Направления подготовки:
«Информатика и вычислительная техника» и «Информационные системы и технологии»

Профили образовательных программ:
«Системотехника и автоматизация проектирования в строительстве»
«Системотехника и информационные технологии управления в строительстве»

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Тема 7. Синхронизация и взаимодействие процессов и потоков

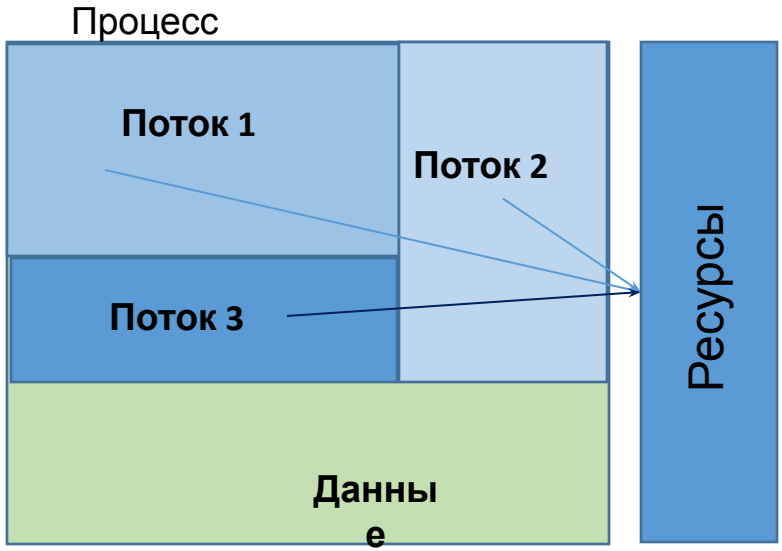
Кафедра «Информационных систем, технологий и автоматизации в строительстве» (ИСТАС)
Составитель: доцент, доц., к.т.н. Иванов Н.А.

Москва
2022

Содержание темы

№	Наименование раздела дисциплины	Тема и содержание лекций
2	Управление процессами и ресурсами	Тема 5. Ресурсы в вычислительных системах. Понятие «ресурс», классификация ресурсов вычислительной системы: разделяемые и закрепляемые, потребляемые и восстанавливаемые. Дескрипторы ресурсов. Динамическое и статическое распределение ресурсов.
		Тема 6. Управление процессами и потоками. Понятия «задача», «процесс», «поток». Состояние процесса. Структура контекста процесса. Идентификатор и дескриптор процесса. Иерархия процессов. Организация учета процессов. Параллельно и последовательно используемые программные модули. Системные и пользовательские процессы. Планирование и диспетчеризация в ОС.
		Тема 7. Синхронизация процессов. Межпроцессное взаимодействие. Состязания, критические области, взаимные блокировки процессов. Необходимость и средства синхронизации. Критические секции. Семафоры. Мьютексы. События.

Процесс и поток



Процесс рассматривается ОС как *заявка на потребление все видов ресурса, кроме одного – процессорного времени.*

Процесс – ***более крупная*** *единица работы* процессора, которая требует для своего выполнения нескольких более мелких работ – ***потоков***.

Поток (***поток выполнения***) - *единица работы процессора*, которой соответствует **фиксированный набор команд**, выполняемых процессором, и которой ***выделяется процессорное время***.

Ранние мультизадачные системы

Процесс состоит из одного потока
ПРОЦЕСС = ПОТОК
Мультипрограммирование на уровне **процессов**

Современные мультизадачные системы

-

Процесс состоит из нескольких потоков
ПРОЦЕСС = {поток1, …., потокК}
Мультипрограммирование на уровне **потоков**

Асимметричная модель. Потоки решают различные задачи и, как правило, не разделяют совместные ресурсы

Симметричная модель. Потоки выполняют одну и туже работу, разделяют общие ресурсы и исполняют одинаковый код

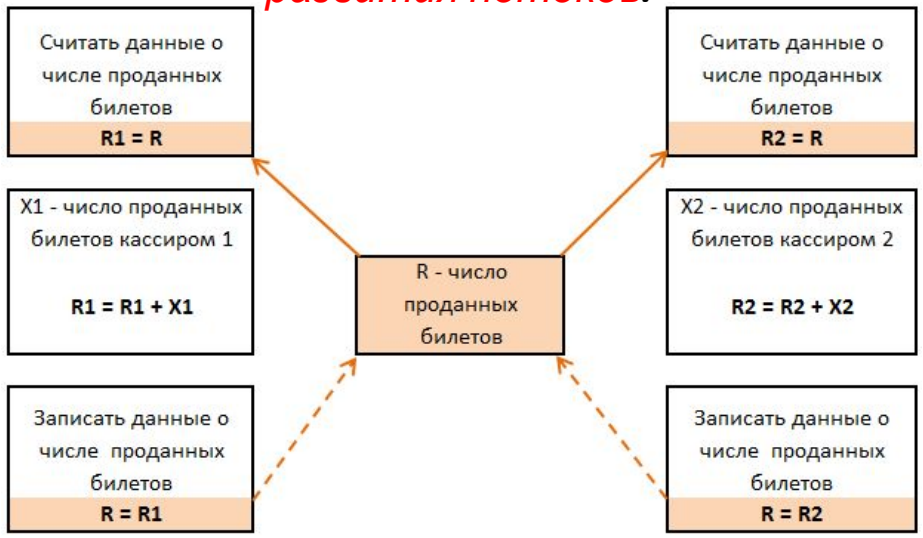
Жизненный цикл потока

Типичные проблемы многопоточных приложений

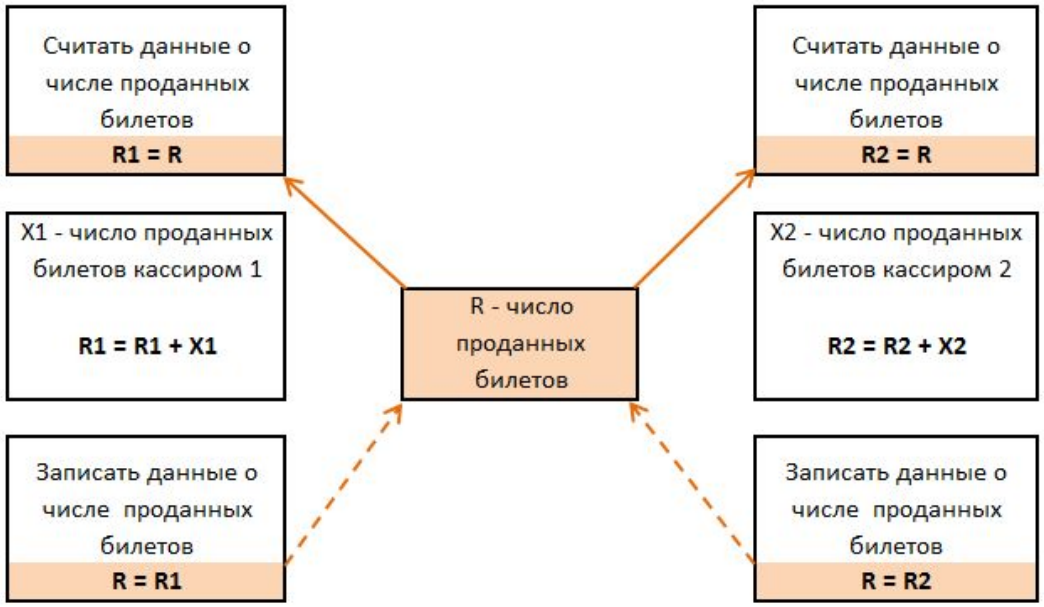
Простота создания потоков неразрывно связана со сложностью их применения.
Две типичные проблемы, с которыми программист может столкнуться при работе с потоками, – это **тупики** (**deadlocks**) и **гонки** (**race conditions**).

Тупик (*взаимная блокировка, дедлок, клинч*) – состояние многозадачной операционной системы, при котором она *не может нормально развиваться*, то есть выполнять некоторые *процессы*, так как они *заблокированы из-за недоступности необходимого ресурса*.

Ситуация **гонок** возникает, когда **два или более потока** пытаются получить доступ к общему ресурсу и изменить его состояние.
При этом конечный результат *зависит от соотношения скоростей развития потоков*.

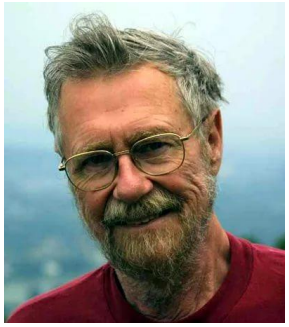


Проблема критической секции



«Критическая секция (**critical section**) – это **участок кода**, требующий *монопольного доступа к каким-то общим данным* (ресурсам), которые не должны быть одновременно использованы более чем одним потоком исполнения.
При нахождении в **критической секции** более одного процесса возникает состояние «***гонки***» »

Материал из Национальной библиотеки им. Н. Э. Баумана

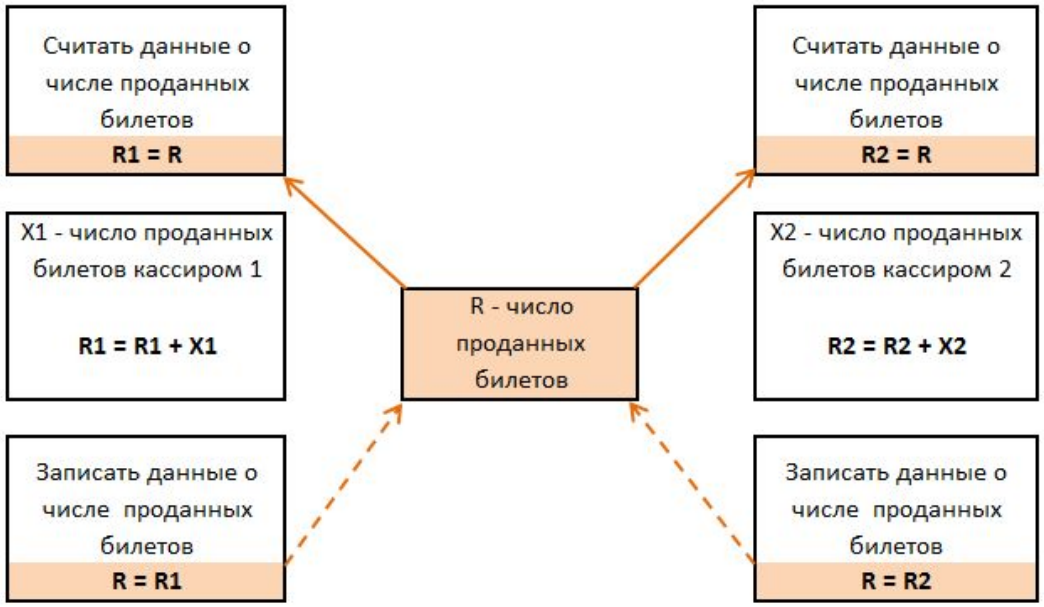


Дейкстра предположил, что ***проблема критической секции будет решена***, если будут выполнены 2 условия:

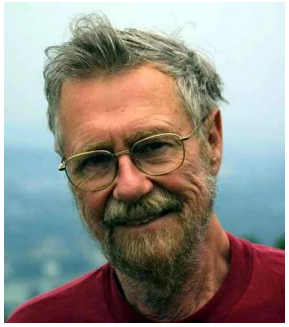
1. Необходимо обеспечить ***взаимное исключение*** (**в текущий момент времени в критической секции может находиться только 1 процесс**).

2. Необходимо устранить 2 вида блокировки:
- процессы **не должны бесконечно долго решать, кто из них войдет первым**.
- процессы, **остановившиеся вне своей критической секции, не должны препятствовать другим процессам входить в свои секции**.

Проблема критической секции. Взаимное исключение



«Критическая секция (**critical section**) – это **участок кода**, требующий *монопольного доступа к каким-то общим данным* (ресурсам), которые не должны быть одновременно использованы более чем одним потоком исполнения.
При нахождении в **критической секции** более одного процесса возникает состояние «***гонки***» »



Эдсгер Дейкстра в работе о межпроцессном взаимодействии.
E.W. Dijkstra, Cooperating Sequential Processes, 1965.

Алгоритм Деккера - *первое известное правильное* решение ***проблемы взаимного исключения*** в параллельном программировании.
Решение позволяет **двум** потокам совместно использовать общий ресурс без конфликтов, используя для связи только общую память.

Теодорус Йозеф (Дирк) Деккер

Алгоритм Деккера для реализации проблемы взаимного исключения

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0   // or 1
```

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

Источник: https://ru.qaz.wiki/wiki/Dekker%27s_algorithm

Если два процесса попытаются войти в критическую секцию одновременно, алгоритм разрешит войти только одному процессу, *в зависимости от того, чья очередь* это сделать.
Если один процесс уже находится в критической секции, другой процесс будет блокирован, ожидая освобождения критической секции.
Это делается с помощью двух флагов, *want_to_enter[0]* и *want_to_enter[1]* , которые указывают на намерение войти в критическую секцию со стороны процессов 0 и 1, соответственно, и переменной очереди *turn*, которая указывает, какой из двух процессов имеет право в текущий момент входить в свою критическую секцию.

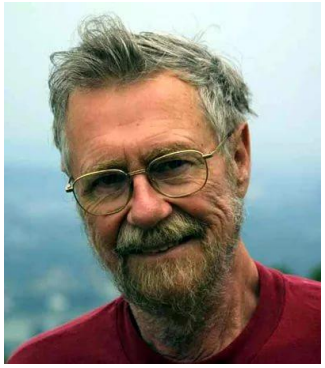
Недостаток:

Решение позволяет **только двум** потокам совместно использовать общий ресурс без конфликтов, используя для связи только общую память.

АЛГОРИТМ ДЕККЕРА В СИНХРОНИЗАЦИИ ПРОЦЕССОВ
http://espressocode.top/dekkers-algorithm-in-process-synchronization/

В поисках решения или как работает железная дорогам…

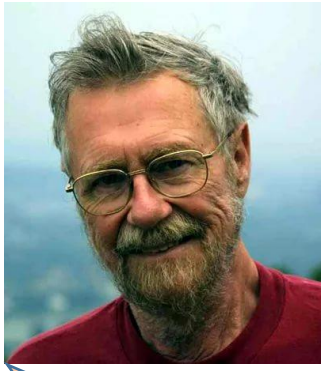
1. Необходимо обеспечить ***взаимное исключение*** (**в текущий момент времени в критической секции может находиться только 1 процесс**).
2. Необходимо устранить 2 вида блокировки:
   - процессы **не должны бесконечно долго решать, кто из них войдет первым**.
   - процессы, **остановившиеся вне своей критической секции, не должны препятствовать другим процессам входить в свои секции**.



Эдсгер Дейкстра , Cooperating Sequential Processes, 1965.

Решение найдено!!!

1. Необходимо обеспечить ***взаимное исключение*** (**в текущий момент времени в критической секции может находиться только 1 процесс**).
2. Необходимо устранить 2 вида блокировки:
   - процессы **не должны бесконечно долго решать, кто из них войдет первым**.
   - процессы, **остановившиеся вне своей критической секции, не должны препятствовать другим процессам входить в свои секции**.

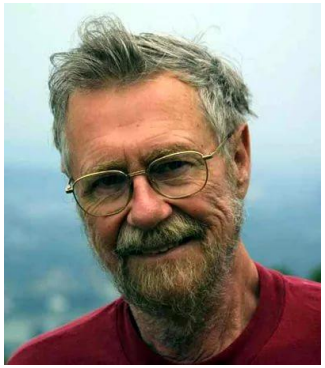


Семафоры!!

Семафоры!!

Эдсгер Дейкстра , Cooperating Sequential Processes, 1965.

Семафоры и семафорные примитивы



Семафор представляет собой **целочисленную неотрицательную переменную**, с которой ассоциирована очередь ожидающих процессов

Обычный семафор может принимать любые целочисленные неотрицательные значения

двоичный семафор – только значения 1 или 0.

Для обычных семафоров иногда задают предельно допустимое верхнее значение N.

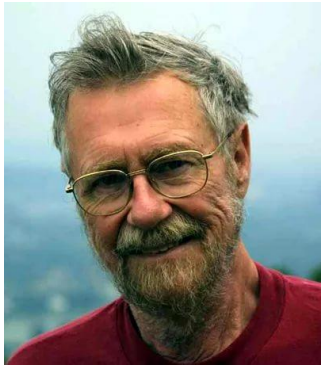
Операции над семафорами

«открытие», **V()** ***V**rijgeven* (освободить)	«закрытие», **P()** ***P**asseren** (пропустить)
Операция V увеличивает значение семафора на 1, если это возможно. В том случае, когда значение семафора соответствует предельному верхнему значению, операция V игнорируется.	Операция P заключается в проверке значения семафора и, если оно положительное, уменьшении значения на 1. Если значение семафора равно 0, то выполнение процесса, затребовавшего эту операцию, приостанавливается.

* Позже Дейкстра связал букву **P** со словом ***p**rolagen*, составленного из голландских слов *proberen* (попытаться) и *verlagen* (уменьшить),
а букву **V** – со словом ***v**erhogen* (увеличить).

Перед использованием семафора его необходимо **инициализировать**.
Как правило при инициализации переменной присваивается значение, показывающие, что **ни один поток не находится внутри критического участка.**

Формализованное описание операций P и V

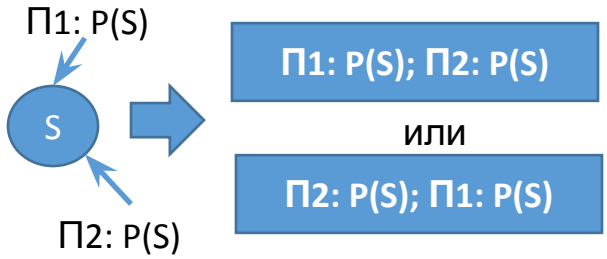


Операции над семафорами	
«открытие», **V()** ***V**rijgeven* (освободить)	«закрытие», **P()** ***P**asseren* (пропустить)
Операция V увеличивает значение семафора на 1, если это возможно. В том случае, когда значение семафора соответствует предельному верхнему значению, операция V игнорируется.	Операция P заключается в проверке значения семафора и, если оно положительное, уменьшении значения на 1. Если значение семафора равно 0, то выполнение процесса, затребовавшего эту операцию, приостанавливается.

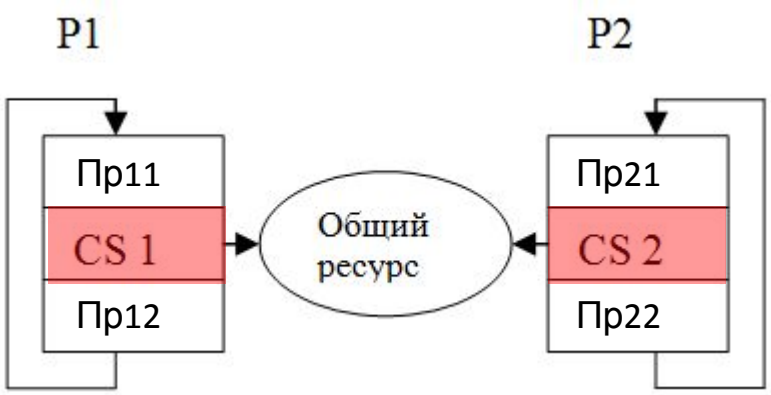
Пусть S – семафор, N – максимальное допустимое значение семафора.
Тогда P(S) и V(S) определяются следующим образом:
P(S) : [Если S > 0; S = S – 1; блокировка процесса]
V(S) : [Если S<N; S=S+1; операция игнорируется]

Существенно, что **проверка значения семафора, изменение его значения и блокировка процесса выполняются как одно неделимое действие**. Поэтому операции над семафорами часто называют ***семафорными примитивами***.

Еще одним важным свойством операций над семафорами является предположение о том, что *одновременное обращение* к одному семафору двух и более процессов рассматривается как **последовательное обращение в произвольном порядке**



Решение проблемы критической секции с помощью семафоров



Постановка задачи:

Пусть есть два процесса P1 и P2 каждый из которых является циклическим.
Процессы обладают равными приоритетами.
Скорости развития процессов не известны.
Процессы взаимодействуют через общую область памяти.
Необходимо так запрограммировать работу процессов, чтобы в текущий момент времени в критической секции находился только один из них.

Решение задачи:

S=1

P1:

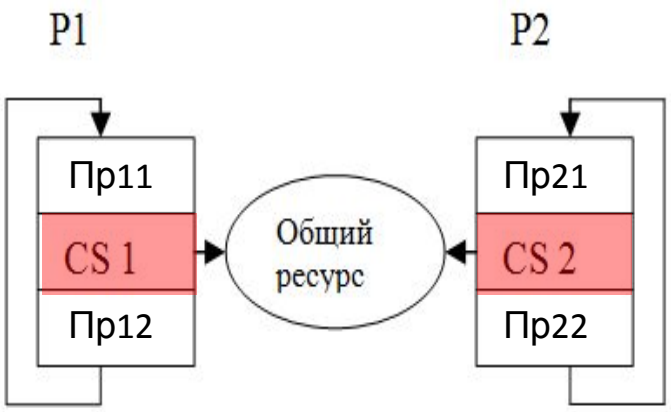
{L1: программа 11; P(S); CS1; V(S); программа 12; goto L1}

P2:

{L2: программа 21; P(S); CS2; V(S); программа 22; goto L2}

S – двоичный семафор; CS1 и CS2 – критические секции процессов P1 и P2 соответственно;
P(S) и V(S) – операции над семафором.

Решение проблемы критической секции с помощью семафоров



Решение задачи:

S=1

P1:
{L1: программа 11; P(S); CS1; V(S); программа 12; goto L1}

P2:
{L2: программа 21; P(S); CS2; V(S); программа 22; goto L2}

ВСПОМНИМ:

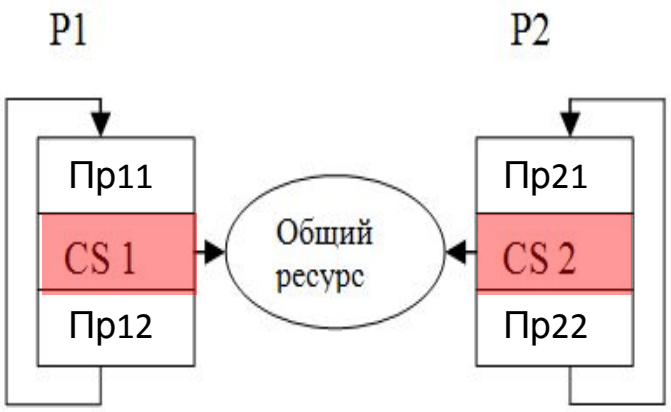
Дейкстра предположил, что ***проблема критической секции будет решена***, если будут выполнены 2 условия:

1. Необходимо обеспечить ***взаимное исключение*** (**в текущий момент времени в критической секции может находиться только 1 процесс**).

2. Необходимо устранить 2 вида блокировки:

- процессы **не должны бесконечно долго решать, кто из них войдет первым**.
- процессы, **остановившиеся вне своей критической секции, не должны препятствовать другим процессам входить в свои секции**.

Решение проблемы критической секции с помощью семафоров



Решение задачи:

S=1

P1:
{L1: программа 11; P(S); CS1; V(S); программа 12; goto L1}

P2:
{L2: программа 21; P(S); CS2; V(S); программа 22; goto L2}

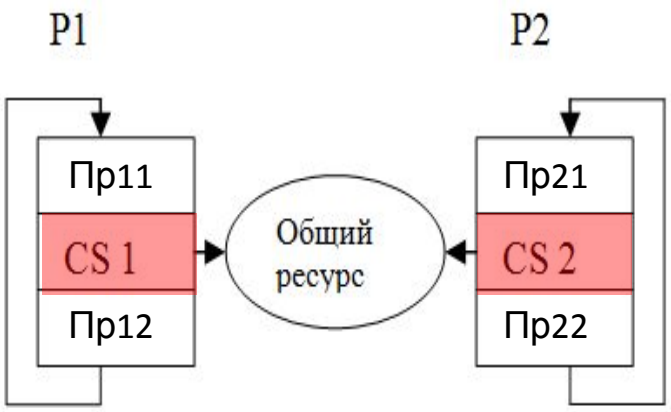
ПРОВЕРИМ:

1. Необходимо обеспечить ***взаимное исключение*** (**в текущий момент времени в критической секции может находиться только 1 процесс**).

Предположим, что процесс **P1**, выполнив часть своего кода «*программа 11*», достиг примитива P(S).
Так как начальное значение семафора ***S*** равно 1, то процесс **P1** благополучно войдет в свою критическую секцию ***CS1***, параллельно уменьшив значение семафора ***S*** до 0.

Если теперь процесс **P2** попытается выполнить примитив P(S), желая войти в критическую секцию ***CS2***, то он будет **заблокирован** на этом примитиве из-за *нулевого* значения семафора ***S***.

Решение проблемы критической секции с помощью семафоров



Решение задачи:

S=1

P1:
{L1: программа 11; P(S); CS1; V(S); программа 12; goto L1}

P2:
{L2: программа 21; P(S); CS2; V(S); программа 22; goto L2}

ПРОВЕРИМ:

2. Необходимо устранить 2 вида блокировки:
- процессы **не должны бесконечно долго решать, кто из них войдет первым**.

Предположим, что при выполнении программы два процесса одновременно подошли к примитивам P(S). В этом случае необходимо вспомнить, что «*одновременное обращение двух и более процессов к семафору рассматривается как последовательное обращение в случайном порядке*».

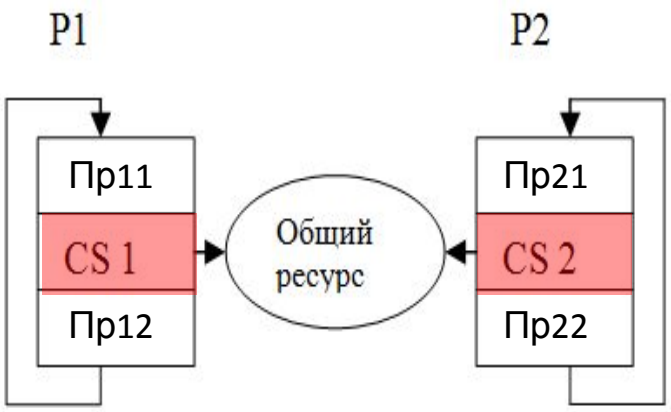
Таким образом, получаем два варианта развития событий.

В первом случае сначала примитив P(S) выполнит процесс **P1**, а затем последует попытка выполнить примитив P(S) процессом **P2**, в результате которой процесс **P2** будет заблокирован.

Во втором случае процессы поменяются местами.

Важно, что в обоих вариантах доступ к своим критическим секциям процессы будут иметь не одновременно!

Решение проблемы критической секции с помощью семафоров



Решение задачи:

S=1

P1:
{L1: программа 11; P(S); CS1; V(S); программа 12; goto L1}

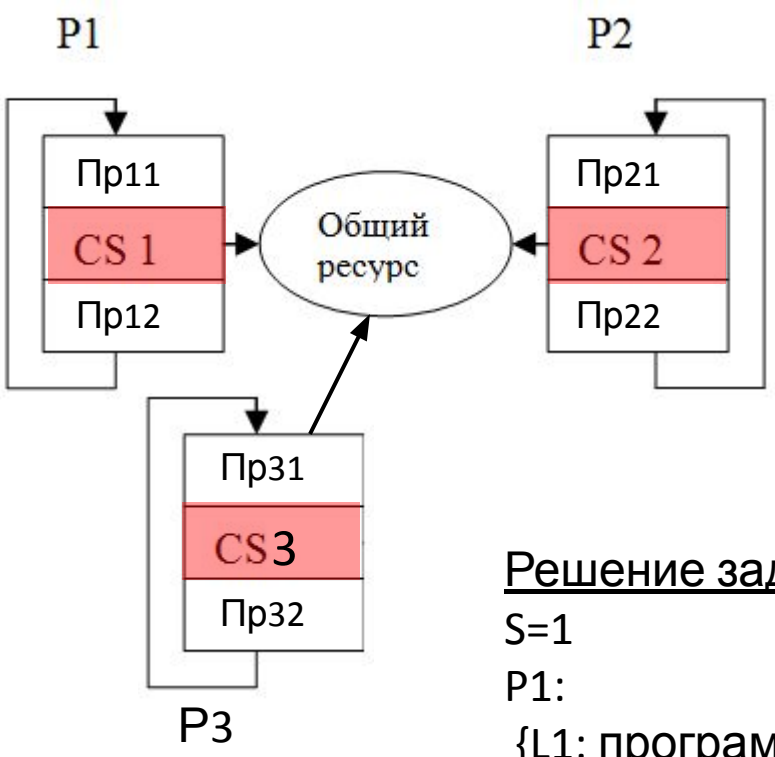
P2:
{L2: программа 21; P(S); CS2; V(S); программа 22; goto L2}

ПРОВЕРИМ:

2. Необходимо устранить 2 вида блокировки:
- процессы, **остановившиеся вне своей критической секции, не должны препятствовать другим процессам входить в свои секции.**

Предположим, что процесс **P1** либо очень долго выполняет часть своего кода «*программа 12*» либо по какой-то причине просто остановился в этой части кода.
В этом случае никаких ограничений на развитие процесса **P2** накладываться не будет, так как процесс **P1** не будет влиять на значение семафора ***S***.
Следовательно, рассматриваемая **блокировка устранена**.

Решение проблемы критической секции с помощью семафоров. Ограничение на число процессов.



Постановка задачи:

Пусть есть два процесса P1 и P2 каждый из которых является циклическим.
Процессы обладают равными приоритетами.
Скорости развития процессов не известны.
Процессы взаимодействуют через общую область памяти.
Необходимо так запрограммировать работу процессов, чтобы в текущий момент времени в критической секции находился только один из них.

Решение задачи:

S=1

P1:

{L1: программа 11; P(S); CS1; V(S); программа 12; goto L1}

P2:

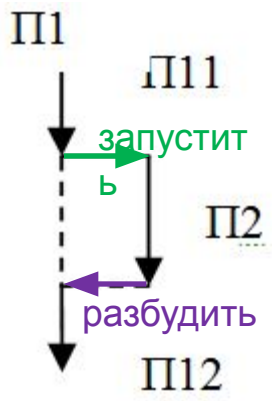
{L2: программа 21; P(S); CS2; V(S); программа 22; goto L2}

P3:

{L3: программа 31; P(S); CS3; V(S); программа 32; goto L3}

S – двоичный семафор; CS1 и CS2 – критические секции процессов P1 и P2 соответственно;
P(S) и V(S) – операции над семафором.

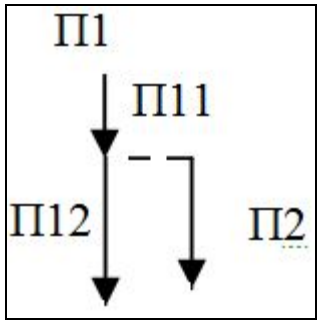
Использование семафоров для синхронизации независимых процессов



Постановка задачи:
Пусть есть два процесса П1 и П2.
Процесс П1 выполняет часть своего кода и затем запускает на выполнение процесс П2. На время выполнения процесса П2 процесс П1 «засыпает». Выполнение процесса П1 возобновляется только после полного завершения процесса П2.

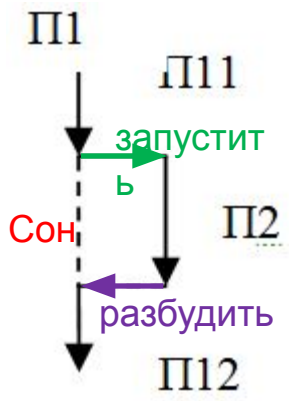
Решение задачи:
Шаг 1.

П1:
{программа 11; Запустить П2; программа 12;}
П2:
{программа 2;}



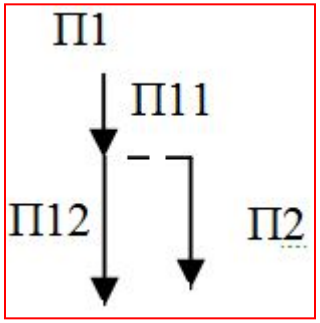
Условие синхронизации не выполнено!!!
Получили параллельное выполнение двух процессов!!!

Использование семафоров для синхронизации независимых процессов



Постановка задачи:
Пусть есть два процесса П1 и П2.
Процесс П1 выполняет часть своего кода и затем запускает на выполнение процесс П2. На время выполнения процесса П2 процесс П1 «засыпает». Выполнение процесса П1 возобновляется только после полного завершения процесса П2.

Решение задачи:
Rem S - двоичный семафор;
Rem P(S) и V(S) – операции над семафором;
S = 0;
П1:
{программа 11; Запустить П2; P(S); программа 12;}
П2:
{программа 2;}
V(S);}



Класс задач «поставщик-потребитель»

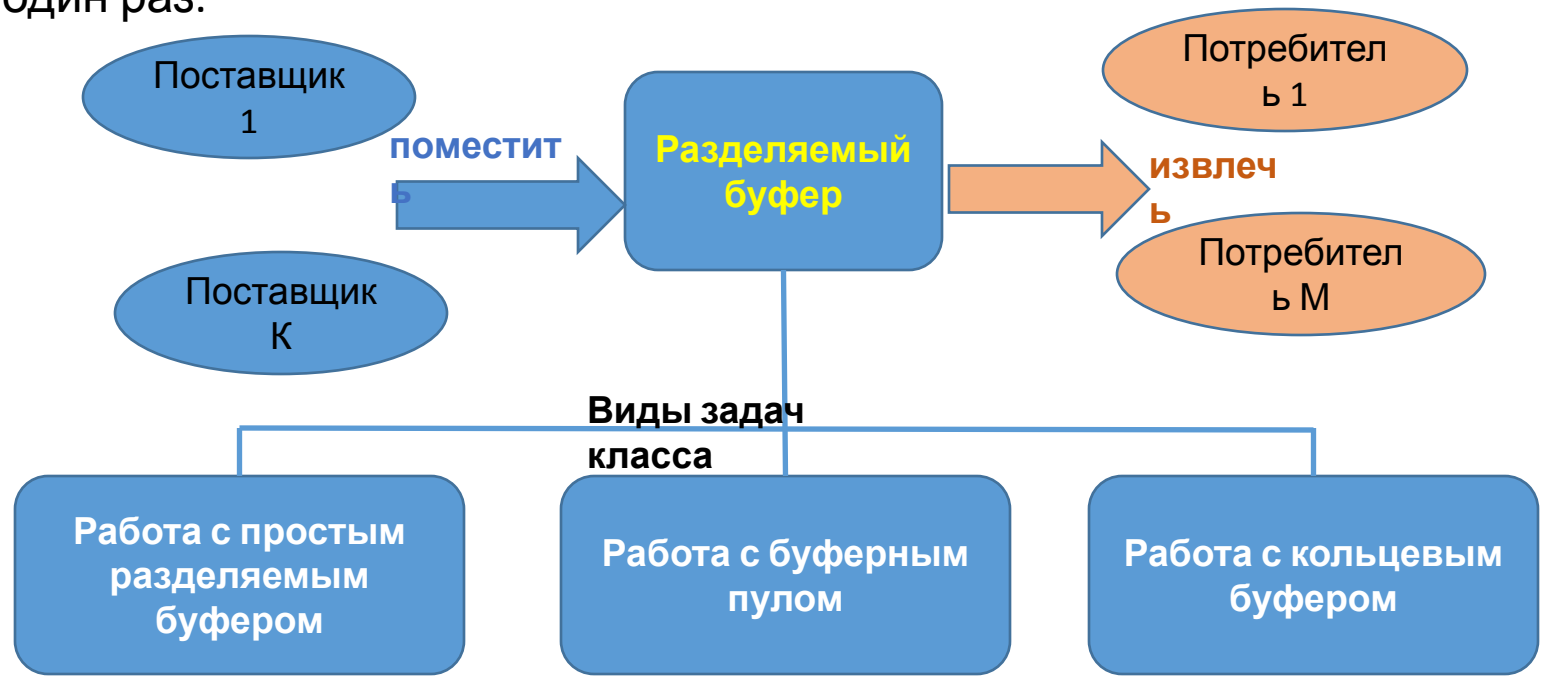
Общее описание класса задач:

В задаче о поставщиках и потребителях ***поставщики*** посылают сообщения, получаемые ***потребителями***.

Процессы общаются через ***разделяемый буфер***, выполняя либо операцию «поместить» либо операцию «извлечь».

Операцию «поместить» выполняет процесс-поставщик, операцию «извлечь» - процесс потребитель.

Необходимо запрограммировать работу процессов так, чтобы сообщения не перезаписывались без прочтения, и каждое из них могло бы быть прочитано только один раз.



Работа с простым разделяемым буфером

Постановка задачи:

Пусть есть два процесса, которые циклически работают с *простым разделяемым буфером**. Первый процесс - «поставщик», - создает информацию и помещает ее в буфер. Второй процесс - «потребитель», - считывает информацию из буфера и выводит ее на печать.

В любой момент времени буфер может быть доступен **только** либо по чтению, либо по записи.

Необходимо запрограммировать работу процессов так, чтобы сообщения не перезаписывались без прочтения, и каждое из них могло бы быть прочитано только один раз.

МОЖЕТ БЫТЬ ЭТО ЗАДАЧА – ЗАДАЧА НА ВЗАИМНОЕ ИСКЛЮЧЕНИЕ?!

S=1;

Поставщик:
{L1: СоздатьДанные;
P(S); **ПоместитьВБуфер**; V(S);
goto L1}

Потребитель:
{L2: P(S); **ИзвлечьИзБуфера**; V(S);
ОбработатьДанные;
goto L2}

---

*простой разделяемый буфер – одна ячейка для приёма-передачи

Работа с простым разделяемым буфером

Постановка задачи:

Пусть есть два процесса, которые циклически работают с *простым разделяемым буфером**. Первый процесс - «поставщик», - создает информацию и помещает ее в буфер. Второй процесс - «потребитель», - считывает информацию из буфера и выводит ее на печать.

В любой момент времени буфер может быть доступен **только** либо по чтению, либо по записи.

Необходимо запрограммировать работу процессов так, чтобы сообщения не перезаписывались без прочтения, и каждое из них могло бы быть прочитано только один раз.

Попробуем *связать семафор с состояниями разделяемого буфера*, которые состоят в том, что он заполняется или опустошается.

Введем **два семафора** «*пустой*» и «*полный*», соответствующие двум указанным выше состояниям буфера.

В начальном состоянии разделяемый буфер пуст, поэтому семафор «пустой» имеет значение 1, а семафор «полный» - 0.

**простой разделяемый буфер – одна ячейка для приёма-передачи*

Работа с простым разделяемым буфером

Постановка задачи:

Пусть есть два процесса, которые циклически работают с *простым разделяемым буфером**. Первый процесс - «поставщик», - создает информацию и помещает ее в буфер. Второй процесс - «потребитель», - считывает информацию из буфера и выводит ее на печать.

В любой момент времени буфер может быть доступен **только** либо по чтению, либо по записи.

Необходимо запрограммировать работу процессов так, чтобы сообщения не перезаписывались без прочтения, и каждое из них могло бы быть прочитано только один раз.

Пустой=1, Полный=0

```
Process Поставщик {
    While (true) {
        Произвести данные
        P(Пустой);
        Поместить данные в буфер
        V(Полный);
    }
}
```

```
Process Потребитель {
    While (true) {
        P(Полный);
        Извлечь данные из буфера
        V(Пустой);
        Обработка данных
    }
}
```

Семафоры «пустой» и «полный» вместе образуют так называемый ***разделенный двоичный семафор***, поскольку в любой момент времени только один из них может иметь значение 1.

ВАЖНО! Рассмотренное выше решение справедливо и при наличии *нескольких* процессов-поставщиков и процессов-потребителей.

Работа с буферным пулом

На практике для повышения производительности системы процессы-потребители и процессы-поставщики взаимодействуют не через единственную ячейку-буфер, а через ***несколько буферных ячеек***, образующих так называемый *буферный пул*.

Наличие буферного пула позволяет каждому из процессов произвести свою операцию с буфером при его частичном заполнении.
Действительно, при наличии в буферном пуле N буферных ячеек процесс-поставщик может выполнить операцию «Поместить данные в буфер» N раз даже тогда, когда процесс-потребитель совсем не освобождает буфер.
Аналогично, процесс-потребитель может выполнять операцию «Извлечь данные из буфера» до тех пор, пока есть хотя бы одна заполненная буферная ячейка, даже в том случае, когда процесс-поставщик перестал заносить данные в буфер.

Работа с буферным пулом

Постановка задачи:

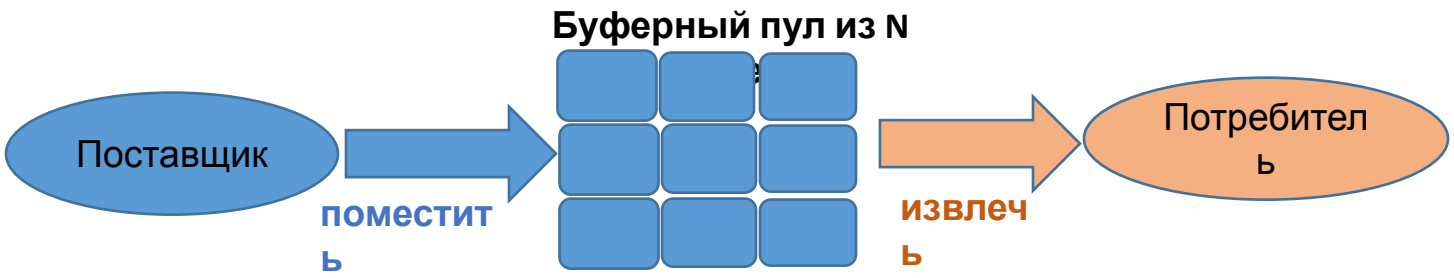
Пусть есть два процесса, которые работают с *буферным пулом*, содержащим N буферных ячеек.

Первый процесс - «поставщик», - создает информацию и помещает ее в одну из свободных ячеек буфера.

Второй процесс - «потребитель», - считывает информацию из буфера и выводит ее на печать.

В любой момент времени буферный пул может быть доступен либо только по чтению, либо только по записи.

Необходимо так запрограммировать систему, чтобы в любой момент времени было известно число свободных и занятых ячеек буфера и в системе было бы не доступно чтение из несуществующей буферной ячейки или запись в несуществующую ячейку буфера.



Работа с буферным пулом

Процесс «поставщик» может быть представлен в виде двух частей: генерирование информации (П11), запись в буфер (П12).

Процесс «потребитель» так же можно разделить на две части: чтение информации из буфера (П21) и вывод информации на печать (П22).

Ниже приведено простейшее описание взаимодействия процессов П1 и П2:
П1: {L1: П11; П12; goto L1}
П2: {L2: П21; П22; goto L2}

Части П11 и П22 процессов никак не влияют на работу другого процесса, а вот части П12 и П21 обращаются к общему ресурсу – буферному пулу.
Организация доступа к буферному пулу для двух процессов – классическая задача о «критической секции». Применив рассмотренное ранее решение этой задачи, получим:

П1: {L1: П11; P(S); П12; V(S); goto L1}
П2: {L2: P(S); П21; V(S); П22; goto L2}

Таким образом, задача доступа к буферному пулу только одного из процессов решена.
Но ответить на вопрос о числе свободных и занятых ячеек буфера системе не представляется возможным.

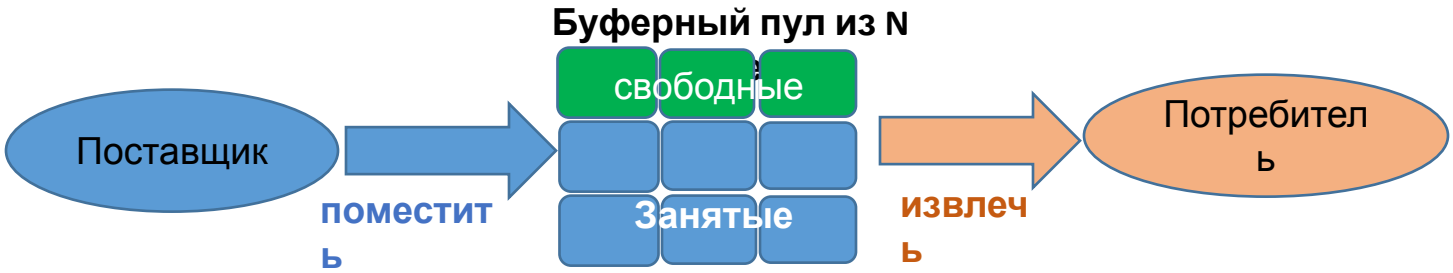
Работа с буферным пулом

Процесс «поставщик» может быть представлен в виде двух частей: генерирование информации (П11), запись в буфер (П12).

Процесс «потребитель» так же можно разделить на две части: чтение информации из буфера (П21) и вывод информации на печать (П22).

Введем два дополнительных семафора ***e*** и ***f***.
Семафор ***e*** будет фиксировать *число свободных буферов*, его значение будет изменяться от N до 0.
Семафор ***f*** будет отражать *число занятых буферов*, его значение будет изменяться от 0 до N. Процесс П1 при помещении очередной порции данных в буферный пул будет *уменьшать* на 1 число свободных буферов и *увеличивать* на 1 число занятых буферов. Процесс П2, извлекая данные из буферного пула, будет, наоборот, *уменьшать* на 1 число занятых буферов и *увеличивать* на 1 число свободных буферов.
При этом важно остановить
процесс П1 при заполнении всех буферов пула,
а процесс П2 – после извлечения данных из последнего занятого буфера пула.

Работа с буферным пулом



Процесс «поставщик» может быть представлен в виде двух частей: генерирование информации (П11), запись в буфер (П12).

Процесс «потребитель» так же можно разделить на две части: чтение информации из буфера (П21) и вывод информации на печать (П22).

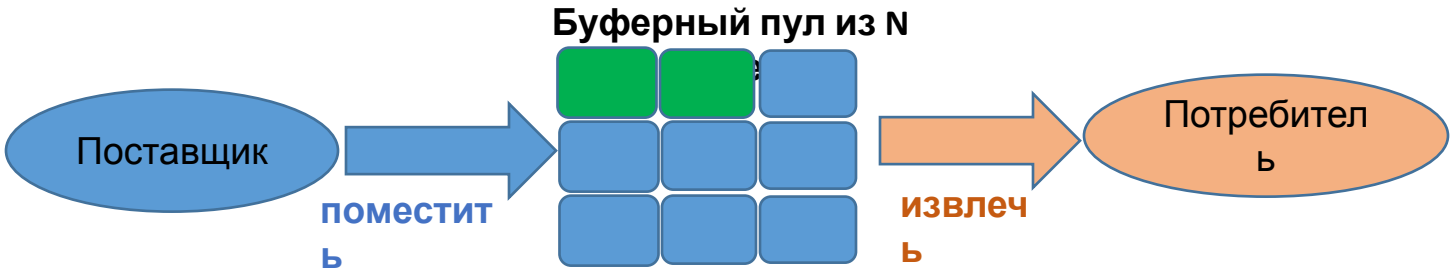
Учитывая вариант решения для задачи «критической секции»
S=1;

П1: {L1: П11; P(S); П12; V(S); goto L1}
П2: {L2: P(S); П21; V(S); П22; goto L2}

получаем:
S= 1; e = N; f =0;

П1: {L1: П11; P(S); P(e); П12; V(f); V(S); goto L1}
П2: {L2: P(S); P(f); П21; V(e); V(S); П22; goto L2}

Работа с буферным пулом



Процесс «поставщик» может быть представлен в виде двух частей: генерирование информации (П11), запись в буфер (П12).

Процесс «потребитель» так же можно разделить на две части: чтение информации из буфера (П21) и вывод информации на печать (П22).

S= 1; e = N; f =0;

П1: {L1: П11; P(S); P(e); П12; V(f); V(S); goto L1}

П2: {L2: P(S); P(f); П21; V(e); V(S); П22; goto L2}

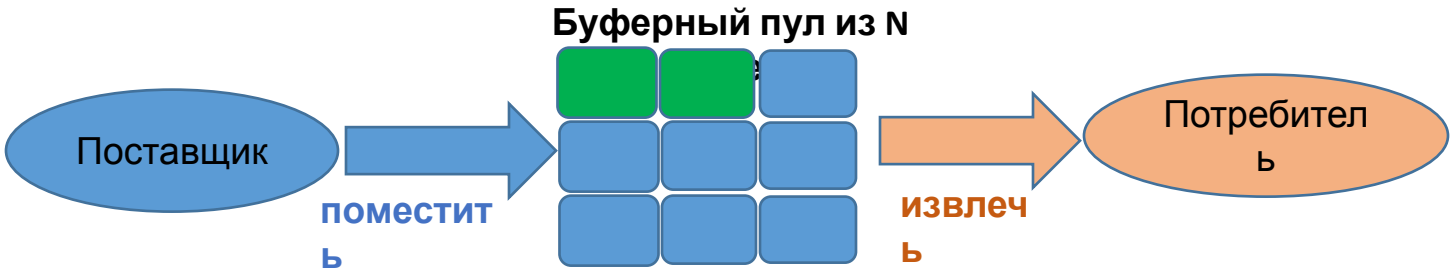
Решение правильное , оно принимается

ИЛИ

Решение не правильное , оно не принимается

?

Работа с буферным пулом



Процесс «поставщик» может быть представлен в виде двух частей: генерирование информации (П11), запись в буфер (П12).

Процесс «потребитель» так же можно разделить на две части: чтение информации из буфера (П21) и вывод информации на печать (П22).

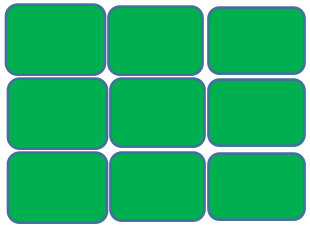
S= 1; e = N; f =0;

П1: {L1: П11; P(S); P(e); П12; V(f); V(S); goto L1}

П2: {L2: P(S); P(f); П21; V(e); V(S); П22; goto L2}

Ситуации

e = N; f = 0



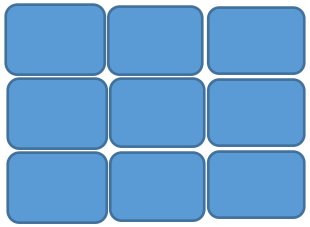
Длительный расчет

П1: {L1: П11; P(S); P(e); П12; V(f); V(S); goto L1}

П2: {L2: P(S); P(f); П21; V(e); V(S); П22; goto L2}

Очередной цикл работы

e = 0; f = N



Очередной цикл работы

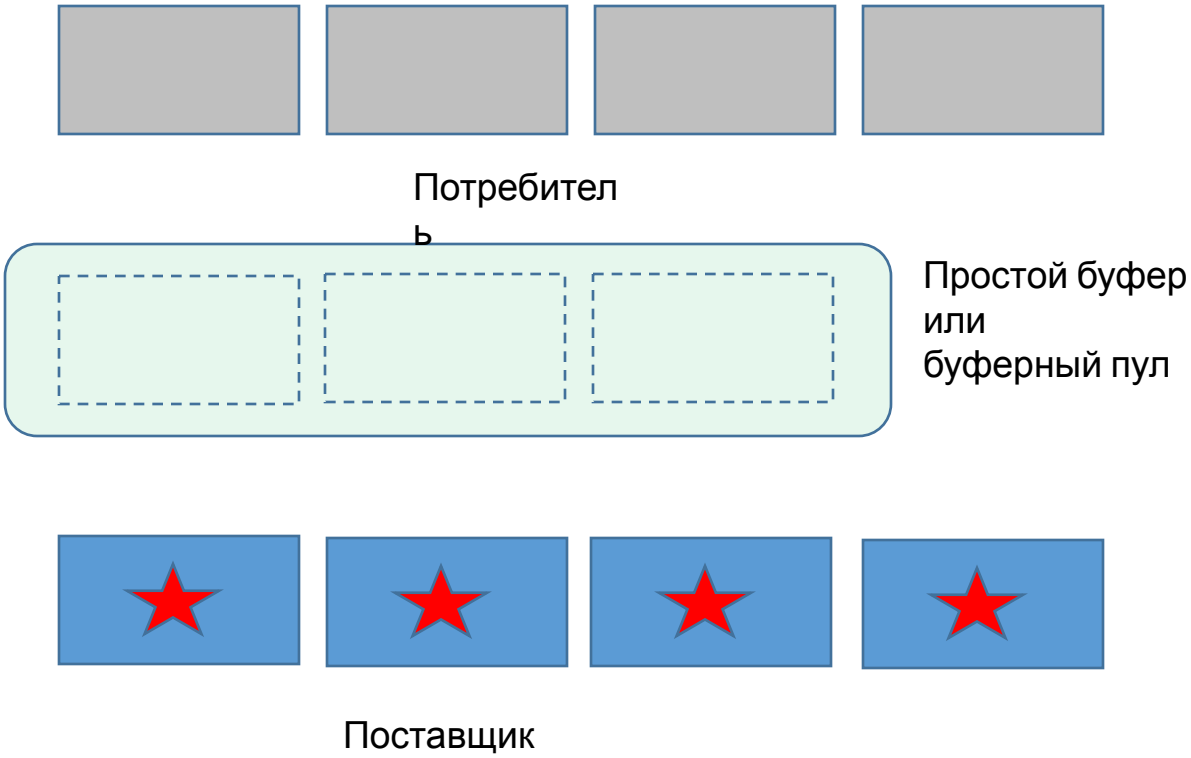
П1: {L1: П11; P(S); P(e); П12; V(f); V(S); goto L1}

П2: {L2: P(S); P(f); П21; V(e); V(S); П22; goto L2}

Длительный вывод данных

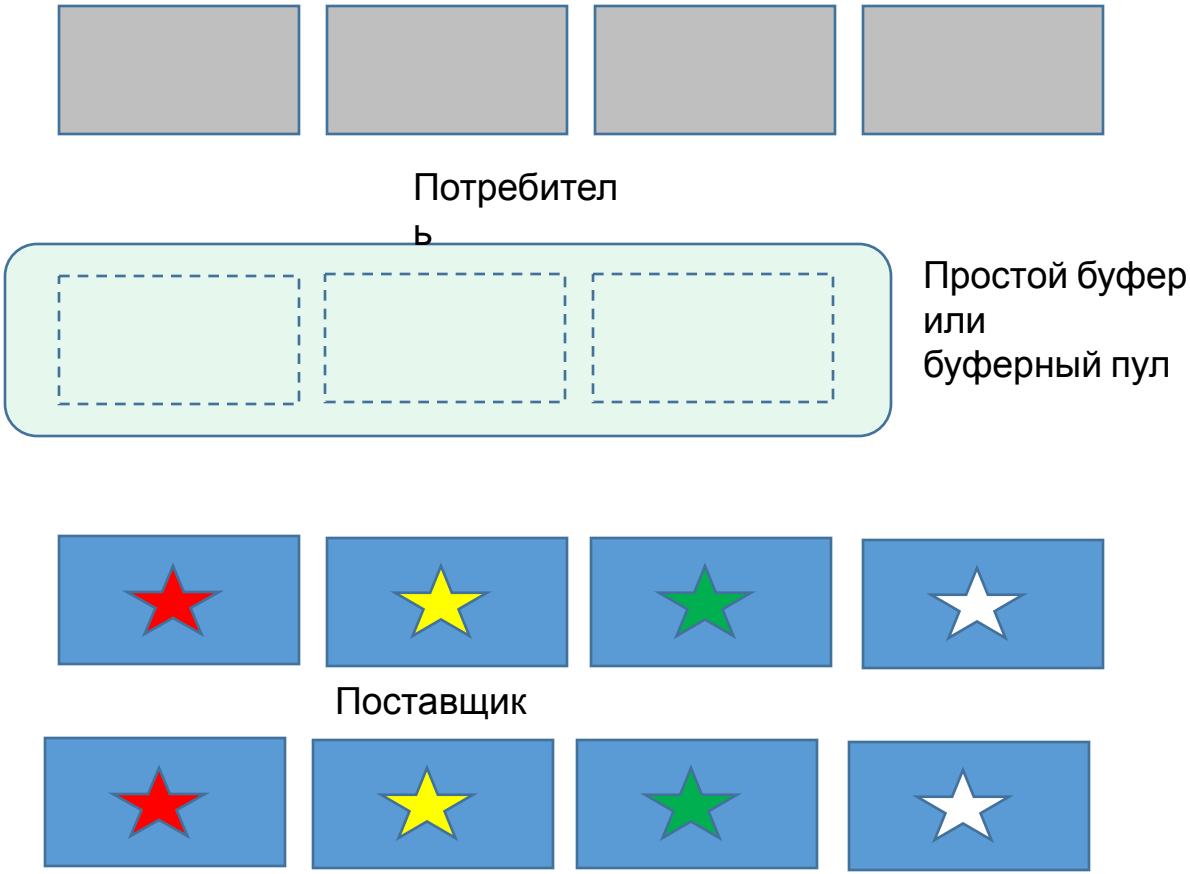
Работа с кольцевым буфером

В ряде случаев для получения нужного результата *последовательность заполнения буферных ячеек данными и последовательность излечения информации из буфера является важной*.



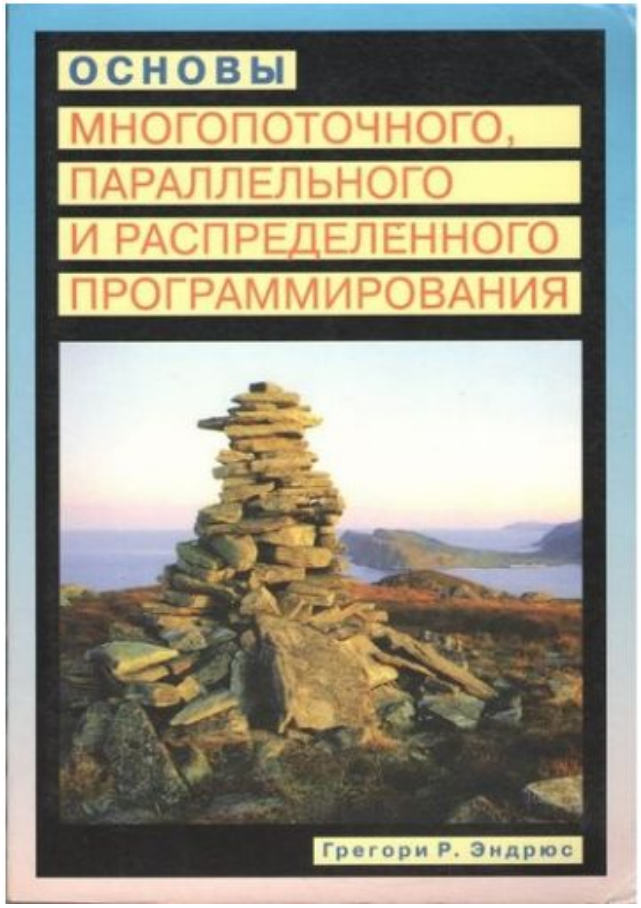
Работа с кольцевым буфером

В ряде случаев для получения нужного результата *последовательность заполнения буферных ячеек данными и последовательность излечения информации из буфера является важной*.



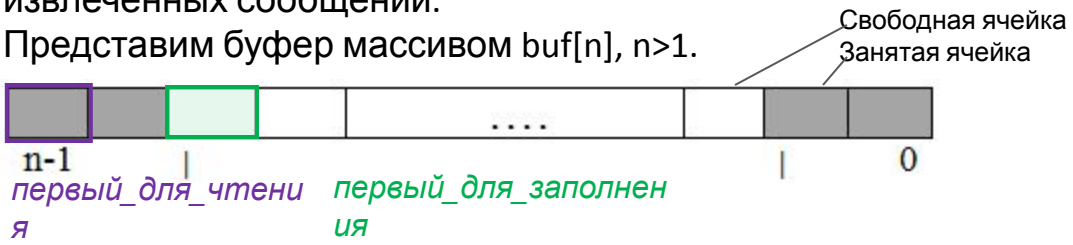
Работа с кольцевым буфером

В ряде случаев для получения нужного результата *последовательность заполнения буферных ячеек данными и последовательность излечения информации из буфера является важной*. Справиться с поставленной проблемой помогает *кольцевой буфер*, который используется как многоэлементный коммуникационный буфер.



Предположим, что есть два процесса: процесс-поставщик, который помещает сообщения в разделяемый буфер, и процесс-потребитель, извлекающий сообщения из буфера. Буфер содержит очередь уже помещенных, но еще не извлеченных сообщений.

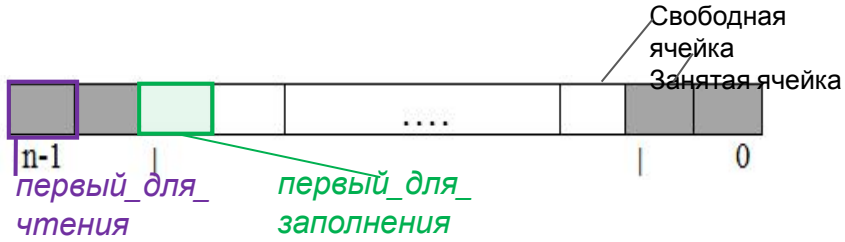
Представим буфер массивом buf[n], n>1.



Пусть переменная *первый_для_чтения* является индексом первого сообщения очереди, а переменная *первый_для_заполнения* – индексом первой пустой ячейки после сообщения в конце очереди. Для того, чтобы значения переменных *первый_для_заполнения* и *первый_для_чтения* всегда были в интервале от 0 до n-1, при их очередном вычислении используется операция «деление по модулю(взятие остатка)».

https://vk.com/doc116553074_496422611?hash=263f78e1c533caf88e

Работа с кольцевым буфером



При таком представлении буфера процесс-производитель помещает в него сообщение в ячейку с индексом ***первый_для_заполнения***, а затем определяется индекс следующей ячейки, подлежащей заполнению:

Buf[первый_для_заполнения]=Сообщение;
первый_для_заполнения = (первый_для_заполнения+1) % n;

Аналогично процесс-потребитель извлекает информацию их ячейки буфера с индексом ***первый_для_чтения***,
а затем определяется индекс следующей ячейки, из которой будет считываться информация:

Данные = Buf[первый_для_чтения];
первый_для_чтения = (первый_для_чтения+1) % n;

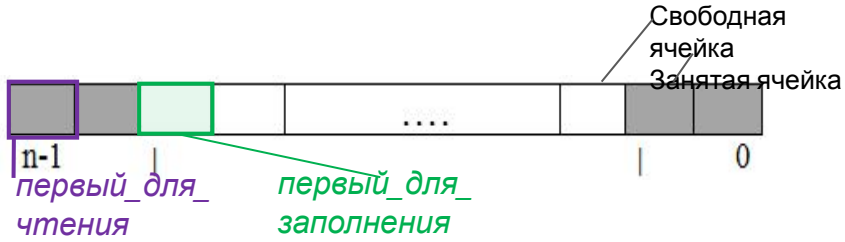
Решение задачи для пары процессов

Buf[n] / массив из n элементов */*
Первый_для_чтения = 0; Первый_для_заполнения = 0;
Пустой = n; Полный = 0; / Семафоры, хранящие число пустых и заполненных ячеек буфера */*

```
Process Поставщик {
  While(true) {
  …..
  создать Сообщение;
  /* поместить созданную информацию в буфер */
  P(Пустой);
  Buf[первый_для_заполнения] = Сообщение;
  первый_для_заполнения = (первый_для_заполнения+1) %
n;
  V(Полный);
                }
}
```

```
Process Потребитель {
  While(true) {
  /* извлечь информацию из буфера */
  P(Полный);
  Данные = Buf[первый_для_чтения];
  первый_для_чтения = (первый_для_чтения+1) %
n
  V(Пустой);
  }
}
```

Работа с кольцевым буфером



При таком представлении буфера процесс-производитель помещает в него сообщение в ячейку с индексом ***первый_для_заполнения***, а затем определяется индекс следующей ячейки, подлежащей заполнению:

Buf[первый_для_заполнения]=Сообщение;
первый_для_заполнения = (первый_для_заполнения+1) % n;

Аналогично процесс-потребитель извлекает информацию их ячейки буфера с индексом ***первый_для_чтения***,
а затем определяется индекс следующей ячейки, из которой будет считываться информация:

Данные = Buf[первый_для_чтения];
первый_для_чтения = (первый_для_чтения+1) % n;

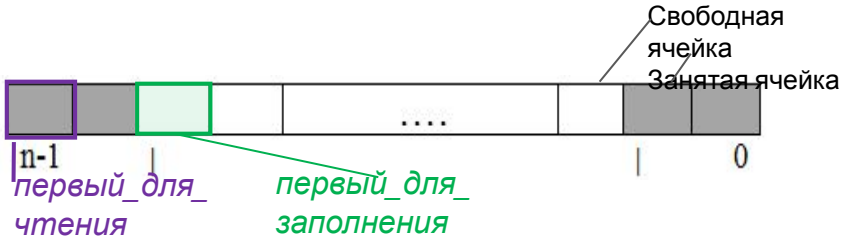
Общее решение задачи ???

Buf[n] / массив из n элементов */*
Первый_для_чтения = 0; Первый_для_заполнения = 0;
Пустой = n; Полный = 0; / Семафоры, хранящие число пустых и заполненных ячеек буфера */*

```
Process Поставщик {
  While(true) {
  …..
  создать Сообщение;
  /* поместить созданную информацию в буфер */
  P(Пустой);
  Buf[первый_для_заполнения] = Сообщение;
  первый_для_заполнения = (первый_для_заполнения+1) %
n;
  V(Полный);
                }
}
```

Представленное **решение справедливо в том случае, когда есть только один процесс-поставщик и один процесс-потребитель**. Предположим, что есть несколько процессов-производителей. Тогда при наличии хотя бы двух свободных ячеек кольцевого буфера два процесса могли бы выполнить операцию помещения информации в буфер. Однако, каждый из них попытался бы поместить данные в одну и ту же ячейку буфера с индексом *первый_для_заполнения*.

Работа с кольцевым буфером



Решение задачи для группы процессов-поставщиков и группы процессов-потребителей

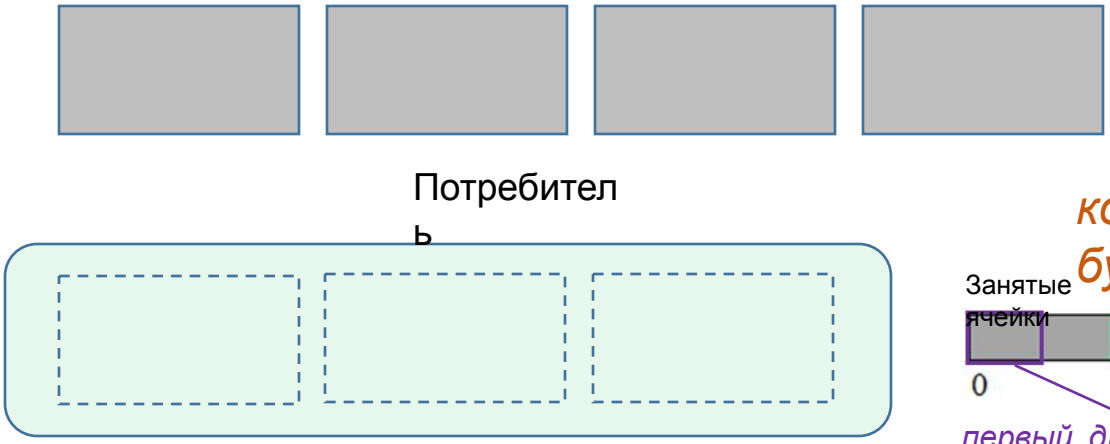
Buf[n] / массив из n элементов */*
Первый_для_чтения = 0; Первый_для_заполнения = 0;
Пустой = n; Полный = 0; / Семафоры, хранящие число пустых и заполненных ячеек буфера */*

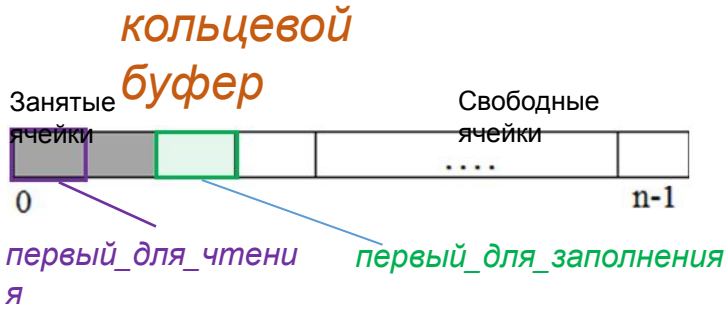
```
Process Поставщик {
  While(true) {




}
}
```

```
Process Потребитель {
  While(true) {




}
}
```

Работа с кольцевым буфером

В ряде случаев для получения нужного результата *последовательность заполнения буферных ячеек данными и последовательность излечения информации из буфера является важной*.

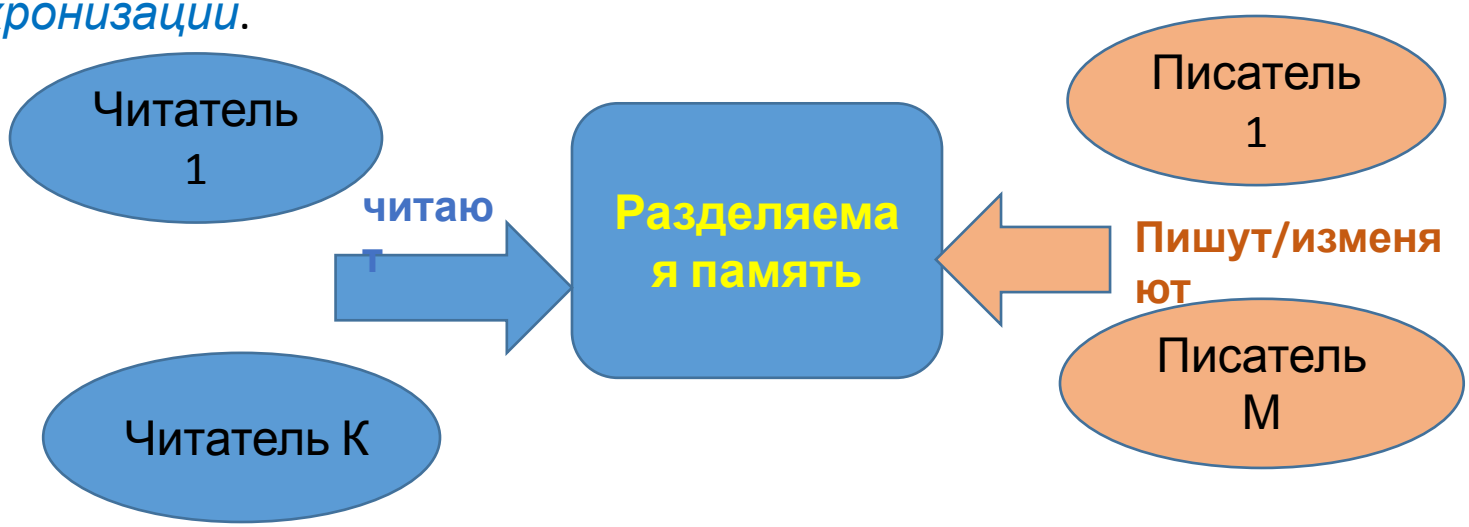




Класс задач «читатели-писатели»

Общее описание класса задач:

Задача о читателях и писателях относится к *классическим задачам синхронизации*.



Области применения

- организация работы с базами данных
- обслуживание файловых систем

Основное отличие от класса «поставщик-потребитель» - для процессов-читателей доступно **одновременное** использование ресурса по чтению.

Класс задач «читатели-писатели»

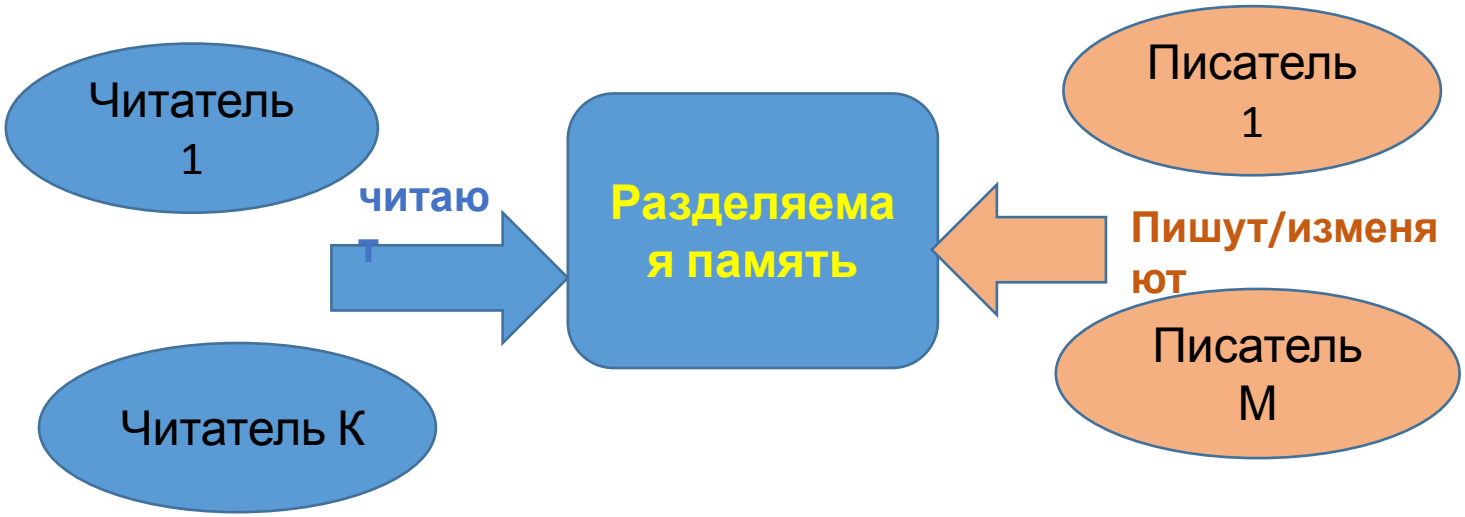
Общее описание класса задач:
Есть две группы процессов.
Процесс первой группы *просматривает данные*, хранящиеся в общей для всех процессов памяти. Такой процесс будем называть «*читатель*».
Процесс второй группы *записывает новые данные или обновляет ранее записанные данные* в общей памяти. ***Для выполнения операции процессу нужен монопольный доступ к ресурсу***.
Такой процесс будем называть «*писатель*».

Процессам-писателям нужен взаимоисключающий доступ к общей памяти.
Доступ процессов-читателей **как группы** также должен быть взаимоисключающим по отношению к любому процессу-писателю.

Наиболее широкое применение на практике нашли два варианта взаимодействия процессов-читателей и процессов-писателей:
взаимодействие с приоритетом «читателей» *и* ***взаимодействие с приоритетом «писателей»***.
В обоих случаях ***необходимо обеспечить взаимное исключение***.
Различия между вариантами взаимодействия процессов ***определяются ограничениями, которые накладываются на процессы-читатели при появлении процесса-писателя***.

В случае приоритета «читателей» действует правило*: если хотя бы один «читатель» уже пользуется ресурсом, то доступ «писателей» к ресурсу закрыт, а вновь приходящие «читатели» беспрепятственно получают доступ к ресурсу*.
В случае приоритета «писателей» правило звучит так: *при появлении запроса от*

Класс задач «читатели-писатели». Приоритет читателей.



Опишем в самом общем виде процесс-читатель и процесс-писатель:

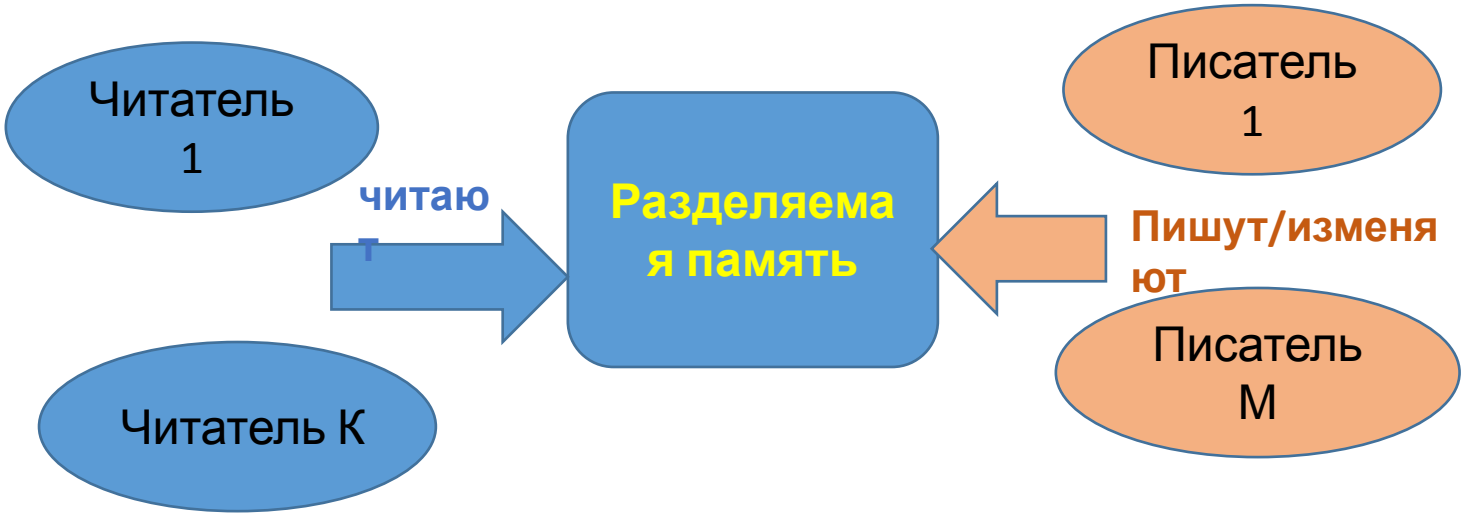
```
Process Читатель
While (true)
{
Чтение данных из общей памяти;
}
```

```
Process Писатель
While(true)
{
Запись данных в  общую память;
}
```

«Чтение_данных» и «Запись_данных» представляют собой ***критические секции процессов***, для которых нужно организовать взаимное исключение при доступе к общей памяти.

Эту задачу решаем введением двоичного семафора S.

Класс задач «читатели-писатели». Приоритет читателей.



s = 1; // семафор для организации взаимного исключения

```
Process Читатель
While (true)
{P(s);
Чтение данных из общей памяти;
V(s);
}
```

```
Process Писатель
While(true)
{P(s);
Запись данных в  общую память;
V(s);
}
```

К сожалению, представленное решение, обеспечив взаимное исключение процессов, не дает возможности нескольким процессам-читателям одновременно иметь доступ к общему ресурсу.

Класс задач «читатели-писатели». Приоритет читателей.

Процесс «Писатель/поставщик» может быть представлен в виде двух частей: ~~генерирование информации (П11)~~, запись в буфер (П12).

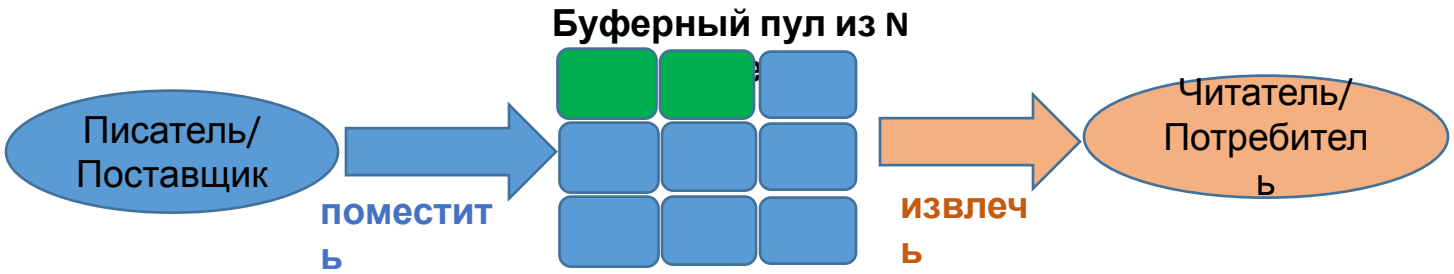
Процесс «Читатель/потребитель» так же можно разделить на две части: чтение информации из буфера (П21) ~~и вывод информации на печать (П22)~~.

s= 1; e = N; f =0;

П2 Читатель: {L2: P(f); P(s); П21; V(e); V(s); ~~П22~~; goto L2}

П1 Писатель: {L1: ~~П11~~; P(e); P(s); П12; V(f); V(s); goto L1}

Класс задач «читатели-писатели». Приоритет читателей.



Процесс «Писатель/поставщик» может быть представлен в виде :, запись в буфер (П12).

Процесс «Читатель/потребитель» можно описать так: чтение информации из буфера (П21).

s= 1; e = N; f =0;

П2 Читатель: {L2: P(f); P(s); П21; V(e); V(s); goto L2}

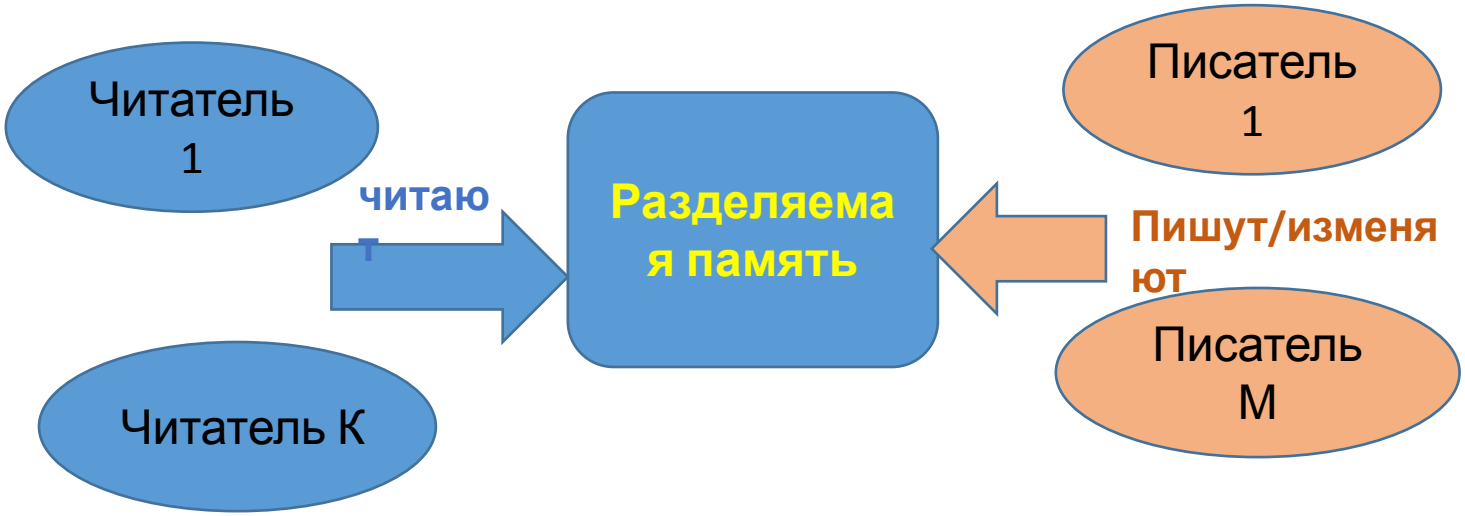
П1 Писатель: {L1: P(e); P(s); П12; V(f); V(s); goto L1}

Ситуации:

Если N>1, то возможна одновременная работа с общим ресурсом процессов двух типов. А писателю нужен МОНОПОЛЬНЫЙ доступ

Если N=1, то не возможна одновременная работа с общим ресурсом нескольких процессов-читателей!!!

Класс задач «читатели-писатели». Приоритет читателей.



w=1; // семафор для обеспечения монопольного доступа писателю
nr = 0; // число активных читателей, целое число

```
Process Читатель
While (true)
{ nr = nr +1;
If  (nr == 1)
{P(w); // заблокировать доступ писателю на работу с общей памятью}
Чтение данных из общей памяти;
 nr = nr - 1;
If  (nr == 0)
{V(w); // разблокировать доступ писателю на работу с общей памятью}
}
```

```
Process Писатель
While(true)
{
// захватить право на работу с общей памятью
P(w);
Запись данных в  общую память;
// вернуть право на работу с общей памятью
V(w);
}
```

Что почитать о семафорах

12. 004.451
С 30

Семафоры и их использование при решении задач учета ресурсов и управления процессами [Электронный ресурс] : методические указания к выполнению практических работ по дисциплине «Операционные системы» для студентов бакалавриата очной формы обучения направления подготовки 09.03.01 Информатика и вычислительная техника / Моск. гос. строит. ун-т, Каф. информационных систем, технологий и автоматизации в строительстве ; [сост. Н.А. Иванов]. - Электрон. текстовые дан. - Москва : МГСУ, 2015. - Б. ц.
УДК 004.451

Кл.слова (ненормированные):
управление процессами -- методические указания

Перейти: http://lib-04.gic.mgsu.ru/lib/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4%D0%B8%D1%87%D0%BA%D0%B8%202015%20-%202/269.pdf

Доп.точки доступа:
Иванов, Н. А. \сост.\
Экземпляры всего: 1
Имеется только электронная копия издания! (1)
Свободны: Имеется только электронная копия издания! (1)

http://lib-04.gic.mgsu.ru/lib/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4%D0%B8%D1%87%D0%BA%D0%B8%202015%20-%202/269.pdf

Синхронизация и средства её обеспечения

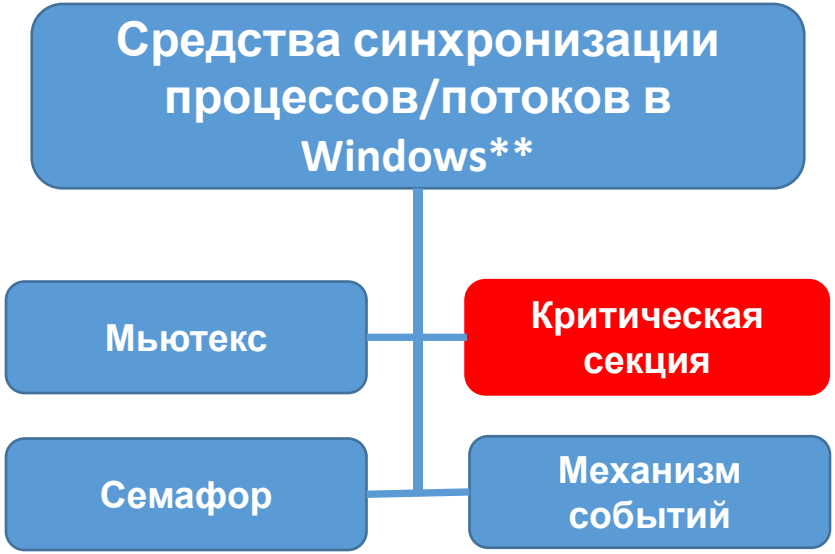
- Управление процессами
  - Планирование процессов
  - Создание процесса
  - Выделение системных ресурсов
  - Синхронизация процессов/потоков
  - Взаимодействие процессов
  - Уничтожение процесса

Синхронизация заключается в **согласовании скоростей развития процессов или потоков** путем приостановки процесса/потока ***до наступления некоторого события*** и последующей его активизации, когда это событие произойдет.

Средства синхронизации процессов/потоков*

- **Мьютекс**
- **Критическая секция**
- **Семафор**
- **Механизм событий**

Синхронизация и средства её обеспечения



Критические секции

Объект - *критическая секция* помогает программисту выделить участок кода, где нить* получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса.

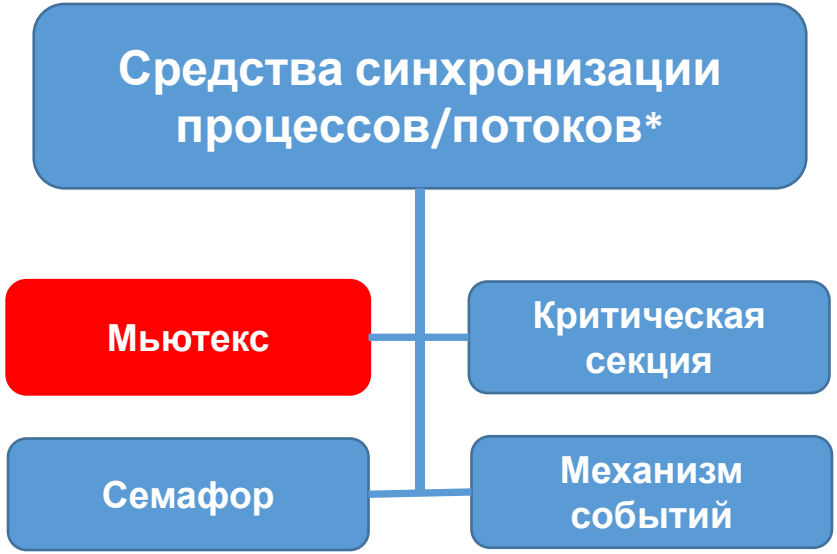
Перед использованием ресурса нить входит в *критическую секцию* (вызывает функцию EnterCriticalSection).

Если после этого какая-либо другая нить попытается войти в ту же самую критическую секцию, ее выполнение приостановится, пока первая нить не покинет секцию с помощью вызова LeaveCriticalSection. ***Используется только для нитей одного процесса***. Порядок входа в критическую секцию не определен.

* Нить = поток

** Источник: Синхронизация процессов и потоков
http://www.codenet.ru/progr/cpp/process-threads-sync.php

Синхронизация и средства её обеспечения



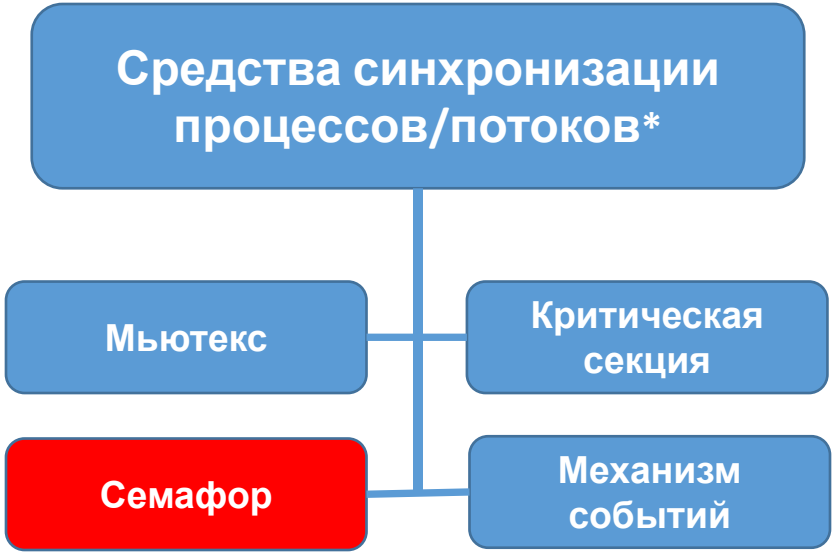
Взаимоисключения / мьютексы
Объекты-взаимоисключения (мьютексы, mutex - от MUTual EXclusion) позволяют координировать взаимное исключение доступа к разделяемому ресурсу. Сигнальное состояние объекта (т.е. состояние "установлен") соответствует моменту времени, когда объект не принадлежит ни одной нити и его можно "захватить". И наоборот, состояние "сброшен" (не сигнальное) соответствует моменту, когда какая-либо нить уже владеет этим объектом. Доступ к объекту разрешается, когда нить, владеющая объектом, освободит его.

Две (или более) нити могут создать мьютекс с одним и тем же именем, вызвав функцию CreateMutex. Первая нить действительно создает мьютекс, а следующие - получают дескриптор уже существующего объекта. Это дает возможность нескольким нитям получить дескриптор одного и того же мьютекса, освобождая программиста от необходимости заботиться о том, кто в действительности создает мьютекс.

Несколько нитей могут получить дескриптор одного и того же мьютекса, что делает возможным взаимодействие между процессами.

Для синхронизации нитей одного процесса применение мьютексов *менее эффективно* чем использование *критических секций*.

Синхронизация и средства её обеспечения



Семафоры

Объект-семафор - это фактически *объект-взаимоисключение со счетчиком*.

Данный объект позволяет "захватить" себя определенному количеству нитей. После этого "захват" будет невозможен, пока одна из ранее "захвативших" семафор нитей не освободит его.

Семафоры применяются для ограничения количества нитей, одновременно работающих с ресурсом.

Объекту при инициализации передается максимальное число нитей, после каждого "захвата" счетчик семафора уменьшается.

Сигнальному состоянию соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается не установленным (сброшенным).

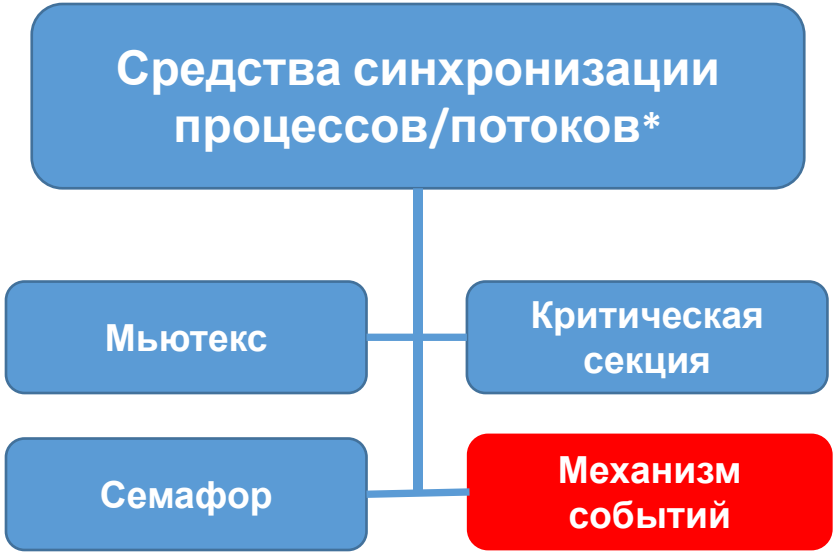
Функции:

CreateSemaphore создает объект-семафор с указанием и максимально возможного начального его значения;

OpenSemaphore возвращает дескриптор существующего семафора, захват семафора производится с помощью ожидающих функций, при этом значение семафора уменьшается на единицу;

ReleaseSemaphore освобождение семафора с увеличением значения семафора на указанное в параметре число.

Синхронизация и средства её обеспечения



События
Объекты-события используются для уведомления ожидающих нитей о наступлении какого-либо события. Различают два вида событий - с *ручным* и *автоматическим* сбросом.
Ручной сброс осуществляется функцией ResetEvent.
События с ручным сбросом используются для уведомления сразу *нескольких* нитей.
При использовании события с ***автосбросом*** уведомление получит и продолжит свое выполнение *только одна* ожидающая нить, остальные будут ожидать дальше.

Функция ***CreateEvent*** создает объект-событие,
функция ***SetEvent*** - устанавливает событие в сигнальное состояние,
функция ***ResetEvent*** - сбрасывает событие.
Функция ***PulseEvent*** устанавливает событие, а после возобновления ожидающих это событие нитей (всех при ручном сбросе и только одной при автоматическом), сбрасывает его. Если ожидающих нитей нет, PulseEvent просто сбрасывает событие.

Взаимодействие процессов

- Управление процессами
  - Создание процесса
  - Уничтожение процесса
  - Планирование процессов
  - Выделение системных ресурсов
  - Синхронизация процессов
  - Взаимодействие процессов

Взаимодействия процессов заключается в *обеспечение силами ОС обмена данными между процессами*.

В общем случае ОС препятствуют непосредственному общению процессов, используя средства по защите процессов друг от друга, в первую очередь, выделение каждому из процессов отдельного адресного пространства. Однако, в ряде случаев *логика работы процессов требует их* ***взаимодействия***.

Операционная система имеет доступ ко всем областям памяти, поэтому она может играть роль посредника в информационном обмене процессов:
при возникновении необходимости в обмене данными процесс обращается с запросом к ОС, по которому ОС, пользуясь своими привилегиями, создает различные системные средства связи, такие, например, как ***каналы, очереди сообщений*** или ***разделяемую память***.
Эти средства относят к классу ***средств межпроцессного взаимодействия***.

Синхронизация и взаимодействие процессов

- Управление процессами
  - Создание процесса
  - Уничтожение процесса
  - Планирование процессов
  - Выделение системных ресурсов
  - Синхронизация процессов
  - Взаимодействие процессов

Многие из ***средств межпроцессного обмена данными*** выполняют также и функции ***синхронизации***:
в том случае, когда данные для процесса-получателя отсутствуют, последний переводится в состояние ожидания средствами ОС, а при поступлении данных от процесса-отправителя процесс-получатель активизируется.

Что почитать о средствах синхронизации в Windows и UNIX

Общие сведения о примитивах синхронизации
https://docs.microsoft.com/ru-ru/dotnet/standard/threading/overview-of-synchronization-primitives

Практикум по системному программированию
http://www.ncfu.ru/export/uploads/imported-from-dle/op/doclinks2015/Metod_PraktSistProgr_10.05.03_19.05.15.pdf

Синхронизация процессов и потоков
http://www.codenet.ru/progr/cpp/process-threads-sync.php

Спасибо за внимание