Лабораторная работа №3
Системные средства синхронизации в ОС GNU/LINUX

Цель работы:

- изучение механизмов синхронизации с использованием сигналов, семафоров, мьютексов и барьеров.
- приобретение практических навыков применения средств синхронизации в многопоточных приложениях

Операционные системы предоставляют средства для синхронизации работы одновременно выполняющихся потоков (одного или разных процессов), которые используются при разработке многопоточных и многопроцессных приложений. В многозадачных системах в условиях параллельного выполнения потоков возникает необходимость координации их действий для избежания таких проблем в ходе вычислительного процесса, как гонки данных, взаимные блокировки, некорректное состояние ресурсов, а также для реализации дополнительных требований при планировании.

Основные назначения средств синхронизации

• **обеспечение целостности данных:** правильное применение средств синхронизации гарантирует, что только один поток будет изменять данные в определённый момент времени, что необходимо для избежания конфликтов и повреждения данных

• **управление доступом к ресурсам:** обеспечивают контроль доступа к общим ресурсам (файлам, устройствам, памяти), предотвращая ситуации, когда несколько потоков одновременно пытаются использовать один и тот же ресурс

• **координация работы потоков**: средства синхронизации позволяют определить порядок выполнения потоков, приостанавливать работу потока до наступления какого-либо события, что необходимо для корректного выполнения программ.

Основные виды средств синхронизации

Мьютексы (Mutex)

Используются для обеспечения взаимного исключения, позволяя только одному потоку выполнять критическую секцию кода в каждый момент времени.

Семафоры (Semaphore)

Позволяют ограничить количество потоков, которые могут одновременно получить доступ к ресурсу. Различают бинарные и счётные семафоры.

Условные переменные (Conditional Variable)

Переводит поток в состояние ожидания до тех пор, пока не выполнится условие. Условная переменная имеет атрибуты, которые определяют характеристики условия. Обычно используются следующие объекты: булева переменная, указывающая, выполняется ли условие, мьютекс для сериализации доступа к булевой переменной, условная переменная для ожидания условия

Барьеры (Barrier)

Используются для приостановки потоков в определённой точке выполнения, требуя, чтобы все потоки достигли этой точки, прежде чем продолжить выполнение.

Блокировки чтения/записи (Read-Write Lock)

Позволяют одновременно нескольким потокам читать данные, но ограничивают запись, что увеличивает производительность в сценариях, где чтение происходит чаще, чем запись.

Механизмы синхронизации помогают разработчикам создавать устойчивые и эффективные многопоточные приложения, обеспечивая корректное взаимодействие между потоками и предотвращая потенциальные ошибки, связанные с конкурентным доступом к данным.

Семафоры

Семафор – переменная определенного типа, которая доступна параллельным процессам для проведения над ней только двух операций:

- ***A(S, n)***– увеличить значение семафора ***S*** на величину ***n***;
- ***D(S, n)***– если значение семафора ***S*** < ***n***, процесс блокируется. В противном случае ***S = S - n***;
- ***Z(S)***– процесс блокируется до тех пор, пока значение семафора ***S*** не станет равным 0.

Семафор играет роль вспомогательного критического ресурса, так как операции ***A*** и ***D*** неделимы при своем выполнении и взаимно исключают друг друга. Семафорный механизм работает по схеме, в которой сначала исследуетсясостояние критического ресурса, а затем уже осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. Основным достоинством семафорных операций является отсутствие состояния «активного ожидания», что может существенно повысить эффективность работы мультипрограммной вычислительной системы.

Для работы с семафорами имеются следующие системные вызовы:
Создание и получение доступа к набору семафоров:

int semget(key_t key, int nsems, int semflg);

Параметр ***ke***y является ключом для массива семафоров, т.е. фактически его именем. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции ***ftok()***, или специальное значение ***IPC_PRIVATE***. Использование значения ***IPC_PRIVATE*** всегда приводит к попытке создания нового массива семафоров с ключом, который не совпадает созначением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции ***ftok()*** ни при одной комбинации ее параметров. Параметр ***nsems*** определяет количество семафоров в создаваемом или уже существующем массиве. В случае если массив с указанным ключом уже имеется,но его размер не совпадает с указанным в параметре ***nsems***, констатируется возникновение ошибки.

Параметр ***semfl g***– флаги – играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву,а также необходимость создания нового массива и поведение системного вызовапри попытке создания. Он является некоторой комбинацией (с помощью операции

побитовое или – "|") следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT— если массива для указанного ключа не существует, он должен быть создан;

IPC_EXCL— применяется совместно с флагом ***IPC_CREAT***. При совместном ихиспользовании и существовании массива с указанным ключом, доступ к массиву не производится и констатируется ошибка, при этом переменная ***errno***, описанная в файле <***errno.h***>, примет значение ***EEXIST***;

0400— разрешено чтение для пользователя, создавшего массив
0200— разрешена запись для пользователя, создавшего массив
0040— разрешено чтение для группы пользователя, создавшего массив
0020— разрешена запись для группы пользователя, создавшего массив
0004— разрешено чтение для всех остальных пользователей
0002— разрешена запись для всех остальных пользователей
Пример: ***semflg= IPC_CREAT | 0022***

Изменение значений семафоров:

int semop(int semid, struct sembuf *sops, int nsops);

Параметр ***semid*** является дескриптором System V IPC для набора семафоров, т. е. значением, которое вернул системный вызов ***semget()*** при создании набора семафоров или при его поиске по ключу. Каждый из ***nsops*** элементов массива, на который указывает параметр ***sops***, определяет операцию, которая должна быть совершена над каким-либо семафором из массива IPC семафоров, и имеет тип структуры:

struct sembuf {

short sem_num;//номер семафора в массиве IPC семафоров (начиная с***0***);
short sem_op; //выполняемая операция;
short sem_flg; // флаги для выполнения операции.
}

Значение элемента структуры ***sem_op*** определяется следующим образом:

- для выполнения операции **A(S,n)** значение должно быть равно ***n***;
- для выполнения операции ***D(S,n)***значение должно быть равно ***-n***;
- для выполнения операции ***Z(S)*** значение должно быть равно ***0***.

Семантика системного вызова подразумевает, что все операции будут в реальности выполнены над семафорами только перед успешным возвращением из системного вызова. Если при выполнении операций ***D*** или ***Z***процесс перешел в состояние ожидания, то он может быть выведен из этого состояния при возникновении следующих форсмажорных ситуаций: массив семафоров былудален из системы; процесс получил сигнал, который должен быть обработан.

Выполнение разнообразных управляющих операций (включая удаление) над набором семафоров:

int semctl(int semid, int semnum, int cmd, union semun arg);

Изначально все семафоры инициируются нулевым значением.

Мьютексы

Мьютекс – экземпляр типа ***pthread_mutex_t*** – по сути является бинарным семафором, т.е. позволяет разделять доступ к ресурсу нескольким потокам. Перед использованием необходимо инициализировать мьютекс функцией ***pthread_mutex_init:***

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

где первый аргумент – указатель на мьютекс, а второй – атрибуты мьютекса. Если указан NULL, то используются атрибуты по умолчанию. В случае удачной инициализации мьютекс переходит в состояние «инициализированный и свободный», а функция возвращает 0. Повторная инициализация инициализированного мьютекса приводит к неопределённому поведению.

Если мьютекс создан статически и не имеет дополнительных параметров, тоон может быть инициализирован с помощью макроса **PTHREAD_MUTEX_INITIALIZER.**

Типичные ошибки, которые могут возникнуть:

- ***EAGAIN*** — недостаточно необходимых ресурсов (кроме памяти) для инициализации мьютекса
- ***ENOMEM*** — недостаточно памяти
- ***EPERM*** — нет прав для выполнения операции
- ***EBUSY*** — попытка инициализировать мьютекс, который уже был инициализирован, но не унечтожен
- ***EINVAL*** — значение ***mutexattr*** невалидно

После использования мьютекса его необходимо уничтожить с помощью функции

int pthread_mutex_destroy(pthread_mutex_t*mutex);

В результате функция возвращает 0 в случае успеха или код ошибки.

После создания мьютекса он может быть захвачен с помощью функций

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);

Функция ***pthread_mutex_lock(),*** если ***mutex*** еще не занят, занимает его, поток становится его обладателем и сразу же выходит. Если мьютекс занят, то блокируется дальнейшее выполнение процесса до освобождения мьютекса.

Функция ***pthread_mutex_trylock()*** идентична по поведению функции ***pthread_mutex_lock(),*** с одним исключением — она не блокирует процесс, если ***mutex з***анят, а возвращает ***EBUSY*** код.

После этого участок кода становится недоступным остальным потокам – их выполнение блокируется до тех пор, пока мьютекс не будет освобождён.

Освобождение должен провести поток, заблокировавший мьютекс, вызовом

int pthread_mutex_unlock(pthread_mutex_t *mutex);

Коды возврата для ***pthread_mutex_lock()****:*

- ***EINVAL*** — мьютекс неправильно инициализирован
- ***EDEADLK*** — мьютекс уже занят текущим процессом.

Коды возврата для ***pthread_mutex_trylock():***

- ***EBUSY*** — мьютекс уже занят
- ***EINVAL*** — мьютекс неправильно инициализирован

Коды возврата для ***pthread_mutex_unlock():***

- ***EINVAL*** — мьютекс неправильно инициализирован
- ***EPERM*** — вызывающий процесс не является обладателем мьютекса.

Барьеры

При инициализации барьера задается какое количество потоков он должен остановить, соответственно, потоки, преодолевающие барьер (дошедшиедо него), не могут продолжать свое выполнение пока на этот барьер не наткнетсяуказанное при инициализации количество потоков. Барьеры предназначены для того чтобы держать потоки - члены определенной группы вместе. Наткнувшись на барьер, поток из группы не сможет продолжать выполнение, пока все члены группы не дойдут до барьера. Обычно барьеры используют для того, чтобы все потоки, выполняющие какой-то параллельный алгоритм, достигли определеннойего точки прежде чем какая-либо из них двинулась дальше.

Стандарт POSIX поддерживает барьеры в качестве опции. В программе они представляются при помощи непрозрачного типа ***pthread_barrier_t***, объекты которого можно инициализировать при помощи вызова функции:

int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t * attr, unsigned count);

где последний параметр ***count*** задает число нитей, которые должны дойти до барьера (вызвать функцию ***pthread_barrier_wait***) прежде чем они смогут продолжить выполнение.

Функция, которая уничтожает барьер:

int pthread_barrier_destroy(pthread_barrier_t *barrier);

Как и в случае остальных примитивов синхронизации данная функция должна вызываться только в случае, если на уничтожаемом барьере не ожидает ни одной нити.

Функция, вызов которой соответствует преодолению барьера:

int pthread_barrier_wait(pthread_barrier_t *barrier);

Поток, вызывавший функцию ***pthread_barrier_wait(),*** блокируется в месте вызова до тех пор, пока количество потоков, вызвавших ***pthread_barrier_wait()*** не станет равным значению ***count,*** которое было задано при инициализации этого барьера. Функция возвращает значение ***PTHREAD_BARRIER_SERIAL_THREAD*** для произвольного синхронизированного потока, и 0 для всех остальных. Функция возврат ***EINVAL,*** если переданный аргумент не ссылается на инициализированный барьер.

Если функция ***pthread_barrier_wait()*** вернула значение ***PTHREAD_BARRIER_SERIAL_THREAD***, барьер может быть уничтожен. То есть,

нельзя просто так уничтожать барьер в каком-то из потоков, так как порядок завершения потоков не детерминирован.

После возврата всех нитей из ***pthread_barrier_wait()*** барьер приходит в исходное состояние, то есть оказывается снова готов к использованию.

Условные переменные

Условная переменная в ***Pthreads*** – это переменная типа ***pthread_cond_t***, которая обеспечивает блокирование одного или нескольких потоков до тех пор, пока не поступит сигнал, или не пройдёт максимально установленное время ожидания. Условная переменная используется совместно с ассоциированным с ней мьютексом.

Для создания условной переменной используется функция

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

где ***cond –*** указатель на переменную типа ***pthread_cond_t,***

attr – аттрибуты условной переменной.

Коды возврата ***int pthread_cond_init():***

- ***EAGAIN*** – у системы нет ресурсов
- ***ENOMEM –*** у системы нет памяти
- ***EBUSY –*** попытка инициализации уже инициализированной условной переменной
- ***EINVAL –*** некорректные параметры ***attr.***

Для уничтожения условной переменной используется функция

int pthread_cond_destroy(pthread_cond_t *cond);

Коды возврата ***pthread_cond_destroy():***

- ***EBUSY –*** попытка уничтожить используемый ресурс
- ***EINVAL –*** плохой аргумент функции

Для ожидания сигнала можно использовать функцию

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

которая ждёт неопределённо долго, или функцию

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);

которая ждёт до момента ***abstime*** (это абсолютное время, а не относительное).

Для того, чтобы послать сигнал, используются две функции

int pthread_cond_signal(pthread_cond_t *cond);

- разблокирует как минимум один поток, заблокированный переменной ***cond,*** и

int pthread_cond_broadcast(pthread_cond_t *cond);

которая разблокирует все потоки, заблокированные с помощью условной переменной ***cond.***

Обе функции могут возвратить ошибочное значение ***EINVAL,*** если аргумент функции не указывает на инициализированную переменную типа ***pthread_cond_t.***

Использование сигналов в ОС GNU/LINUX

Сигналы не могут непосредственно переносить информацию, что ограничивает их применимость в качестве общего механизма межпроцессного взаимодействия. Каждому типу сигналов присвоено мнемоническое имя (например, ***SIGINT***), которое указывает, для чего обычно используется сигнал этого типа. Имена

сигналов определены в стандартном заголовочном файле ***<signal.h>*** при помощи директивы препроцессора ***#define***. Этим именам соответствуют небольшие положительные целые числа. С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прерывает исполнение, иуправление передается функции-обработчику сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Типы сигналов принято задавать специальными символьными константами. Системный вызов ***kill()*** предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

#include
<sys/types.h> ***#include***
<signal.h>

int kill(pid_t pid, int signal);

Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал. Аргумент ***pid*** указывает процесс, которому посылается сигнал, а аргумент ***sig*** – какой сигнал посылается. В зависимости от значения аргументов:

pid > 0 сигнал посылается процессу с идентификатором ***pid***;

pid=0 сигнал посылается всем процессам в группе, к которой принадлежит посылающий процесс;

pid=-1 и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователяпроцесса, посылающего сигнал.

pid = -1 и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов ***с pid = 0*** и ***pid = 1***).

pid < 0, но не ***–1***, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента ***pid*** (если позволяют привилегии).

Если ***sig = 0***, то производится проверка на ошибку, а сигнал не посылается. Это можно использовать для проверки правильности аргумента ***pid*** (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Системные вызовы для установки собственного обработчика сигналов:

#include <signal.h>
void (*signal (int sig, void (*handler) (int)))(int);
int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);

Структура ***sigaction*** имеет следующий формат:

```
struct sigaction {
void (*sa_handler)(int);
void (*sa_sigaction)(int, siginfo_t *, void *); sigset_t sa_mask;
int sa_flags;
 void (*sa_restorer)(void);
```

Системный вызов ***signal*** служит для изменения реакции процесса на какой-либо сигнал. Параметр ***sig*** – это номер сигнала, обработку которого предстоит изменить. Параметр ***handler*** описывает новый способ обработки сигнала – это может быть указатель на пользовательскую функцию-обработчик сигнала, специальное значение ***SIG_DFL*** (восстановить реакцию процесса на сигнал ***sig*** по умолчанию) или специальное значение ***SIG_IGN*** (игнорировать поступивший сигнал ***sig***). Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Пример пользовательской обработки сигнала ***SIGUSR:***

void *my_handler(int nsig) { код функции-обработчика сигнала ***}***

int main() {

(void) signal(SIGUSR1, my_handler); }

Системный вызов ***sigaction*** используется для изменения действий процессапри получении соответствующего сигнала. Параметр ***sig*** задает номер сигнала и может быть равен любому номеру. Если параметр ***act*** не равен нулю, то новое действие, связянное с сигналом ***sig***, устанавливается соответственно ***act***. Если ***oldact*** не равен нулю, то предыдущее действие записывается в ***oldact***.

Большинство типов сигналов ***UNIX*** предназначены для использования ядром, хотя есть несколько сигналов, которые посылаются от процесса к процессу:

SIGALRM – сигнал таймера (***alarm clock***). Посылается процессу ядром при срабатывании таймера. Каждый процесс может устанавливать не менее трех таймеров. Первый из них измеряет прошедшее реальное время. Этот таймер устанавливается самим процессом при помощи системного вызова ***alarm()***;

SIGCHLD – сигнал останова или завершения дочернего процесса (***child process terminated or stopped***). Если дочерний процесс останавливается или завершается, то ядро сообщит об этом родительскому процессу, послав ему данный сигнал. По умолчанию родительский процесс игнорирует этот сигнал, поэтому, если в родительском процессе необходимо получать сведения о завершении дочерних процессов, то нужно перехватывать этот сигнал;

SIGHUP – сигнал освобождения линии (***hangup signal***). Посылается ядром всем процессам, подключенным к управляющему терминалу (***control terminal***) при отключении терминала. Он также посылается всем членам сеанса, еслизавершает работу лидер сеанса (обычно процесс командного интерпретатора), связанного с управляющим терминалом;

SIGINT – сигнал прерывания программы (***interrupt***). Посылается ядром всем процессам сеанса, связанного с терминалом, когда пользователь нажимает клавишу прерывания. Это также обычный способ остановки выполняющейся программы;

SIGKILL – сигнал уничтожения процесса (***kill***). Это довольно специфический сигнал, который посылается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнал процесса;

SIGPIPE – сигнал, посылаемый процессу при попытке записи в соединение (канал или сокет) при отсутствии или обрыве соединения с читающей стороной;

SIGPOLL – сигнал о возникновении одного из опрашиваемых событий (***pollable event***). Этот сигнал генерируется ядром, когда некоторый открытый дескриптор файла становится готовым для ввода или вывода;

SIGPROF – сигнал профилирующего таймера (***profiling time expired***). Как было упомянуто для сигнала ***SIGALRM***, любой процесс может установить не менее трех таймеров. Второй из этих таймеров может использоваться для измерения времени выполнения процесса в пользовательском и системном режимах. Этот сигнал генерируется, когда истекает время, установленное в этомтаймере, и поэтому может быть использован средством профилирования программы;

SIGQUIT – сигнал о выходе (***quit***). Очень похожий на сигнал ***SIGINT***, этот сигнал посылается ядром, когда пользователь нажимает клавишу выхода используемого терминала. В отличие от ***SIGINT***, этот сигнал приводит к аварийному завершению и сбросу образа памяти;

SIGSTOP – сигнал останова (***stop executing***). Это сигнал управления заданиями, который останавливает процесс. Его, как и сигнал ***SIGKILL***, нельзя проигнорировать или перехватить;

SIGTERM – программный сигнал завершения (***software termination signal***). Программист может использовать этот сигнал для того, чтобы дать процессу время для «наведения порядка», прежде чем посылать ему сигнал ***SIGKILL***;

SIGTRAP – сигнал трассировочного прерывания (***trace trap***). Это особый сигнал, который в сочетании с системным вызовом ptrace используется отладчиками, такими как ***sdb, adb, gdb***;

SIGTSTP – терминальный сигнал остановки (***terminal stop signal***). Он формируется при нажатии специальной клавиши останова;

SIGTTIN – сигнал о попытке ввода с терминала фоновым процессом (***background process attempting read***). Если процесс выполняется в фоновом режиме и пытается выполнить чтение с управляющего терминала, то ему посылается этот сигнал. Действие сигнала по умолчанию – остановка процесса;

SIGTTOU – сигнал о попытке вывода на терминал фоновым процессом (***background process attempting write***). Аналогичен сигналу ***SIGTTIN***, но генерируется, если фоновый процесс пытается выполнить запись в управляющий терминал. Действие сигнала по умолчанию – остановка процесса;

SIGURG – сигнал о поступлении в буфер сокета срочных данных (***high bandwidth data is available at a socket***). Он сообщает процессу, что по сетевому соединению получены срочные внеочередные данные;

SIGUSR1 и ***SIGUSR2*** – пользовательские сигналы (***user defined signals 1 and2***). Так же, как и сигнал ***SIGTERM***, эти сигналы никогда не посылаются ядром имогут использоваться для любых целей по выбору пользователя;

SIGVTALRM – сигнал виртуального таймера (***virtual timer expired***). Третий таймер можно установить так, чтобы он измерял время, которое процесс выполняет в пользовательском режиме.

Наборы сигналов определяются при помощи типа ***sigset_t***, который определен в заголовочном файле <***signal.h***>. Выбрать определенные сигналы можно, начав либо с полного набора сигналов и удалив ненужные сигналы, либос пустого набора,

включив в него нужные. Инициализация пустого и полного набора сигналов выполняется при помощи процедур ***sigemptyset*** и ***sigfillset*** соответственно. После инициализации с наборами сигналов можно оперировать при помощи процедур ***sigaddset*** и ***sigdelset***, соответственно добавляющих и удаляющих указанные вами сигналы.

Описание данных процедур:

#include <signal.h>

/* Инициализация*/

int sigemptyset (sigset_t *set);

int sigfillset (sigset_t *set);

/*Добавление и удаление сигналов*/

int sigaddset (sigset_t *set, int signo);

int sigdelset (sigset_t *set, int signo);

Процедуры ***sigemptyset*** и ***sigfillset*** имеют единственный параметр – указательна переменную типа ***sigset_t***. Вызов ***sigemptyset*** инициализирует набор ***set***, исключив из него все сигналы. И, наоборот, вызов ***sigfillset*** инициализирует набор, на который указывает ***set***, включив в него все сигналы. Приложения должны вызывать ***sigemptyset*** или ***sigfillset*** хотя бы один раз для каждой переменной типа ***sigset_t***.

Процедуры ***sigaddset*** и ***sigdelset*** принимают в качестве параметров указательна инициализированный набор сигналов и номер сигнала, который должен быть добавлен или удален. Второй параметр, ***signo***, может быть символическим именем константы, таким как ***SIGINT***, или настоящим номером сигнала, но в последнем случае программа окажется системно-зависимой.

Задания на лабораторную работу

1. Изучите теоретическую часть лабораторной работы.

2. Исследуйте на конкретном примере особенности 3-х по выбору из указанных методов синхронизации потоков:

1) семафоры
2) мьютексы
3) сигналы
4) условные переменные
5) барьеры

Задачи для синхронизации придумайте самостоятельно, исходя из особенностей методов.

Примечание

1. Задачи для каждого метода синхронизации должны быть различными.
2. Задачи должны наглядно демонстрировать выбранный метод синхронизации и учитывать особенности его применения.

Студент, сдающий работу, должен АРГУМЕНТИРОВАННО обосновать выбранную задачу и метод синхронизации.