Лабораторная работа №7

«Потоки. Синхронизация I»

На данном занятии следует изучить проблемы, которые могут возникнуть вследствие совместной работы нескольких потоков с одним «объектом» и рассмотреть некоторые пути их решения: *мьютексы* и *блокировки для чтения и записи*.

Основные задания

Задание №1 (Состояние гонки)

Напишите программу, которая показывает возможность повреждения данных (переменная длинного целого типа с нулевым начальным значением) несколькими одновременно работающими потоками (демонстрирует поведение, описывающееся понятием *race conditions*). Программа запускает некоторое количество пар потоков-потомков. Поток из одной части этой пары заданное количество раз увеличивает значение переменной на 1, поток из другой части пары такое же количество раз уменьшает значение переменной на 1. Основной поток ожидает завершения работы всех потоков — потомков и выводит финальное значение изменяемой переменной. Количество потоков и количество операций, которые выполняет каждый поток, передаются в программу с помощью коротких опций. Если какая-то опция не задана, то предусмотреть для нее значение по умолчанию. Программа, должна заданное количество раз выполнить эти действия и каждый раз вывести то финальное значение переменной, какое получится в результате каждого запуска.

Проанализируйте поведение программы в зависимости от значения параметров.

Задание №2 (Мьютексы)

Напишите программу, которая находит численное значение определенного интеграла $\int_a^b f(x)\,\mathrm{d}\,x$ методом средних прямоугольников с помощью заданного количества параллельно работающих потоков. Количество потоков p, с помощью которых проводятся расчеты, функция вычисления интеграла получает при запуске программы через параметры командной строки.

Замечание: Для такого многопотокового режима работы программы можно воспользоваться свойством аддитивности интеграла:

$$\int_a^b f(x)\,\mathrm{d}\,x = \sum_{i=0}^{p-1} \int_{a_i}^{b_i} f(x)\,\mathrm{d}\,x,$$

где $a_i = a + i \cdot h$, $b_i = a_i + h$, $h = (b-a)/p$.

Таким образом, участок интегрирования $[a,b]$ разбивается на части, число которых совпадает с количеством работающих потоков. Каждый поток вычисляет свою часть интеграла

(интеграл на своем участке $[a_i, b_i]$) и прибавляет его к переменной, в которой «накапливается» результат. Для исключения несанкционированного доступа со стороны работающих потоков к «накапливающей» переменной необходимо защитить ее с помощью мьютекса.

Замечание: Для численного вычисления определенного интеграла

$$\int_a^b f(x)\,\mathrm{d}\,x,$$

где $f(x)$ непрерывна на отрезке $[a, b]$ можно воспользоваться *составным методом средних прямоугольников*. Это исключительно простой метод, дающий хорошую точность вычислений. Для выполнения численно интегрирования разобьем интервал интегрирования $[a, b]$ точками на n равных частей с шагом $h = \frac{b-a}{n}$. Для каждой точки из этого интервала ($x_0 = a$, $x_1 = x_0 + h$, $\ldots$, $x_i = x_{i-1} + h$, $x_n = b$) вычислим значение интегрируемой функции $f(x)$. Тогда приближенное значение интеграла можно вычислить по формуле составного метода средних прямоугольников.

$$\int_a^b f(x)\,\mathrm{d}\,x \approx I_n = h \cdot \sum_{i=0}^{n-1} f\left(x_i + \frac{h}{2}\right)$$

Будем считать, что интеграл вычислен с заданной точностью ε, если при последовательном удвоении количества отрезков разбиения n будет выполнено условие $|I_n - I_{2n}| < \varepsilon$. Здесь I_n и I_{2n} — интегральные суммы, посчитанные для n и $2n$ отрезков разбиения.

Замечание: Для удобства работы можно сначала написать «однопотоковую» версию программы, отладить ее, а затем преобразовать в многопотоковую. Для тестирования работы программы можно вычислить интеграл от какой-нибудь простой функции, например:

$$\int_0^2 (4 - x^2)\,\mathrm{d}\,x \approx 5.33333$$

Задание №3 (Блокировки чтения и записи)

Существует массив, количество элементов которого задается с помощью короткой опции при запуске программы. Если при запуске значение не указано, то создается массив из заданного по умолчанию количества элементов. Заданное количество присоединенных потоков-писателей записывают в него информацию (псевдослучайные числа из требуемого диапазона в элементы, выбранные псевдослучайным образом), заданное количество присоединенных потоков-читателей (большее, чем писателей) считывает информацию (сохраненные в массиве числа из элементов, выбранных псевдослучайным образом). Отсоединенный поток с заданной периодичностью выводит состояние массива в стандартный поток вывода. Для удобства работы можно

организовать необходимые случайные задержки. Согласованную работу системы (синхронизированный доступ к массиву) осуществите с помощью блокировок чтения-записи.

Рекомендации по выполнению

Состояние гонки

Программирование потоков — достаточно сложная задача, т. к. предполагается, что большинство потоков выполняется «одновременно». Например, приложению невозможно предсказать, когда система предоставит доступ к процессору одному потоку, а когда — другому. Длительность этого доступа может быть как достаточно большой, так и очень короткой, в зависимости от того, как часто система переключает задания. Если в системе есть несколько процессоров, потоки могут выполняться одновременно в буквальном смысле.

Большинство ошибок, возникающих при работе с потоками, связано с тем, что потоки обращаются к одним и тем же данным. Одно из главных достоинств потоков (простота обмена данными между собой), одновременно является их недостатком. При наличии нескольких потоков управления, совместно использующих одни и те же данные, необходимо гарантировать, что каждый из потоков будет видеть эти данные в непротиворечивом состоянии. Если каждый из потоков использует переменные, которые не используются в других потоках, то проблем не возникает. Аналогично, если переменная доступна одновременно нескольким потокам только для чтения, то здесь также отсутствует проблема. Однако, если один поток изменяет значение переменной, читать или изменять которое могут также другие потоки, то 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 может прочитать 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. 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.

Подобного рода ошибки называются *состоянием гонки* (*race conditions*): потоки преследуют друг друга в попытке изменить одни и те же данные, или, более точно, ситуация, в которой одновременные манипуляции нескольких потоков с ресурсом приводят к неоднозначным результатам. Чтобы исключить возможность гонки, необходимо сделать операции *атомарными* (*atomic*). Атомарная операция неделима и непрерывна; если она началась, то уже не может быть приостановлена или прервана до тех пор, пока не завершится. Выполнение других операций в это время становится невозможным. Для этого можно использовать блокировки, которые позволят только одному потоку работать с переменной в один момент времени. Таким образом, изменение переменной производится атомарно, и подобной гонки между потоками не будет.

Мьютексы

Можно защитить данные и ограничить доступ к ним одним потоком в один момент времени с помощью *интерфейса взаимоисключений* (*MUTual-EXclusion*). *Мьютекс* (*mutex*) — это блокировка, которая устанавливается перед обращением к разделяемому ресурсу и

снимается после выполнения требуемой последовательности операций. Если мьютекс заперт, то любой другой поток, который попытается запереть его, будет заблокирован до тех пор, пока мьютекс не будет отперт. Если в момент, когда отпирается мьютекс, заблокированными окажутся несколько потоков, все они будут запущены и первый из них, который успеет запереть мьютекс, продолжит работу. Все остальные потоки обнаружат, что мьютекс по-прежнему заперт, 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 будут соблюдать одни и те же правила доступа к данным. Операционная система никак не упорядочивает доступ к данным. Если мы позволим одному потоку производить действия с разделяемыми данными, предварительно не ограничив доступ к ним (без попытки «захватить» мьютекс), то остальные потоки могут обнаружить эти данные в противоречивом состоянии, даже если перед обращением к ним будут устанавливать блокировку.

Для создания мьютекса следует определить переменную типа pthread_mutex_t. Прежде чем использовать переменную-мьютекс, ее сначала необходимо инициализировать. Для этого можно либо присвоить переменной значение константы PTHREAD_MUTEX_INITIALIZER (только для статически размещаемых мьютексов) или передать указатель на эту переменную функции pthread_mutex_init. Если мьютекс размещается в динамической памяти (например, размещен с помощью функции malloc), то прежде чем освободить занимаемую память, необходимо вызвать функцию pthread_mutex_destroy.

```
#include <pthread.h>

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

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Обе функции возвращают значение 0 в случае успешного вызова, и код ошибки в случае неудачи.

При инициализации мьютекса функцией pthread_mutex_init в качестве второго аргумента используется указатель на объект атрибутов мьютекса. Чтобы инициализировать мьютекс со значениями атрибутов по умолчанию, следует указать значение NULL в аргументе attr. Конкретные значения атрибутов мьютексов рассмотрим на данном занятии чуть позже.

Мьютекс запирается при помощи вызова функции pthread_mutex_lock. Если мьютекс уже заперт, вызывающий поток будет заблокирован до тех пор, пока мьютекс не будет отперт. Мьютекс отпирается вызовом функции pthread_mutex_unlock.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthreadjnutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Все три функции возвращают значение 0 в случае успешного вызова, и код ошибки в

случае неудачи.

Если поток не должен блокироваться при попытке запереть мьютекс, он может воспользоваться функцией pthread_mutex_trylock. Если к моменту вызова этой функции мьютекс будет отперт, функция запрет мьютекс и вернет значение 0. В противном случае pthread_mutex_trylock вернет код ошибки EBUSY.

Рассмотрим пример применения мьютексов для защиты структуры данных. Если более чем один поток работает с данными, размещаемыми динамически, мы можем предусмотреть в структуре данных счетчик ссылок на объект, чтобы освобождать память только в том случае, когда все потоки завершат работу с объектом.

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

struct foo {
	int f_count;
	pthread_mutex_t f_lock;
	/* другие поля структуры */
};

struct foo * foo_alloc(void) { /* размещает объект в динамической памяти */
	struct foo *fp;

	if ((fp = malloc(sizeof(struct foo))) != NULL) {
		fp->f_count = 1;
		if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
			free(fp);
			return(NULL);
		}
		/* продолжение инициализации */
	}
	return(fp);
}

void foo_hold(struct foo *fp) { /* наращивает счетчик ссылок на объект */
	pthread_mutex_lock(&fp->f_lock);
	fp->f_count++;
	pthread_mutex_unlock(&fp->f_lock);
}

void foo_releaze(struct foo *fp) { /* освобождает ссылку на объект */
	pthread_mutex_lock(&fp->f_lock);
	if (--fp->f_count == 0) { /* последняя ссылка */
		pthread_mutex_unlock(&fp->f_lock);
		pthread_mutex_destroy(&fp->f_lock);
		free(fp);
	} else {
		pthread_mutex_unlock(&fp->f_lock);
```

```
        }
}
```

Мьютекс запирается перед увеличением или уменьшением счетчика ссылок и перед его проверкой на равенство нулю. При инициализации счетчика ссылок значением 1 в функции foo_alloc запирать мьютекс нет необходимости, поскольку пока только поток, который размещает структуру, имеет к ней доступ. Если бы в этой точке структура включалась в некий список, она могла бы быть обнаружена другими потоками, и тогда пришлось бы сначала запереть мьютекс.

Прежде чем приступить к работе с объектом, поток должен увеличить счетчик ссылок на него. По окончании работы с объектом поток должен удалить ссылку. Когда удаляется последняя ссылка, память, занимаемая объектом, освобождается.

Предотвращение тупиковых ситуаций

Поток может попасть в *тупиковую ситуацию* (*deadlock*), если, например, попытается дважды захватить один и тот же мьютекс. Бывают и менее очевидные способы создать такую тупиковую ситуацию. Например, тупиковая ситуация может возникнуть в случае, когда в программе используется более одного мьютекса и мы позволим одному потоку удерживать первый мьютекс и пытаться запереть второй мьютекс, в то время как некий другой поток аналогичным образом может удерживать второй мьютекс и пытается запереть первый. В результате ни один из потоков не сможет продолжить работу, поскольку каждый из них будет ждать освобождения ресурса, захваченного другим потоком, и возникает тупиковая ситуация.

Тупиковых ситуаций можно избежать, жестко определив порядок, в котором производится захват ресурсов. 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, то можно попробовать иной подход. Например, при невозможности потока запереть мьютекс можно отпереть все захваченные им 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, то можно продолжить работу. Однако, если мьютекс запереть не удалось, можно отпереть уже захваченные мьютексы, освободить занятые ресурсы и повторить попытку немного позже.

Блокировки чтения и записи

Блокировки чтения-записи похожи на мьютексы, за исключением того, что они допускают более высокую степень параллелизма. Мьютексы могут иметь всего два состояния, закрытое и открытое, и только один поток может владеть мьютексом в каждый момент времени. *Блокировки чтения-записи могут иметь три состояния*: *режим блокировки для чтения*, *режим блокировки для записи* и *отсутствие блокировки*. Режим блокировки для записи может установить только один поток, но установка режима блокировки для чтения доступна нескольким потокам одновременно.

Если блокировка чтения-записи установлена в режиме блокировки для записи, все потоки, которые будут пытаться захватить эту блокировку, будут приостановлены до тех пор, пока

блокировка не будет снята. Если блокировка чтения-записи установлена в режиме блокировки для чтения, все потоки, которые будут пытаться захватить эту блокировку для чтения, получат доступ к ресурсу, но если какой-либо поток попытается установить режим блокировки для записи, он будет приостановлен до тех пор, пока не будет снята последняя блокировка для чтения. Различные реализации блокировок чтения-записи могут значительно различаться, но обычно, если блокировка для чтения уже установлена и имеется поток, который пытается установить блокировку для записи, то остальные потоки, которые пытаются получить блокировку для чтения, будут приостановлены. Это предотвращает возможность блокирования пишущих потоков непрекращающимися запросами на получение блокировки для чтения.

Блокировки чтения-записи хорошо подходят для ситуаций, когда чтение данных производится намного чаще, чем запись. Когда блокировка чтения-записи установлена в режиме для записи, можно безопасно выполнять модификацию защищаемых ею данных, поскольку только один поток может владеть блокировкой для записи. Когда блокировка чтения-записи установлена в режиме для чтения, защищаемые ею данные могут быть безопасно прочитаны несколькими потоками, если эти потоки смогли получить блокировку для чтения.

Блокировки чтения-записи еще называют совместно-исключающими блокировками. Когда блокировка чтения-записи установлена в режиме для чтения, то говорят, что блокировка находится в режиме совместного использования. Когда блокировка чтения-записи установлена в режиме для записи, то говорят, что блокировка находится в режиме исключительного использования.

Стандарт *POSIX* определяет механизм блокировки для чтения и записи посредством типа pthread_rwlock_t. Как и в случае с мьютексами, блокировки чтения-записи должны быть инициализированы перед их использованием и разрушены перед освобождением занимаемой ими памяти. Для статически размещенных блокировок чтения-записи, аналогично статически размещенным мьютексам, можно воспользоваться специальным инициализатором PTHREAD_RWLOCK_INITIALIZER. Иначе, можно воспользоваться функциями:

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t * attr);

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Обе функции возвращают 0 в случае успешного завершения, или код ошибки в случае неудачи.

Функция pthread_rwlock_init инициализирует блокировку чтения-записи. Если в аргументе attr передается пустой указатель, блокировка инициализируется с атрибутами по умолчанию. Атрибуты блокировок чтения-записи мы рассмотрим позже.

Перед освобождением памяти, занимаемой блокировкой чтения-записи, нужно вызвать функцию pthread_rwlock_destroy, чтобы освободить все занимаемые блокировкой ресурсы. Функция pthread_rwlock_init размещает все необходимые для блокировки ресурсы, а pthread_rwlock_destroy освобождает их. Если освободить память, занимаемую блокировкой чтения-записи, без предварительного обращения к функции pthread_rwlock_destroy, то все ресурсы, занимаемые блокировкой, будут потеряны для системы.

Чтобы установить блокировку в режиме для чтения, необходимо вызвать функцию pthread_rwlock_rdlock. Чтобы установить блокировку в режиме для записи, необходимо вызвать функцию pthread_rwlock_wrlock. Независимо от того, в каком режиме установлена блокировка чтения-записи, снятие блокировки выполняется функцией pthread_rwlock_unlock.

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Все три функции возвращают 0 в случае успешного завершения, либо код ошибки в случае неудачи.

Реализации могут ограничивать количество блокировок, установленных в режиме совместного использования, поэтому обязательно нужно проверять значение, возвращаемое функцией pthread_rwlock_rdlock. Даже когда функции pthread_rwlock_wrlock и pthread_rwlock_unlock возвращают код ошибки, нет необходимости проверять возвращаемые значения этих функций, если схема наложения блокировок разработана надлежащим образом. Эти функции могут вернуть код ошибки только в том случае, когда блокировка не была инициализирована или когда может возникнуть тупиковая ситуация при попытке повторно установить уже установленную блокировку.

Стандарт *Single UNIX Specification* определяет дополнительные версии примитивов для работы с блокировками, которые могут использоваться для проверки состояния блокировки.

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

Обе функции возвращают 0 в случае успешного завершения, или код ошибки в случае неудачи.

Если блокировка была успешно установлена, эти функции возвращают значение 0. В противном случае они возвращают код ошибки EBUSY. Эти функции могут использоваться в тех случаях, когда невозможно заранее предопределить порядок установки блокировок, чтобы избежать тупиковых ситуаций, которые мы обсуждали ранее.

Рассмотрим пример применения блокировок чтения-записи. Очередь запросов на выполнение заданий защищается единственной блокировкой чтения-записи.

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

struct job {
        struct job *j_next;
        struct job *j_prev;
        pthread_t j_id; /* сообщает, какой поток выполняет это задание */
        /* ... другие поля структуры ... */
};
```

```
struct queue {
        struct job *q_head;
        struct job *q_tail;
        pthread_rwlock_t q_lock;
};


/* Инициализация очереди. */
int queue_init(struct queue *qp) {
        int err;
        qp->q_head = NULL;
        qp->q_tail = NULL;
        err = pthread_rwlock_init(&qp->q_lock, NULL);
        if (err != 0)
                return(err);
        /* ... продолжение инициализации ... */
        return(0);
}

/* Добавить задание в начало очереди. */
void job_insert(struct queue *qp, struct job *jp) {
        pthread_rwlock_wrlock(&qp->q_lock);
        jp->j_next = qp->q_head;
        jp->j_prev = NULL;
        if (qp->q_head != NULL)
                qp->q_head->j_prev = jp;
        else
                qp->q_tail = jp; /* список был пуст */
        qp->q_head = jp;
        pthread_rwlock_unlock(&qp->q_lock);
}

/* Добавить задание в конец очереди. */
void job_append(struct queue *qp, struct job *jp) {
        pthread_rwlock_wrlock(&qp->q_lock);
        jp->j_next = NULL;
        jp->j_prev = qp->q_tail;
        if (qp->q_tail != NULL)
                qp->q_tail->j_next = jp;
        else
                qp->q_head = jp;  /* список был пуст */
        qp->q_tail = jp;
        pthread_rwlock_unlock(&qp->q_lock);
}


/* Удалить задание из очереди. */
void job_remove(struct queue *qp, struct job *jp) {
```

```
        pthread_rwlock_wrlock(&qp->q_lock);
        if (jp == qp->q_head) {
                qp->q_head = jp->j_next;
                if (qp->q_tail == jp)
                        qp->q_tail = NULL;
        } else if (jp == qp->q_tail) {
                qp->q_tail = jp->j_prev;
                if (qp->q_head == jp)
                        qp->q_head = NULL;
        } else {
                jp->j_prev->j_next = jp->j_next;
                jp->j_next->j_prev = jp->j_prev;
        }
        pthread_rwlock_unlock(&qp->q_lock);
}


/* Найти задание для потока с заданным идентификатором. */
struct job * job_find(struct queue *qp, pthread_t id) {
        struct job *jp;

        if (pthread_rwlock_rdlock(&qp->q_lock) != 0)
                return(NULL);
        for (jp = qp->q_head; jp != NULL; jp = jp->j_next)
                if (pthread_equal(jp->j_id, id))
                break;
        pthread_rwlock_unlock(&qp->q_lock);
        return(jp);
}
```

В этом примере блокировка чтения-записи очереди устанавливается в режиме для записи только тогда, когда необходимо добавить новое задание в очередь или удалить задание из очереди. Когда нужно выполнить поиск задания в очереди, мы устанавливаем блокировку в режиме для чтения, допуская возможность поиска заданий несколькими рабочими потоками одновременно. В данном случае использование блокировки чтения-записи дает прирост производительности.

Рабочие потоки извлекают из очереди только те задания, которые соответствуют их идентификаторам. Поскольку сама структура с заданием используется только одним потоком, для организации доступа к ней не требуется дополнительных блокировок.

Атрибуты мьютексов

Работа с атрибутами мьютексов аналогична работе с рассмотренными на прошлом занятии атрибутами потоков. Для инициализации структуры pthread_mutexattr_t используется функция pthread_mutexattr_init, а для ее разрушения - pthread_mutexattr_destroy.

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

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Обе функции возвращают 0 в случае успешного завершения, или код ошибки в случае неудачи.

Функция pthread_mutexattr_init инициализирует структуру pthread_mutexattr_t значениями атрибутов мьютексов по умолчанию. Для нас представляют интерес два атрибута: process-shared и type. Согласно стандарту *POSIX* атрибут process-shared является необязательным - если этот атрибут поддерживается на заданной платформе, то будет определен и символ _POSIX_THREAD_PROCESS_SHARED. Проверку во время выполнения можно произвести с помощью функции sysconf, передав ей параметр _SC_THREAD_PROCESS_SHARED. Хотя *POSIX*-совместимые системы не обязаны поддерживать этот атрибут, стандарт *Single UNIX Specification* требует обязательной поддержки этого атрибута в операционных системах, отвечающих требованиям *XSI*.

Внутри процесса множество потоков могут иметь доступ к одному и тому же объекту синхронизации. Такое поведение определено по умолчанию. В этом случае атрибут мьютекса process-shared имеет значение PTHREAD_PROCESS_PRIVATE.

Существуют механизмы, позволяющие независимым друг от друга процессам отображать одну и ту же область памяти в свои собственные адресные пространства. Доступ к данным, совместно используемым несколькими процессами, обычно требует синхронизации, так же как и доступ к совместно используемым данным из нескольких потоков. Если атрибут process-shared установлен в значение PTHREAD_PROCESS_SHARED, следовательно, мьютекс размещается в области памяти, разделяемой между несколькими процессами, и может использоваться для синхронизации этих процессов.

Получить значение атрибута process-shared структуры pthread_mutexattr_t можно с помощью функции pthread_mutexattr_getpshared. Чтобы изменить значение этого атрибута, следует использовать функцию pthread_mutexattr_setpshared.

```
#include <pthread.h>
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr,
                                 int * restrict pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

Обе функции возвращают 0 в случае успешного завершения, или код ошибки в случае неудачи.

Атрибут process-shared позволяет библиотеке pthread выбрать более оптимальную реализацию мьютекса в случае, когда этот атрибут имеет значение PTHREAD_PROCESS_PRIVATE, которое принимается по умолчанию для многопоточных приложений. Таким образом можно ограничить использование более ресурсоемкой реализации случаем, когда мьютексы совместно используются несколькими процессами.

Атрибут type позволяет указать тип мьютекса. Стандарт *POSIX* определяет четыре типа. Тип PTHREAD_MUTEX_NORMAL - это стандартный мьютекс, который не производит дополнительных проверок на наличие ошибок или тупиковых ситуаций. Мьютексы типа PTHREAD_MUTEX_ERRORCHECK производят проверку наличия ошибок.

Мьютексы типа PTHREAD_MUTEX_RECURSIVE позволяют одному и тому же потоку

многократно запирать мьютекс, не отпирая его. Рекурсивные мьютексы содержат счетчик, в котором хранится количество запираний мьютекса. Мьютекс будет освобожден только тогда, когда количество отпираний совпадет с количеством запираний. Так, если имеется рекурсивный мьютекс, который был заперт дважды, и вы отперли его один раз, то мьютекс все равно останется заблокированным до тех пор, пока вы не отопрете его второй раз.

И, наконец, мьютексы типа PTHREAD_MUTEX_DEFAULT могут использоваться для назначения семантики мьютекса по умолчанию. Реализации могут самостоятельно определять, какому из трех предыдущих типов соответствует данный тип мьютекса. Так, например, в ОС *Linux* этот тип мьютекса соответствует типу PTHREAD_MUTEX_NORMAL.

Поведение мьютексов этих четырех типов показано в **табл**. Колонка «*Попытка отпирания другим потоком*» соответствует ситуации, когда производится попытка отпирания мьютекса, запертого другим потоком. Колонка «*Попытка отпирания незапертого мьютекса*» соответствует ситуации, когда поток пытается отпереть незапертый мьютекс, что обычно объясняется ошибкой в алгоритме.

Чтобы получить значение атрибута type, можно использовать функцию pthread_mutexattr_gettype, а чтобы изменить его - функцию pthread_mutexattr_settype.

```
#include <pthread.h>
int pthread_mutexattr_gettype(const pthread_mutexattr_t * restrict attr,
                              int * restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

Обе функции возвращают 0 в случае успешного завершения, или код ошибки в случае неудачи.

Таблица. Поведение мьютексов различного типа

Тип	Повторное запирание без отпирания	Попытка отпирания другим потоком	Попытка отпирания незапертого мьютекса
PTHREAD_MUTEX_NORMAL	Тупиковая ситуация	Не определено	Не определено
PTHREAD_MUTEX_ERRORCHECK	Возвращает код ошибки	Возвращает код ошибки	Возвращает код ошибки
PTHREAD_MUTEX_RECURSIVE	Допускается	Возвращает код ошибки	Возвращает код ошибки
PTHREAD_MUTEX_DEFAULT	Не определено	Не определено	Не определено

Атрибуты блокировок чтения-записи

Блокировки чтения-записи, подобно мьютексам, также имеют атрибуты. Для инициализации структуры pthread_rwlockattr_t используется функция pthread_rwlockattr_init, а для ее разрушения - функция pthread_rwlockattr_ destroy.

```
#include <pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Обе функции возвращают 0 в случае успешного завершения, или код ошибки в случае неудачи.

Единственный атрибут, поддерживаемый блокировками чтения-записи, - это атрибут process-shared, полностью идентичный аналогичному атрибуту мьютексов. Как и в случае с мьютексами, для обслуживания атрибута process-shared блокировок чтения-записи используется пара функций: pthread_rwlockattr_getpshared и pthread_rwlockattr_setpshared.

```
#include <pthread.h>
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
                                  int * restrict pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

Обе функции возвращают 0 в случае успешного завершения, или код ошибки в случае неудачи.

Хотя стандарт *POSIX* определяет всего один атрибут для блокировок чтения-записи, тем не менее реализации могут свободно добавлять собственные нестандартные атрибуты.

Более полную информацию о рассмотренных примитивах синхронизации можно получить на сайте документации *man*-страниц (см. функции pthread_...):

http://man7.org/linux/man-pages/dir_all_alphabetic.html#letter_p