *[взаимодействующие процессы](https://lms.kgeu.ru/mod/page/view.php?id=227402" \o "Взаимодействующие процессы)* непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется ***симметричной прямой адресацией*** . **Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные**. Если только один из *взаимодействующих процессов*, например передающий, указывает имя своего партнера по *кооперации*, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например ожидает получения информации от произвольного источника, то такая схема адресации называется ***асимметричной прямой адресацией*** .

При ***непрямой адресации*** данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, откуда они могут быть затем изъяты каким-либо другим процессом. Примером такого объекта может служить обычная доска объявлений или рекламная газета. При этом передающий процесс не знает, как именно идентифицируется процесс, который получит информацию, а принимающий процесс не имеет представления об идентификаторе процесса, от которого он должен ее получить.

При использовании *прямой адресации* связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных *инициализирующих действий*. Единственное, что нужно для использования средства связи, – это знать, как идентифицируются процессы, участвующие в обмене данными.

При использовании *непрямой адресации* инициализация средства связи может и не требоваться. Информация, которой должен обладать процесс для взаимодействия с другими процессами, – это некий идентификатор промежуточного объекта для хранения данных, если он, конечно, не является единственным и неповторимым в вычислительной системе для всех процессов.

#### Информационная валентность процессов и средств связи

Следующий важный вопрос – это вопрос об информационной валентности связи. Слово "валентность" здесь использовано по аналогии с химией. Сколько процессов может быть одновременно ассоциировано с конкретным средством связи? Сколько таких средств связи может быть задействовано между двумя процессами?

Понятно, что при *прямой адресации* только одно фиксированное средство связи может быть задействовано для обмена данными между двумя процессами, и только эти два процесса могут быть ассоциированы с ним. При *непрямой адресации* может существовать более двух процессов, использующих один и тот же объект для данных, и более одного объекта может быть использовано двумя процессами.

К этой же группе вопросов следует отнести и вопрос о направленности связи. Является ли связь однонаправленной или двунаправленной? Под однонаправленной связью мы будем понимать связь, при которой каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи. При двунаправленной связи каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных. В коммуникационных системах принято называть однонаправленную связь ***симплексной*** , двунаправленную связь с поочередной передачей информации в разных направлениях – ***полудуплексной*** , а двунаправленную связь с возможностью одновременной передачи информации в разных направлениях – ***дуплексной*** . *Прямая* и *непрямая адресация* не имеет непосредственного отношения к направленности связи.

## Нити исполнения

Рассмотренные выше аспекты логической реализации относятся к средствам связи, ориентированным на организацию взаимодействия различных процессов. Однако усилия, направленные на ускорение решения задач в рамках классических операционных систем, привели к появлению совершенно иных механизмов, к изменению самого понятия "процесс".

В свое время внедрение идеи мультипрограммирования позволило повысить пропускную способность компьютерных систем, т. е. уменьшить среднее *время ожидания* результатов работы процессов. Но любой отдельно взятый процесс в мультипрограммной системе никогда не может быть выполнен быстрее, чем при работе в однопрограммном режиме на том же вычислительном комплексе. Тем не менее, если *алгоритм* решения задачи обладает определенным *внутренним параллелизмом*, мы могли бы ускорить его работу, организовав взаимодействие нескольких процессов. Рассмотрим следующий пример. Пусть у нас есть следующая *программа* на псевдоязыке программирования:

Ввести массив a

Ввести массив b

Ввести массив c

a = a + b

c = a + c

Вывести массив c

При выполнении такой программы в рамках одного процесса этот процесс четырежды будет блокироваться, ожидая окончания операций ввода-вывода. Но наш *алгоритм* обладает *внутренним параллелизмом*. *Вычисление* суммы массивов a + b можно было бы выполнять параллельно с ожиданием окончания *операции* ввода массива c.

Ввести массив a

Ожидание окончания операции ввода

Ввести массив b

Ожидание окончания операции ввода

Ввести массив с

Ожидание окончания операции ввода a = a + b

c = a + c

Вывести массив с

Ожидание окончания операции вывода

Такое совмещение операций *по* времени можно было бы реализовать, используя два *взаимодействующих процесса*. Для простоты будем полагать, что средством коммуникации между ними служит *разделяемая память*. Тогда наши процессы могут выглядеть следующим образом.

Процесс 1 Процесс 2

Ввести массив a Ожидание ввода

Ожидание окончания массивов a и b

 операции ввода

Ввести массив b

Ожидание окончания

 операции ввода

Ввести массив с

Ожидание окончания a = a + b

 операции ввода

c = a + c

Вывести массив с

Ожидание окончания

 операции вывода

Казалось бы, мы предложили конкретный способ ускорения решения задачи. Однако в действительности дело обстоит не так просто. Второй процесс должен быть создан, оба процесса должны сообщить операционной системе, что им необходима *память*, которую они могли бы разделить с другим процессом, и, наконец, нельзя забывать о переключении контекста. Поэтому реальное поведение процессов будет выглядеть примерно так.

Процесс 1 Процесс 2

Создать процесс 2

 Переключение контекста

 Выделение общей

 памяти

 Ожидание ввода

 a и b

 Переключение контекста

Выделение общей памяти

Ввести массив a

Ожидание окончания

 операции ввода

Ввести массив b

Ожидание окончания

 операции ввода

Ввести массив с

Ожидание окончания

 операции ввода

 Переключение контекста

 a = a + b

 Переключение контекста

c = a + c

Вывести массив с

Ожидание окончания

 операции вывода

Очевидно, что мы можем не только не выиграть во времени при решении задачи, но даже и проиграть, так как временные потери на создание процесса, выделение общей памяти и переключение контекста могут превысить выигрыш, полученный за счет совмещения операций.

Для того чтобы реализовать нашу идею, введем новую абстракцию внутри понятия "процесс" – *нить исполнения* или просто *нить* (в англоязычной литературе используется термин *thread* ). *Нити процесса* разделяют его программный код, *глобальные переменные* и системные ресурсы, но каждая *нить* имеет собственный программный *счетчик*, свое содержимое регистров и свой *стек*. Теперь процесс представляется как совокупность взаимодействующих *нитей* и выделенных ему ресурсов. Процесс, содержащий всего одну *нить исполнения*, идентичен процессу в том смысле, который мы употребляли ранее. Для таких процессов мы в дальнейшем будем использовать термин "традиционный процесс". Иногда *нити* называют облегченными процессами или мини-процессами, так как во многих отношениях они подобны традиционным процессам. *Нити*, как и процессы, могут порождать нити-потомки, правда, только внутри своего процесса, и переходить из одного состояния в другое. Состояния *нитей* аналогичны состояниям традиционных процессов. Из состояния **рождение** процесс приходит содержащим всего одну *нить исполнения*. Другие *нити процесса* будут являться потомками этой нити-прародительницы. Мы можем считать, что процесс находится в состоянии **готовность**, если хотя бы одна из его *нитей* находится в состоянии **готовность** и ни одна из *нитей* не находится в состоянии **исполнение**. Мы можем считать, что процесс находится в состоянии **исполнение**, если одна из его *нитей* находится в состоянии **исполнение**. Процесс будет находиться в состоянии **ожидание**, если все его *нити* находятся в состоянии **ожидание**. Наконец, процесс находится в состоянии **закончил исполнение**, если все его *нити* находятся в состоянии **закончила исполнение**. Пока одна *нить процесса* заблокирована, другая *нить* того же процесса может выполняться. *Нити* разделяют *процессор* так же, как это делали традиционные процессы

## Механизмы синхронизации

 Допустим, что в вычислительной системе находятся два взаимодействующих процесса: один из них – H – с высоким приоритетом, другой – L – с низким приоритетом. Пусть *планировщик* устроен так, что процесс с высоким приоритетом вытесняет низкоприоритетный процесс всякий раз, когда он готов к исполнению, и занимает *процессор* на все время своего *CPU burst* (если не появится процесс с еще большим приоритетом). Тогда в случае, если процесс L находится в своей критической секции, а процесс H, получив *процессор*, подошел ко входу в критическую область, мы получаем тупиковую ситуацию. Процесс H не может войти в критическую область, находясь в цикле, а процесс L не получает управления, чтобы покинуть критический участок.

Для того чтобы не допустить возникновения подобных проблем, были разработаны различные *механизмы синхронизации*

## Критическая секция

Важным понятием при изучении способов синхронизации процессов является понятие *критической секции* (*critical section*) программы. ***Критическая секция*** – это часть программы, исполнение которой может привести к возникновению *race condition* для определенного набора программ. Чтобы исключить эффект гонок *по* отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей *критической секции*, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию *взаимоисключения* для *критических секций* программ. Реализация *взаимоисключения* для *критических секций* программ с практической точки зрения означает, что *по* отношению к другим процессам, участвующим во взаимодействии, *критическая секция* начинает выполняться как *атомарная операция*. Давайте рассмотрим следующий пример, в котором псевдопараллельные [взаимодействующие процессы](https://lms.kgeu.ru/mod/page/view.php?id=227402) представлены действиями различных студентов (*таблица* 5.1):

Здесь *критический участок* для каждого процесса – от *операции* "Обнаруживает, что хлеба нет" до *операции* "Возвращается в комнату" включительно. В результате отсутствия *взаимоисключения* мы из ситуации "Нет хлеба" попадаем в ситуацию "Слишком много хлеба". Если бы этот *критический участок* выполнялся как *атомарная операция* – "Достает два батона хлеба", то проблема образования излишков была бы снята.

|  |
| --- |
| **Таблица 5.1.** |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  |  |
| 17-07 | Обнаруживает,что хлеба нет |  |  |
| 17-09 | Уходит в магазин |  |  |
| 17-11 |  | Приходит в комнату |  |
| 17-13 |  | Обнаруживает, что хлеба нет |  |
| 17-15 |  | Уходит в магазин |  |
| 17-17 |  |  | Приходит в комнату |
| 17-19 |  |  | Обнаруживает,что хлеба нет |
| 17-21 |  |  | Уходит в магазин |
| 17-23 | Приходит в магазин |  |  |
| 17-25 | Покупает 2 батона на всех |  |  |
| 17-27 | Уходит из магазина |  |  |
| 17-29 |  | Приходит в магазин |  |
| 17-31 |  | Покупает 2 батона на всех |  |
| 17-33 |  | Уходит из магазина |  |
| 17-35 |  |  | Приходит в магазин |
| 17-37 |  |  | Покупает 2 батона на всех |
| 17-39 |  |  | Уходит из магазина |
| 17-41 | Возвращается в комнату |  |  |
| 17-43 |  |  |  |
| 17-45 |  |  |  |
| 17-47 |  | Возвращается в комнату |  |
| 17-49 |  |  |  |
| 17-51 |  |  |  |
| 17-53 |  |  | Возвращается в комнату |

Сделать процесс добывания хлеба *атомарной операцией* можно было бы следующим образом: перед началом этого процесса закрыть дверь изнутри на засов и уходить добывать хлеб через окно, а *по* окончании процесса вернуться в комнату через окно и отодвинуть засов. Тогда пока один студент добывает хлеб, все остальные находятся в состоянии ожидания под дверью (*таблица* 5.2).

|  |
| --- |
| **Таблица 5.2.** |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  |  |
| 17-07 | Достает два батона хлеба |  |  |
| 17-43 |  | Приходит в комнату |  |
| 17-47 |  |  | Приходит в комнату |

Итак, для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем *критическом участке*, другие процессы не могли войти в свои *критические участки*. Мы видим, что *критический участок* должен сопровождаться прологом ( entry section ) – "закрыть дверь изнутри на засов" – и эпилогом ( exit section ) – "отодвинуть засов", которые не имеют отношения к *активности*одиночного процесса. Во *время выполнения* пролога процесс должен, в частности, получить разрешение на вход в *критический участок*, а во *время выполнения* эпилога – сообщить другим процессам, что он покинул *критическую секцию*.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

while (some condition) {

 entry section

 critical section

 exit section

 remainder section

}

Здесь под *remainder* section понимаются все *атомарные операции*, не входящие в *критическую секцию*.

Оставшаяся часть этой лекции посвящена различным способам программной организации пролога и эпилога *критического участка* в случае, когда очередность доступа к *критическому участку* не имеет значения.

**Алгоритмы синхронизации**

Предположим, что надежная *связь* процессов организована, и они умеют обмениваться информацией. Нужно ли нам предпринимать еще какие-либо действия для организации правильного решения задачи взаимодействующими процессами? Нужно ли изменять их внутреннее поведение? Разъяснению этих вопросов и посвящена данная лекция.

**Interleaving, race condition и взаимоисключения**

Давайте временно отвлечемся от операционных систем, процессов и нитей исполнения и поговорим о некоторых " *активностях* ". Под ***активностями*** мы будем понимать последовательное выполнение ряда действий, направленных на достижение определенной цели. *Активности* могут иметь *место* в программном и техническом обеспечении, в обычной деятельности людей и животных. Мы будем разбивать *активности* на некоторые неделимые, или *атомарные*, *операции*. Например, *активность* "приготовление бутерброда" можно разбить на следующие *атомарные операции*:

1. Отрезать ломтик хлеба.
2. Отрезать ломтик колбасы.
3. Намазать ломтик хлеба маслом.
4. Положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые *операции* могут иметь внутренние невидимые действия (взять батон хлеба в левую руку, взять нож в правую руку, произвести отрезание). Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.

Пусть имеется две *активности*

P: a b c

Q: d e f

где a, b, c, d, e, f – *атомарные операции*. При последовательном выполнении *активностей* мы получаем такую последовательность атомарных действий:

PQ: a b c d e f

Что произойдет при исполнении этих *активностей* псевдопараллельно, в режиме разделения времени? *Активности* могут расслоиться на неделимые *операции* с различным чередованием, то есть может произойти то, что на английском языке принято называть словом *interleaving*. Возможные варианты чередования:

а b c d e f

a b d c e f

a b d e c f

a b d e f c

a d b c e f

......

d e f a b c

*Атомарные операции* *активностей* могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри *активностей*. Так как псевдопараллельное выполнение двух *активностей* приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две *активности* P и Q, состоящие из двух *атомарных операций* каждая:

P: x=2 Q: x=3

 y=x-1 y=x+1

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для *активностей* общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). . Мы будем говорить, что набор *активностей* (например, программ) ***детерминирован***, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он ***недетерминирован*** . Выше приведен пример *недетерминированного набора программ*. Понятно, что *детерминированный набор* *активностей* можно безбоязненно выполнять в режиме разделения времени. Для *недетерминированного набора* такое *исполнение* нежелательно.

Можно ли до получения результатов определить, является ли *набор* *активностей* детерминированным или нет? Для этого существуют достаточные *условия Бернстайна*. Изложим их применительно к программам с *разделяемыми переменными*.

Введем наборы входных и выходных переменных программы. Для каждой *атомарной операции* наборы входных и выходных переменных – это наборы переменных, которые *атомарная операция* считывает и записывает. Набор входных переменных программы R(P) (R от слова *read*) - суть *объединение* наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова *write*) - суть *объединение* наборов выходных переменных для всех ее неделимых действий. Например, для программы

P: x=u+v

 y=x\*w

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что *переменная* x присутствует как в R(P), так и в W(P).

Теперь сформулируем *условия Бернстайна*.

Если для двух данных *активностей* P и Q:

* пересечение W(P) и W(Q) пусто,
* пересечение W(P) с R(Q) пусто,
* пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Случай двух *активностей* естественным образом обобщается на их большее количество.

*Условия Бернстайна* информативны, но слишком жестки. *По* сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы *детерминированный набор* образовывали *активности*, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований *атомарных операций*, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный *доступ* программ к некоторым данным.

Про *недетерминированный набор программ* (и *активностей* вообще) говорят, что он имеет *race condition* ( *состояние гонки,* состояние состязания). В приведенном выше примере процессы состязаются за *вычисление* значений переменных x и y.

Задачу *упорядоченного доступа* к *разделяемым данным* (устранение *race condition*) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется *взаимоисключением (mutual exclusion)* . Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними *взаимоисключениями* уже не обойтись, нужна *взаимосинхронизация* поведения программ.

## Программные алгоритмы организации взаимодействия процессов

### Требования, предъявляемые к алгоритмам

Организация *взаимоисключения* для *критических участков*, конечно, позволит избежать возникновения *race condition*, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих *критические участки*, если они могут проходить их в произвольном порядке.

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд *взаимоисключения*. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test ) являются *атомарными операциями*.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем *критическом участке*, то не существует никаких других процессов, которые исполняются в соответствующих *критических секциях*. Это условие получило название условия *взаимоисключения (mutual exclusion)*.
4. Процессы, которые находятся вне своих *критических участков* и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные *критические участки*. Если нет процессов в *критических секциях* и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в *remainder* section, должны принимать решение о том, какой процесс войдет в свою *критическую секцию*. Такое решение не должно приниматься бесконечно долго. Это условие получило название ***условия прогресса (progress)*** .
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой *критический участок*. От того момента, когда процесс запросил разрешение на вход в *критическую секцию*, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои *критические участки* лишь ограниченное число раз. Это условие получило название ***условия ограниченного ожидания (bound waiting)*** .

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа организации пролога и эпилога для *критической секции*.

#### Запрет прерываний

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

while (some condition) {

 запретить все прерывания

 critical section

 разрешить все прерывания

 remainder section

}

Поскольку выход процесса из состояния **исполнение** без его завершения осуществляется по прерыванию, внутри *критической секции*никто не может вмешаться в его работу. Однако такое решение может иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыслу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к *критическим секциям* внутри самой операционной системы, например при обновлении содержимого *PCB*.

#### Переменная-замок

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в *критическую секцию* только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из *критической секции* процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе " *[Критическая секция](https://lms.kgeu.ru/mod/page/view.php?id=227406" \o "Критическая секция)* ").

shared int lock = 0;

/\* shared означает, что \*/

/\* переменная является разделяемой \*/

while (some condition) {

 while(lock); lock = 1;

 critical section

 lock = 0;

 remainder section

}

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию *взаимоисключения*, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в *критический участок*. Мы получаем два процесса, одновременно выполняющих свои *критические секции*.

#### Строгое чередование

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для *критического участка*, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

shared int turn = 0;

while (some condition) {

 while(turn != i);

 critical section

 turn = 1-i;

 remainder section

}

Очевидно, что *взаимоисключение* гарантируется, процессы входят в *критическую секцию* строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет *условию прогресса*. Например, если значение turn равно 1, и процесс P0 готов войти в *критический участок*, он не может сделать этого, даже если процесс P1 находится в **remainder section**.

#### Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в *критический участок*

shared int ready[2] = {0, 0};

Когда i-й процесс готов войти в *критическую секцию*, он присваивает элементу массива ready[i] значение равное 1. После выхода из *критической секции* он, естественно, сбрасывает это значение в 0. Процесс не входит в *критическую секцию*, если другой процесс уже готов к входу в *критическую секцию* или находится в ней.

while (some condition) {

 ready[i] = 1;

 while(ready[1-i]);

 critical section

 ready[i] = 0;

 remainder section

}

Полученный алгоритм обеспечивает *взаимоисключение*, позволяет процессу, готовому к входу в *критический участок*, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает *условие прогресса*. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в *критическую секцию*. Возникает ситуация, которую принято называть тупиковой (*deadlock*). (Подробнее о тупиковых ситуациях рассказывается в лекции 7.)

#### Алгоритм Петерсона

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

shared int ready[2] = {0, 0};

shared int turn;

while (some condition) {

 ready[i] = 1;

 turn =1-i;

 while(ready[1-i] && turn == 1-i);

 critical section

 ready[i] = 0;

 remainder section

}

При исполнении пролога *критической секции* процесс Pi заявляет о своей готовности выполнить *критический участок* и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в *критическом участке* продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия *взаимоисключения* методом от противного. Пусть оба процесса одновременно оказались внутри своих *критических секций*. Заметим, что процесс Pi может войти в *критическую секцию*, только если ready[1-i] == 0 или turn == i. Заметим также, что если оба процесса выполняют свои *критические секции* одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в *критические секции* из состояния, когда они оба одновременно находились в процессе выполнения цикла while? Нет, так как в этом случае переменная turn должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных измениться не могут). Пусть процесс P0 первым вошел в *критический участок*, тогда процесс P1 должен был выполнить перед вхождением в цикл while по крайней мере один предваряющий оператор (turn = 0;). Однако после этого он не может выйти из цикла до окончания *критического участка* процесса P0, так как при входе в цикл ready[0] == 1 и turn == 0, и эти значения не могут измениться до тех пор, пока процесс P0 не покинет свой *критический участок*. Мы пришли к противоречию. Следовательно, имеет место *взаимоисключение*.

Докажем выполнение *условия прогресса*. Возьмем, без ограничения общности, процесс P0. Заметим, что он не может войти в свою *критическую секцию* только при совместном выполнении условий ready[1] == 1 и turn == 1. Если процесс P1 не готов к выполнению *критического участка*, то ready[1] == 0, и процесс P0 может осуществить вход. Если процесс P1 готов к выполнению *критического участка*, то ready[1] == 1 и переменная turn имеет значение 0 либо 1, позволяя процессу P0 либо процессу P1начать выполнение *критической секции*. Если процесс P1 завершил выполнение *критического участка*, то он сбросит свой флаг готовности ready[1] == 0, разрешая процессу P0 приступить к выполнению критической работы. Таким образом, *условие прогресса*выполняется.

Отсюда же вытекает выполнение *условия ограниченного ожидания*. Так как в процессе ожидания разрешения на вход процесс P0 не изменяет значения переменных, он сможет начать исполнение своего *критического участка* после не более чем одного прохода по *критической секции* процесса P1.

#### Алгоритм булочной (Bakery algorithm)

*Алгоритм Петерсона* дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название *алгоритм булочной*, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера *алгоритм булочной*не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива

shared enum {false, true} choosing[n];

shared int number[n];

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения

(a,b) < (c,d), если a < c

или если a == c и b < d

max(a0, a1, ...., an) – это число k такое, что

k >= ai для всех i = 0, ...,n

Структура процесса Pi для *алгоритма булочной* приведена ниже

while (some condition) {

 choosing[i] = true;

 number[i] = max(number[0], ...,

 number[n-1]) + 1;

 choosing[i] = false;

 for(j = 0; j < n; j++){

 while(choosing[j]);

 while(number[j] != 0 && (number[j],j) <

 (number[i],i));

 }

 critical section

 number[i] = 0;

 remainder section

}

Доказательство того, что этот алгоритм удовлетворяет условиям 1 – 5, выполните самостоятельно в качестве упражнения.

#### Аппаратная поддержка взаимоисключений

Наличие аппаратной поддержки *взаимоисключений* позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались к общепринятому hardware для решения задачи реализации *взаимоисключений*, когда говорили об использовании механизма запрета/разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как *атомарные операции*. Давайте обсудим, как концепции таких команд могут использоваться для реализации *взаимоисключений*.

#### Команда Test-and-Set (проверить и присвоить 1)

О выполнении команды *Test-and-Set* , осуществляющей проверку значения логической переменной с одновременной установкой ее значения в 1, можно думать как о выполнении функции

int Test\_and\_Set (int \*target){

 int tmp = \*target;

 \*target = 1;

 return tmp;

}

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-замка, так чтобы он обеспечивал *взаимоисключения*

shared int lock = 0;

while (some condition) {

 while(Test\_and\_Set(&lock));

 critical section

 lock = 0;

 remainder section

}

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет *условию ограниченного ожидания* для алгоритмов. Подумайте, как его следует изменить для соблюдения всех условий.

#### Команда Swap (обменять значения)

Выполнение команды *Swap* , обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией:

void Swap (int \*a, int \*b){

 int tmp = \*a;

 \*a = \*b;

 \*b = tmp;

}

Применяя атомарную команду *Swap* , мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную key, локальную для каждого процесса:

shared int lock = 0;

int key;

while (some condition) {

 key = 1;

 do Swap(&lock,&key);

 while (key);

 critical section

 lock = 0;

 remainder section

}

### Семафоры

Одним из первых *механизмов*, предложенных для синхронизации поведения процессов, стали *семафоры*, концепцию которых описал Дейкстра (Dijkstra) в 1965 году.

#### Концепция семафоров

*Семафор* представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Классическое определение этих операций выглядит следующим образом:

P(S): пока S == 0 процесс блокируется;

 S = S – 1;

V(S): S = S + 1;

Эта запись означает следующее: при выполнении операции P над *семафором* S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции Vнад *семафором* S к его значению просто прибавляется 1. В момент создания *семафор* может быть инициализирован любым неотрицательным значением.

Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях реализуются с помощью специальных системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

#### Решение проблемы producer-consumer с помощью семафоров

Одной из типовых задач, требующих организации взаимодействия процессов, является задача ***producer-consumer*** (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда. На этом уровне деятельность потребителя и производителя можно описать следующим образом.

Producer: while(1) {

 produce\_item;

 put\_item;

 }

Consumer: while(1) {

 get\_item;

 consume\_item;

 }

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации. Если буфер пуст, то потребитель должен дожидаться нового [сообщения](https://lms.kgeu.ru/mod/page/view.php?id=227411). Как можно реализовать эти условия с помощью *семафоров*? Возьмем три *семафора*: empty, full и *mutex*. *Семафор*full будем использовать для гарантии того, что потребитель будет ждать, пока в буфере появится информация. *Семафор* empty будем использовать для организации ожидания производителя при заполненном буфере, а *семафор* *mutex* – для организации взаимоисключения на критических участках, которыми являются действия put\_item и get\_item (операции "положить информацию" и "взять информацию" не могут пересекаться, так как в этом случае возникнет опасность искажения информации). Тогда решение задачи на C-подобном языке выглядит так:

Semaphore mutex = 1;

 Semaphore empty = N; /\* где N – емкость буфера\*/

 Semaphore full = 0;

Producer:

 while(1) {

 produce\_item;

 P(empty);

 P(mutex);

 put\_item;

 V(mutex);

 V(full);

 }

Consumer:

 while(1) {

 P(full);

 P(mutex);

 get\_item;

 V(mutex);

 V(empty);

 consume\_item;

 }

Легко убедиться, что это действительно корректное решение поставленной задачи. Попутно заметим, что *семафоры* использовались здесь для достижения двух целей: организации взаимоисключения на критическом участке и *взаимосинхронизации* скорости работы процессов.

## Мониторы

Хотя решение *задачи producer-consumer* с помощью *семафоров* выглядит достаточно изящно, *программирование* с их использованием требует повышенной осторожности и внимания, чем отчасти напоминает *программирование* на языке Ассемблера. Допустим, что в рассмотренном примере мы случайно поменяли местами *операции* P, сначала выполнив операцию для *семафора* *mutex*, а уже затем для *семафоров* full и empty. Допустим теперь, что потребитель, войдя в свой критический участок ( *mutex* сброшен), обнаруживает, что *буфер* пуст. Он блокируется и начинает ждать появления сообщений. Но производитель не может войти в критический участок для передачи информации, так как тот заблокирован потребителем. Получаем тупиковую ситуацию.

В сложных программах произвести *анализ* правильности использования *семафоров* с карандашом в руках становится очень непросто. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от *interleavingатомарных операций*, и ошибки могут быть трудновоспроизводимы. Для того чтобы облегчить работу программистов, в 1974 году Хором (Hoare) был предложен механизм еще более высокого уровня, чем [*семафоры*](https://lms.kgeu.ru/mod/page/view.php?id=227409), получивший название *мониторов*. Мы с вами рассмотрим конструкцию, несколько отличающуюся от оригинальной.

***Мониторы*** представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. *Монитор* обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов, принадлежащих *монитору*. В свою очередь, эти функции-методы могут использовать в работе только данные, находящиеся внутри *монитора*, и свои параметры. На абстрактном уровне можно описать структуру *монитора* следующим образом:

monitor monitor\_name {

 описание внутренних переменных ;

 void m1(...){...

 }

 void m2(...){...

 }

 ...

 void mn(...){...

 }

 {

 блок инициализации

 внутренних переменных;

 }

}

Здесь функции m1,..., mn представляют собой функции-методы *монитора*, а блок инициализации внутренних переменных содержит *операции*, которые выполняются один и только один раз: при создании *монитора* или при самом первом вызове какой-либо функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т. е. находиться в состоянии **готовность** или **исполнение**, внутри данного *монитора*. Поскольку *мониторы* представляют собой особые конструкции языка программирования, *компилятор* может отличить *вызов функции*, принадлежащей *монитору*, от вызовов других функций и обработать его специальным образом, добавив к нему *пролог* и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на *компилятор*, а не на программиста, работа программиста при использовании *мониторов*существенно упрощается, а *вероятность* возникновения ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно *семафорам* full и empty в предыдущем примере. Для этого в *мониторах* было введено понятие условных переменных (condition variables)1, над которыми можно совершать две *операции* wait и signal, отчасти похожие на *операции* P и V над *семафорами*.

Если *функция* *монитора* не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию wait над какой-либо *условной переменной*. При этом процесс, выполнивший операцию wait, блокируется, становится неактивным, и другой процесс получает возможность войти в *монитор*.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию signal над той же самой *условной переменной*. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались *операции* signal для этой переменной, то активным становится только один из них. Что можно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри *монитора*? Хор предложил, чтобы пробужденный процесс подавлял *исполнение* разбудившего процесса, пока он сам не покинет *монитор*. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает *монитор* немедленно после исполнения *операции* signal. Мы будем придерживаться подхода Хансена.

**Необходимо отметить,** что условные переменные, в отличие от семафоров Дейкстры, не умеют запоминать предысторию. Это означает, что операция signal всегда должна выполняться после операции wait. Если операция signal выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна. Следовательно, выполнение операции wait всегда будет приводить к блокированию процесса.

Давайте применим концепцию *мониторов* к решению задачи производитель-потребитель.

monitor ProducerConsumer {

 condition full, empty;

 int count;

 void put() {

 if(count == N) full.wait;

 put\_item;

 count += 1;

 if(count == 1) empty.signal;

 }

 void get() {

 if (count == 0) empty.wait;

 get\_item();

 count -= 1;

 if(count == N-1) full.signal;

 }

 {

 count = 0;

 }

 }

Producer:

 while(1) {

 produce\_item;

 ProducerConsumer.put();

 }

Consumer:

 while(1) {

 ProducerConsumer.get();

 consume\_item;

 }

Легко убедиться, что приведенный пример действительно решает поставленную задачу.

Реализация *мониторов* требует разработки специальных языков программирования и компиляторов для них. *Мониторы* встречаются в таких языках, как параллельный Евклид, параллельный *Паскаль*, *Java* и т. д. Эмуляция *мониторов* с помощью системных вызовов для обычных широко используемых языков программирования не так проста, как эмуляция *семафоров*. Поэтому можно пользоваться еще одним механизмом со скрытыми взаимоисключениями, механизмом, о котором мы уже упоминали, – передачей сообщений.

### Сообщения

Для *прямой* и *непрямой адресации* достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – send и receive. В случае *прямой* адресации мы будем обозначать их так:

|  |
| --- |
| send(P, message) – послать сообщение message процессу P ; |
| receive(Q, message) – получить сообщение message от процесса Q. |

В случае *непрямой адресации* мы будем обозначать их так:

|  |
| --- |
| send(A, message) – послать сообщение message в почтовый ящик A ; |
| receive(A, message) – получить сообщение message из почтового ящика A. |

Примитивы send и receive уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный *буфер*. Реализация решения задачи *producer-consumer* для таких примитивов становится неприлично тривиальной. Надо отметить, что, несмотря на простоту использования, *передача сообщений* в пределах одного компьютера происходит существенно медленнее, чем работа с *семафорами* и *мониторами*.

### Эквивалентность семафоров, мониторов и сообщений

Мы рассмотрели три высокоуровневых механизма, использующихся для организации взаимодействия процессов. Можно показать, что в рамках одной вычислительной системы, когда процессы имеют возможность использовать *разделяемую память*, все они эквивалентны. Это означает, что любые два из предложенных механизмов могут быть реализованы на базе третьего, оставшегося механизма.

#### Реализация мониторов и передачи сообщений с помощью семафоров

Рассмотрим сначала, как реализовать [*мониторы*](https://lms.kgeu.ru/mod/page/view.php?id=227410) с помощью *семафоров*. Для этого нам нужно уметь реализовывать взаимоисключения при входе в *монитор* и *условные переменные*. Возьмем *семафор* *mutex* с начальным значением 1 для реализации взаимоисключения при входе в *монитор* и по одному *семафору* ci для каждой *условной переменной*. Кроме того, для каждой *условной переменной*заведем счетчик fi для индикации наличия ожидающих процессов. Когда процесс входит в *монитор*, компилятор будет генерировать вызов функции monitor\_enter, которая выполняет операцию P над *семафором* *mutex* для данного *монитора*. При нормальном выходе из *монитора* (то есть при выходе без вызова операции signal для *условной переменной* ) компилятор будет генерировать вызов функции monitor\_exit, которая выполняет операцию V над этим *семафором*.

Для выполнения операции wait над *условной переменной* компилятор будет генерировать вызов функции wait, которая выполняет операцию V для *семафора* *mutex*, разрешая другим процессам входить в *монитор*, и выполняет операцию P над соответствующим *семафором* ci, блокируя вызвавший процесс. Для выполнения операции signal над *условной переменной* компилятор будет генерировать вызов функции signal\_exit, которая выполняет операцию V над ассоциированным *семафором* ci (если есть процессы, ожидающие соответствующего события), и выход из *монитора*, минуя функцию monitor\_exit.

Semaphore mutex = 1;

void monitor\_enter(){

 P(mutex);

}

void monitor\_exit(){

 V(mutex);

}

Semaphore ci = 0;

int fi = 0;

void wait(i){

 fi=fi + 1;

 V(mutex);

 P(ci);

 fi=fi - 1;

}

void signal\_exit(i){

 if (fi)V(ci);

 else V(mutex);

}

Заметим, что при выполнении функции signal\_exit, если кто-либо ожидал этого события, процесс покидает *монитор* без увеличения значения *семафора* *mutex*, не разрешая тем самым всем процессам, кроме разбуженного, войти в *монитор*. Это увеличение совершит разбуженный процесс, когда покинет *монитор* обычным способом или когда выполнит новую операцию wait над какой-либо *условной переменной*.

Рассмотрим теперь, как реализовать передачу сообщений, используя [*семафоры*](https://lms.kgeu.ru/mod/page/view.php?id=227409). Для простоты опишем реализацию только одной *очереди сообщений*. Выделим в разделяемой памяти достаточно большую область под хранение сообщений, там же будем записывать, сколько пустых и заполненных ячеек находится в буфере, хранить ссылки на списки процессов, ожидающих чтения и памяти. Взаимоисключение при работе с разделяемой памятью будем обеспечивать *семафором* *mutex*. Также заведем по одному *семафору*ci на взаимодействующий процесс, для того чтобы обеспечивать *блокирование процесса* при попытке чтения из пустого буфера или при попытке записи в переполненный буфер. Посмотрим, как такой механизм будет работать. Начнем с процесса, желающего получить сообщение.

Процесс-получатель с номером i прежде всего выполняет операцию P(*mutex*), получая в монопольное владение разделяемую память. После чего он проверяет, есть ли в буфере сообщения. Если нет, то он заносит себя в список процессов, ожидающих сообщения, выполняет V(*mutex*) и P(ci). Если сообщение в буфере есть, то он читает его, изменяет счетчики буфера и проверяет, есть ли процессы в списке процессов, жаждущих записи. Если таких процессов нет, то выполняется V(*mutex*), и процесс-получатель выходит из критической области. Если такой процесс есть (с номером j ), то он удаляется из этого списка, выполняется V для его *семафора* cj, и мы выходим из критического района. Проснувшийся процесс начинает выполняться в критическом районе, так как *mutex* у нас имеет значение 0 и никто более не может попасть в критический район. При выходе из критического района именно разбуженный процесс произведет вызов V(*mutex*).

Как строится работа процесса-отправителя с номером i? Процесс, посылающий сообщение, тоже ждет, пока он не сможет иметь монополию на использование разделяемой памяти, выполнив операцию P(*mutex*). Далее он проверяет, есть ли место в буфере, и если да, то помещает сообщение в буфер, изменяет счетчики и смотрит, есть ли процессы, ожидающие сообщения. Если нет, выполняет V(*mutex*) и выходит из критической области, если есть, "будит" один из них (с номером j ), вызывая V(cj), с одновременным удалением этого процесса из списка процессов, ожидающих сообщений, и выходит из критического региона без вызова V(*mutex*), предоставляя тем самым возможность разбуженному процессу прочитать сообщение. Если места в буфере нет, то процесс-отправитель заносит себя в очередь процессов, ожидающих возможности записи, и вызывает V(*mutex*) и P(ci).

#### Реализация семафоров и передачи сообщений с помощью мониторов

Нам достаточно показать, что с помощью *мониторов* можно реализовать [*семафоры*](https://lms.kgeu.ru/mod/page/view.php?id=227409), так как получать из *семафоров* сообщения мы уже умеем.

Самый простой способ такой реализации выглядит следующим образом. Заведем внутри *монитора* переменную-счетчик, связанный с эмулируемым *семафором* список блокируемых процессов и по одной *условной переменной* на каждый процесс. При выполнении операции P над *семафором* вызывающий процесс проверяет значение счетчика. Если оно больше нуля, уменьшает его на 1 и выходит из *монитора*. Если оно равно 0, процесс добавляет себя в очередь процессов, ожидающих события, и выполняет операцию wait над своей *условной переменной*. При выполнении операции V над *семафором* процесс увеличивает значение счетчика, проверяет, есть ли процессы, ожидающие этого события, и если есть, удаляет один из них из списка и выполняет операцию signal для *условной переменной*, соответствующей процессу.

#### Реализация семафоров и мониторов с помощью очередей сообщений

Покажем, наконец, как реализовать [*семафоры*](https://lms.kgeu.ru/mod/page/view.php?id=227409) с помощью *очередей сообщений*. Для этого воспользуемся более хитрой конструкцией, введя новый синхронизирующий процесс. Этот процесс имеет счетчик и очередь для процессов, ожидающих включения *семафора*. Для того чтобы выполнить операции P и V, процессы посылают синхронизирующему процессу сообщения, в которых указывают свои потребности, после чего ожидают получения подтверждения от синхронизирующего процесса.

После получения сообщения синхронизирующий процесс проверяет значение счетчика, чтобы выяснить, можно ли совершить требуемую операцию. Операция V всегда может быть выполнена, в то время как операция P может потребовать блокирования процесса. Если операция может быть совершена, то она выполняется, и синхронизирующий процесс посылает подтверждающее сообщение. Если процесс должен быть блокирован, то его идентификатор заносится в очередь блокированных процессов, и подтверждение не посылается. Позднее, когда какой-либо из других процессов выполнит операцию V, один из блокированных процессов удаляется из очереди ожидания и получает соответствующее подтверждение.

Поскольку мы показали ранее, как из *семафоров* построить [*мониторы*](https://lms.kgeu.ru/mod/page/view.php?id=227410), мы доказали эквивалентность *мониторов*, *семафоров* и сообщений.

### Заключение

Для организации синхронизации процессов могут применяться специальные *механизмы* высокого уровня, блокирующие процесс, ожидающий входа в критическую секцию или наступления своей очереди для использования совместного ресурса. К таким механизмам относятся, например, [*семафоры*](https://lms.kgeu.ru/mod/page/view.php?id=227409), [*мониторы*](https://lms.kgeu.ru/mod/page/view.php?id=227410) и сообщения. Все эти конструкции являются эквивалентными, т. е., используя любую из них, можно реализовать две оставшиеся.