# Лабораторная работа №6
# «Потоки. Управление»

На данном занятии необходимо продолжить изучение потоков исполнения и познакомиться с дополнительными возможностями управления: рассмотреть атрибуты потоков, изучить способ отмены потока, познакомиться с потоковыми данными и понять назначение обработчиков очистки.

## *Основные задания*

### Задание №1 ( Атрибуты потока )

Напишите программу, которая с помощью командной строки получает параметр — количество создаваемых потоков. Из основной функции вызывается функция, куда передается рабочий массив, в которой создается заданное количество расчетных потоков – потомков и ожидается завершение их работы. (В упрощенном варианте можно все это сделать в функции main.) Каждый расчетный поток получает в качестве аргумента номер потока выполнения (отсчет с 0), засыпает на это количество секунд и генерирует псевдослучайное число в диапазоне $[1,10]$. Затем, полученное псевдослучайное число заносится в глобальный массив целых чисел в ячейку, соответствующую номеру потока, а поток засыпает на это сгенерированное количество секунд. Параллельно с расчетными потоками запускается отсоединенный поток, который постоянно с периодом 1 секунда выводит на экран сообщение, отображающее значения, хранящиеся в массиве. Когда массив станет полностью заполненным, отсоединенный поток завершает свою работу. Предусмотреть информационные сообщения, поясняющие работу программы.

### Задание №2 ( Асинхронно отменяемый поток )

Напишите программу, которая с помощью командной строки получает параметр — время задержки. В основной программе создается поток – потомок и ожидается завершение его работы. Поток–потомок устанавливается в режим асинхронно отменяемого потока. Поток выводит строку с текстом (включающим номер итерации) бесконечное количество раз и «засыпает» на секунду после каждого вывода. После создания потока основной поток засыпает на заданное количество секунд (время задержки) и пытается отменить работающий поток. Затем анализируется статус завершения потока и выводится сообщение о том, как же завершился поток — в результате отмены или обычным образом.

### Задание №3 ( Не отменяемый поток )

Напишите программу, которая с помощью командной строки получает параметр — время задержки. В основной программе создается поток – потомок и ожидается завершение его работы. Поток–потомок устанавливается в не отменяемое состояние. Поток выводит строку с текстом (включающим номер итерации) заданное количество раз (оно равно удвоенному времени задержки) и «засыпает» на секунду после каждого вывода. После создания потока основной поток засыпает на заданное количество секунд (время задержки) и пытается отменить работающий поток. Затем анализируется статус завершения потока и выводится сообщение о том, как же завершился поток — в результате отмены или обычным образом.

## Задание №4 ( Синхронно отменяемый поток )

Напишите программу, которая с помощью командной строки получает параметр — время задержки. В основной программе создается поток – потомок и ожидается завершение его работы. Данный поток – потомок устанавливается в режим синхронно отменяемого потока. В нем вычисляется значение числа $\pi$ по формуле Лейбница:

$$\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}-\ldots$$

Вычисления продолжаются до $n=100000$. Текущее значение числа $\pi$ выводится в стандартный поток вывода. Вычисление очередного слагаемого и добавление его в сумму происходит в не отменяемой секции. Сразу после этого формируется возможная точка выхода.

Основной поток ожидает заданное при запуске программы количество секунд и посылает команду на отмену потока. Затем анализируется: в результате чего завершился поток. Если в результате отмены — выводится сообщение. Если в поток досчитал до конца, то выводится результат.

## Задание №5 ( Потоковые данные )

Напишите программу, которая с помощью командной строки получает параметр — количество создаваемых потоков. В основной программе (для этого можно сделать отдельную функцию) создается заданное количество потоков – потомков и ожидается завершение их работы. Каждый поток выводит в стандартный поток вывода случайное количество строк, со случайными числами. Случайные числа генерируются из заданного диапазона. Каждая строка включает в себя идентификатор потока, текстовое сообщение и сгенерированное псевдослучайное число. Неизменяемая часть строки хранится в области потоковых данных для каждого потока. Используйте функцию sleep, чтобы сделать работу программы более наглядной.

## Задание №6 ( Обработчики очистки )

Модифицируйте программу из **Задания №2** так, чтобы поток – потомок перед своим завершением выводил в стандартный поток ошибок сообщение об этом с данными, полученными из завершаемого потока. Используйте функции pthread_cleanup_push, pthread_cleanup_pop.

## *Рекомендации по выполнению*

## Атрибуты потока

На прошлом практическом занятии при вызове функции pthread_create в качестве второго аргумента было указано значение NULL. Данный аргумент предназначен для передачи указателя на объект *атрибутов потока*. Эти потоковые атрибуты предназначены для тонкой настройки поведения отдельных потоков. Если указатель равен NULL, то создаваемый поток исполнения настраивается на основании стандартных атрибутов. Для указания нестандартных атрибутов необходимо создать экземпляр структуры pthread_attr_t и передать функции pthread_create указатель на него.

Структура pthread_attr_t используется для того, чтобы изменить значения атрибутов по умолчанию и связать эти атрибуты с создаваемым потоком. Для инициализации структуры pthread_attr_t следует использовать функцию pthread_init_attr. После вызова этой

функции структура pthread_attr_t будет заполнена значениями атрибутов по умолчанию. Эти значения зависят от конкретной реализации. Для разрушения структуры pthread_attr_t используется функция pthread_ attr_destroy. Функция pthread_attr_destroy корректно освободит все ресурсы, которые были использованы функцией pthread_attr_init. Кроме того, pthread_attr_destroy заполнит структуру ошибочными значениями, чтобы функция pthread_create возвращала ошибку при случайном использовании такой структуры.

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Обе функции возвращают 0 в случае успешного вызова и код ошибки в случае неудачного вызова.

Структура pthread_attr_t непрозрачна для приложения. Это означает, что приложение ничего не должно знать о внутреннем устройстве структуры, что повышает переносимость создаваемых приложений. Поэтому, согласно стандарту *POSIX* определяются отдельные функции для получения и изменения значений каждого атрибута.

Для большинства приложений *LINUX* интерес представляет единственный атрибут detachstate — *статус отсоединения потока*. Поток может быть создан как *ожидаемый* (*JOINABLE*) (таким поток создается по умолчанию) или *отсоединенный* (*DETACHED*). Ожидаемый поток, подобно процессу, после своего завершения не удаляется автоматически операционной системой *LINUX*. Код его завершения хранится где-то в системе, пока какой-нибудь другой поток не вызовет функцию pthread_join(), чтобы запросить это значение. Только тогда ресурсы потока считаются освобожденными. В отличие от ожидаемого потока, отсоединенный поток, завершившись, сразу уничтожается. Другие потоки не могут вызвать по отношению к нему функцию pthread_join() и получить возвращаемое им значение.

Таким образом, если заранее известно, что в код завершения потока дальнейшем не потребуется, и не нужно будет специально ждать его завершения, то можно сразу же создать и запустить поток в отсоединенном состоянии. Для этого нужно изменить значение атрибута detachstate в структуре pthread_attr_t. Для этого предназначена функция pthread_attr_setdetachstate, которой передается одно из двух возможных значений — PTHREAD_CREATE_DETACHED, чтобы запустить поток в отсоединенном состоянии, или PTHREAD_CREATE_JOINABLE, чтобы запустить поток в нормальном состоянии, в котором приложение сможет получить код завершения потока.

Чтобы получить текущее состояние атрибута detachstate, можно воспользоваться функцией pthread_attr_getdetachstate. По адресу, который передается во втором аргументе, функция запишет одно из двух возможных значений: PTHREAD_CREATE_DETACHED или PTHREAD_CREATE_JOINABLE, в зависимости от значения атрибута в структуре pthread_attr_t.

```
#include <pthread.h>
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

Обе функции возвращают либо значение 0 в случае успешного вызова, либо код ошибки в случае неудачи. Первый аргумент функций — указатель на объект атрибутов потока, второй — требуемый статус.

Таким образом, для задания собственного атрибута потока, следует выполнить такую последовательность действий:

1. Создайте объект типа pthread_attr_t.

2. Вызовите функцию pthread_attr_init(), передав ей указатель на объект. Эта функция присваивает неинициализированным атрибутам стандартные значения.
3. Запишите в объект требуемые значения атрибутов.
4. Передайте указатель на объект в функцию pthread_create().
5. Вызовите функцию pthread_attr_destroy(), чтобы удалить объект из памяти. Сама переменная pthread_attr_t не удаляется; ее можно проиициалиаировать повторно с помощью функции pthread_attr_init().

Один и тот же объект может быть использован для запуска нескольких потоков, причем нет необходимости хранить объект после того, как поток был создан.

Рассмотрим фрагмент программы, создающей отсоединенный поток:

```
#include <pthread.h>

void* thread_function(void* thread_arg) {
	/* Потоковая функция... */
}

int main() {
	pthread_attr_t attr;
	pthread_t thread;

	pthread_attr_init (&attr);
	pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
	pthread_create(&thread, &attr, &thread_function, NULL);
	pthread_attr_destroy (&attr);
	/* Выполнение задания в основном потоке... */
	/* Уже нет необходимости ожидать завершения потока-потомка */
	return 0;
}
```

Следует отметить, что даже если поток был создан ожидаемым, его позднее можно сделать отсоединенным. Для этого нужно знать его идентификатор и вызвать функцию:

```
int pthread_detach(pthread_t thread)
```

Функция возвращает 0 в случае успешного вызова и код ошибки в случае неудачного вызова. Обратное преобразование потока (из отсоединенного состояния в ожидаемое) невозможно.

## Отмена потока

Обычно поток завершается при выходе из потоковой функции или вследствие вызова функции pthread_exit(). Кроме этого есть еще одна возможность преждевременно завершить работу потока. Любой поток может послать другому потоку запрос на завершение. Это называется *отменой*, или *принудительным завершением* потока. Для этого предусмотрена функция pthread_cancel(), имеющая следующий прототип:

```
int pthread_cancel(pthread_t THREAD_ID);
```

Функция pthread_cancel() возвращает 0 при удачном завершении, ненулевое

значение сигнализирует об ошибке. По умолчанию вызов функции pthread_cancel заставляет указанный поток вести себя так, как будто бы он вызвал функцию pthread_exit с аргументом PTHREAD_CANCELED. Однако поток может отвергнуть запрос или как-то иначе отреагировать на него.

Таким образом, чтобы отменить поток, следует вызвать функцию pthread_cancel(), передав ей идентификатор требуемого потока. Функция pthread_cancel() возвращает управление сразу, но это не означает немедленного завершения потока. Основная задача рассматриваемой функции — доставка потоку запроса на удаление, а не фактическое его удаление. Далее следует дождаться завершения потока, вызвав функцию pthread_join(). Для обычных потоков это нужно делать обязательно, чтобы освободить занятые потоком ресурсов. Если же поток является отсоединенным, то ожидать его завершения не нужно. Отмененный поток возвращает специальное значение PTHREAD_CANCELED.

Во многих случаях поток выполняет код, который нельзя просто взять и прервать. Например, поток может выделить какие-то ресурсы, поработать с ними, а затем удалить. Если отмена потока произойдет где-то посередине, освободить занятые ресурсы станет невозможно, вследствие чего они окажутся потерянными для системы. Чтобы учесть эту ситуацию, поток должен решить, где и когда он может быть отменен.

С точки зрения возможности отмены поток находится в одном из трех состояний.

- *Асинхронно отменяемый*. Такой поток можно отменить в любой точке его выполнения.
- *Синхронно отменяемый*. Поток можно отменить, но не везде. Запрос на отмену помещается в очередь, и поток отменяется только по достижении определенной точки.
- *Не отменяемый*. Попытки отменить поток игнорируются.

Первоначально поток является синхронно отменяемым.

Для управления состоянием потока предназначены два атрибута потоков, которые не входят в состав структуры pthread_attr_t. Это *атрибут возможности принудительного завершения потока* (*cancelability state*) и *атрибут типа принудительного завершения* (*cancelability type*). Эти атрибуты определяют поведение потока в ответ на вызов функции pthread_ cancel.

Атрибут *cancelability state* может иметь два значения: PTHREAD_CANCEL_ENABLE и PTHREAD_CANCEL_DISABLE. Поток может изменить значение этого атрибута при помощи вызова функции pthread_setcancelstate.

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate);
```

Функция возвращает либо 0 в случае успешного завершения, либо код ошибки в случае неудачи.

В одной атомарной операции функция pthread_setcancelstate изменяет значение атрибута *cancelability state* в соответствии со значением аргумента state и сохраняет прежнее значение атрибута по адресу, который передается в аргументе oldstate.

Следует отметить, что функция pthread_cancel не ждет, пока поток завершит работу. По умолчанию поток продолжает работу после вызова этой функции, пока не достигнет точки выхода. Точка выхода — это место, где поток может обнаружить запрос на принудительное завершение и откликнуться на него. Стандарт *POSIX* назначает точками выхода целый ряд функций, полный список которых можно найти в документации по функции pthread_cancel.

В момент запуска потока значение его атрибута *cancelability state* устанавливается равным PTHREAD_CANCEL_ENABLE. Если поток установит значение этого атрибута

равным PTHREAD_CANCEL_DISABLE, то вызов функции pthread_cancel не будет приводить к завершению потока. Вместо этого запрос на принудительное завершение становится в режим ожидания. Когда поток опять разрешит возможность принудительного завершения, он откликнется на ожидающий запрос в ближайшей точке выхода.

Если поток достаточно продолжительное время не обращается к функциям, которые являются стандартными точками выхода (например, при выполнении объемных вычислений), то можно определить свою собственную точку выхода с помощью функции pthread_testcancel.

```
#include <pthread.h>
void pthread_testcancel(void);
```

Функция pthread_testcancel проверяет наличие ожидающего запроса на принудительное завершение, и если таковой имеется и при этом атрибут *cancelability state* разрешает принудительное завершение, то поток завершит свою работу. Но если возможность принудительного завершения потока запрещена, вызов функции pthread_testcancel не оказывает никакого влияния.

По умолчанию для потока устанавливается тип принудительного завершения, известный как *отложенный выход*. После вызова функции pthread_cancel поток не завершается немедленно, он продолжает работу до тех пор, пока не достигнет ближайшей точки выхода. Изменить тип принудительного завершения можно с помощью функции pthread_setcanceltype.

```
#include <pthread.h>
int pthread_setcanceltype(int type, int *oldtype);
```

Функция возвращает либо 0 в случае успешного завершения, либо код ошибки в случае неудачи.

Аргумент type может содержать либо значение PTHREAD_CANCEL_DEFERRED (*отложенное завершение потока*), либо PTHREAD_CANCEL_ASYNCHRONOUS (*асинхронное завершение потока*). Функция pthread_setcanceltype устанавливает значение атрибута в соответствии с аргументом type и возвращает предыдущее значение атрибута в переменной, на которую указывает аргумент oldtype.

Асинхронное завершение потока отличается от отложенного тем, что поток может быть принудительно завершен в любой момент времени. В этом случае поток будет завершен вне зависимости от того, достиг он точки выхода или нет.

Рассмотрим отдельно различные виды потоков.

### *Синхронные и асинхронные потоки*

Асинхронно отменяемый поток «свободен» в любое время. Синхронно отменяемый поток, наоборот, бывает «свободным», только когда ему «удобно». Соответствующие места в программе называются *точками выхода*. Запрос на отмену помещается в очередь и находится в ней до тех пор, пока поток не достигнет следующей точки выхода.

Чтобы сделать поток асинхронно отменяемым, воспользуйтесь функцией pthread_setcanceltype(). Эта функция влияет на тот поток, в котором она была вызвана. Первый ее аргумент должен быть PTHREAD_CANCEL_ASYNCHRONOUS в случае асинхронных потоков и PTHREAD_CANCEL_DEFERRED — в случае синхронных потоков. Второй аргумент— это указатель на переменную, в которую записывается предыдущее состояние потока. Если предыдущее состояние потока сохранять не нужно, то в качестве соответствующего параметра нужно указать NULL.

Как уже было сказано, точки выхода создается с помощью функции pthread_testcancel(). Все, что она делает — это обрабатывает отложенный запрос на отмену в синхронном потоке. Ее следует периодически вызывать в потоковой функции в ходе длительных вычислений, там, где поток можно завершить без риска потери ресурсов или других побочных эффектов.

Как уже было сказано, некоторые функции неявно создают точки отмены. О них можно узнать на *man*-странице, посвященной функции pthread_cancel(). Эти функции могут быть вызваны в других функциях, которые, таким образом, станут точками выхода.

### *Не отменяемые потоки*

Поток может отказаться удаляться, вызвав функцию pthread_setcancelstate(). Как и в случае функции pthread_setcanceltype(), это оказывает влияние только на вызывающий поток. Первый аргумент функции должен быть PTHREAD_CANCEL_DISABLE, если нужно запретить отмену потока, и PTHREAD_CANCEL_ENABLE в противном случае. Второй аргумент —это указатель на переменную, в которую записывается предыдущее состояние потока. Если предыдущее состояние потока сохранять не нужно, то в качестве соответствующего параметра нужно указать NULL.

Вот как можно запретить отмену потока:

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
```

Рассмотрим фрагмент шаблона программы:

```
int old_cancel_state;
/* Начало не отменяемой области */
	pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_cancel_state);
	/* Обязательные действия*/
/* Конец не отменяемой области */
	pthread_setcancelstate(old_cancel_state, NULL);
```

По окончании не отменяемой области восстанавливается предыдущее состояние потока, а не режим PTHREAD_CANCEL_ENABLE. Это позволяет безопасно вызывать функцию process_transaction() из другой критической секции.

### *Пример, демонстрирующий отмену потока.*

Функцию pthread_cancel() можно вызывать из любого места программы, и на момент возврата из pthread_join() часто необходимо знать, как завершился поток: своим естественным путем или был отменен другим потоком. Эта информация очень нужна, когда вызывающая сторона ждет от потока возвращаемых данных. Чтобы узнать, был ли поток отменен, нужно прочитать возвращаемое этим потоком значение и сравнить его с константой PTHREAD_CANCELED. Эта константа является не целым числом, а безтиповым указателем. Рассмотрим пример, показывающий, как выполняется такая проверка.

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

void * any_func(void * arg) {
	while(1) {
		fprintf(stderr, ".");
```

```
                sleep(1);
        }
        return NULL;
}

int main(void) {
        pthread_t thread;
        void * result;
        if (pthread_create(&thread, NULL,
                        &any_func, NULL) != 0) {
                fprintf(stderr, "Error\n");
                return 1;
        }
        sleep(5);
        pthread_cancel(thread);

        if (!pthread_equal(pthread_self(), thread))
                pthread_join(thread, &result) ;

        if (result == PTHREAD_CANCELED)
                fprintf(stderr, "Canceled\n");

        return 0;
}
```

## Потоковые данные

### *Однократная инициализация*

Иногда, в многопотоковом приложении необходимо гарантированно выполнить какое-то действие (обычно инициализацию) только один раз, независимо от того, сколько потоков создано. Если потоки создаются из основной программы, то это, обычно, легко сделать — инициализация просто выполняется перед созданием тех потоков, которые от нее зависят. Однако, для библиотечной функции обычно это сделать невозможно, т. к. программа может уже создать потоки исполнения перед первым вызовом функции библиотеки. Поэтому, такой библиотечной функции необходим какой-то стандартный метод выполнения однократной инициализации, когда он первый раз вызывается из любого уже работающего потока. Такую однократную инициализацию можно выполнить с помощью функции:

```
#include <pthread.h>
int pthread_once (pthread_once_t * once_control, void (* init) (void));
```

Функция возвращает 0 в случае успешного завершения и натуральное число — номер ошибки в случае ошибки.

Функция pthread_once() использует аргумент once_control типа pthread_once_t, для того, чтобы гарантированно вызвать один раз функцию, переданную во втором аргументе init. Таким образом, эта функция будет вызвана тольно один раз, независимо от того, сколько раз, и из какого количества потоков выполнялся вызов функции pthread_once(). Функция init имеет следующий вид:

```
void init(void) {
    . . . .
}
```

Аргумент once_control является указателем на переменную, которая должна быть статически инициализирована специальным значением PTHREAD_ONCE_INIT:

```
pthread_once_t once_var = PTHREAD_ONCE_INIT;
```

Первый вызов функции pthread_once(), в котором указан адрес переменной типа pthread_once_t, изменяет значение этой переменной таким образом, что все последующие вызовы pthread_once() уже не будут вызывать функцию init.

Очень часто функция pthread_once() используется для работы с собственными данными потока.

### *Собственные потоковые данные*

В отличие от процессов, все потоки программы делят общее адресное пространство. Это означает, что если один поток модифицирует ячейку памяти (например, глобальную переменную), то это изменение отразится на всех остальных потоках. Таким образом, потоки могут работать с одними и теми же данными, не используя механизмы межзадачного взаимодействия.

Тем не менее у каждого потока — свой собственный стек вызова. Это позволяет всем потокам выполнять разный код, а также вызывать функции традиционным способом. При каждом вызове функции в любом потоке создается отдельный набор локальных переменных, которые сохраняются в стеке этого потока.

В случае многопотоковой программы можно использовать не только локальные и глобальные данные, но также и еще один, специальный класс, характерный только для многопотоковых программ: *собственные данные потоков* (*thread-specific data*). Эти данные очень похожи на глобальные, за исключением того, что они являются приватными, собственными данными каждого потока.

Такие «локально глобальные» данные потока нужны для того, чтобы каждый поток мог обладать некоторым набором данных, принадлежащих ему одному, и не беспокоиться по поводу синхронизации при работе с этими данными. С этой целью операционная система *LINUX* предоставляет потокам *область потоковых данных (thread-specific data area)*. Переменные, сохраняемые в этой области, дублируются для каждого потока, что позволяет потокам свободно работать с ними, не мешая друг другу. Доступ к потоковым данным нельзя получить с помощью ссылок на обычные переменные, ведь у потоков общее адресное пространство. В *LINUX* имеются специальные функции для чтения и записи значений, хранящихся в области потоковых данных.

Можно создать сколько угодно потоковых переменных, при этом все они должны иметь тип void*. Ссылка на каждую переменную осуществляется по *ключу*. Перед размещением локальных данных потока мы должны создать *ключ*, который будет идентифицировать данные. Этот ключ будет использоваться для получения доступа к локальным данным потока. Создается такой ключ вызовом функции pthread_key_create.

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *));
```

Функция возвращает 0 в случае успешного завершения, или код ошибки в случае неудачи.

Созданный ключ сохраняется по адресу keyp. Один и тот же ключ используется различными потоками в процессе, но каждый поток будет ассоциировать с ключом отдельный, собственный набор локальных данных. После создания ключа адрес локальных данных для каждого потока устанавливается равным NULL.

Кроме того, функция pthread_key_create может связать с созданным ключом очистки ключа. Она будет автоматически вызываться при уничтожении потока; ей передается значение ключа, соответствующее данному потоку. Это очень удобно, так как функция очистки вызывается даже в случае отмены потока в произвольной точке. (Если поток завершил работу вызовом функции pthread_exit или возвращает управление из запускающей процедуры в результате нормального завершения автоматически вызывается функция очистки. Но если поток вызывает функцию exit, _exit, _Exit, abort или завершает работу аварийно, то функция очистки не вызывается.) Если потоковая переменная равна NULL, функция очистки не вызывается. Если же такая функция не нужна, задайте в качестве второго параметра функции pthread_key_create() значение NULL.

Поток может создать необходимое количество ключей. Каждый ключ может быть ассоциирован с функцией очистки. Это могут быть отдельные функции для каждого из ключей или, наоборот, все ключи могут быть ассоциированы с одной и той же функцией очистки. Каждая реализация операционной системы может накладывать свои ограничения на количество ключей, создаваемых процессом (параметр PTHREAD_KEYS_MAX).

Порядок вызова функции очистки при завершении потока зависит от реализации. В функции очистки допускается вызов функций, которые могут создавать новые локальные данные потока и ассоциировать их с ключом. После вызова всех функций очистки система проверяет, не сохранились ли какие-либо непустые указатели на локальные данные потока, и если таковые будут обнаружены, функции очистки будут вызваны снова. Этот процесс будет повторяться снова и снова, пока не будут обнулены все указатели на локальные данные или не будет достигнуто максимально возможное количество итераций PTHREAD_DESTRUCTOR_ITERATIONS.

Можно разорвать связь ключа с локальными данными для всех потоков, вызвав функцию pthread_key_delete.

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key):
```

Функция возвращает 0 в случае успешного завершения, или код ошибки в случае неудачи.

Следует отметить, что вызов функции pthread_key_delete не приводит к вызову функции очистки, ассоциированной с данным ключом. Чтобы освободить память, занимаемую локальными данными потока, следует предусмотреть все необходимые действия в самом приложении.

После того как ключ создан, он должен быть ассоциирован с локальными данными потока с помощью функции pthread_setspecific. Чтобы по заданному ключу получить адрес области памяти с локальными данными потока, следует обратиться к функции pthread_getspecific.

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key. const void *value);
```

Функция возвращает 0 в случае успешного завершения, или код ошибки в случае неудачи.

```
void *pthread_getspecific(pthread_key_t key);
```

Функция возвращает указатель на область памяти с локальными данными или NULL, если ключ не ассоциирован с локальными данными

Если с ключом не были ассоциированы локальные данные потока, то функция pthread_getspecific будет возвращать значение NULL. Мы можем использовать это обстоятельство, чтобы определить, следует ли вызывать функцию pthread_setspecific.

Схема программы, использующей потоковые данные.

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

/* The key used to associate a log file pointer with each thread. */
static pthread_key_t thread_log_key;

/* Write MESSAGE to the log file for the current thread. */
void write_to_thread_log (const char* message) {
	FILE* thread_log = (FILE*) pthread_getspecific(thread_log_key);
	fprintf (thread_log, "%s\n", message);
}

/* Close the log file pointer THREAD_LOG. */
void close_thread_log(void* thread_log) {
	fclose((FILE*)thread_log);
}

void* thread_function (void* args) {
	char thread_log_filename[20];
	FILE* thread_log;

	/* Generate the filename for this thread's log file. */
	sprintf(thread_log_filename, "thread%d.log", (int) pthread_self());
	/* Open the log file. */
	thread_log = fopen (thread_log_filename, "w");
	/* Store the file pointer in thread-specific data under thread_log_key. */
	pthread_setspecific(thread_log_key, thread_log);
	write_to_thread_log("Thread starting.");
	/* Do work here... */
	return NULL;
}

int main() {
	int i;
	pthread_t threads[5];

	/* Create a key to associate thread log file pointers in
	thread-specific data. Use close_thread_log to clean up the file
	pointers. */
	pthread_key_create(&thread_log_key, close_thread_log);
	/* Create threads to do the work. */
	for (i = 0; i < 5; ++i)
```

```
        pthread_create(&(threads[i]), NULL, thread_function, NULL);
    /* Wait for all threads to finish. */
    for (i = 0; i < 5; ++i)
        pthread_join(threads[i], NULL);
    return 0;
}
```

Обратите внимание на то, что в функции thread_function() не нужно закрывать журнальный файл. Просто когда создавался ключ, функция close_thread_log() была назначена функцией очистки данного ключа. Когда бы поток ни завершился, операционная система *LINUX* вызовет эту функцию, передав ей значение ключа, соответствующее данному потоку. В функции close_thread_log() и происходит закрытие файла.

## Обработчики очистки

Функции очистки ключей гарантируют, что в случае завершения или отмены потока не произойдет потери ресурсов. Но иногда возникает необходимость в создании функции, которая будет связана не с ключом, дублируемым между потоками, а с обычным ресурсом. Такая функция называется *обработчиком очистки* (*cleanup handlers*). Для этого поток назначает функцию, которая будет автоматически вызвана в момент его завершения, примерно так же, как это делается для процессов с помощью функции atexit, которая регистрирует функции, запускаемые при завершении процесса. Поток может зарегистрировать несколько таких функций обработки выхода. Обработчики заносятся в стек — это означает, что они будут вызываться в порядке, обратном порядку их регистрации.

```
#include <pthread.h>
void pthread_cleanup_push(void (*rtn)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

Функция pthread_cleanup_push регистрирует функцию rtn, которая будет вызвана с аргументом arg, когда поток выполнит одно из следующих действий:

- Вызовет функцию pthread_exit;
- Ответит на запрос о принудительном завершении
- Вызовет функцию pthread_cleanup_pop с ненулевым аргументом execute.

Если аргумент execute имеет значение 0, то функция обработки выхода из потока вызываться не будет. Если он не равен нулю, при отмене регистрации выполняется операция очистки. В любом случае функция pthread_cleanup_pop удаляет функцию-обработчик, зарегистрированную последним обращением к функции pthread_cleanup_push.

Есть важное ограничение, связанное с этими функциями. Оно заключается в том, что данные функции могут быть реализованы в виде макроопределений и тогда они должны использоваться в паре, в пределах одной и той же области видимости в потоке. Макроопределение функции pthread_cleanup_push может включать в себя символ {, и тогда парная ей скобка } будет находиться в макроопределении pthread_cleanup_pop.

Рассмотрим фрагмент программы, в котором обработчик очистки применяется для удаления динамического буфера при завершении потока.

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

/* Allocate a temporary buffer. */
```

```
void* allocate_buffer(size_t size) {
	return malloc(size);
}

/* Deallocate a temporary buffer. */
void deallocate_buffer(void* buffer) {
	free(buffer);
}

void do_some_work() {
	/* Allocate a temporary buffer. */
	void* temp_buffer = allocate_buffer(1024);
	/* Register a cleanup handler for this buffer, to deallocate it in
	case the thread exits or is cancelled. */
	pthread_cleanup_push(deallocate_buffer, temp_buffer);

	/* Do some work here that might call pthread_exit or
	might be cancelled... */

	/* Unregister the cleanup handler. Because we pass a nonzero value,
	this actually performs the cleanup by calling
	deallocate_buffer. */
	pthread_cleanup_pop(1);
}
```

В данном случае функции phread_cleanup_pop() передается ненулевой аргумент, поэтому функция очистки deallocate_buffer() вызывается автоматически. В данном простейшем случае можно было в качестве обработчика непосредственно использовать стандартную библиотечную функцию free().