Министерство образования Республики Беларусь

Учреждение образования:

“ Белорусский государственный университет информатики и радиоэлектроники”

Кафедра интеллектуальных информационных технологий

**Отчет по лабораторной работе №3**

На тему

Алгоритмы синхронизации процессов

**Работу выполнил:**

студент 3 курса

группы xxxxxx

Name Name

**Проверил:**

Name Name

Минск 2022

**Тема:**

Алгоритмы синхронизации процессов

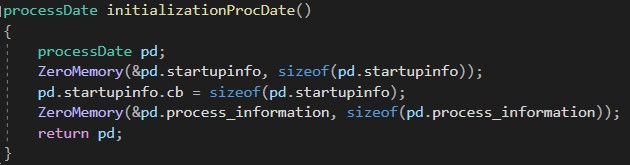
**Цель:**

Изучение алгоритмов синхронизации процессов

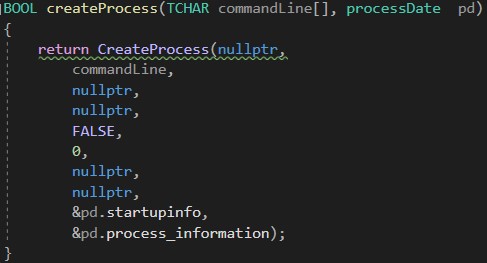
**Задание:**

Написать программу p, которая бы при запуске создавала два дочерних процесса p1 и p2. Программа p должна генерировать случайные числа и записывать их в буфер. Процессы p1 и p2 должны читать числа из буфера, p1 должен выводить числа на консоль, а p2 — в файл. Взаимодействие между процессами реализовать с помощью объектов ядра.

**Методы:**



Этот метод используется для инициализации процессов, чтобы в дальнейшем, это зарезервированное место заполненное нулями, с помощью функции ZeroMemory (первый аргумент это указатель на блок памяти, второй аргумент размер в байтах), на месте которых, будет создан процесс

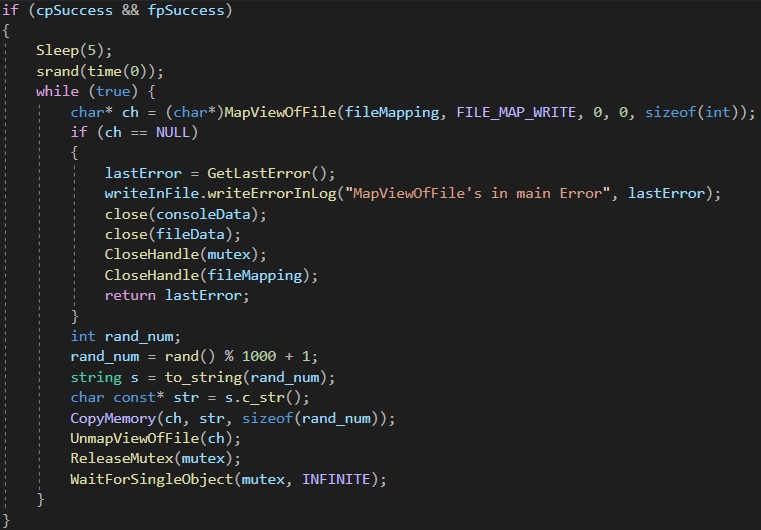


Этот метод используется как основная функция запуска процесса, в которой мы указываем имя исполняемого модуля и ранее инициализированные данные под процессы, такие как:

1. lpStartupInfo используется для настройки свойств процесса, например расположения окон и заголовок.

2. lpProcessInformation Структура PROCESS\_INFORMATION с информацией о процессе.

В случае успеха функция вернет TRUE, если нет, то FALSE.

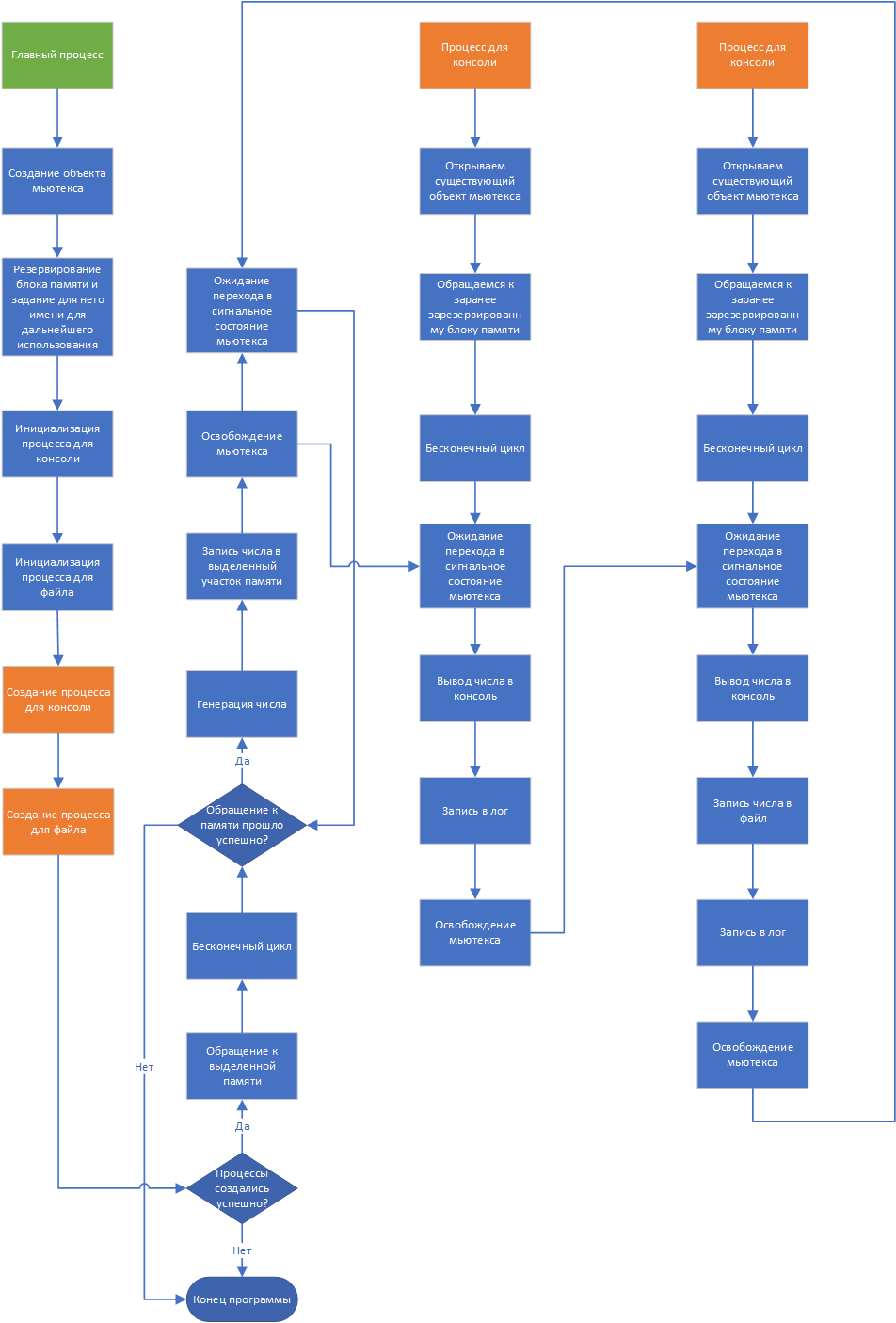


Главные строки, которые, в случае успешного создания двух процессов, один для вывода в консоль, а другой в файл, программа генерирует рандомные числа и записывает их в заранее зарезервированный блок памяти, размер которого является 4 байта. Затем обращаемся к этому блоку памяти с помощью функции MapViewOfFile, с аргументом записи и с помощью функции CopyMemory, записываем число в этот блок памяти.

После записи числа в блок памяти, освобождается заранее созданный мьютекс функцией ReleaseMutex. Для доступа к объекту мьютекс используется ожидающая функция WaitForSingleObject().

С помощью этих двух функций, дается доступ двум другим процессам (сначала процессу вывода в консоль, потому что она был создан раньше), к числам записанных в блок памяти.

**Блок схема**

****

**Выбор метода синхронизации**

Был выбран метод синхронизации мьютекс, так как это простые блокировки, полученные перед тем, как войти в критическую секцию и затем отпустить ее. Поскольку только один поток находится в своем критическом разделе в любой момент времени, условия гонки отсутствуют, а данные всегда остаются согласованными.

**Контрольные вопросы:**

1. Поясните, в чем состоит проблема синхронизации процессов/потоков

При наличии нескольких потоков управления, совместно использующих одни и те же данные, необходимо гарантировать, что все потоки будут видеть стабильное представление этих данных. Если каждый из потоков использует переменные, которые не модифицируются в других потоках, проблем не возникает. Аналогично, если переменная доступна одновременно нескольким потокам только для чтения, здесь также отсутствует проблема сохранения непротиворечивости.

Однако если один поток изменяет значение переменной, читать или изменять которое могут также другие потоки, необходимо синхронизировать доступ к переменной, чтобы гарантировать, что потоки не будут получать неверное значение переменной при одновременном доступе к ней. Когда поток изменяет значение переменной, существует потенциальная опасность, что другой поток прочитает еще не до конца записанное значение. На аппаратных платформах, где запись в память осуществляется более чем за один цикл, может произойти так, что между двумя циклами записи вклинится цикл чтения. Разумеется, такое поведение во многом зависит от аппаратной архитектуры, но при написании переносимых программ мы не можем полагаться на то, что они будут выполняться только на определенной платформе.

Приводится пример гипотетической ситуации, когда два потока одновременно выполняют запись и чтение значения одной и той же переменной. В данном примере поток A читает значение переменной и затем записывает в нее новое значение, но операция записи производится за два цикла. Если поток B прочитает значение этой же переменной между двумя циклами записи, он обнаружит переменную в несогласованном состоянии. Для решения этой проблемы потоки должны использовать блокировки, которые позволят только одному потоку работать с переменной в один момент времени.

1. Что такое “эффект гонок”?

Состояние гонки. Race condition. Другое название: гонки данных (data race). Ошибка программирования многозадачной системы, при которой работа системы зависит от того, в каком порядке выполняются части кода. Состояние гонки является классическим гейзенбагом. Состояние гонки возникает тогда, когда несколько потоков многопоточного приложения пытаются одновременно получить доступ к данным, причем хотя бы один поток выполняет запись. Состояния гонки могут давать непредсказуемые результаты, и зачастую их сложно выявить. Иногда последствия состояния гонки проявляются только через большой промежуток времени и в совсем другой части приложения. Кроме того, ошибки такого рода невероятно сложно воспроизвести повторно. Для предотвращения состояния гонки используются приемы синхронизации, позволяющие правильно упорядочить операции, выполняемые разными потоками. Приведем пример. Пусть, один поток выполняет над общей переменной x операцию x = x + 3, а второй поток - операцию x = x + 5. Данные операции для каждого потока фактически разбиваются на три отдельных подоперации: считать x из памяти, увеличить x, записать x в память. В зависимости от взаимного порядка выполнения потоками под операций финальное значение переменной x может быть больше исходного на 3, 5 или 8.

Основными последствиями могут быть:

● утечки памяти[2],

● ошибки сегментирования[2],

● порча данных[2],

● уязвимости[2],

● взаимные блокировки,

● утечки других ресурсов, например файловых дескрипторов.

1. Что такое клинч?

Клинч, дедлок (deadlock), смертельные объятия - разные названия одной из самых серьезных проблем, возникающих при параллельном программировании. Клинч может возникнуть в ситуации, когда два или более параллельно выполняемых потока конкурируют за обладание двумя или более общими ресурсами. При клинче каждый из потоков успевает захватить один из общих ресурсов. Для окончания работы каждому потоку необходимы другие ресурсы, захваченные другими потоками. В результате, никто из потоков не может завершить свою работу, все стоят в очередях, которые не двигаются, - работа замирает - приложение "зависает". Это худшее, что может случиться с приложением.

Плохо, когда приложение завершается в результате возникновения исключительной ситуации, для которой не предусмотрена нормальная обработка. Еще хуже, когда приложение "нормально" завершается, но выдает некорректные результаты. Совсем плохо, когда приложение "зависает", не выдает результатов и завершить его приходится "грубым" способом с помощью внешнего прерывания (например, с помощью диспетчера задач).

Поясним ситуацию клинча на примере двух потоков P1 и P2 и двух ресурсов R1 и R2. Пусть поток P1, входя в критическую секцию, захватывает ресурс R1, блокируя его для использования потоком P2. Аналогично, работая параллельно, поток P2, входя в критическую секцию, захватывает ресурс R2, блокируя его для использования потоком P1. Потоку P1 в какой-то момент работы в критической секции становится необходимым ресурс R2, но этот ресурс заблокирован и поток становится в очередь, прерывая свое выполнение. Симметричная ситуация возникает с потоком P2. Как два боксера, войдя в клинч, не могут разойтись, без вмешательства судьи, так и потоки не смогут продолжить выполнение без внешнего вмешательства.

1. Что такое взаимоисключение?

Мы можем защитить данные и ограничить доступ к ним одним потоком в один момент времени с помощью взаимоисключений (mutual-exclusion) интерфейса pthreads. Мьютекс (mutex) — это фактически блокировка, которая устанавливается (запирается) перед обращением к разделяемому ресурсу и снимается (отпирается) после выполнения требуемой последовательности операций. Если мьютекс заперт, любой другой поток, который попытается запереть его, будет заблокирован, пока мьютекс не будет отперт. Если в момент отпирания мьютекса сразу несколько потоков будут находиться в заблокированном состоянии, все они будут запущены, и первый, кто успеет запереть мьютекс, продолжит работу. Все остальные потоки обнаружат запертый мьютекс и опять перейдут в режим ожидания. Таким образом, доступ к ресурсу сможет получить одновременно только один поток. Такой механизм взаимоисключений будет корректно работать только при условии, что все потоки приложения соблюдают одни и те же правила доступа к данным. Операционная система никак не упорядочивает доступ к данным. Если мы позволим одному потоку производить действия с общими данными, предварительно не ограничив доступ к ним, остальные потоки могут обнаружить эти данные в противоречивом состоянии, даже если перед обращением к ним будут устанавливать блокировку.

1. Что такое критическая секция с точки зрения теории операционных систем?

Важным понятием при изучении способов синхронизации процессов является понятие критической секции (critical section) программы. Критическая секция - это часть программы, выполнение которой может привести к возникновению race condition. 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. 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.

1. Что такое объект ядра? Какие объекты ядра операционной системы windows или Linux вы знаете?

Ядро — центральная часть операционной системы, управляющая выполнением процессов, ресурсами вычислительной системы и предоставляющая процессам координированный доступ к этим ресурсам.

Как основополагающий элемент ОС, ядро представляет собой наиболее низкий уровень абстракции для доступа приложений к ресурсам вычислительной системы, необходимым для их работы.

К объектам ядра ОС относятся:

● Процессы

● Потоки

● Файлы

● События

● Семафоры

● Мьютексы

● Каналы

● Файлы, проецируемые в память

Каждый объект ядра — это блок памяти, выделенный ядром и доступный только ему. В блоке размещается структура данных, поля которой содержат информацию об объекте. Некоторые поля (например, дескриптор защиты и счётчик количества пользователей) присутствуют во всех объектах, но большая их часть специфична для объектов конкретного типа. Приложение не имеет прямого доступа к структурам данных, представляющим собой объекты ядра. Оперировать объектами ядра приложение может только через специальные функции Windows.

Когда вы вызываете функцию, создающую объект ядра (например, CreateFile), она возвращает дескриптор созданного объекта. Этот дескриптор может быть использован любым потоком вашего процесса. Для большей надежности операционной системы Microsoft сделано так, что значения дескрипторов объектов ядра действительны только в адресном пространстве процесса, их создавшего. Поэтому все попытки передачи такого дескриптора другому процессу и его использования в другом процессе приводят к ошибкам.

Объекты принадлежат ядру, а не процессу. Другими словами, если процесс создает какой-либо объект ядра, а затем завершает свою работу, то объект ядра может быть и не разрушен. В большинстве случаев объект все же разрушается, но если этим объектом пользуется другой процесс, то ядро не позволит разрушить объект до тех пор, пока второй процесс не откажется от него.

Ядру известно, сколько процессов использует конкретный объект ядра, поскольку в каждом объекте есть счётчик пользователей. В момент создания объекта ядра счётчику присваивается единичное значение. Когда к этому объекту обращается другой процесс, значение счётчика увеличивается на единицу. А когда какой-то процесс завершается, счётчики всех используемых им объектов ядра уменьшаются на единицу. Как только счётчик пользователей объекта обнуляется, ядро уничтожает этот объект.

1. Какие есть подходы к предотвращению, последствий тупиков?

Поток может попасть в тупиковую ситуацию (deadlock), если попытается дважды захватить один и тот же мьютекс, но есть и менее очевидные способы. Например, тупиковая ситуация может возникнуть, когда в программе используется несколько мьютексов и один поток, удерживая первый мьютекс, пытается запереть второй, в то время как другой поток аналогично удерживает второй мьютекс и пытается запереть первый. В результате ни один из потоков не сможет продолжить работу, поскольку каждый будет ждать освобождения ресурса, захваченного другим потоком. Возникнет тупиковая ситуация. Тупиковых ситуаций можно избежать, жестко определив порядок захвата ресурсов. Приведем пример. Предположим, что есть два мьютекса, A и B, которые необходимо запереть одновременно. Если все потоки сначала будут запирать мьютекс A, а потом B, тупиковой ситуации с этими мьютексами никогда не возникнет. Аналогично, если все потоки сначала будут запирать мьютекс B, а потом A, тупиковой ситуации с этими мьютексами также никогда не возникнет. Опасность попадания в тупиковую ситуацию возникает, только когда разные потоки могут попытаться запереть мьютексы в разном порядке. Иногда архитектура приложения не позволяет заранее предопределить порядок захвата мьютексов. Если программа использует достаточно много мьютексов и структур данных, а доступные функции, которые работают с ними, не укладываются в достаточно простую иерархию, придется попробовать иной подход. Например, при невозможности запереть мьютекс можно отпереть захваченные мьютексы и повторить попытку немного позже. В этом случае во избежание блокировки потока можно использовать функцию pthread\_mutex\_trylock. Если мьютекс удалось запереть с помощью pthread\_mutex\_trylock, можно продолжить работу. Если мьютекс запереть не удалось, можно отпереть уже захваченные мьютексы, освободить занятые ресурсы и повторить попытку немного позже.

1. Как можно диагностировать наличие в системе большой очереди процессов или тупика? В чем различие этих ситуаций?

Тупиковые ситуации надо отличать от простых очередей, хотя те и другие возникают при совместном использовании ресурсов и внешне выглядят похоже: поток приостанавлива­ется и ждет освобождения ресурса. Однако очередь — это нормальное явление, неотъемле­мый признак высокого коэффициента использования ресурсов при случайном поступле­нии запросов. Очередь появляется тогда, когда ресурс недоступен в данный момент, но освободится через некоторое время, позволив потоку продолжить выполнение. Тупик же, что видно из его названия, является в некотором роде неразрешимой ситуацией. Необхо­димым условием возникновения тупика является потребность потока сразу в нескольких ресурсах.

Невозможность потоков завершить начатую работу из-за возникновения вза­имных блокировок снижает производительность вычислительной системы. По­этому проблеме предотвращения тупиков уделяется большое внимание. На тот случай, когда взаимная блокировка все же возникает, система должна предоста­вить администратору-оператору средства, с помощью которых он смог бы распо­знать тупик, отличить его от обычной блокировки из-за временной недоступности ресурсов. И наконец, если тупик диагностирован, то нужны средства для снятия взаимных блокировок и восстановления нормального вычислительного процесса.

1. Чем объект синхронизации критическая секция отличается от объекта синхронизации семафор в операционной системе windows.

Критические секции:

Объект-критическая секция помогает программисту выделить участок кода, где нить (thread) 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.

Существует также функция TryEnterCriticalSection, которая проверяет, занята ли критическая секция в данный момент. С ее помощью нить в процессе ожидания доступа к ресурсу может не блокироваться, а выполнять какие-то полезные действия.

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).

Функция 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.

1. Что такое файлы, проецируемые в память? Какие базовые механизмы операционных систем реализованы через данный подход?

Проецируемые в память файлы позволяют работать с файлами на диске так, как будто бы они целиком располагаются в виртуальной памяти. Для поддержки этого механизма операционная система используя менеджер виртуальной памяти и драйвер файловой системы проецирует файл на адресное пространство процесса.

Самый низкоуровневый механизм совместного использования данных в одной системе — проецирование файла в память. На нём так или иначе базируются все другие механизмы разделения данных. Поэтому, если необходимо получить максимальное быстродействие с минимумом издержек, лучше всего применять именно проецирование.

Совместное использование данных в этом случае происходит по следующей схеме. Несколько процессов проецируют в память представления одного и того же объекта «проекция файла», то есть делят одни и те же страницы физической памяти. Поэтому, когда один процесс записывает данные в представление общего объекта «проекция файла», эти изменения немедленно отражаются в других процессах. Но для этого все процессы должны использовать одинаковое имя объекта «проекция файла».

Проецирование файла данных в адресное пространство процесса предоставляет разработчику мощный механизм работы с файлами. Спроецировав файл на адресное пространство процесса, программа получает возможность работать с ним, как с массивом. Это очень удобно при манипуляциях с большими потоками данных.

Как только файл спроецирован в память, к нему можно обращаться так, будто он целиком в нее загружен.

**Применение:**

● загрузка и выполнение EXE- и DLL-файлов. Это позволяет существенно экономить как на размере страничного файла, так и на времени, необходимом для подготовки приложения к выполнению,

● доступ к файлу данных, размещенному на диске. Это позволяет обойтись без операций файлового ввода-вывода и буферизации его содержимого,

● разделение данных между несколькими процессами, выполняемыми на одной машине (В Windows есть и другие методы для совместного доступа разных процессов к одним данным — но все они так или иначе реализованы на основе проецируемых в память файлов.)