Приложение 1. Отчет по лабораторной работе

**Федеральное государственное автономное образовательное учреждение высшего профессионального образования   
"Национальный исследовательский университет   
"Высшая школа экономики"**

  Факультет компьютерных наук

Департамент программной инженерии

Отчёт по лабораторной работе

**Лабораторные работы 6-7**

**Средства System V IPC.**

**Организация работы с разделяемой памятью в UNIX.**

**Понятие нитей исполнения (thread)**

по курсу «Операционные системы»

Выполнил:

Принял

Фомин Сергей Сергеевич

Оценка:

Москва 2020г.

Оглавление

[Задание 1 3](#_Toc39448804)

[Задание 2 6](#_Toc39448805)

[Задание 3 8](#_Toc39448806)

[Задание 4 11](#_Toc39448807)

[Задание 5 14](#_Toc39448808)

[Задание 6 15](#_Toc39448809)

**Для каждого задания указывается:**

1. Формулировка задания.
2. Исходный текст программы.
3. Формат вызова программы.
4. Результат работы программы.
5. Объяснения полученных результатов.

# Задание 1

Откомпилируйте программы 06-1а.с и 06-1b.c, запустите их несколько раз. Проанализируйте полученные результаты.

(5 баллов)

Исходный текст программы:

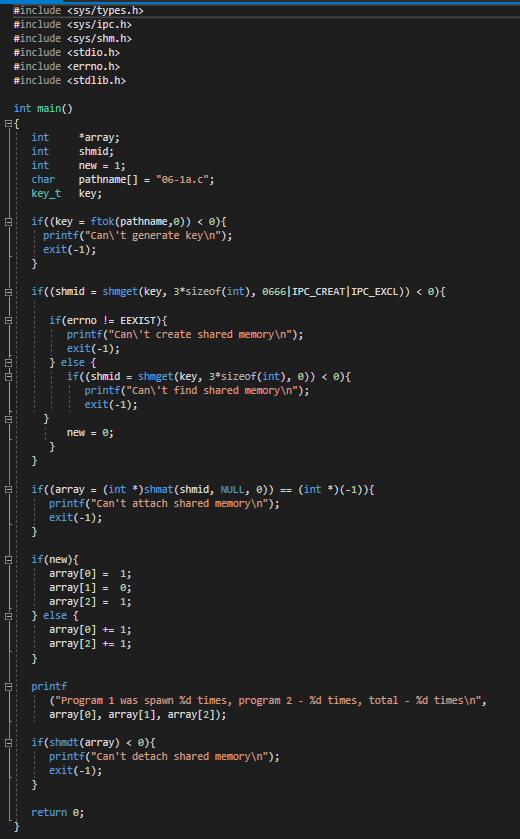


Рисунок . 06-1а.с

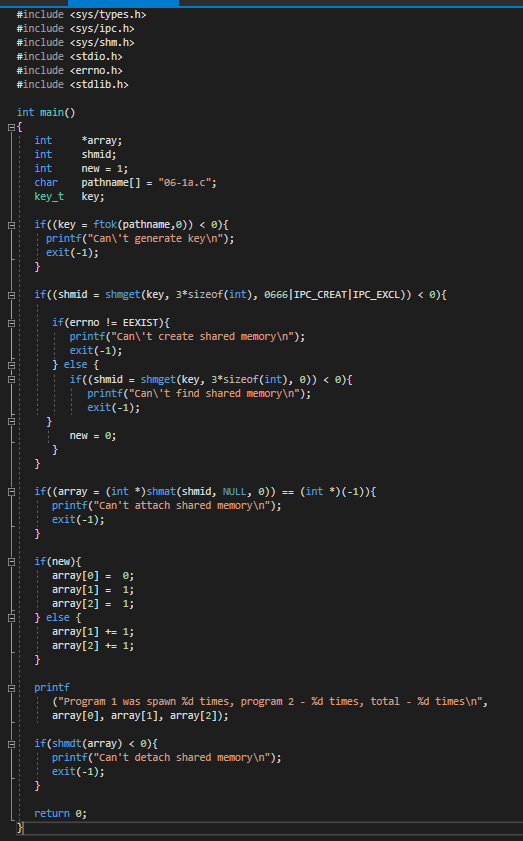


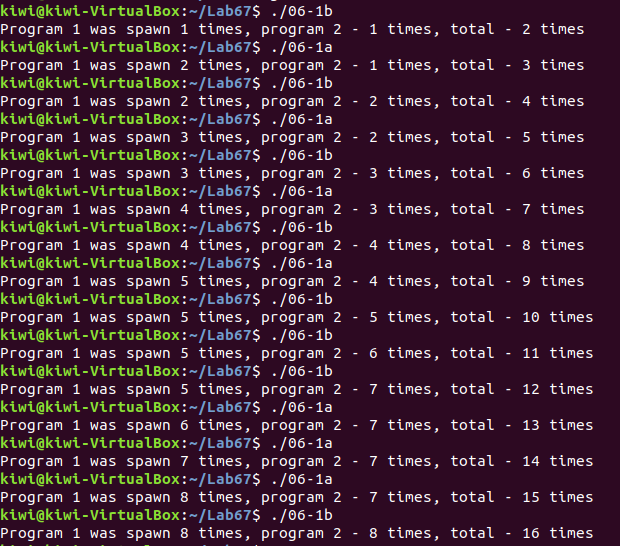
Рисунок . 06-1b.c

Формат вызова:





Результат работы:



Объяснение: функция ftok служит для преобразования имени существующего файла и небольшого целого числа, например, порядкового номера экземпляра средств связи, в ключ System V IPC. shmget предназначен для выполнения операции доступа к сегменту разделяемой памяти и, в случае его успешного завершения, возвращает дескриптор System V IPC для этого сегмента (целое неотрицательное число, однозначно характеризующее сегмент внутри вычислительной системы и использующееся в дальнейшем для других операций с ним). В качестве флагов поставляется комбинация прав доступа к создаваемому сегменту, флагов IPC\_CREAT и IPC\_EXCL. Если сегмент для данного ключа еще не существует, то система будет пытаться создать его с указанными правами доступа. Если же вдруг он уже существовал, то мы просто получим его дескриптор. Флаг IPC\_EXCL гарантирует нормальное завершение системного вызова только в том случае, если сегмент действительно был создан (т. е. ранее он не существовал), если же сегмент существовал, то системный вызов завершится с ошибкой, и значение системной переменной errno, описанной в файле errno.h, будет установлено в EEXIST.

Системный вызов shmat предназначен для включения области разделяемой памяти в адресное пространство текущего процесса. shmdt предназначен для исключения области разделяемой памяти из адресного пространства текущего процесса.

В программе организуется разделяемая память для массива из трех целых чисел, где первый элемент массива является счетчиком числа запусков программы a, второй – счетчиком числа запусков программы b, третий – счетчиком числа запусков обеих программ. В случае, если возникает ошибка отличная от EEXIST, происходит выход из программы, иначе получаем дескриптор сегмента и флаг необходимости инициализации элементов массива, следовательно увеличиваем значения элементов массива. Разные программы увеличивают значения элементов одного и того же массива, выполняясь отдельно, но при этом находясь в одном сегменте.

# Задание 2

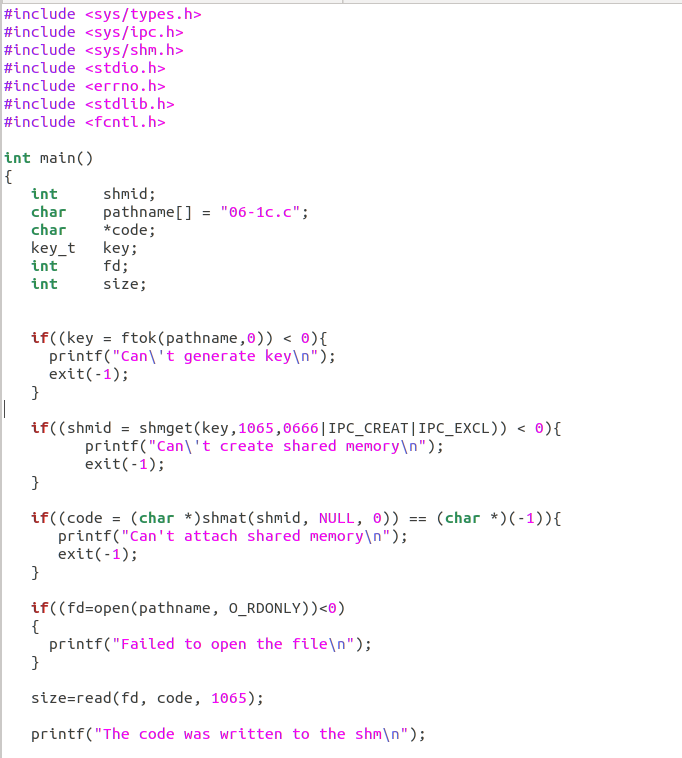
Напишите две программы, осуществляющие взаимодействие через разделяемую память. **Первая программа** должна создавать сегмент разделяемой памяти и копировать туда собственный исходный текст, **вторая программа** должна брать оттуда этот текст, печатать его на экране и удалять сегмент разделяемой памяти из системы.

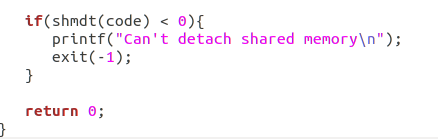
Прокомментируйте основные этапы работы программ.

(8 баллов)

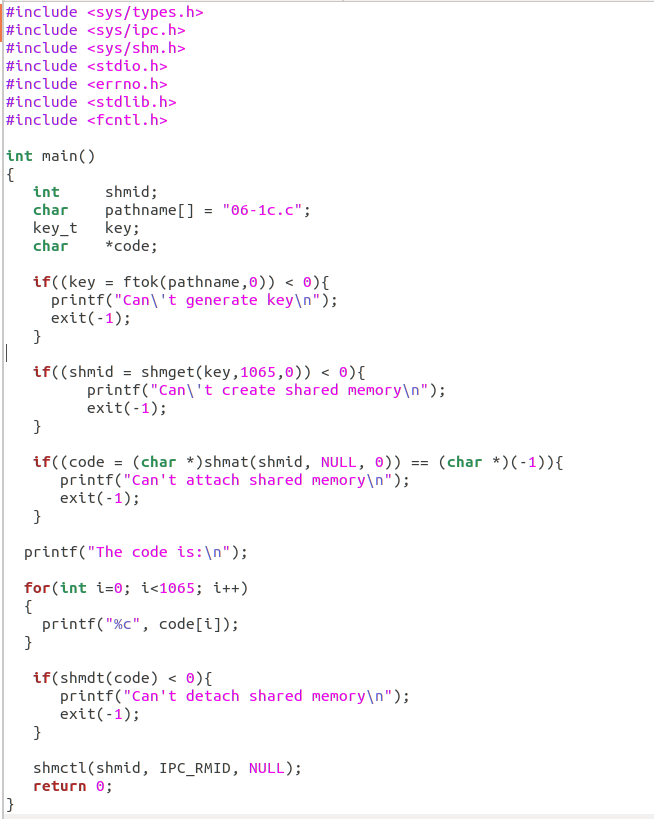
Исходный текст программы:

Запись кода:





Чтение:



Формат вызова:

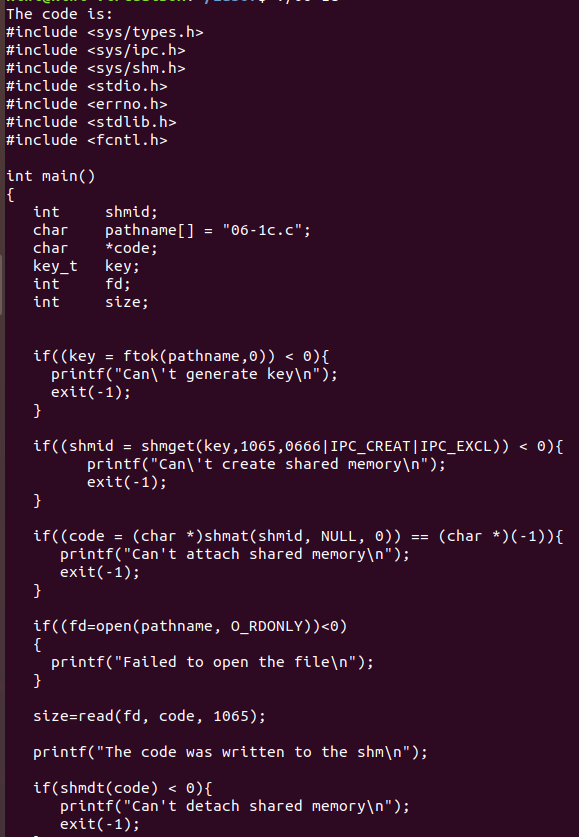


Рисунок . запись файла



Рисунок . чтение и вывод на экран

Результат работы:





Объяснение:

В программе записи кода происходит генерация IPC ключа, после чего мы пытаемся создать сегмент разделяемой памяти. После включения области в адресное пространство текущего процесса, происходит открытие файла и его запись в, после которой происходит исключение области из текущего адресного пространства.  
В программе чтения записанного ранее кода происходит получение ключа и дескриптора сегмента, который также как и ранее включается в адресное пространство текущего процесса (который записан в указатель code), далее происходит вывод code (т.е. записанного в первой программе сегмента), после которой происходит исключение области из текущего адресного пространства и удаление сегмента разделяемой памяти из системы.

# Задание 3

Откомпилируйте программы 06-3а.с и 06-3b.c их и запустите любую из них один раз для создания и инициализации разделяемой памяти. Затем запустите другую и, пока она находится в цикле, запустите (например, с другого виртуального терминала) снова первую программу. Объясните полученный результат.

(5 баллов)

Исходный текст программы:

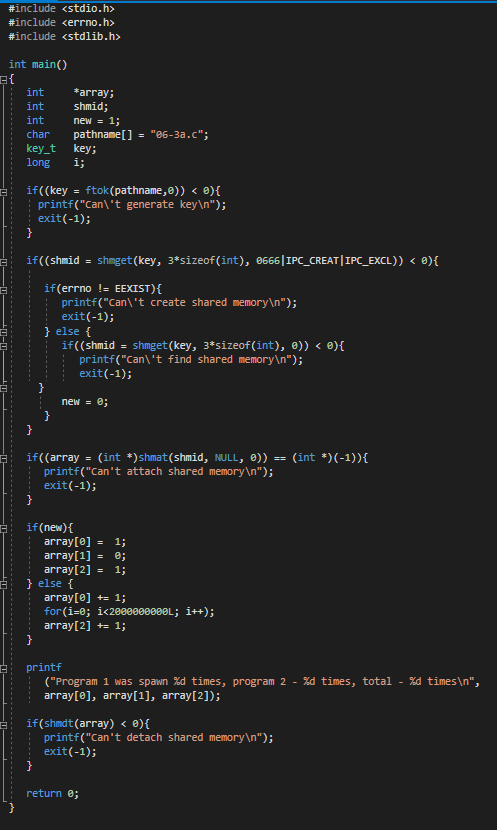


Рисунок . 06-3a.c

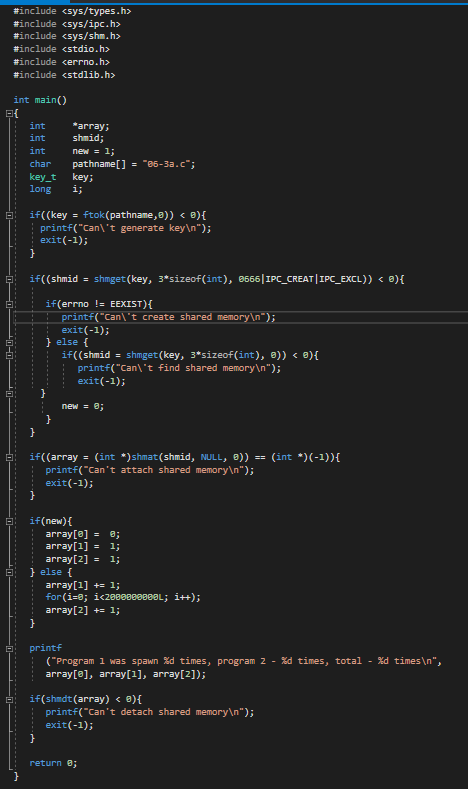


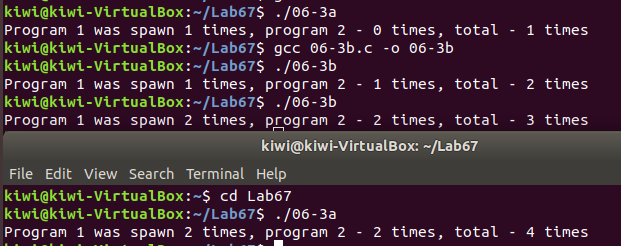
Рисунок . 03-b.c

Формат вызова:





Результат работы:



Объяснение: Количество запусков по-отдельности не соответствует количеству запусков вместе. Это связано с тем, что любые неатомарные операции, связанные с изменением содержимого разделяемой памяти, представляют собой критическую секцию процесса или нити исполнения. Для написания корректно работающих программ необходимо обеспечивать взаимоисключение при работе с разделяемой памятью и, может быть, взаимную очередность доступа к ней. Это можно сделать с помощью, например, алгоритма Петерсона или алгоритма булочной.

# Задание 4

**Задача повышенной сложности**: модифицируйте программы 06-3а.с и 06-3b.c для корректной работы с помощью алгоритма Петерсона. Считать, что инициализация разделяемой памяти осуществляется вне критической секции.

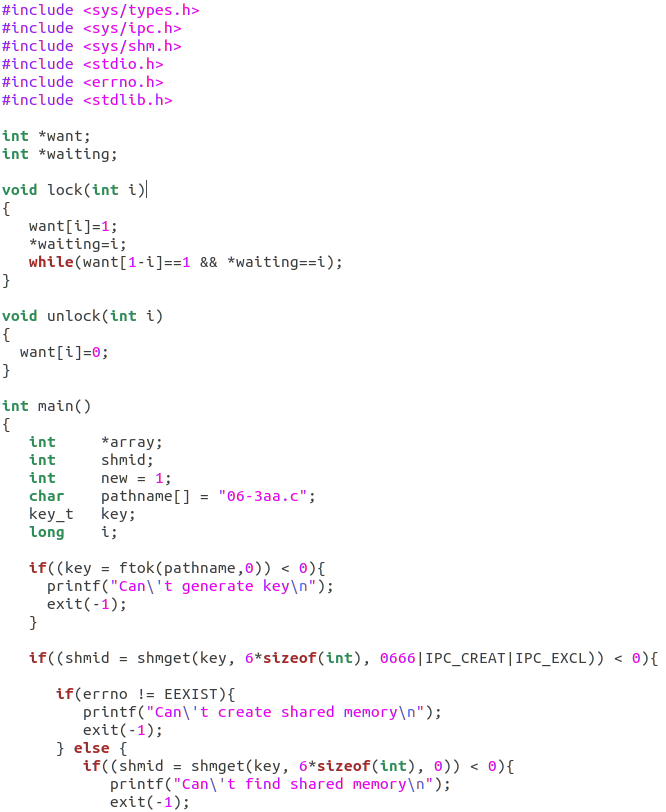
Приведите описание алгоритма Петерсона.

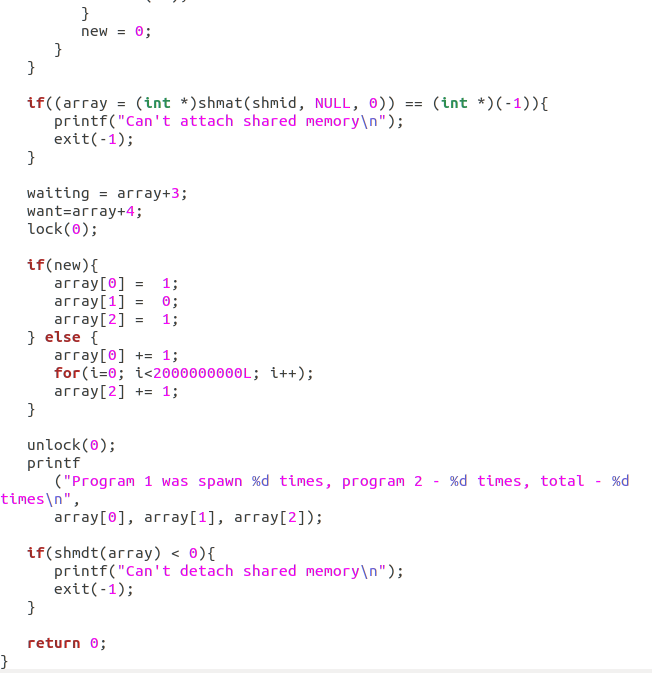
Новый для группы вариант 10 баллов.

Повторение уже предложенного варианта 6 баллов.

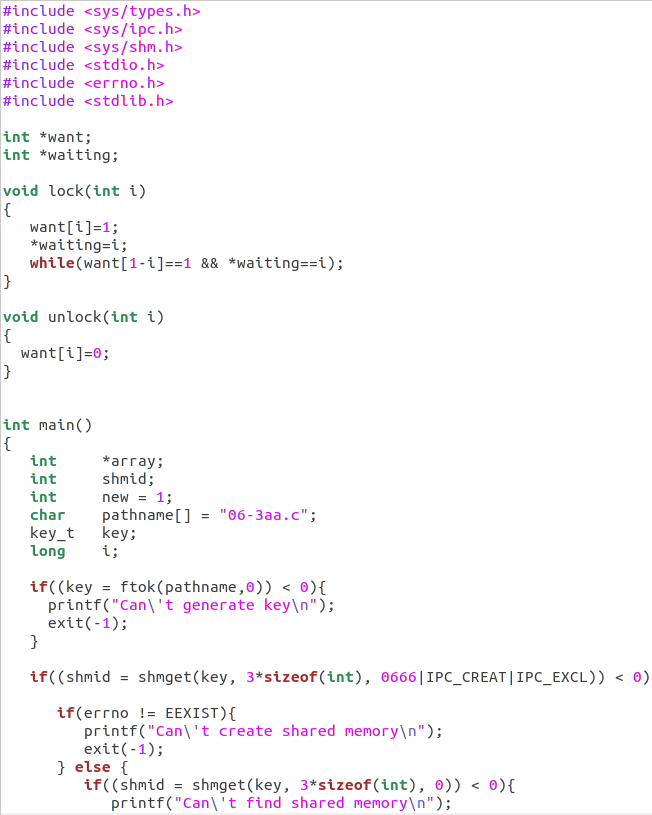
Исходный текст программы:

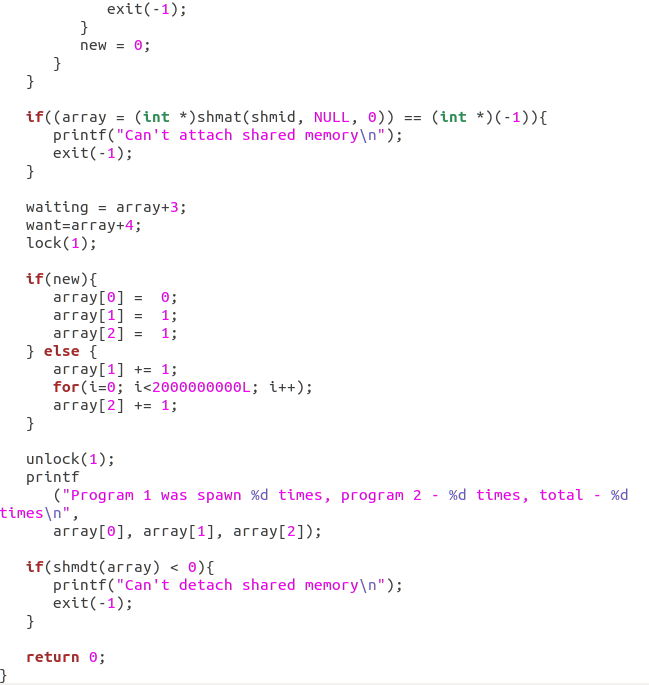
06-3.а:





06-3b.c



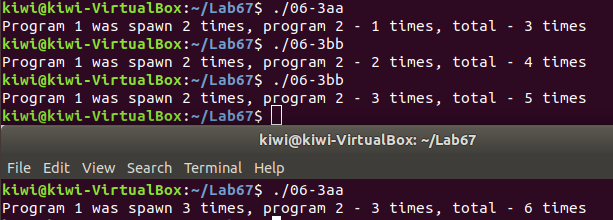


Формат вызова:





Результат работы:



Объяснение: Алгоритм Петерсона позволяет реализовать ожидание процессом своей очереди входа в критическую область, находящуюся между функциями lock(in i) или unlock(int i). Таким образом процесс 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 (т.е. доступ к критической области имеет одновременно только один процесс). Так как в процессе ожидания разрешения на вход один процесс не изменяет значения переменных, он сможет начать исполнение своего критического участка после не более чем одного прохода по критической секции другого процесса.

Описание алгоритма Петерсона:

Поток с номером 0\* вызывает lock, задавая этим индикатор своей «заинтересованности», устанавливая флаг очереди так, чтобы уступить очередь исполнения потоку номер 1\*\*. Поскольку последний пока еще не «заинтересован» в попадании в критическую область, выполнение сразу же возвращается из lock, и поток 0 входит в неё. Теперь lock вызывается потоком 1, для которого также выполняются описанные выше действия. Но так как поток 0 все еще «заинтересован» (want[0] == 1), выполнение остается в lock — поток 1 в ожидании. Как только поток 0 вызывает unlock и сбрасывает флаг своей «заинтересованности», поток 1 входит в критическую область и в конце сам вызывает unlock.

Потоки почти одновременно вызывают lock, устанавливая тем самым флаг своей «заинтересованности» и уступая очередь выполнения конкурирующему потоку посредством установки значения переменной waiting. Поскольку последним это делает поток 1, ему уже придется ждать в цикле, в то время как поток 0 беспрепятственно входит в критическую область кода. После того, как поток 0 выходит из критической области и сбрасывает флаг своей «заинтересованности», поток 1 продолжает своё исполнение и в конце сам сбрасывает соответствующий флаг вызовом unlock.

Потоки 0 и 1 никогда не могут попасть в критическую секцию в один момент времени: если 0 вошёл в секцию, он установил want[0] в 1 (true). Тогда либо want[1] == 0 (тогда поток 1 не в критической секции), либо waiting == 1 (тогда поток 1 пытается войти в критическую секцию и крутится в цикле), либо поток 1 пытается войти в критическую секцию после установки want[1] == true, но до установки waiting и цикла. Таким образом, если оба процесса находятся в критической секции, должно быть want[0]==1 && want[1]==1 && waiting == 0 && waiting == 1, но такого не может быть одновременно.

Возможна ситуация, когда один процесс будет несколько раз подряд захватывать себе критическую секцию, а другой, изъявивший желание попасть туда, будет ждать. В таком случае процесс не будет ждать дольше, чем один вход в критическую секцию: после выполнения unlock и повторного захода в lock процесс установит себя как ждущего и попадёт в цикл, который не завершится, пока не отработает другой процесс.

\*программа 06-3а  
\*\*программа 06-3b

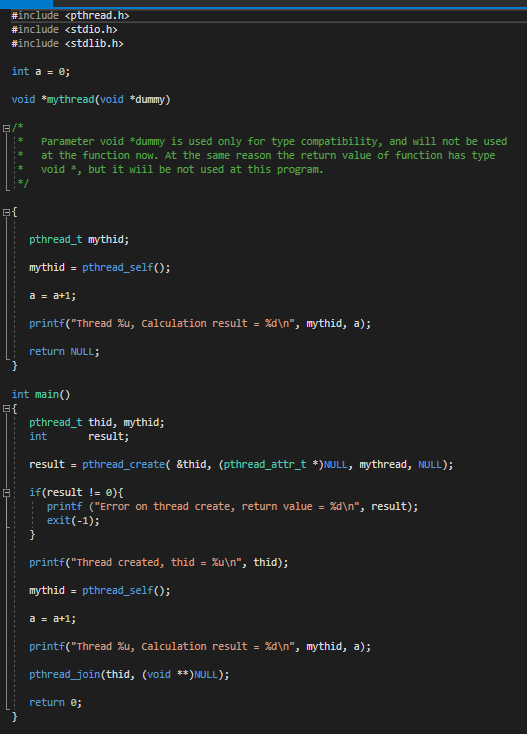
# Задание 5

Откомпилируйте программу 06-2.c для иллюстрации работы **двух нитей исполнения** и запустите её на исполнение.

Объясните полученный результат.

(5 баллов)

Исходный текст программы:

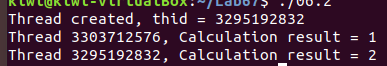


Формат вызова:





Результат работы:



Объяснение:

Программа печатает два разных значения, так как переменная a является разделяемой, и каждый thread прибавляет 1 к одной и той же переменной.

Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек, другими словами нити – это код, исполняющийся параллельно основному коду начиная с момента создания нити. Поскольку глобальные переменные у нитей исполнения являются общими, они могут использовать их как элементы разделяемой памяти.

Каждая нить исполнения, как и процесс, имеет в системе уникальный номер – идентификатор thread'a. Нить исполнения, создаваемую при рождении нового процесса, принято называть начальной или главной нитью исполнения этого процесса.

Нити исполнения могут порождать нити-потомки внутри своего процесса. Для создания новой нити исполнения (thread'а) внутри текущего процесса служит функция pthread\_create. Данная функция в случае ошибки возвращает положительное значение (а не отрицательное, как это происходит обычно).

Нить исполнения может быть завершена вызовом функции pthread\_exit, с помощью возврата из функции, ассоциированной с нитью исполнения или при возврате из функции main().

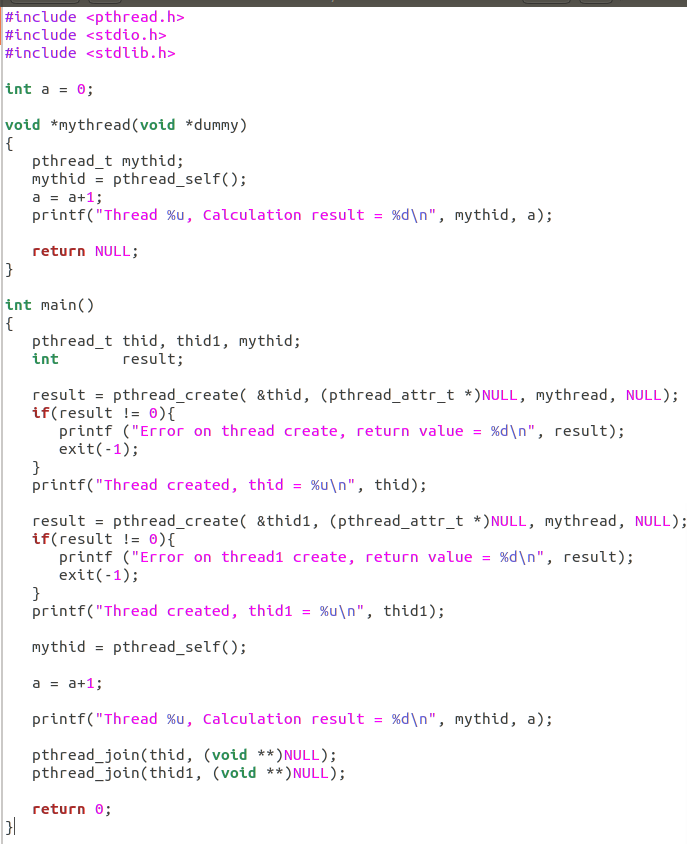
Функция pthread\_join блокирует работу вызвавшей ее нити исполнения до завершения thread'а с идентификатором thread. После разблокирования в указатель, расположенный по адресу status\_addr, заносится адрес, который вернул завершившийся thread либо при выходе из ассоциированной с ним функции, либо при выполнении функции pthread\_exit().

# Задание 6

Модифицируйте предыдущую программу (06-2.c), добавив к ней третью нить исполнения.

(3 балла)

Исходный текст программы:

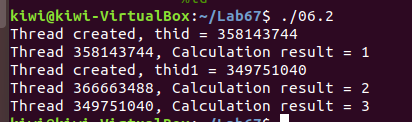


Формат вызова:





Результат работы:



Объяснение:

В процессе работы программы создаётся 3 thread’a: основной и 2 созданных нами.

Программа печатает три разных значения, так как переменная a является разделяемой, и каждый thread прибавляет 1 к одной и той же переменной.