4.1 Взаимодействие между процессами

Ситуации, когда приходится процессам взаимодействовать:

- Передача информации от одного процесса другому
- Контроль над деятельностью процессов (например: когда они борются за один ресурс)
- Согласование действий процессов (например: когда один процесс поставляет данные, а другой их выводит на печать. Если согласованности не будет, то второй процесс может начать печать раньше, чем поступят данные).

Два вторых случая относятся и к потокам. В первом случае у потоков нет проблем, т.к. они используют общее адресное пространство.

4.1.1 Передача информации от одного процесса другому

Передача может осуществляться несколькими способами:

- Разделяемая память
- **Каналы** (трубы), это псевдофайл, в который один процесс пишет, а другой читает.
- **Сокеты** - поддерживаемый ядром механизм, скрывающий особенности среды и позволяющий единообразно взаимодействовать процессам, как на одном компьютере, так и в сети.
- Почтовые ящики (только в Windows), однонаправленные, возможность широковещательной рассылки.
- Вызов удаленной процедуры, процесс **A** может вызвать процедуру в процессе **B**, и получить обратно данные.

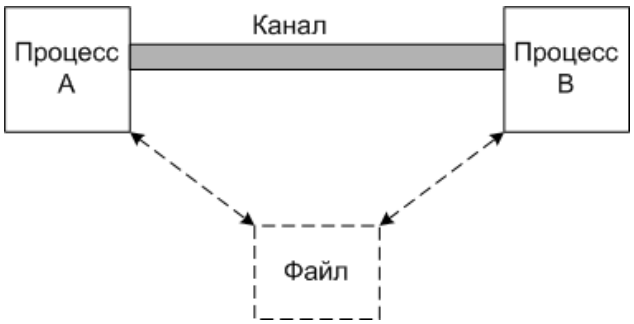


Схема для канала

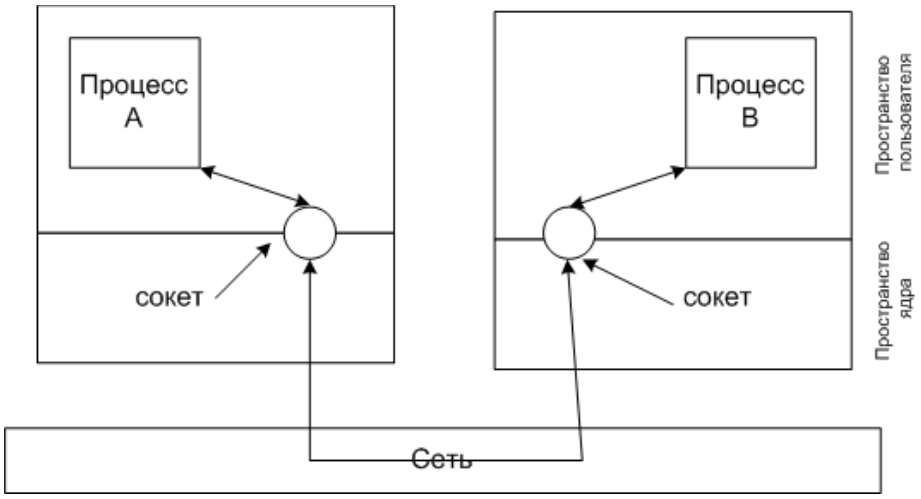


Схема для сокетов

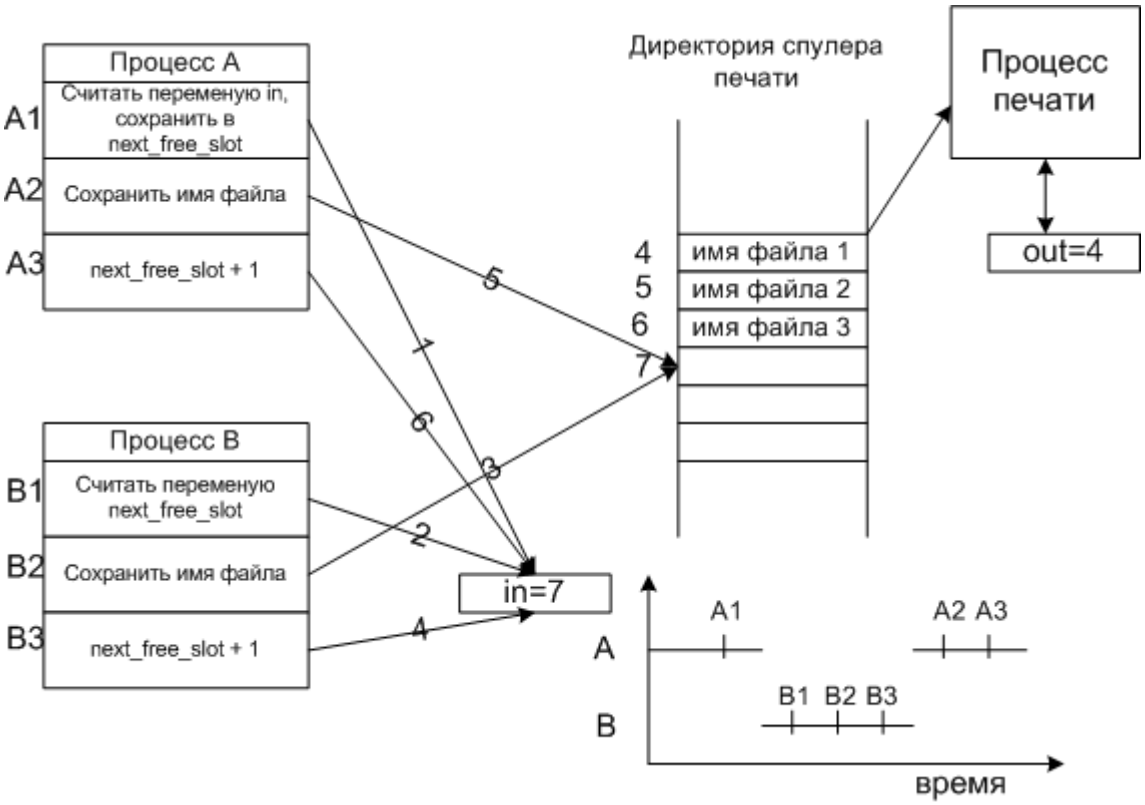
4.1.2 Состояние состязания

Состояние состязания - ситуация когда несколько процессов считывают или записывают данные (в память или файл) одновременно.

Рассмотрим пример, когда два процесса пытаются распечатать файл. Для этого им нужно поместить имя файла в спулер печати, в свободный сегмент.

in - переменная указывающая на следующий свободный сегмент

out - переменная указывающая на следующее имя файла для печати



Пример состязания

Распишем события по пунктам.

1. Процесс **A** считывает переменную in (равную 7), и сохраняет ее в своей переменной next_free_slot.
2. Происходит прерывание по таймеру, и процессор переключается на процесс **B**.
3. Процесс **B** считывает переменную in (равную 7), и сохраняет ее в своей переменной next_free_slot.
4. Процесс **B** сохраняет имя файла в сегменте **7.**
5. Процесс **B** увеличивает переменную next_free_slot на единицу (next_free_slot+1), и заменяет значение in на 8.
6. Управление переходит процессу **A**, и продолжает с того места на котором остановился.
7. Процесс **A** сохраняет имя файла в сегменте **7**, затирая имя файла процесса **B**.
8. Процесс **A** увеличивает переменную next_free_slot на единицу (next_free_slot+1), и заменяет значение in на 8.

Как видно из этой ситуации, файл процесса **B** не будет напечатан.

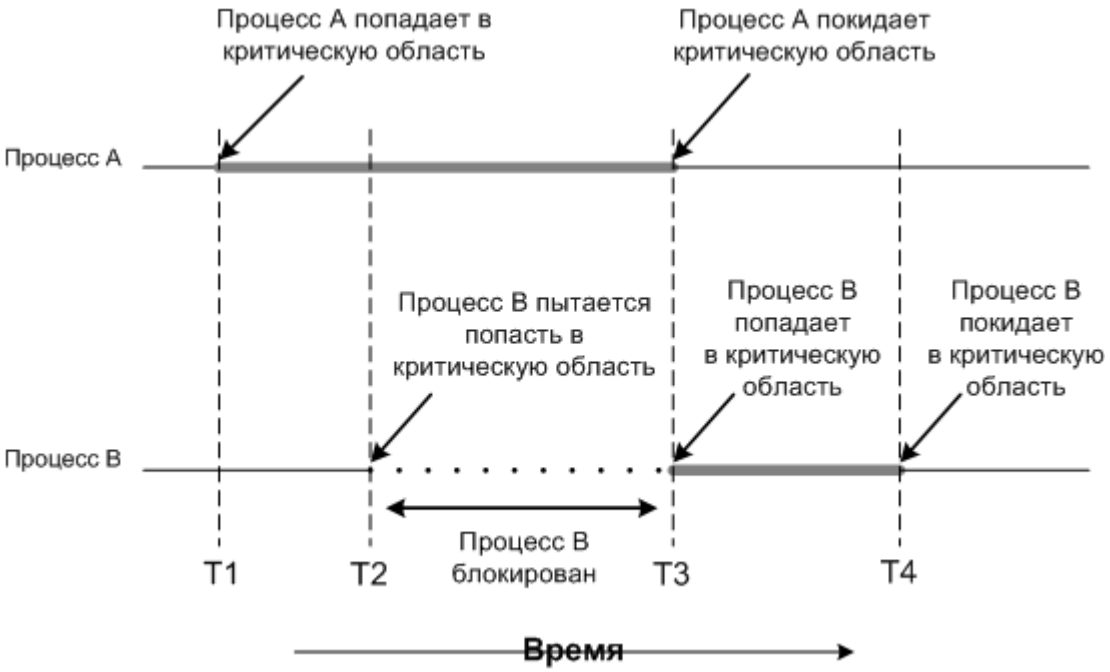
4.1.3 Критические области

Критическая область - часть программы, в которой есть обращение к совместно используемым данным.

Условия избегания состязания и эффективной работы процессов:

1. Два процесса не должны одновременно находиться в критических областях.
2. Процесс, находящийся вне критической области, не может блокировать другие процессы.
3. Невозможна ситуация, когда процесс вечно ждет (зависает) попадания в критическую область.

Пример:



Взаимное исключение с использованием критических областей

4.1.4 Взаимное исключение с активным ожиданием

Рассмотрим методы взаимного исключения

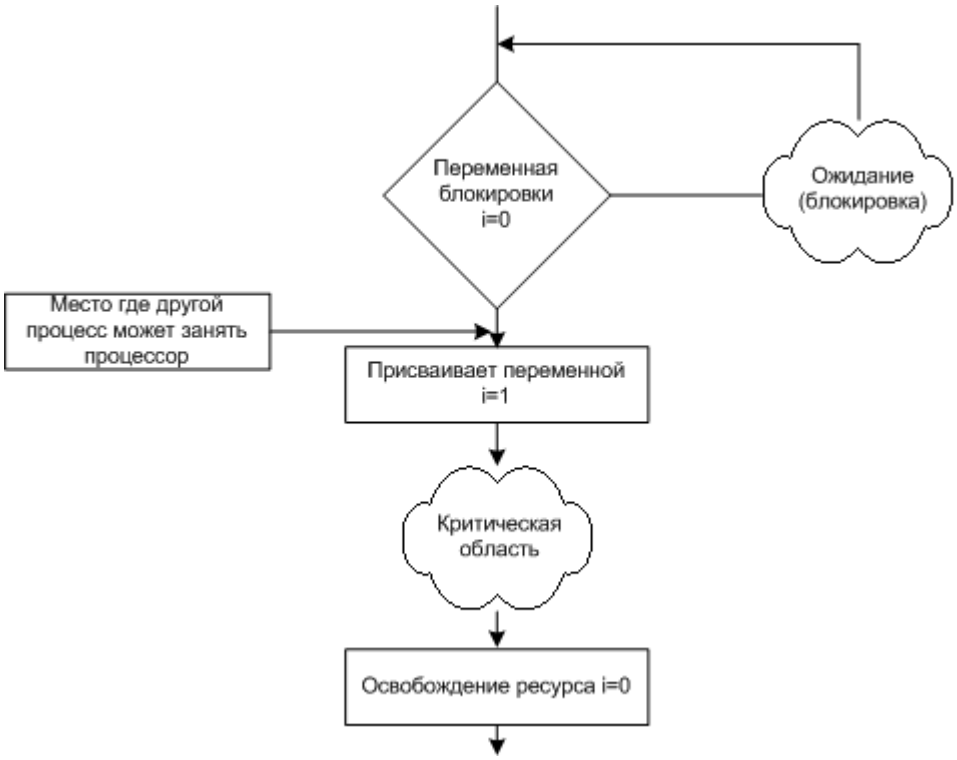
Запрещение прерываний

Заключается в запрещении всех прерываний при входе процесса в критическую область.

Недостаток этого метода в том, что если произойдет сбой процесса, то он не сможет снять запрет на прерывания.

Переменные блокировки

Вводится понятие переменной блокировки, т.е. если значение этой переменной равно, например 1, то ресурс занят другим процессом, и второй процесс переходит в режим ожидания (блокируется) до тех пор, пока переменная не примет значение 0.

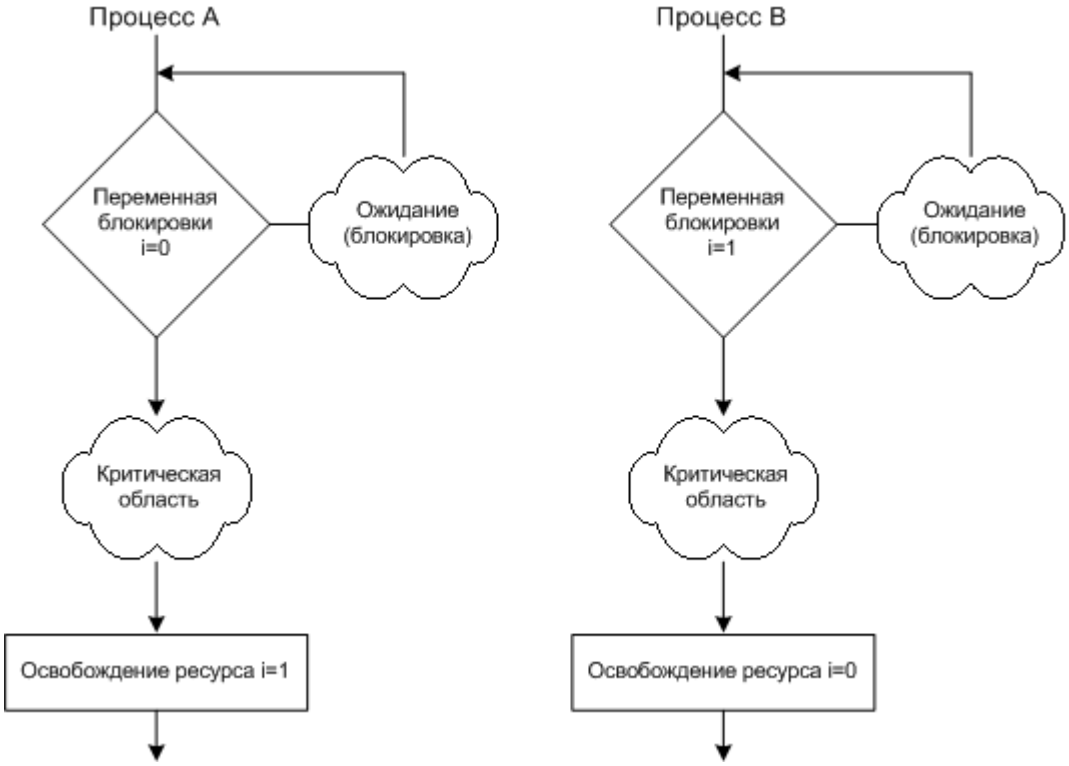


метод блокирующих переменных

Проблема, как и с процессом печати, после того как первый процесс считает 0, второй может занять процессор и тоже считать 0. Заблокированный процесс находится в режиме **активного ожидания**, постоянно проверяя, не изменилась ли переменная блокировки.

Строгое чередование

В этой модели, процессы могут выполняться строго по очереди, используя переменную.



Строгое чередование

Недостатки метода:

- Заблокированный процесс постоянно находится в цикле, проверяя, не изменилась ли переменная.
- Противоречит третьему условию, когда процесс, находящийся вне критической области, может блокировать другие процессы.

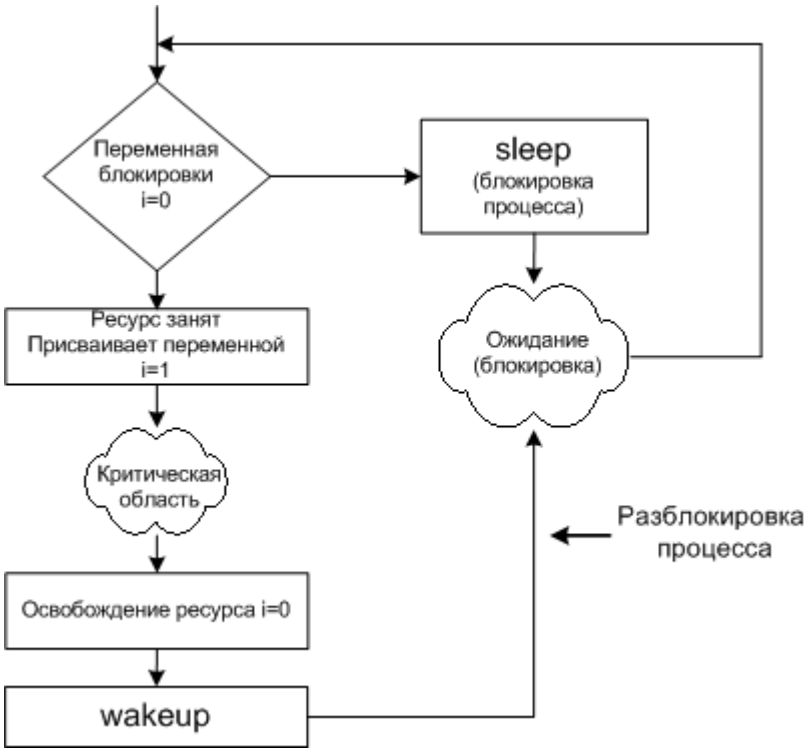
Существуют еще алгоритмы с активным ожиданием (**алгоритм Петерсона, команда TSL**), но у всех них есть общий недостаток - расходуется бесцельно время процессора на циклы проверки изменения переменной.

4.1.5 Примитивы взаимодействия процессов

Вводится понятия двух примитивов.

sleep - системный запрос, в результате которого вызывающий процесс блокируется, пока его не запустит другой процесс.

wakeup - системный запрос, в результате которого блокированный процесс будет запущен.



Применение примитивов

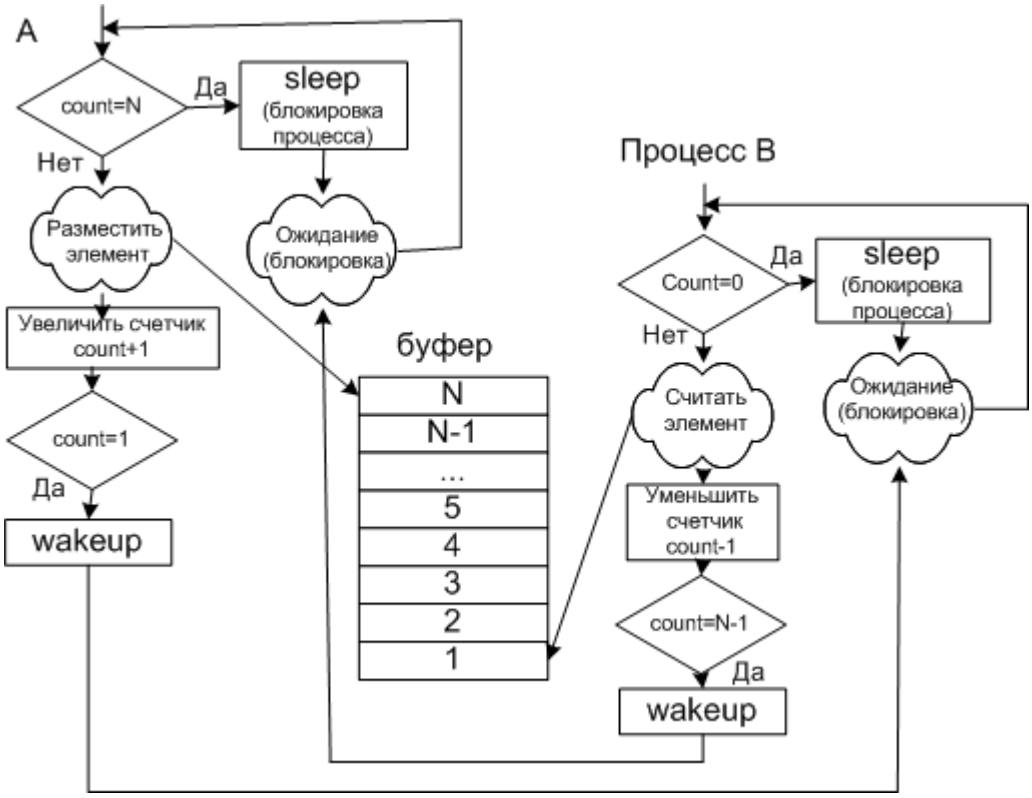
Основное преимущество - это отсутствие активного ожидания..

Проблема заключается в следующем, если спулер пуст, то wakeup срабатывает в пустую.

Проблема переполненного буфера (проблема производителя и потребителя)

Рассмотрим два процесса, которые совместно используют буфер ограниченного размера, один процесс пишет в буфер, другой считывает данные.

Чтобы первый процесс не писал, когда буфер полный, а второй не считывал, когда он пуст, вводится переменная *count* для подсчета количества элементов в буфере.



Проблема переполненного буфера

В этой ситуации оба процесса могут попасть в состояние ожидания, если пропадет сигнал активации.

Алгоритм такой ситуации:

1. Процесс **B**, считал count=0 (заблокироваться он еще не успел)
2. Планировщик передал управление процессу **A**
3. Процесс **A**, выполнил все вплоть до wakeup, пытаясь разблокировать процесс **B** (но он не заблокирован, wakeup срабатывает впустую)
4. Планировщик передал управление процессу **B**
5. И он заблокировался, и больше сигнала на разблокировку не получит
6. Процесс **A** в конце концов заполнит буфер и заблокируется, но сигнала на разблокировку не получит.

4.1.6 Семафоры

Семафоры - переменные для подсчета сигналов запуска, сохраненных на будущее.

Были предложены две операции **down** и **up** (аналоги sleep и wakeup).

Прежде чем заблокировать процесс, down проверяет семафор, если он равен нулю, то он блокирует процесс, если нет, то процесс снова становится активным, и уменьшает семафор на единицу.

up увеличит значение семафора на 1 или разблокирует процесс находящийся в ожидании..

down уменьшает значение семафора на 1 или блокирует процесс, если семафор =0.

down и up выполняются как **элементарное действие**, т.е. процесс не может быть блокирован во время выполнения этих операций. Значит, у операционной системы должен быть запрет на все прерывания, и перевод процесса в режим ожидания.

Решение проблемы переполненного буфера с помощью семафора

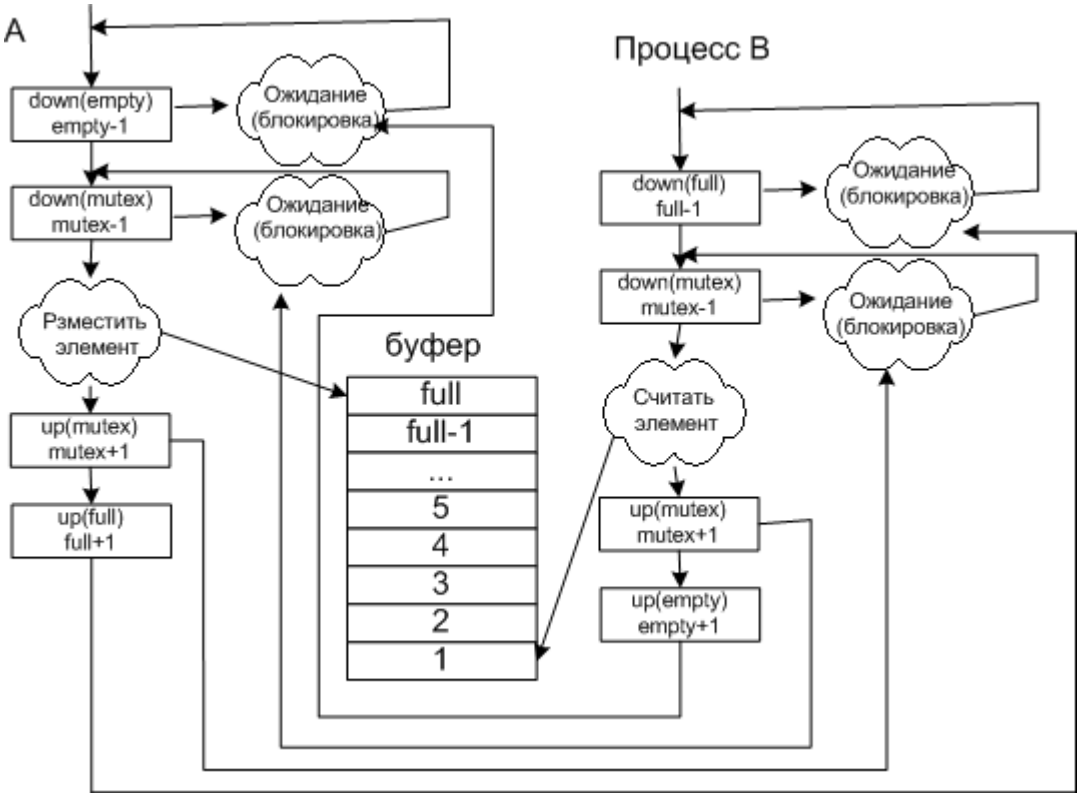
Применим три семафора:

full - подсчет заполненных сегментов (в начале = 0)

empty - подсчет пустых сегментов (в начале = количеству сегментов)

mutex - для исключения одновременного доступа к буферу двух процессов. (в начале = 1)

Мьютекс упрощенная версия семафора, он управляет доступом к ресурсу. Показывает, блокирован или нет ресурс.



Решение проблемы переполненного буфера с помощью семафора

Применение семафоров для устройств ввода/вывода

Для устройств ввода/вывода семафор выставляется равный нулю. После запуска управляющего процесса выполняется down, и т.к. семафор равен нулю, процесс блокируется. Когда нужно активизировать процесс управления, выполняется up.