Лабораторное занятие №7
«Потоки. Основные операции»

На данном занятии необходимо познакомиться с таким понятием, как поток исполнения (*thread*) и рассмотреть основные операции: создание и завершение работы потока, передача параметров потоку и возвращение вычисленного результата, ожидание одним потоком завершения другого потока и определение идентификатора потока.

Основные задания

Задание №1 (Создание потока)

Напишите программу, которая создает новый поток исполнения. При создании используйте атрибуты потока по умолчанию. Первоначальный и вновь созданный поток должны вывести в стандартный поток вывода десять строк текста (Родительский поток выводит строки вида «Main Thread. Iteration: 1», новый поток: «Child Thread. Iteration: 1»). После вывода строки каждый поток «засыпает» на заданное количество секунд (например, от 0 до 2). Посмотреть как этот параметр влияет на выполнение программы.

Задание №2 (Передача параметров потоку)

Напишите программу, которая создает четыре потока, выполняющие одну и ту же потоковую функцию. Эта функция должна должна принять в качестве параметров имя потока name, базовую строку для вывода str и количество повторений строки num и вывести в стандартный поток вывода последовательность текстовых строк, сформированных по правилу: Thread name. Str i, где i — целое число, определяющее повторение строки. Каждый поток должен получить свой набор параметров.

Задание №3 (Ожидание завершения потока)

Напишите программу, которая создает два новых потока исполнения. Один из потоков — потомков выводит в стандартный поток вывода заданное количество раз заданную строку (аналогичную строкам из **Задания №1**) Другой поток генерирует заданное количество целых псевдослучайных чисел из заданного диапазона и выводит их в стандартный поток вывода в виде, аналогичном выводу первого потока — потомка. Если во время работы данного потока будет получено заданное псевдослучайное число, то поток преждевременно завершает свою работу с выдачей соответствующего сообщения. Основной, родительский поток, дожидается окончания работы всех дочерних потоков и выводит в стандартный поток вывода свое сообщение.

Задание №4 (Возвращение потоком результата)

Напишите программу, которая создает два новых потока исполнения. Один из новых потоков исполнения вычисляет заданное количество первых чисел Каталана, другой новый поток исполнения вычисляет заданное количество первых простых чисел. Затем, вычисленный массив чисел возвращается каждым потоком — потомком в ожидающий завершения основной

поток, который выводит результаты расчетов в стандартный поток вывода.

Дополнение:

Попробовать разные способы возвращения потоком вычисленных значений.

Замечания:

Числа Каталана — это числовая последовательность, встречающаяся во многих задачах комбинаторики; n – е число Каталана может быть вычислено по формуле:

$$C_n = \frac{(2 \cdot n)!}{(n+1)! \cdot n!}, \quad n \geqslant 0 .$$

Приведем несколько первых чисел Каталана ($n = 0, 1, 2, \ldots$)

1 , 1 , 2 , 5 , 14 , 42 , 132 , 429 , 1430 , 4862 , 16796 , 58786 , …

Следует отметить, что числа Каталана достаточно быстро возрастают вместе со своим номером.

Простое число — это натуральное число, большее 1 , имеющее ровно два различных натуральных делителя: единицу и самого себя.

Рекомендации по выполнению

Потоки (*threads*) позволяют одновременно выполнять некоторые действия в контексте одной программы. Потоки работают параллельно. Ядро *Linux* планирует их работу асинхронно, прерывая время от времени каждый из них, чтобы дать шанс на выполнение остальным.

С концептуальной точки зрения поток существует внутри процесса, являясь более мелкой единицей управления программой. При вызове программы ядро *Linux* создает для нее новый процесс, а в нем единственный поток — последовательно выполняющий программный код. Этот поток может создавать дополнительные потоки. Все они находятся в одном процессе, выполняя ту же самую программу. В отличие от создания нового процесса, создание нового потока не требует копирования виртуальной памяти, дескрипторов файлов и т.п. Оба потока — старый и новый — имеют доступ к общей виртуальной памяти, общим дескрипторам файлов и другим системным ресурсам. Если, к примеру, один поток меняет значение переменной, это изменение отражается на другом потоке. В связи с тем что процесс и все его потоки могут одновременно выполнять только одну программу, то если один из потоков вызывает функцию семейства `exec()`, все остальные потоки завершаются, а новая программа может создавать собственные потоки.

Для работы с потоками в операционной системе *Linux* предназначена библиотека *Pthreads*, соответствующая стандарту *POSIX*. Все функции и типы данных этой библиотеки объявлены в заголовочном файле `<pthread.h>`, который следует включать во все исходные файлы программы на языке *Cu*, использующие потоки. Эти функции не входят в стандартную библиотеку языка *Cu*, поэтому при компоновке программы нужно подключить данную библиотеку, для чего следует указать опцию `-lpthread` в командной строке вызова компилятора `gcc`.

Создание потока: pthread_create()

Для создания потока в операционной системе *Linux* следует выполнить несколько действий:

1. Создается функция с особой сигнатурой, которая называется *потоковой функцией*.
2. При помощи функции pthread_create() создается поток, в котором начинает параллельно остальной программе выполняться потоковая функция.
3. Вызывающая сторона продолжает выполнять какие-то действия, не дожидаясь завершения потоковой функции.

Каждому потоку в процессе присваивается идентификатор. Эти идентификаторы существуют локально в рамках текущего процесса. Для их хранения предусмотрен специальный тип pthread_t, который становится доступным при включении в программу заголовочного файла pthread.h.

Для создания потока и запуска потоковой функции используется функция pthread_create(), объявленная в заголовочном файле pthread.h следующим образом:

```
int pthread_create(pthread_t * THREAD_ID, void * ATTR,
                   void *(*THREAD_FUNC)(void*), void * ARG);
```

Функция, pthread_create() принимает четыре аргумента:

- после выхода из функции по адресу, сохнаняемому в THREAD_ID, помещается идентификатор нового потока (если новый поток был успешно создан);
- безтиповый указатель ATTR служит для указания *атрибутов потока*. Если этот аргумент равен NULL, то поток создается с атрибутами по умолчанию. На данном занятии мы не будем передавать потокам специальные атрибуты;
- аргумент PTHREAD_FUNC является указателем на потоковую функцию. Это обычная функция, возвращающая бестиповый указатель (void*) и принимающая бестиповый указатель в качестве единственного аргумента;
- аргумент ARG — это бестиповый указатель, содержащий аргументы, передаваемые потоковой функции; если потоковой функции аргументы не нужны, то в качестве ARG можно указать NULL.

Этот процесс можно представить так: программа при запуске создает один поток, а функция pthread_create() позволяет создавать дополнительные потоки. Это означает, что основную программу следует трактовать как "*родительский поток*". Следует также помнить, что если любой из потоков завершает программу, то все остальные потоки сразу же завершаются. Это отличает потоки от процессов.

Следует отметить, что функции семейства pthread, как правило, возвращают код ошибки в случае неудачи. Они не изменяют значение переменной errno подобно другим функциям *POSIX*. Экземпляр переменной errno для каждого потока предоставляется только для сохранения совместимости с существующими функциями, которые используют эту переменную. Вообще, при работе с потоками принято возвращать код ошибки из функций, что дает возможность локализовать ошибку, а не полагаться на некоторую глобальную переменную, которая могла быть изменена в результате побочного эффекта.

Рассмотрим теперь небольшой пример.

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

void * any_func(void * args) {
        fprintf(stderr, "Hello World\n");
        sleep(5);
        return NULL;
}

int main(void) {
        pthread_t thread;
        int result;

        result = pthread_create(&thread, NULL, &any_func, NULL);

        if (result != 0) {
                fprintf(stderr, "Error\n");
                return 1;
        }

        fprintf(stderr, "Goodbye World\n");
        while(1);

        return 0;
}
```

Функции main() и any_func() работают параллельно, поэтому видимой задержки между выводом двух сообщений не происходит. Бесконечный цикл в этой программе нужен для предотвращения ее преждевременного завершения. Уже было сказано, что при завершении программы одним из потоков все остальные потоки немедленно завершаются. Если бы в нашей программе не было бесконечного цикла, то возврат из функции main() мог бы произойти раньше, чем вывод сообщения "Hello World".

Для передачи данных в поток используется четвертый аргумент функции pthread_create(). Этот указатель автоматически становится аргументом потоковой функции, но поскольку его тип void*, то данные содержатся не в самом аргументе. Он должен лишь указывать на какую-то переменную, структуру или массив. Лучше всего создать для каждой потоковой функции собственную структуру, в которой определялись бы данные, ожидаемые потоковой функцией. С помощью потокового аргумента можно использовать одну и ту же потоковую функцию с разными потоками. Все они будут выполнять один и тот же код, но с разными данными. Следующий пример демонстрирует эту возможность.

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

void * any_func (void * arg) {
        int a = *(int*) arg;
```

```
        fprintf(stderr, "Hello World "
                                "with argument=%d\n", a);
        return NULL;
}

int main(int argc, char ** argv) {
        pthread_t thread;
        int arg, result;

        if (argc < 2) {
                fprintf(stderr, "Too few arguments\n");
                return 1;
        }

        arg = atoi(argv[1]);
        result = pthread_create(&thread, NULL, &any_func, &arg);

        fprintf(stderr, "Goodbye World\n");
        while(1);

        return 0;
}
```

В приведенном примере первый аргумент программы (`argv[1]`) преобразуется в целое число, которое затем передается в потоковую функцию. Если требуется передать в поток несколько аргументов, то их можно разместить в структуре, указатель на которую также передается в потоковую функцию. Этот подход продемонстрирован в следующем примере.

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

struct thread_arg {
        char * str;
        int num;
};

void * any_func(void * arg) {
        struct thread_arg targ =
                *(struct thread_arg *) arg;
        fprintf(stderr, "str=%s\n", targ.str);
        fprintf(stderr, "num=%d\n", targ.num);
        return NULL;
}

int main(void) {
        pthread_t thread;
        int result;
        struct thread_arg targ;
```

```
        targ.str = "Hello World";
        targ.num = 2007;

        result = pthread_create(&thread, NULL, &any_func, &targ);

        while(1);
        return 0;
}
```

В приведенном примере потоковую функцию any_func() можно было бы реализовать и таким образом:

```
void * any_func(void * arg) {
        struct thread_arg * targ =
                (struct thread_arg *) arg;
        fprintf(stderr, "str=%s\n", targ->str);
        fprintf(stderr, "num=%d\n", targ->num);
        return NULL;
}
```

Один процесс может создать произвольное количество потоков (в пределах разумного). Приведенная ниже программа демонстрирует создание двух независимо работающих потоков.

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

void * thread_func1(void * arg) {
        fprintf(stderr, "thread1: %s\n", (char*) arg);
        sleep (5);
        return NULL;
}

void * thread_func2(void * arg) {
        fprintf(stderr, "thread2: %s\n", (char*) arg);
        sleep (5);
        return NULL;
}

int main (void) {
        pthread_t thread1, thread2;
        char * thread1_str = "Thread1";
        char * thread2_str = "Thread2";

        if (pthread_create(&thread1, NULL,
                        &thread_func1, thread1_str) != 0) {
                fprintf(stderr, "Error (thread1)\n");
                return 1;
```

```
        }

        if (pthread_create(&thread2, NULL,
                        &thread_func2, thread2_str) != 0) {
                fprintf(stderr, "Error (thread2)\n");
                return 1;
        }

        fprintf(stderr, "Hello World\n");
        while(1)
        return 0;
}
```

Следует отметить, что идентификаторы потоков и потоковые параметры можно хранить как в отдельных переменных, так и в соответствующий массивах.

Завершение потока: pthread_exit()

Программа обычно завершается либо при помощи возврата из функции main(), либо посредством вызова exit(). Потоки ведут себя аналогично: они могут завершаться или возвратом из потоковой функции, или вызовом специальной функции pthread_exit(), которая объявлена в заголовочном файле pthread.h следующим образом:

```
void pthread_exit(void * RESULT);
```

Функция pthread_exit() особо полезна, если потоковая функция вызывает другие функции. Функция никогда не возвращается в вызвавший ее поток. Аргумент RESULT представляет собой нетипизированный указатель, аналогичный аргументу, передаваемому запускающей процедуре. Этот указатель смогут получить другие потоки процесса, вызвавшие рассмотренную далее функцию pthread_join(). Он, по сути, представляет собой возвращаемое значение потока. Применение аргумента RESULT мы рассмотрим чуть позже. Если же возвращаемое значение потока не нужно, то через аргумент RESULT можно передать пустой указатель. Пока мы и будем указывать NULL. Рассмотрим пример, демонстрирующий работу функции pthread_exit().

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

void print_msg (void) {
        fprintf (stderr, "Hello World\n");
        pthread_exit(NULL);
}

void * any_func (void * arg) {
        print_msg();
        fprintf(stderr, "End of any_func()\n");
        return 0;
}
```

```
int main (void) {
        pthread_t thread;
        if (pthread_create(&thread, NULL,
                        &any_func, NULL)  != 0) {
                fprintf(stderr, "Error\n");
                return 1;
        }
        while(1);
        return 0;
}
```

Ожидание потока: pthread_join()

Функция pthread_join() позволяет синхронизировать потоки. Она объявлена в заголовочном файле pthread.h следующим образом:

```
int pthread_join(pthread_t THREAD_ID, void ** DATA);
```

Эта функция блокирует вызывающий поток до тех пор, пока не завершится поток с идентификатором THREAD_ID. По адресу DATA помещаются данные, возвращаемые потоком через функцию pthread_exit() или через инструкцию return потоковой функции.

При удачном завершении pthread_join() возвращает 0, любое другое значение сигнализирует об ошибке.

Следует отметить, что информация о завершении потока продолжает храниться в текущем процессе до тех пор, пока не будет вызвана pthread_join(). Например, после вызова pthread_create() родительский поток может начать выполнять какие-то своидействия и после завершения этой работы вызвать pthread_join(). Если дочерний поток к тому времени завершится, то вызывающая сторона получит результат и продолжит выполнение. Если же поток-потомок будет еще работать, то потоку-родителю придется подождать его завершения.

Следующий пример показывает, как работает функция pthread_join().

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

#define A_COUNT             15
#define B_COUNT             10

void * print_b(void * arg) {
        int i;
        for (i = 0; i < B_COUNT; i++) {
                fprintf(stderr, "B");
                sleep(1);
        }
        fprintf(stderr, "C");
        return NULL;
```

```
}

int main(void) {
        pthread_t thread;
        int i;

        if (pthread_create(&thread, NULL,
                        &print_b, NULL) != 0) {
                fprintf(stderr, "Error\n");
                return 1;
        }

        for (i =0; i < A_COUNT; i++) {
                fprintf(stderr, "A");
                sleep(1);
        }

        if (pthread_join(thread, NULL) != 0) {
                fprintf(stderr, "Join error\n");
                return 1;
        }

        fprintf(stderr, "D\n");
        return 0;
}
```

Символы A печатает родительский поток. Символы B выводятся порожденным потоком. Когда этот поток завершается, выводится символ C. А когда завершается программа, печатается символ D. Изменяя значения A_COUNT и B_COUNT, можно визуально наблюдать, как работают потоки в различных ситуациях.

Завершение функции pthread_join() с кодом 0 гарантирует завершение потока. После этого можно читать возвращенное потоком значение. Приведенный чуть ниже пример показывает, как это делается.

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

void * any_func(void * arg) {
        int a = *(int *) arg;
        a++;
        return (void *) a;
}

int main (void) {
        pthread_t thread;
        int parg = 2007, pdata;

        if (pthread_create(&thread, NULL,
```

```
                                &any_func, &parg) != 0) {
                        fprintf(stderr, "Error\n");
                        return 1;
                }

                pthread_join(thread, (void *) &pdata);
                printf("%d\n", pdata);

                return 0;
}
```

Функцию pthread_join() может вызывать любой поток, работающий внутри текущего процесса. Рассмотрим пример, иллюстрирующий это.

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

#define A_COUNT             10
#define B_COUNT             25
#define C_COUNT             10

void * print_b(void * arg) {
        int i;
        for (i = 0; i < B_COUNT; i++) {
                fprintf(stderr, "B");
                sleep (1);
        }
        fprintf(stderr, "(end-of-B)\n");

        return NULL;
}

void * print_c(void * arg) {
        pthread_t thread = *(pthread_t *) arg;
        int i;
        for (i = 0; i < C_COUNT; i++) {
                fprintf(stderr, "C");
                sleep(1);
        }
        fprintf(stderr, "(end-of-C)\n");
        pthread_join(thread, NULL);

        return NULL;
}

int main (void) {
        pthread_t thread1, thread2;
        int i;
```

```
        if (pthread_create(&thread1, NULL,
                           &print_b, NULL) != 0) {
                fprintf(stderr, "Error (thread1)\n");
                return 1;
        }

        if (pthread_create(&thread2, NULL,
                           &print_c, &thread1) != 0} {
                fprintf(stderr, "Error (thread2)\n");
                return 1;
        }

        for (i = 0; i < A_COUNT; i++) {
                fprintf(stderr, "A");
                sleep(1);
        }
        fprintf(stderr, "(end-of-A)\n");

        pthread_join(thread2, NULL);
        fprintf(stderr, "(end-of-all)\n");

        return 0;
}
```

В этой программе основной поток создает два новых потока. При этом поток, выводящий символы C, вызывает функцию pthread_join() для потока, печатающего символы B.

Получение информации о потоке: pthread_self(), pthread_equal()

Для процессов существует системный вызов getpid(), возвращающий идентификатор текущего процесса. Подобная функция существует и для потоков. Это функция pthread_self(), имеющая следующий прототип:

```
pthread_t pthread_self(void);
```

Для сравнения идентификаторов двух процессов используется оператор ==. Тип pthread_t является целым числом, но к идентификаторам потоков не рекомендуется применять математические операции. Для сравнения идентификаторов двух потоков служит функция pthread_equal(), объявленная в заголовочном файле pthread.h следующим образом:

```
int pthread_equal(pthread_t THREAD1, pthread_t THREAD2);
```

Если идентификаторы относятся к одному потоку, то эта функция возвращает ненулевое значение. Если THREAD1 и THREAD2 являются идентификаторами разных потоков, то pthread_equal() возвращает 0.

Чаще всего эти функции используют для того, чтобы уберечь поток от вызова pthread_equal() для самого себя. Рассмотрим пример демонстрирующий работу функций

pthread_self() и pthread_equal().

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

void * any_func(void * arg) {
        pthread_t thread = * (pthread_t *) arg;

        if (pthread_equal(pthread_self(), thread) != 0)
                fprintf(stderr, "1\n");

        return NULL;
}

int main(void) {
        pthread_t thread;
        if (pthread_create(&thread, NULL,
                        &any_func, &thread) != 0) {
                fprintf(stderr, "Error\n");
                return 1;
        }
        if (pthread_equal(pthread_self(), thread) != 0)
                fprintf(stderr, "2\n");

        pthread_join(thread, NULL);
        return 0;
}
```

Очевидно, что эта программа всегда выводит на экран единицу и никогда не выведет двойку.

Таким образом, для того, чтобы поток не вызвал функцию pthread_join(), чтобы ждать самого себя (в подобной ситуации возвращается код ошибки EDEADLK), можно применять проверку:

```
if (!pthread_equal(pthread_self(), any_thread))
        pthread_join(any_thread, NULL);
```

Замечание: во многих примерах этого занятия информация, предназначенная для вывода, направляласть в поток ошибок stderr. Так было сделано из-за того, что поток ошибок не буферизированный и пользователь сможет сразу после помещения в поток увидеть информацию на экране. Стандартный поток вывода stdout буфуризированный, но можно отключить буферизацию вызовом функции setbuf(stdout, NULL) перед первым выводом и направлять информацию в него.