3. ЛАБОРАТОРНАЯ РАБОТА № 3. ПРАКТИЧЕСКОЕ ЗНАКОМСТВО С ПОТОКАМИ И СИНХРОНИЗАЦИЕЙ ПОТОКОВ В ОС *UNIX*

ЦЕЛЬ РАБОТЫ

Ознакомиться с подсистемой управления потоками в операционной системе *Unix* и основными программными средствами для создания, управления и удаления потоков.

ЗАДАНИЕ

Изучить основные программные средства управления потоками ОС *Unix*, а также способы синхронизации потоков. Разработать приложения для многопоточных вычислений с использованием синхронизации посредством мьютексов, семафоров и условных переменных.

3.1. УПРАВЛЕНИЕ ПОТОКАМИ

3.1.1. Понятие потока

Поток (*thread*) можно определить как часть процесса, включающая управляющую последовательность команд и использующая системные ресурсы этого процесса.

Существует две основных категории реализации потоков:

Пользовательские потоки – потоки, реализуемые через специальные библиотеки потоков и работающие в пользовательском пространстве.

Потоки уровня ядра – потоки, реализуемые через системные вызовы и работающие в пространстве ядра.

Каждый процесс имеет, как минимум, один поток, при этом самый первый поток, создаваемый при рождении нового процесса, принято называть начальным или главным потоком этого процесса.

Основное отличие процесса от потока заключается в способе использования системных ресурсов. Дочерний процесс практически независим от родительского, для него системой выделяется отдельное адресное пространство, и он на равных правах с родительским процессом «конкурирует» за процессорное время. При этом можно уничтожить родительский процесс, не затронув дочерний, который может выполняться и после завершения родительского процесса.

В отличие от дочернего процесса, поток, порожденный данным процессом, полностью зависим от процесса и завершение процесса влечет уничтожение всех созданных им потоков, поскольку происходит освобождение системных ресурсов, выделенных для этого процесса. Еще одним отличием является невозможность изменить права доступа для потока, тогда как дочерний процесс в редких случаях (например, при смене пароля для входа в систему) может обладать правами доступа отличными от прав родительского процесса.

Первая подсистема потоков в *Linux* появилась в 1996 году и называлась *LinuxThreads*, ее автором является *Ксавье Лерой* (*Xavier Leroy*). Разработанная им библиотека *LinuxThreads* была попыткой организовать поддержку потоков в *Linux* в то время, когда ядро системы еще не предоставляло никаких специальных механизмов для работы с потоками. Позднее разработку потоков для *Linux* вели сразу две конкурирующие группы – *NGPT* (*New Generation POSIX Threads*) и *NPTL* (*Native POSIX Thread Library*). В 2002 году группа *NGPT* фактически присоединилась к *NPTL* и теперь реализация потоков *NPTL* является стандартом *Linux*. Подсистема потоков *Linux* стремится соответствовать требованиям стандартов *POSIX*, поэтому новые многопоточные приложения *Linux* должны без проблем компилироваться на новых *POSIX*-совместимых системах.

3.1.2. Преимущества и недостатки использования потоков

Потоки часто становятся источниками программных ошибок особого рода. Эти ошибки возникают при использовании потоками разделяемых ресурсов системы (например, общего адресного пространства) и являются частным случаем более широкого класса ошибок – ошибок синхронизации. Если задача разделена между независимыми процессами, то доступом к их общим ресурсам управляет ОС, и вероятность ошибок из-за конфликтов доступа снижается. Впрочем, разделение задачи между несколькими независимыми процессами само по себе не защищает от других разновидностей ошибок синхронизации. В пользу потоков можно указать то, что накладные расходы на создание нового потока в многопоточном приложении ниже, чем накладные расходы на создание нового самостоятельного процесса. Уровень контроля со стороны родительского процесса над порожденными потоками в многопоточном приложении выше, чем уровень контроля приложения над дочерними процессами. Кроме того, в результате работы многопоточных программ, как правило, не остаются незавершенными дочерние процессы,

зависящие от родительского (процессы-«зомби»), как может произойти при использовании многопроцессного приложения.

Преимущества и недостатки использования нескольких потоков во время выполнения программы приведены в табл. 3.1.

Таблица 3.1

Преимущества и недостатки использования потоков

Преимущества	Недостатки
Потоки удобно использовать при необходимости выполнения в процессе нескольких действий сразу (пример: одновременная обработка на сервере запросов нескольких пользователей)	Создание многопоточного приложения может потребовать значительных усилий на этапе проектирования из-за необходимости синхронизации потоков
Ускорение работы приложений, использующих ввод, обработку и вывод данных за счет возможности распределения этих операций по отдельным потокам. Это дает возможность не прекращать выполнение программы во время возможных простоев из-за ожидания при чтении/записи данных	Отладка многопоточной программы значительно сложнее, чем отладка однопоточной из-за сложности контроля исполнения различных потоков
Как правило, переключение между потоками происходит быстрее и требует меньших затрат системных ресурсов, по сравнению с переключением между процессами	Параллельные вычисления, реализованные в виде многопоточного приложения, на компьютере с одним процессором не обязательно будут выполняться быстрее, чем аналогичная по функциональности однопоточная программа

3.1.3. Программирование потоков

В *Linux* каждый поток на самом деле является процессом, и для того, чтобы создать новый поток, нужно создать новый процесс. Однако для создания дополнительных потоков используются процессы особого типа. Эти процессы представляют собой обычные дочерние процессы главного процесса, но они разделяют с главным процессом адресное пространство, файловые дескрипторы и обработчики сигналов. Для обозначения процессов этого типа применяется специальный термин –

легкие процессы (*lightweight processes*). Поскольку для потоков не требуется создавать собственную копию адресного пространства (и других ресурсов) своего процесса-родителя, создание нового легкого процесса требует значительно меньших затрат, чем создание полновесного дочернего процесса.

Спецификация *POSIX* 1003.1*c* требует, чтобы все потоки многопоточного приложения имели один идентификатор, однако в *Linux* у каждого процесса, в том числе и у процессов-потоков, есть свой идентификатор.

Основные функции для работы с потоками

Для работы с потоками используются следующие основные функции

- *pthread_create* – создание потока;
- *pthread_join* – блокирование работы вызвавшего функцию процесса или потока в ожидании завершения потока;
- *pthread_cancel* – досрочное завершение потока из другого потока или процесса;
- *pthread_exit* – завершает поток, код завершения передается функции *pthread_join*. Данная функция подобна функции *exit*, однако вызов *exit* в «основном» процессе программы приведет к завершению всей программы.

Запуск и завершение потока

Потоки создаются функцией *pthread_create*, имеющей следующую сигнатуру

```
int pthread_create (pthread_t* tid, pthread_attr_t* attr,
void*(*function)(void*), void* arg)
```

Данная функция определена в заголовочном файле *<pthread.h>*. Первый параметр этой функции представляет собой указатель на переменную типа *pthread_t*, которая служит идентификатором создаваемого потока. Второй параметр – указатель на переменную типа *pthread_attr_t* – используется для установки атрибутов потока. Третьим параметром функции *pthread_create* должен быть адрес функции потока. Эта функция играет для потока ту же роль, что функция *main* для главной программы. Четвертый параметр функции *pthread_create* имеет тип *void**. Этот параметр может использоваться для передачи значения в функцию потока. Вскоре после вызова *pthread_create* функция потока будет запущена на выполнение параллельно с другими потоками программы.

Новый поток запускается не сразу после вызова *pthread_create*, потому что перед тем, как запустить новую функцию потока, нужно выполнить некоторые подготовительные действия, а поток-родитель при этом продолжает выполняться. Необходимо учитывать данное обстоятельство при разработке многопоточного приложения, в противном случае возможны серьезные ошибки при выполнении программы. Если при создании потока возникла ошибка, то функция *pthread_create* возвращает ненулевое значение, соответствующее номеру ошибки.

Функция потока должна иметь сигнатуру вида

```
void* func_name(void* arg)
```

Имя функции может быть любым. Аргумент *arg* является указателем, который передается в последнем параметре функции *pthread_create*. Функция потока может вернуть значение, которое затем может быть обработано другим потоком или процессом.

Функция, вызываемая из функции потока, должна обладать свойством реентерабельности (этим же свойством должны обладать рекурсивные функции). Реентерабельная функция – это функция, которая может быть вызвана повторно, в то время, когда она уже вызвана. Такие функции используют локальные переменные (и локально выделенную память) в тех случаях, когда их нереентерабельные аналоги могут воспользоваться глобальными переменными.

Завершение функции потока происходит в следующих случаях

- функция потока вызвала функцию *pthread_exit*;
- функция потока достигла точки выхода;
- поток был досрочно завершен другим потоком или процессом.

Функция *pthread_exit* объявлена в заголовочном файле <*pthread.h*> и ее сигнатура имеет вид

```
void pthread_exit(void *retval)
```

Аргументом функции является указатель на возвращаемый объект. Нельзя возвращать указатель на стековый (нединамический) объект, объявленный в теле функции потока, либо на динамический объект, создаваемый и удаляемый в теле функции, т. к. после завершения потока все стековые объекты его функции удаляются. В итоге указатель будет содержать адрес памяти с неопределенным содержимым, что может привести к серьезной ошибке.

В случае, если необходимо дождаться завершения потока в теле родительского процесса, вызывается функция *pthread_join* следующего вида

```
int pthread_join(pthread_t th, void** thread_return)
```

Первый аргумент *th* необходим для указания ожидаемого потока, значение этого аргумента определяется в результате выполнения функции *pthread_create*. В качестве второго аргумента *thread_return* выступает указатель на аргумент функции *pthread_exit* либо *NULL*, если поток ничего не возвращает.

Досрочное завершение потока

Функции потоков можно рассматривать как вспомогательные программы, находящиеся под управлением функции *main*. Точно так же, как при управлении процессами иногда возникает необходимость досрочно завершить процесс, многопоточной программе может понадобиться досрочно завершить один из потоков. Для досрочного завершения потока можно воспользоваться функцией *pthread_cancel*

```
int pthread_cancel(pthread_t thread)
```

Единственным аргументом этой функции является идентификатор потока – *thread*. Функция *pthread_cancel* возвращает 0 в случае успеха и ненулевое значение (код ошибки) в случае ошибки.

Несмотря на то, что *pthread_cancel* может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков. В теле функции потока можно не только самостоятельно выбрать порядок завершения в ответ на вызов *pthread_cancel*, но и вовсе игнорировать этот вызов. Поэтому вызов функции *pthread_cancel* следует рассматривать как запрос на выполнение досрочного завершения потока.

Функция *pthread_setcancelstate* определяет, будет ли поток реагировать на обращение к нему с помощью *pthread_cancel* или не будет. Сигнатура функции имеет вид

```
int pthread_setcancelstate(int state, int* oldstate)
```

Аргумент *state* может принимать два значения

- *PTHREAD_CANCEL_DISABLE* – запрет завершения потока;
- *PTHREAD_CANCEL_ENABLE* – разрешение на завершение потока.

Во второй аргумент *oldstate* записывается указатель на предыдущее значение аргумента *state*. С помощью функции *pthread_setcancelstate*

можно указывать участки кода потока, во время исполнения которых поток нельзя завершить вызовом функции *pthread_cancel*

```
//...
// участок функции, который можно досрочно завершить
//...
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
//...
// участок функции, который нельзя досрочно завершить
//...
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
//...
// участок функции, который можно досрочно завершить
//...
```

Функция *pthread_testcancel* создает точку возможного досрочного завершения потока (точку отмены). Такие точки необходимы для корректного завершения потока, т. к. даже если досрочное завершение разрешено, поток, получивший запрос на досрочное завершение, часто может завершить работу не сразу. Если поток находится в режиме отложенного досрочного завершения (именно этот режим установлен по умолчанию), он выполнит запрос на досрочное завершение, только достигнув одной из точек отмены. Сигнатура функции *pthread_testcancel*

```
void pthread_testcancel()
```

В соответствии со стандартом *POSIX* точками отмены являются вызовы многих «обычных» функций, например *open*, *pause* и *write*.

Тем не менее, выполнение потока может быть прервано принудительно, не дожидаясь точек отмены. Для этого необходимо перевести поток в режим немедленного завершения, что делается с помощью вызова функции

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL)
```

Вызов функции

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL)
```

снова переводит поток в режим отложенного досрочного завершения.

3.1.4. Синхронизация потоков

При выполнении нескольких потоков во многих случаях необходимо синхронизировать их взаимодействие. Существует несколько способов синхронизации потоков

- взаимные исключения – мьютексы (*Mutex – MUTial EXclusion*);
- переменные состояния;
- семафоры.

Механизм использования переменных состояния и семафоров в многопоточных приложениях аналогичен механизму использования этих методов синхронизации в многопроцессных приложениях.

Механизм мьютексов представляет общий метод синхронизации выполнения потоков. Мьютекс можно определить как объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. В любой момент мьютексом может владеть только один поток.

Использование мьютексов гарантирует, что только один поток в некоторый момент времени выполняет критическую секцию кода. Мьютексы можно использовать и в однопоточном коде.

Доступны следующие действия с мьютексом: инициализация, удаление, захват или открытие, попытка захвата.

Объекты синхронизации потоков являются переменными в памяти процесса и обладают живучестью объектов процесса. Потоки в различных процессах могут связаться друг с другом через объекты синхронизации, помещенные в разделяемую память потоков, даже в случае, когда потоки в различных процессах невидимы друг для друга.

Объекты синхронизации можно также разместить в файлах, где они будут существовать независимо от создавшего их процесса.

Необходимость в синхронизации потоков возникает в следующих случаях

1. Если синхронизация – это единственный способ гарантировать последовательность разделяемых (общих) данных.
2. Если потоки в двух или более процессах могут использовать единственный объект синхронизации совместно. При этом объект синхронизации должен инициализироваться только одним из взаимодействующих процессов, потому что повторная инициализация объекта синхронизации устанавливает его в открытое состояние.
3. Если синхронизация может гарантировать достоверность изменяющихся данных.
4. Если процесс может отобразить файл и существует поток в этом процессе, который получает уникальный доступ к записям. Как только

установлена блокировка, любой другой поток в любом процессе, отображающем файл, который пытается установить блокировку, блокируется, пока запись в файл не будет закончена.

Функции синхронизации потоков с использованием мьютексов

Для синхронизации потоков с использованием мьютексов используются следующие основные функции:

- *pthread_mutex_init* – инициализирует взаимоисключающую блокировку;
- *pthread_mutex_destroy* – удаляет взаимоисключающую блокировку;
- *pthread_mutex_lock* – устанавливает блокировку. В случае, если блокировка была установлена другим потоком, текущий поток останавливается до снятия блокировки другим процессом;
- *pthread_mutex_unlock* – снимает блокировку.

Инициализация и удаление объекта атрибутов мьютекса

Атрибуты мьютекса могут быть связаны с каждым потоком. Чтобы изменить атрибуты мьютекса по умолчанию, можно объявить и инициализировать объект атрибутов мьютекса, а затем изменить определенные значения. Часто атрибуты мьютекса устанавливаются в одном месте в начале приложения, чтобы быстро найти и изменить их. После того, как сформированы атрибуты мьютекса, можно его инициализировать.

Функция

```
int pthread_mutexattr_init (pthread_mutexattr_t* mattr)
```

используется, чтобы инициализировать объект атрибутов *mattr* значениями по умолчанию. Память для каждого объекта атрибутов выделяется системой поддержки потоков во время выполнения.

Пример вызова функции *pthread_mutexattr_init*

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
/* инициализация атрибутов значениями по умолчанию */
ret = pthread_mutexattr_init(&mattr);
```

Для корректного удаления объекта атрибутов, созданного с помощью функции *pthread_mutexattr_init*, необходимо вызвать

функцию *pthread_mutexattr_destroy*. В противном случае возможна утечка памяти, так как тип *pthread_mutexattr_t* является закрытым. Она возвращает 0 после успешного завершения или другое значение, если произошла ошибка. Пример вызова функции *pthread_mutexattr_destroy*

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
/* удаление атрибутов */
ret = pthread_mutexattr_destroy(&mattr);
```

Область видимости мьютекса

Областью видимости мьютекса может быть либо некоторый процесс, либо вся система. В первом случае оперировать мьютексом могут только потоки, созданные процессом, в котором создан и мьютекс. Во втором случае мьютекс существует в разделяемой памяти и может быть разделен среди потоков нескольких процессов. По умолчанию мьютекс создается в области видимости процесса и обладает живучестью процесса.

Чтобы установить область видимости атрибутов мьютекса используется функция

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t* mattr, int)
```

Первый аргумент *mattr* является указателем на объект атрибутов, для которого устанавливается область видимости. Второй аргумент является константой, обозначающей устанавливаемую область видимости: *PTHREAD_PROCESS_PRIVATE* для области видимости процесса и *PTHREAD_PROCESS_SHARED* для области видимости системы. Мьютекс, созданный в области видимости системы, должен существовать в разделяемой памяти.

Пример вызова функции *pthread_mutexattr_setpshared*

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
ret = pthread_mutexattr_init(&mattr);
/* переустановка на значение по умолчанию: private */
ret = pthread_mutexattr_setpshared
        (&mattr, PTHREAD_PROCESS_PRIVATE);
```

Функция

pthread_mutexattr_getpshared(pthread_mutexattr_t *mattr, int *pshared)

используется для получения текущей области видимости мьютекса потока

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int pshared, ret;
/* получить атрибут pshared для мьютекса */
ret = pthread_mutexattr_getpshared(&mattr, &pshared);
```

Инициализация мьютекса

Функция *pthread_mutex_init* предназначена для инициализации мьютекса

int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);

Мьютекс, на который указывает первый аргумент *mp*, инициализируется значением по умолчанию, если второй аргумент *mattr* равен *NULL*, или определенными атрибутами, которые уже установлены с помощью *pthread_mutexattr_init*.

Функция *pthread_mutex_init* возвращает 0 после успешного завершения или другое значение, если произошла ошибка. Пример использования функции *pthread_mutexattr_init*

```
#include <pthread.h>
pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;
/* инициализация мьютекса значением по умолчанию */
ret = pthread_mutex_init(&mp, NULL);
```

Когда мьютекс инициализируется, он находится в открытом (разблокированном) состоянии. Статически определенные мьютексы могут инициализироваться непосредственно значениями по умолчанию с помощью константы *PTHREAD_MUTEX_INITIALIZER*. Пример инициализации

```
/* инициализация атрибутов мьютекса по умолчанию*/
ret = pthread_mutexattr_init(&mattr);
/* смена значений mattr с помощью функций */
ret = pthread_mutexattr_setpshared
```

```
(&mattr, PTHREAD_PROCESS_SHARED);
/* инициализация мьютекса произвольными значениями */
ret = pthread_mutex_init(&mp, &mattr);
```

Запирание (захват) мьютекса

Функция *pthread_mute_lock* используется для запирания или захвата мьютекса. Аргументом функции является адрес запираемого мьютекса. Если мьютекс уже заперт, вызывающий поток блокируется и мьютекс ставится в очередь приоритетов. Когда происходит возврат из *pthread_mute_lock*, мьютекс запирается, а вызывающий поток становится его владельцем. Функция *pthread_mute_lock* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример вызова

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_mutex_lock(&mp);
```

Для открытия (раблокировки) мьютекса используется функция *pthread_mutex_unlock*. При этом мьютекс должен быть закрыт, а вызывающий поток должен быть владельцем мьютекса, то есть тем, кто его запирал. Пока любые другие потоки ждут доступа к мьютексу, его поток-владелец, находящийся в начале очереди, не блокирован. Функция *pthread_mutex_unlock* возвращает 0 после успешного завершения или другое значение, если произошла ошибка. Пример вызова

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_mutex_unlock(&mp);
```

Существует способ захвата мьютекса без блокирования потока. Функция *pthread_mutex_trylock* пытается провести запирание мьютекса. Она является неблокирующей версией *pthread_mutex_lock*. Если мьютекс уже закрыт, вызов возвращает ошибку, однако поток, вызвавший эту функцию, не блокируется. В противном случае мьютекс закрывается, а вызывающий процесс становится его владельцем. Функция *pthread_mutex_trylock* возвращает 0 после успешного завершения или другое значение, если произошла ошибка. Пример вызова

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_ mutex_trylock(&mp);
```

Захват через мьютекс не должен повторно инициализироваться или удаляться, пока другие потоки могут его использовать. Если мьютекс инициализируется повторно или удаляется, приложение должно убедиться, что в настоящее время этот мьютекс не используется.

Удаление мьютекса

Функция *pthread_mutex_destroy* используется для удаления мьютекса в любом состоянии. Функция *pthread_mutex_destroy* возвращает 0 после успешного завершения или другое значение, если произошла ошибка. Пример вызова

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_mutex_destroy(&mp);
```

Иерархия блокировок

Иногда может возникнуть необходимость одновременного доступа к нескольким ресурсам. При этом возникает проблема, заключающаяся в том, что два потока пытаются захватить оба ресурса, но запирают соответствующие мьютексы в различном порядке.

В приведенном ниже примере два потока запирают мьютексы 1 и 2 и возникает тупик при попытке запереть один из мьютексов.

Таблица 3.2

Пример «тупика» для двух потоков

Поток 1	Поток 2
/* использует ресурс 1 */ *pthread_mutex_lock(&m1)*; /* теперь захватывает ресурсы 2+1*/ *pthread_mutex_lock(&m2)*;	/* использует ресурс 2 */ *pthread_mutex_lock(&m2);* /* теперь захватывает ресурсы 1+2*/ *pthread_mutex_lock(&m1)*;

Наилучшим способом избежать проблем является запирание нескольких мьютексов в одном и том же порядке во всех потоках. Эта

техника называется иерархией блокировок: мьютексы упорядочиваются путем назначения каждому своего номера. После этого придерживаются правила – если мьютекс с номером *n* уже заперт, то нельзя запирать мьютекс с номером меньше *n*.

Если блокировка всегда выполняется в указанном порядке, тупик не возникнет. Однако эта техника может использоваться не всегда, поскольку иногда требуется запирать мьютексы в порядке, отличном от порядка их номеров.

Чтобы предотвратить тупик в этой ситуации, лучше использовать функцию *pthread_mutex_trylock*. Один из потоков должен освободить свой мьютекс, если он обнаруживает, что может возникнуть тупик.

Ниже проиллюстрировано использование условной блокировки

```
// Поток 1:
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m2);
/* обработка */
/* нет обработки */
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
// Поток 2:
for (; ;) {
pthread_mutex_lock(&m2);
if (pthread_mutex_trylock(&m1)==0)
/* захват! */
break;
/* мьютекс уже заперт */
pthread_mutex_unlock(&m2);
}
/* нет обработки */
pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);
```

В примере выше поток 1 запирает мьютексы в нужном порядке, а поток 2 пытается закрыть их по-своему. Чтобы убедиться, что тупик не возникнет, поток 2 должен аккуратно обращаться с мьютексом 1; если поток блокировался, ожидая мьютекс, который будет освобожден, он, вероятно, только что вызвал тупик с потоком 1. Чтобы гарантировать,

что это не случится, поток 2 вызывает *pthread_mutex_trylock*, который запирает мьютекс, если тот свободен. Если мьютекс уже заперт, поток 2 получает сообщение об ошибке. В этом случае поток 2 должен освободить мьютекс 2, чтобы поток 1 мог запереть его, а затем освободить оба мьютекса.

Синхронизация с использованием семафора

Семафор предназначен для синхронизации потоков по действиям и данным, и в общем случае способ использования семафора сходен со способом использования мьютексов.

Семафор (S) – это защищенная переменная, значения которой можно опрашивать и менять только при помощи специальных операций $P(S)$ и $V(S)$ и операции инициализации. Семафор может принимать целое неотрицательное значение. При выполнении потоком операции P над семафором S значение семафора уменьшается на 1 при $S > 0$, или поток блокируется, «ожидая на семафоре», при $S = 0$. При выполнении операции $V(S)$ происходит пробуждение одного из потоков, ожидающих на семафоре S, а если таковых нет, то значение семафора увеличивается на 1. В простом случае, когда семафор работает в режиме 2-х состояний ($S > 0$ и $S = 0$), его алгоритм работы полностью совпадает с алгоритмом мьютекса.

Как следует из вышесказанного, при входе в критическую секцию поток должен выполнять операцию $P(S)$, а при выходе из критической секции операцию $V(S)$.

Прототипы функций для манипуляции с семафорами описываются в файле <*semaphore.h*>. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий:

- *int sem_init(sem_t* sem, int pshared, unsigned int value)* – инициализация семафора *sem* значением *value*. В качестве *pshared* всегда необходимо указывать 0.
- *int sem_wait(sem_t* sem)* – «ожидание на семафоре». Выполнение потока блокируется до тех пор, пока значение семафора не станет положительным. При этом значение семафора уменьшается на 1.
- *int sem_post(sem_t* sem)* – увеличивает значение семафора *sem* на 1.
- *int sem_destroy(sem_t* sem)* – уничтожает семафор *sem*.
- *int sem_trywait(sem_t* sem)* – неблокирующий вариант функции *sem_wait*. При этом вместо блокировки вызвавшего потока функция возвращает управление с кодом ошибки в качестве результата работы.

Синхронизация с использованием условной переменной

Условная переменная позволяет потокам ожидать выполнения некоторого условия (события), связанного с разделяемыми данными. Над условными переменными определены две основные операции: информирование о наступлении события и ожидание события. При выполнении операции «информирование» один из потоков, ожидающих значения условной переменной, возобновляет свою работу.

Условная переменная всегда используется совместно с мьютексом. Перед выполнением операции «ожидание» поток должен заблокировать мьютекс. При выполнении операции «ожидание» указанный мьютекс автоматически разблокируется. Перед возобновлением ожидающего потока выполняется автоматическая блокировка мьютекса, позволяющая потоку войти в критическую секцию, после критической секции рекомендуется разблокировать мьютекс. При подаче сигнала другим потокам рекомендуется функцию «сигнализации» так же защитить мьютексом.

Прототипы функций для работы с условными переменными содержатся в файле *pthread.h*. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий

- *pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* attr)* – инициализирует условную переменную *cond* с указанными атрибутами *attr* или с атрибутами по умолчанию (при указании 0 в качестве *attr*).
- *int pthread_cond_destroy(pthread_cond_t* cond)* – уничтожает условную переменную cond.
- *int pthread_cond_signal(pthread_cond_t* cond)* – информирование о наступлении события потоков, ожидающих на условной переменной *cond*.
- *int pthread_cond_broadcast(pthread_cond_t* cond)* – информирование о наступлении события потоков, ожидающих на условной переменной *cond*. При этом возобновлены будут все ожидающие потоки.
- *int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex)* – ожидание события на условной переменной cond.

Рассмотренных средств достаточно для решения разнообразных задач синхронизации потоков. Вместе с тем они обеспечивают взаимоисключение на низком уровне и не наполнены семантическим смыслом. При непосредственном их использовании легко допустить ошибки различного рода: забыть выйти из критической секции, использовать примитив не по назначению, реализовать вложенное

использование примитива и т. д. При этом операции с мьютексами, семафорам и условными переменными оказываются разбросанными по всему программному коду приложения, что повышает вероятность появления ошибки и усложняет ее поиск и устранение.

3.1.5. Компиляция многопоточной программы

Для компиляции и сборки многопоточной программы необходимо иметь следующее

- стандартный компилятор *C* (*cc, gcc, g++*т. д.);
- файлы заголовков: *<thread.h>, <pthread.h>, <errno.h>, <limits.h>, <signal.h>, <unistd.h>*;
- библиотеку реализации потоков (*libpthread*);
- другие библиотеки, совместимые с многопоточными приложениями (*libc*, *libm*, *libw*, *libintl*, *libnsl*, *libsocket*, *libmalloc*, *libmapmalloc* и др.).

Файл заголовка <*pthread.h*>, используемый с библиотекой *lpthread*, компилирует код, который является совместимым с интерфейсами многопоточности, определенными стандартом *POSIX* 1003.1*c*.

Для компиляции программы, использующей потоки и реентерабельные системные функции, необходимо дополнительно указать в строке вызова компилятора следующие аргументы:

```
-D_REENTERANT –lpthread.
```

Команда компиляции *D* включает макрос *_REENTERANT*. Этот макрос указывает, что вместо обычных функций стандартной библиотеки к программе должны быть подключены их реентерабельные аналоги. Реентерабельный вариант библиотеки *glibc* написан таким образом, чтобы реализованные в ней реентерабельные функции как можно меньше отличались от их обычных аналогов. Также в строке вызова компилятора могут дополнительно указываться пути для поиска заголовочных файлов (ключ «*I*») и путь для поиска библиотек (ключ *L*). Для компоновщика указывается «*l*», что программа должна быть связана с библиотекой *libpthread*, которая содержит все специальные функции, необходимые для работы с потоками.

3.1.6. Особенности отладки многопоточной программы

Отладка многопоточной программы сложнее, чем отладка однопоточной. Ниже приведены наиболее типичные ошибки исходного кода, которые могут вызвать ошибки исполнения в многопоточных программах

1. Доступ к глобальной памяти без использования механизмов синхронизации.

2. Создание тупиков, вызванных двумя потоками, пробующими получить права на одну и ту же пару глобальных ресурсов в различном порядке (один поток управляет одним ресурсом, а второй управляет другим ресурсом и ни один неможет продолжать выполнение до освобождения нужного им ресурса).

3. Попытка повторно получить доступ к уже захваченному ресурсу (рекурсивный тупик).

4. Создание скрытого промежутка при синхронизации. Это происходит, когда сегмент кода, защищенный механизмом синхронизации, содержит вызов функции, которая освобождает и затем повторно создает механизм синхронизации прежде, чем управление возвращается к вызывающему потоку. В результате вызывающему «кажется», что глобальные данные были защищены, хотя это не так.

5. Невнимание к тому факту, что потоки по умолчанию создаются с типом *PTHREAD_CREATE_JOINABLE* и ресурсы таких потоков должны быть утилизированы (возвращены родительскому процессу) посредством вызова функции *pthread_join*. Обратите внимание, что *pthread_exit* не освобождает выделенную память.

6. Создание глубоко вложенных, рекурсивных обращений и использование больших автоматических массивов может вызвать проблемы, потому что многопоточные программы имеют более ограниченный размер стека, чем однопоточные.

7. Определение неадекватного размера стека или использование стека не по умолчанию.

3.1.7. Примеры практической реализации

Пример программы с использованием потока.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
int i = 0;
void* thread_func(void *arg) {
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        for (i=0; i < 4; i++)  {
                        printf("I'm still running!\n");
                        }
```

```
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
        pthread_testcancel();
        printf("YOU WILL NOT STOP ME!!!\n");
}
int main(int argc, char * argv[]) {
        pthread_t thread;
        pthread_create(&thread, NULL, thread_func, NULL);
        pthread_cancel(thread);
        printf("Requested to cancel the thread\n");
        pthread_join(thread, NULL);
        printf("The thread is stopped.\n");
return EXIT_SUCCESS;
}
```

Пример реализации многопоточной программы.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define NUM_THREADS 6
void *thread_function(void *arg);
int main() {
int res;
int lots_of_threads;
pthread_t a_thread[NUM_THREADS];
void *thread_result;
srand ( (insigned)time(NULL) );
for (lots_of_threads = 0; lots_of_threads < NUM_THREADS;
lots_of_threads++)
        {
          res = pthread_create (&(a_thread[lots_of_threads]),NULL,
                        thread_function, (void *)&lots_of_threads);
```

```
            if (res != 0) {
                        perror("Thread creation failed");
                        exit(EXIT_FAILURE);
                    }
            }
    printf("Waiting for threads to finish...\n");
    for (lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0;
    lots_of_threads--)
        {
          res = pthread_join (a_thread[lots_of_threads],&thread_result);
          if (res == 0) printf("Picked up a thread\n");
                                    else perror("pthread_join failed");
    }
    printf("All done\n");
    exit(EXIT_SUCCESS);
    }

    void *thread_function(void *arg) {
        int my_number = *(int *)arg;
        int rand_num;
        printf ("thread_function is running. Argument was %d\n",
    my_number);
        rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
        printf ("Bye from %d\n", my_number);
        pthread_exit(NULL);
    }
```

Пример использования мьютекса для контроля доступа к переменной. В приведенном ниже коде функция *increment_count* использует мьютекс, чтобы гарантировать атомарность (целостность) модификации разделяемой переменной *count*. Функция *get_count()* использует мьютекс, чтобы гарантировать, что переменная *count* атомарно считывается.

```
#include <pthread.h>
```

```
pthread_mutex_t count_mutex;
long count;
void increment_count() {
        pthread_mutex_lock(&count_mutex);
        count = count + 1;
        pthread_mutex_unlock(&count_mutex);
}

long get_count() {
        long c;
        pthread_mutex_lock(&count_mutex);
        c = count;
        pthread_mutex_unlock(&count_mutex);
        return (c);
}
```

Пример многопоточной программы с синхронизацией с использованием мьютексов.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#define SIZE_I 2
#define SIZE_J 2
float X[SIZE_I][SIZE_J];
float S[SIZE_I][SIZE_J];
int count = 0;          // глобальный счетчик
struct DATA_ {
    double x;
    int i;
    int z;
};
typedef struct DATA_ DATA;
```

```
pthread_mutex_t lock; //Исключающая блокировка
// Функция для вычислений
double f(double x) { return x*x; }
// Потоковая функция для вычислений
void *calc_thr (void *arg) {
    DATA* a = (DATA*) arg;
    X[a->i][a->z] = f(a->x);
    // установка блокировки
    pthread_mutex_lock(&lock);
    // изменение глобальной переменной
    count ++;
    // снятие блокировки
    pthread_mutex_unlock(&lock);
    delete a;
    return NULL;
}
// Потоковая функция для ввода
void *input_thr(void *arg) {
    DATA* a = (DATA*) arg;
    printf("S[%d][%d]:", a->i, a->z);
    scanf("%f", &S[a->i][a->z]);
    delete a;
    return NULL;
}
int main() {
//массив идентификаторов потоков
pthread_t thr[ SIZE_I * SIZE_J ];
// инициализация мьютекса
pthread_mutex_init(&lock, NULL);
DATA *arg;
// Ввод данных для обработки
for (int i=0;i<SIZE_I; i++) {
```

```
      for (int z=0; z<SIZE_J; z++) {
            arg = new DATA;
            arg->i = i;
            arg->z = z;
// создание потока для ввода элементов матрицы
pthread_create (&thr[ i* SIZE_J + z ], NULL, input_thr, (void *)arg);
}      // for (int z=0; z<SIZE_J; P ++z)
}      // for (int i=0;i<SIZE_I; P ++i)
// Ожидание завершения всех потоков ввода данных
// идентификаторы потоков хранятся в массиве thr
for(int i = 0; i < SIZE_I*SIZE_J; i++) pthread_join (thr[i], NULL);
// Вычисление элементов матрицы
pthread_t thread;
printf("Start calculation\n");
for (int i=0;i<SIZE_I; i++) {
      for (int z=0; z<SIZE_J; z++) {
            arg = new DATA;
            arg->i = i;
            arg->z = z;
            arg->x = S[i][z];
            // создание потока для вычислений
            pthread_create (&thread, NULL, calc_thr, (void *)arg);
            // перевод потока в отсоединенный режим
            pthread_detach(thread);
            // for (int z=0; z<SIZE_J; z++)
}      // for (int i=0;i<SIZE_I; i++)
do {
// Основной процесс "засыпает" на 1с
// Проверка состояния вычислений
printf("finished %d threads.\n", count);
} while ( count < SIZE_I*SIZE_J);
// Вывод результатов
```

```
for (int i=0;i<SIZE_I; i++) {
     for (int z=0; z<SIZE_J; z++) {
          printf("X[%d][%d] = %f\t", i, z, X[i][z]);
}
printf("\n");
}
// удаление мьютекса
pthread_mutex_destroy(&lock);
return 0;
}
```

Пример многопоточной программы с синхронизацией семафорами.

```
#include "main.h"
#include <iostream.h>
#include <semaphore.h>
#include <fstream.h>
#include <stdio.h>
#include <error.h>
void* WriteToFile(void*);
int errno;
sem_t psem;
ofstream qfwrite;

int main(int argc, char **argv) {
pthread_t tidA,tidB;
int n;
char filename[] = "./rezult.txt";
qfwrite.open(&filename[0]);
sem_init(&psem,0,0);
sem_post(&psem))
        pthread_create(&tidA,NULL,&WriteToFile,(void*)100));
        pthread_create(&tidB,NULL,&WriteToFile,(void*)100));
        pthread_join(tidA,NULL));
```

```
        pthread_join(tidB,NULL));
sem_destroy(&psem));
qfwrite.close();
}
void* WriteToFile(void *f){
int max = (int)f;
for (int i=0; i<=max; i++)
 {
     sem_wait(&psem);
     qfwrite<<pthread_self()<<"-writetofilecounter i="<<i<<endl;
     qfwrite<<flush;
     sem_post(&psem);
 }
return NULL;
}
```

Пример многопоточной программы с синхронизацией с использованием условных переменных. Ниже приведен фрагмент программы, использующей семафоры для синхронизации записи (*writer*) и чтения (*reader*) данных в буфер *data* и из него, емкость буфера – 1 запись.

```
#include "main.h"
#include <iostream.h>
#include <semaphore.h>
#include <fstream.h>
#include <stdio.h>
#include <error.h>
...
int full;
pthread_mutex_t mx;
pthead_cond_t cond;
int data;
void *writer(void *)
{
```

```
        while(1)
        {
           int t= write_to_buffer ();
           pthread_mutex_lock(&mx)
           while(full) {
               pthread_cond_wait(&cond, &mx);
            }
        data=t;
        full=1;
        pthread_cond_signal(&mx);
        pthread_mutex_unlock(&mx);
        }
return NULL;
}

void * reader(void *)
{
        while (1)
        {
           int t;
           pthread_mutex_lock(&mx);
           while (!full)
           {
              pthread_cond_wait(&cond, &mx);
            }
         t=data;
         full=0;
         pthread_cond_signal(&mx);
         pthread_mutex_unlock(&mx);
         read_from _buffer();
        }
return NULL;
```

```
}
...
```

3.2. ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическим материалом.
2. Разработать три многопоточные программы с использованием минимум двух потоков и различных средств синхронизации. Например: два потока записывают и читают информацию из одного файла; два потока увеличивают значение общей переменной; два потока с различной частотой считывают и записывают данные в общий буфер памяти.
3. Необходимо обеспечить синхронизированную работу потоков в критической секции с использованием:
   - мьютексов;
   - семафоров;
   - условных переменных.
4. Убедиться в результативности применения средств синхронизации потоков, сравнив результаты работы программ с использованием и без использования средств синхронизации.

3.3. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчет должен содержать следующие разделы:

1. Титульный лист, оформленный согласно утвержденному образцу.
2. Цели выполняемой лабораторной работы.
3. Задание на лабораторную работу.
4. Исходные тексты созданных программ.
5. Результаты работы программ с использованием средств синхронизации. Результаты работы программ без использования средств синхронизации.
6. Выводы (с пояснением различий результатов работы программ при использовании и без использования средств синхронизации).