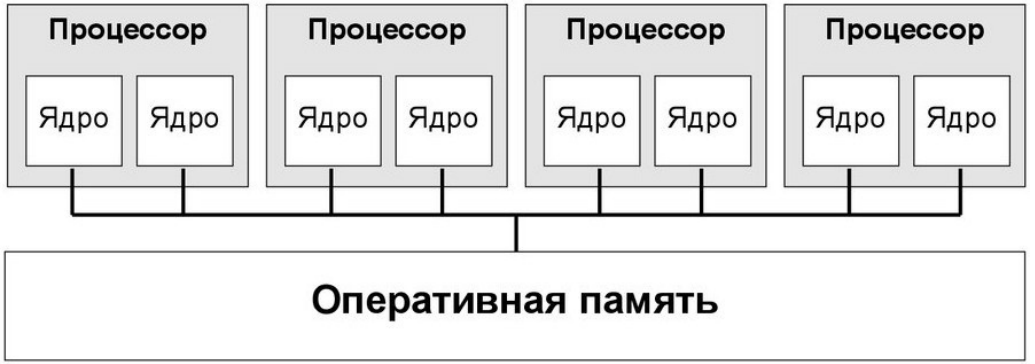
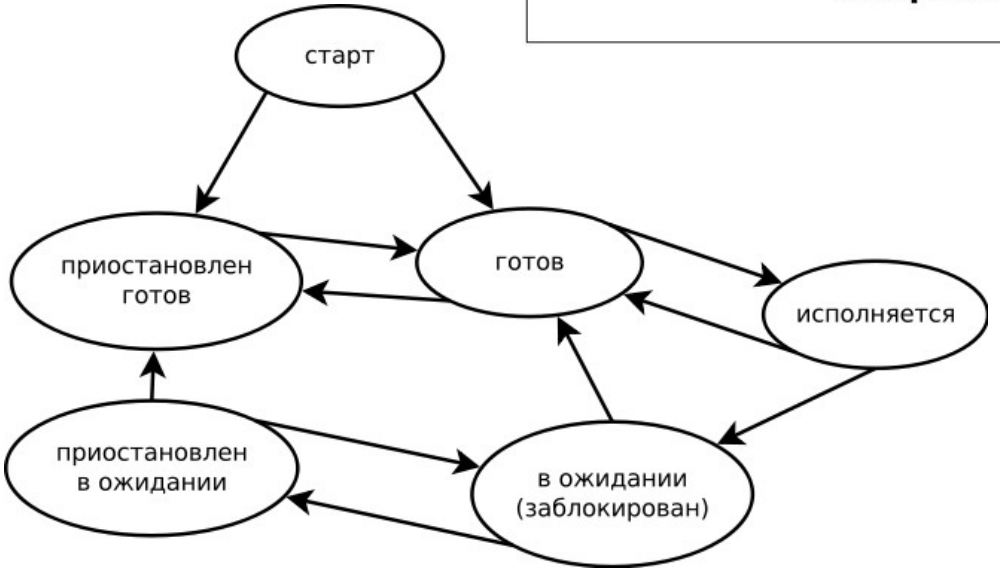
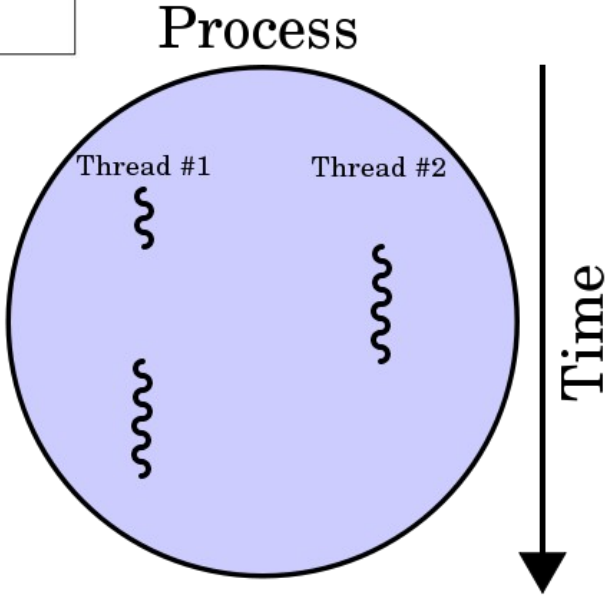
Архитектуры параллельных ВС (многопоточность)





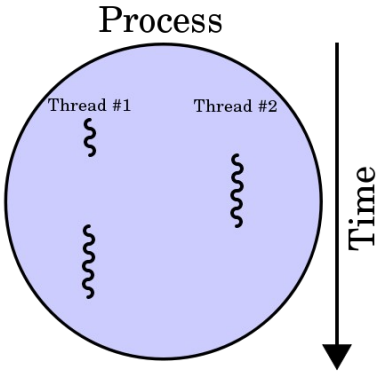


Определения

Многопото́чность (Multithreading) — свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких ***потоков***, выполняющихся «параллельно», то есть *без предписанного порядка во времени*. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

Потоки называют также ***потоками выполнения (thread of execution)***, ***«нитями» (буквальный перевод англ. thread)*** или неформально ***«тредами»***.

Определения



Пото́к выполне́ния (thread — нить, ПВ) — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

Реализация ПВ и процессов в разных ОС отличается, но в большинстве случаев ПВ находится внутри процесса. Несколько ПВ могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов.

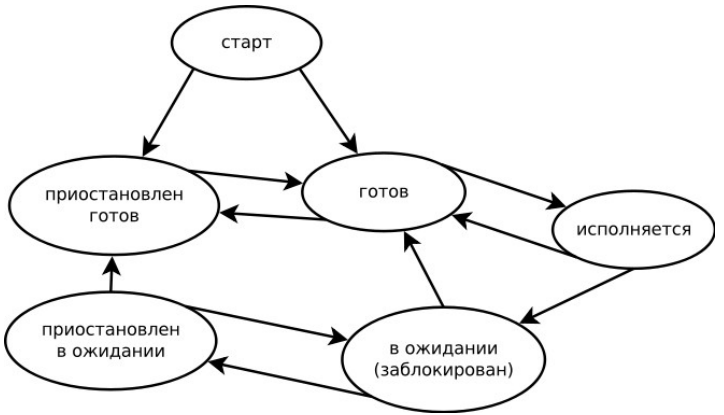
Аналогия: несколько вместе работающим поваров. Все они готовят одно блюдо, читают одну и ту же кулинарную книгу с одним и тем же рецептом и следуют его указаниям, причём не обязательно все они читают на одной и той же странице.

Определения

Проце́сс — выполнение пассивных инструкций компьютерной программы на процессоре.

Стандарт ISO 9000:2000 Definitions определяет **процесс** как совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.

Компьютерная программа сама по себе — лишь пассивная последовательность инструкций. В то время как **процесс** — непосредственное выполнение этих инструкций.



Также, **процессом** называют выполняющуюся программу и все её элементы: адресное пространство, глобальные переменные, регистры, стек, открытые файлы и т.д. (***контекст процесса***).

Определения

Отличие потоков от процессов:

- процессы, как правило, независимы, тогда как потоки выполнения существуют как составные элементы процессов
- процессы несут значительно больше информации о состоянии, тогда как несколько потоков выполнения внутри процесса совместно используют информацию о состоянии, а также память и другие вычислительные ресурсы
- процессы имеют отдельные адресные пространства, тогда как потоки выполнения совместно используют их адресное пространство
- процессы взаимодействуют только через предоставляемые системой механизмы связей между процессами
- переключение контекста между потоками выполнения в одном процессе, как правило, быстрее, чем переключение контекста между процессами.

Библиотека POSIX Threads

POSIX Threads — стандарт POSIX-реализации потоков (нитей) выполнения.
Стандарт POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995) определяет API для управления потоками, их синхронизации и планирования.
Реализации существуют для большого числа UNIX-подобных ОС (GNU/Linux, Solaris, FreeBSD, OpenBSD, NetBSD, OS X), а также для Microsoft Windows и других ОС.
Библиотеки, реализующие этот стандарт (и функции этого стандарта), обычно называются ***Pthreads*** *(функции имеют приставку «pthread_»).*

Библиотека POSIX Threads

```
#include <pthread.h>

int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
        void *(*start_routine)
(void *),
    void *arg);
```

https://reposcope.com/man/ru/3/pthread_create

pthread_create(3) - Russkiy

Linux
2018-04-30

ИМЯ

pthread_create - создаёт новую нить

ОБЗОР

```
#include <pthread.h>


int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

Компилируется и компонуется вместе с *-pthread*.

ОПИСАНИЕ

Функция **pthread_create**() запускает новую нить в вызвавшем процессе. Новая нить начинает выполнение вызовом *start_routine*(); значение *arg* является единственным аргументом *start_routine*().

Новая нить завершает работу в одном из следующих случаев:

Библиотека POSIX Threads

```
#include <pthread.h>

int pthread_join(
    pthread_t thread,
    void **retval);
```

https://reposcope.com/man/ru/3/pthread_join

Печать символов с использованием потоков (C-стиль)

```
// функция main создаёт один новый поток
// для печати “ABCD...”,
// а основной поток печатает “abcd...”.

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

void wait_thread (void);
void* thread_func (void*);
```

Печать символов с использованием потоков (C-стиль)

```
int main () {
    pthread_t thread;
    if (pthread_create(&thread,NULL,
                       thread_func,NULL))
       return EXIT_FAILURE;
    for (unsigned int i = 0; i < 20; i++) {
        puts("abcdefghijklmnoprstuvwxyz");
        wait_thread();
    }
    if (pthread_join(thread,NULL))
       return EXIT_FAILURE;
    return EXIT_SUCCESS;
}
```

Печать символов с использованием потоков (C-стиль)

```
void wait_thread (void) {
    time_t start_time = time(NULL);
    while(time(NULL) == start_time) {}
}

void* thread_func (void* vptr_args) {
    for (unsigned int i = 0; i < 20; i++) {
        fputs(
          "ABCDEFGHIJKLMNOPRSTUVWXYZ\n",stderr);
        wait_thread();
    }
    return NULL;
}
```

sum(a[i]^2) – непосредственный возврат результата

```
/*
 * Дана последовательность натуральных чисел a0, ...
 * Создать многопоточное приложение для поиска суммы квадратов sum(a[i]^2).
 * В приложении вычисления должны независимо выполнять четыре потока.
*/

#include <iostream>
#include <iomanip>
#include <limits>
#include <ctime>
#include <pthread.h>
```

sum(a[i]^2) – непосредственный возврат результата

```
double *A ;  //последовательность чисел a0...
const unsigned int arrSize = 10000000;
const int threadNumber = 4; // Количество потоков


//стартовая функция для дочерних потоков
void *func(void *param) { //вычисление суммы квадратов в потоке
    // Смещение в потоке для начала массива
    unsigned int shift = arrSize / threadNumber;
    int p = (*(int*)param )*shift;
    double *sum = new double;
    *sum = 0;
    for(unsigned int i = p ; i < p+shift ; i++) {
        *sum+=A[i]*A[i];
    }
    return (void*)sum ;
}
```

sum(a[i]^2) – непосредственный возврат результата

```
int main() {
    double rez = 0.0 ; //для записи окончательного результата

    // заполнение массива A
    A = new double[arrSize];
    if(A == nullptr) {
        std::cout <<
            "Incorrect size of vector = " << arrSize << "\n";
        return 1;
    }

    for(int i = 0; i < arrSize; ++i) {
        A[i] = double(i);
    }
```

sum(a[i]^2) – непосредственный возврат результата

```
pthread_t thread[threadNumber];
int number[threadNumber] ;

clock_t start_time =  clock(); // начальное время

//создание четырех дочерних потоков
for (int i=0 ; i<threadNumber ; i++) {
    number[i]=i ; //для передачи параметра потоку
    pthread_create(&thread[i], nullptr,
                   func, (void*)(number+i)) ;
}
```

sum(a[i]^2) – непосредственный возврат результата

```
    double *sum ;
    for (int i = 0 ; i < threadNumber; i++)
        pthread_join(thread[i],(void **)&sum) ;
        rez += *sum ;
        delete sum ;
    }
    clock_t end_time = clock(); // конечное время
    std::cout << "Сумма квадратов = " << rez << "\n" ;

    std::cout << "Время счета и сборки = "
              << end_time - start_time << "\n";
    delete[] A;
    return 0;
}
```

sum(a[i]^2) – возврат результата через аргумент

```
struct Package {
    // Указатель на начало области обработки
    double* array;
    int threadNum;  // Номер потока
    double sum;     // Формируемая частичная сумма
} ;

double *A ;  //последовательность чисел a0...
const unsigned int arrSize = 10000000;

const int threadNumber = 4; // Количество потоков
```

sum(a[i]^2) – возврат результата через аргумент

```
//стартовая функция для дочерних потоков
void *func(void *param) {//вычисление суммы квадратов
    // Востановление структуры
    Package* p = (Package*)param;
    p->sum = 0.0;
    for(unsigned int i = p->threadNum ;
                  i < arrSize; i+=threadNumber) {
        p->sum += p->array[i] * p->array[i];
    }
    return nullptr;
}
```

sum(a[i]^2) – возврат результата через аргумент

```
int main() {
    double rez = 0.0 ;

    // заполнение массива A
    A = new double[arrSize];
    if(A == nullptr) {
        std::cout << "Incorrect size of vector = "
                  << arrSize << "\n";
        return 1;
    }
    for(int i = 0; i < arrSize; ++i) {
        A[i] = double(i);
    }
```

sum(a[i]^2) – возврат результата через аргумент

```
pthread_t thread[threadNumber];
Package pack[threadNumber];

clock_t start_time =  clock(); // начальное время

//создание дочерних потоков
for (int i=0 ; i<threadNumber ; i++) {
    // Формирование структуры для передачи потоку
    pack[i].array = A;
    pack[i].threadNum = i;
    pthread_create(&thread[i],
                   nullptr, func, (void*)&pack[i]) ;
}
```

sum(a[i]^2) – возврат результата через аргумент

```
    for (int i = 0 ; i < threadNumber; i++) {
        pthread_join(thread[i], nullptr) ;
        rez += pack[i].sum;
    }
    clock_t end_time = clock(); // конечное время

    //вывод результата
    std::cout << "Сумма квадратов = "
              << rez << "\n" ;
    std::cout << "Время счета и сборки = "
              << end_time - start_time << "\n";
     delete[] A;
    return 0;
}
```

Обертка из классов над pthread

```
// Создание четырех потоков с использованием
// оберток из классов

#include <cstdlib>
#include <iostream>
#include <memory>
#include <unistd.h>
#include <pthread.h>
```

Обертка из классов над pthread

```
class Thread {
public:
    virtual ~Thread () {}
    virtual void run () = 0;
    int start () {
        return pthread_create(&_ThreadId, nullptr,
               Thread::thread_func, this );
    }
    int wait () {return pthread_join( _ThreadId, NULL );}
protected:
    pthread_t _ThreadId;
    static void* thread_func(void* d) {
        (static_cast <Thread*>(d))->run();
        return nullptr;
    }
};
```

Обертка из классов над pthread

```
class TestingThread : public Thread {
public:
    TestingThread (const char* pcszText) :
                        _pcszText( pcszText ) {}
    virtual void run () {
        for (unsigned int i = 0; i < 30; i++) {
            std::cout << _pcszText << "\n";
            sleep(1);
        }
    }
protected:
    const char* _pcszText;
};
```

Обертка из классов над pthread

```
int main (int argc, char *argv[], char *envp[]) {
    TestingThread Thread_a("abcdefghijklmnoprstuvwxyz");
    TestingThread Thread_A("ABCDEFGHIJKLMNOPRSTUVWXYZ");
    TestingThread
              Thread_0("01234567890123456789012345678 9");
    TestingThread
              Thread_9("98765432109876543210987654321 0");
    return
        Thread_a.start() || Thread_A.start() ||
        Thread_0.start() || Thread_9.start() ||
        Thread_a.wait()  || Thread_A.wait()  ||
        Thread_0.wait()  || Thread_9.wait()
                          ? EXIT_FAILURE : EXIT_SUCCESS;
}
```

Многопоточность в C++

```
#include <iostream>
#include <thread>

void hello() {
    std::cout << "Hello Concurrent World\n";
}

int main() {
    std::thread t(hello);
    t.join();
}
```

sum(a[i]^2) – C++: использование <thread>

```
#include <iostream>
#include <iomanip>
#include <limits>
#include <ctime>
#include <thread>

double *A ;  //последовательность чисел a0...
const unsigned int arrSize = 10000000;
const int threadNumber = 4; // Количество потоков
```

sum(a[i]^2) – C++ использование <thread>

```
//стартовая функция для дочерних потоков
//void sqsum(int iTread, int iTN,
//           double *arr, int size, double *sum) {
void sqsum(int iTread, int iTN,
           double *arr, int size, double &sum) {
    for(int i = iTread; i < size; i+=iTN) {
        //*sum += arr[i] * arr[i];
        sum += arr[i] * arr[i];
    }
}
```

sum(a[i]^2) – C++ использование <thread>

```
int main() {
    // заполнение массива A
    A = new double[arrSize];
    if(A == nullptr) {
        std::cout << "Incorrect size of vector = "
                  << arrSize << "\n";
        return 1;
    }
    for(int i = 0; i < arrSize; ++i) {
        A[i] = double(i);
    }
```

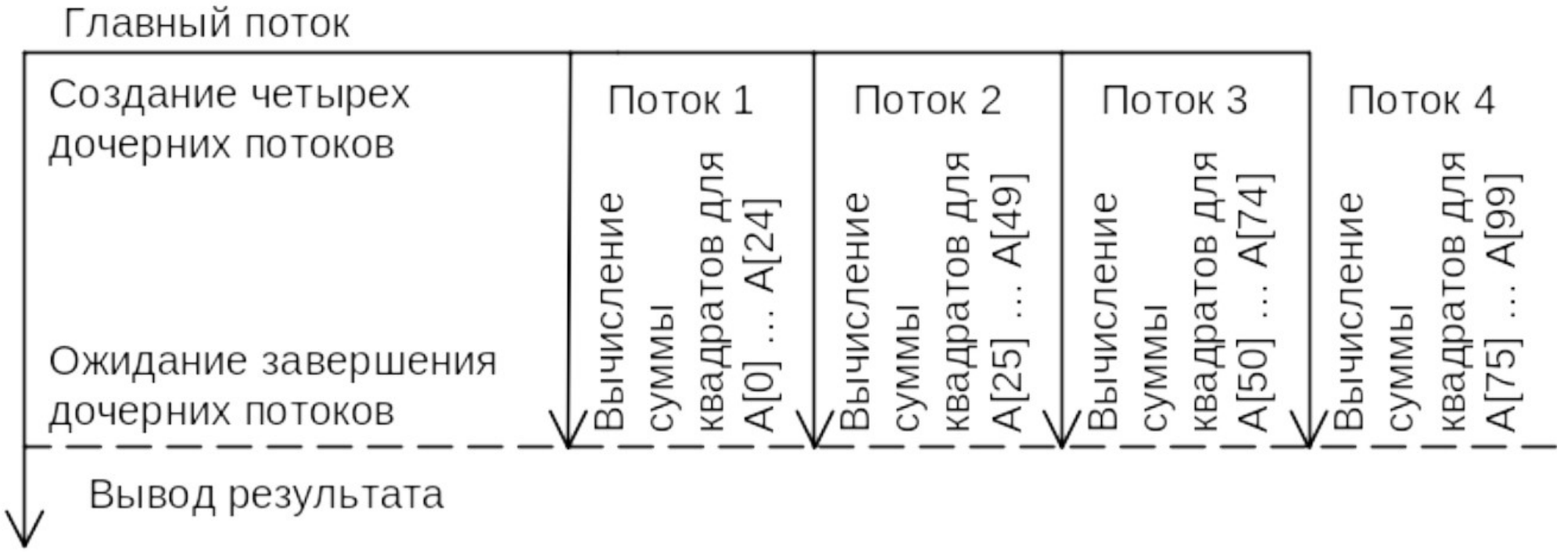
sum(a[i]^2) - C++ использование <thread>

```
std::thread *thr[threadNumber];
double sum[threadNumber];

clock_t start_time =  clock(); // начальное время

// Создание потоков
for (int i=0 ; i<threadNumber ; i++) {
    //thr[i] = new std::thread{sqsum, i,
    //     threadNumber, A, arrSize, (sum+i)};
    thr[i] = new std::thread{sqsum, i,
                  threadNumber, A, arrSize,
                  std::ref(sum[i])};
}
```

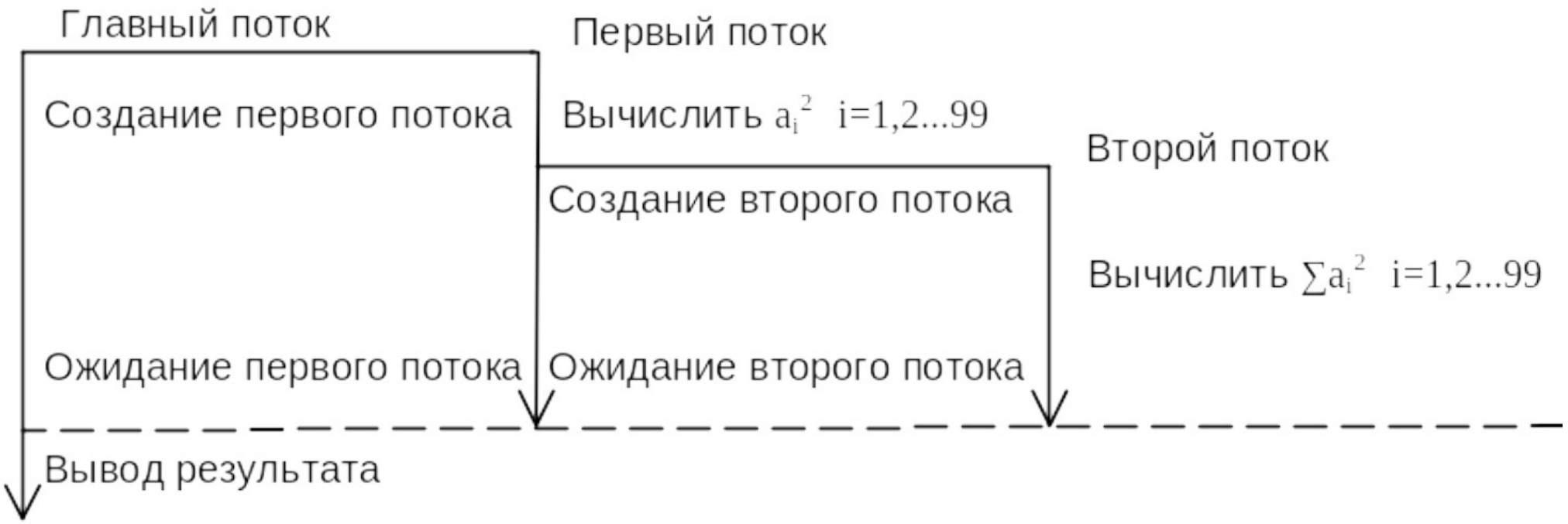
sum(a[i]^2) – C++ использование <thread>

```
    double rez = 0.0 ; //для результата
    // Завершение потоков
    for (int i=0 ; i<threadNumber ; i++) {
        thr[i]->join();
        rez += sum[i];
        delete thr[i];
    }
    clock_t end_time = clock(); // конечное время
    std::cout << "Сумма квадратов = "
                << rez << "\n" ;
    std::cout << "Время счета и сборки = "
                << end_time - start_time << "\n";
    delete[] A;
    return 0;
}
```

Разбиение процесса на независимые потоки



Иерархическое выполнение потоков



Параллелизм – parallelism

Параллелизм - concurrency

Модели многопоточных приложений

Итеративный параллелизм
Используется для реализации нескольких потоков (часто идентичных), каждый из которых содержит циклы. Потоки программы, описываются итеративными функциями и работают совместно над решением одной задачи.

Рекурсивный параллелизм
Используется в программах с одной или несколькими рекурсивными процедурами, вызов которых независим. Это технология «разделяй-и-властвуй» или «перебор-с-возвратами».

Модели многопоточных приложений

Производители и потребители
Парадигма взаимодействующих неравноправных потоков. Одни из потоков «производят» данные, другие их «потребляют». Часто такие потоки организуются в **конвейер**, через который проходит информация. Каждый поток конвейера потребляет выход своего предшественника и производит входные данные для своего последователя. Другой распространенный способ организации потоков – древовидная структура или сети слияния, на этом основан, в частности, принцип **дихотомии**.

Клиенты и серверы
Еще один способ взаимодействия неравноправных потоков. Клиентский поток запрашивает сервер и ждет ответа. Серверный поток ожидает запроса от клиента, затем действует в соответствии с поступившим запросом.

Модели многопоточных приложений

Управляющий и рабочие
Модель организации вычислений, при которой существует поток, координирующий работу всех остальных потоков. Как правило, управляющий поток распределяет данные, собирает и анализирует результаты.

Взаимодействующие равные
Модель, в которой исключен не занимающийся непосредственными вычислениями управляющий поток. Распределение работ в таком приложении либо фиксировано заранее, либо динамически определяется во время выполнения. Одним из распространенных способов динамического распределения работ является **«портфель задач»**. Портфель задач, как правило, реализуется с помощью разделяемой переменной, доступ к которой в один момент времени имеет только один процесс.

Механизмы синхронизации. Мьютексы

Двоичные семафоры (мьютексы) могут находиться в двух состояниях, открытом и закрытом.

Поток может закрыть только открытый мьютекс

Если поток пытается закрыть уже закрытый мьютекс, то его выполнение приостанавливаетс до тех пор, пока мьютекс не станет открытым.

Таким образом, двоичные семафоры блокируют выполнение других потоков, когда это необходимо

*Векторное произведение матриц: C = A * B*

Найти результат произведения матриц A*B, где A и B – матрицы 3x3.
Записать в общую очередь промежуточные результаты вычислений потоков, с указанием идентификатора потока или индивидуального номера потока, присваиваемого потоку пользователем.
Защитить операции с общей очередью посредством двоичного семафора. Вывести общую очередь, после завершения основных вычислений.

Обсуждение

Поскольку каждый поток записывает результаты вычислений в общую очередь, то одновременная запись в очередь несколькими потоками или передача управления к другому потоку, когда операция записи не закончена, могут привести, например, к потере записи нескольких результатов или разрушению связей между элементами очереди.

Для предотвращения подобных ситуаций воспользуемся ***двоичным семафором***, с помощью которого будет защищена операция записи в очередь.

Векторное произведение матриц: C = A * B

```
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t mutex ; //двоичный семафор
int A[3][3] = {
    1, -2, 0,
    4, 6, 2,
    -3, 4, -2
};
int B[3][3] = {
    0, 2, 0,
    1, 1, 1,
    5, -3, 10
};
int C[3][3];  //результирующая матрица
```

Векторное произведение матриц: C = A * B

```
struct result  {     //структура очереди
    char str[100];  //для записи результата вычислений
    result *next;   //указатель на следующий элемент
} ;

result *head ;   //указатель на первый элемент очереди
result *newrez ; //указатель элементов очереди
result *rez ;    //указатель на последний элемент очереди
```

Векторное произведение матриц: C = A * B

```
//стартовая функция для дочерних потоков
void* func(void *param) {
    //номер строки, заполняемой потоком
    int p=*(int *)param ;
    //вычисление элементов матрицы, стоящих в строке p
    for (int i=0 ; i<3 ; i++) {
        C[p][i]=0 ;
        for (int j=0 ; j<3 ; j++) {
            C[p][i]+=A[p][j]*B[j][i] ;
        }
        //протокол входа в КС: закрыть двоичный семафор
        pthread_mutex_lock(&mutex) ;
```

Векторное произведение матриц: C = A * B

```
        //начало критической секции
        // – запись результата в очередь
        newrez = new result ;
        sprintf(newrez->str,
           "Поток %d: вычислен элемент [%d][%d] = %d\0",
           p, p, i, C[p][i]
        ) ;

        //конец критической секции
        //протокол выхода из КС:
        pthread_mutex_unlock(&mutex) ;
        //открыть двоичный семафор
    }
    return nullptr;
}
```

Векторное произведение матриц: C = A * B

```
int main() {
    head = new result ;
    rez = head ; //создание первого элемента очереди
    //инициализация двоичного семафора
    pthread_mutex_init(&mutex, NULL) ;
    //идентификаторы для дочерних потоков
    pthread_t mythread1,  mythread2 ;
    int num[3] ;
    for (int i=0 ; i<3 ; i++) {
        num[i]=i ; //номера строк для потоков
    }
```

Векторное произведение матриц: C = A * B

```
//создание дочерних потоков
pthread_create(&mythread1, NULL, func,
               (void *)(num+1)) ;
pthread_create(&mythread2, NULL, func,
               (void *)(num+2)) ;
//заполнение первой строки результирующей матрицы
func((void *)num) ;
//ожидание завершения дочерних потоков
pthread_join(mythread1,NULL) ;
pthread_join(mythread2,NULL) ;
```

Векторное произведение матриц: C = A * B

```
    rez = head->next ;
    while (rez!=NULL) { //вывод очереди
        printf("\n%s", rez->str) ;
        rez = rez->next ;
    }
    // вывод результата вычислений всех потоков
    printf("\n") ;
    for (int i=0 ; i<3 ; i++) {
        for (int j=0 ; j<3 ; j++) {
            printf( "%d ",C[i][j]) ;
        }
        printf("\n") ;
    }
    return 0;
}
```

Механизмы синхронизации. Семафоры. Защита критических секций, условная синхронизация

Семафор

Содержит неотрицательное целое значение.

Любой поток может изменять значение семафора. Когда поток пытается уменьшить значение семафора, происходит следующее:
- если значение больше нуля, то оно уменьшается;
- если значение равно нулю, поток приостанавливается до того момента, когда значение семафора станет положительным.

При положительном значении поток продолжает работу.

Операция увеличения значения семафора является **неблокирующей**.

Механизмы синхронизации. Семафоры. Защита критических секций, условная синхронизация

Критической секцией многопоточного алгоритма чаще всего являются операции записи в общие данные, которые могут быть изменены или открыты для чтения несколькими потоками.
Для защиты критических секций могут быть использованы **мьютексы** или **семафоры**.

*Синхронизация потоков носит иногда более сложный характер. Например, при наступлении какого-то **условия** необходимо приостановить поток до изменения ситуации (другим процессом) или, наоборот, при наступлении какого-то условия следует запустить поток.*

Одним из механизмов ***условной синхронизации*** процессов также являются **семафоры**.

Задача о кольцевом буфере

Потоки ***производители*** и ***потребители*** разделяют кольцевой буфер, состоящий из **N** ячеек. Производители передают сообщение потребителям, помещая его в конец очереди буфера. Потребители сообщение извлекают из начала очереди буфера. Создать многопоточное приложение с потоками писателями и читателями. Предотвратить такие ситуации как, изъятие сообщения из пустой очереди или помещение сообщения в полный буфер.

При решении задачи использовать семафоры.

Задача о кольцевом буфере

Обсуждение. Пусть буфер – это целочисленный массив из **N** элементов. В задаче 2 критических секции.
Первая связана с операциями чтения-записи нескольких потоков в общий буфер.
Вторая критическая секция определяется тем, что буфер являются конечным.
Запись должна производиться только в те ячейки, которые являются свободными или уже прочитаны потоками-читателями (*условная взаимная синхронизация*).

Задача о кольцевом буфере

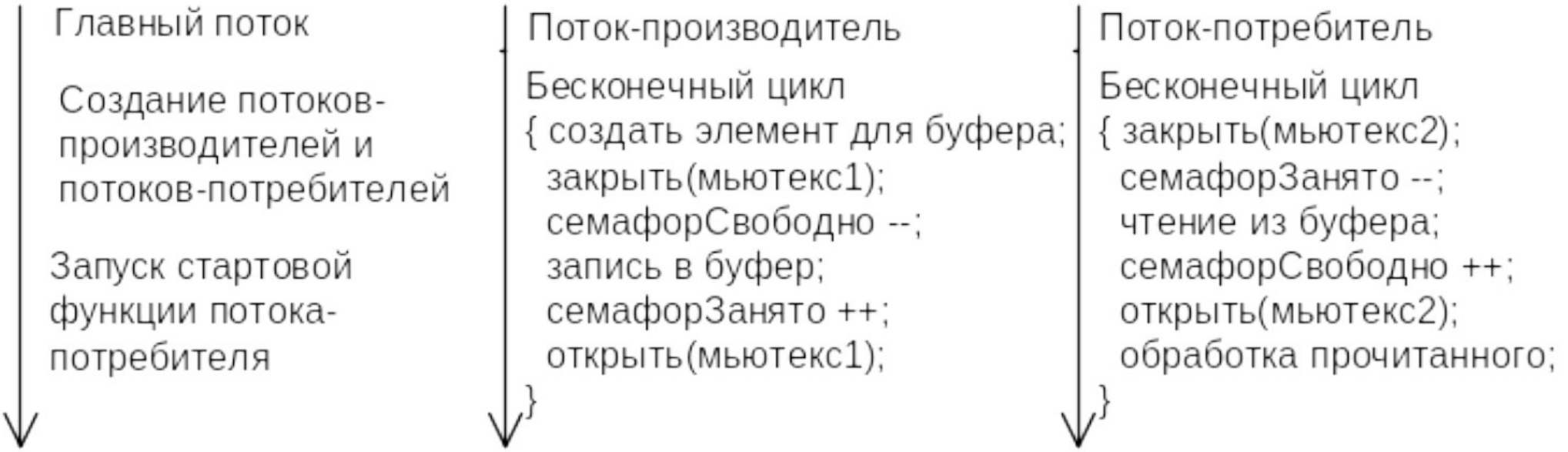
Для защиты первой критической секции воспользуемся двумя мьютексами.
Первый сделает возможным запись в буфер только для одного потока-писателя.
Второй сделает возможным чтение из буфера только для одного потока-читателя.
Операция чтения должна быть **защищена**, т. к. она осущствляет запись данных об освобождении ячейки. Иначе операция записи может стать невозможной или некорректной из-за переполнения буфера. Чтение и запись могут проходить параллельно, т.к.всегда используют разные ячейки.

Задача о кольцевом буфере

Для условной синхронизации используем 2 семафора.
Первый показывает, сколько ячеек в буфере свободно. Ячейка свободна, когда в нее еще не осуществлялась запись или она была прочитана.
Второй семафор показывает, сколько ячеек в буфере занято. Запись не может быть выполнена, пока число занятых ячеек = **N** (или число свободных ячеек = 0).
Операция чтения не может быть выполнена, пока число свободных ячеек = **N** (или число занятых ячеек = 0).

Для блокировки воспользуемся условиями, заключенными в скобки, исходя из особенностей поведения семафоров.

Задача о кольцевом буфере. Схема решения



Задача о кольцевом буфере. Семафоры

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

const int bufSize = 10;
int buf[bufSize] ; //буфер
int front = 0 ;    //индекс для чтения из буфера
int rear = 0 ;     //индекс для записи в буфер
```

Задача о кольцевом буфере. Семафоры

```
sem_t  empty ; //семафор, отображающий как  буфер пуст
sem_t  full ;  //семафор, отображающий как буфер полон

pthread_mutex_t mutexD ; //мьютекс для операции записи
pthread_mutex_t mutexF ; //мьютекс для операции чтения
unsigned int seed = 101; // инициализатор ГСЧ
```

Задача о кольцевом буфере. Семафоры

```
//стартовая функция потоков – производителей (писателей)
void *Producer(void *param) {
    int pNum = *((int*)param);
    while (1) {
        //создать элемент для буфера
        int data = rand() % 11 - 5 ;
        //поместить элемент в буфер
        pthread_mutex_lock(&mutexD) ; //защита операции записи
        sem_wait(&empty) ; // свободных ячеек уменьшить на 1
        buf[rear] = data ;
        rear = (rear+1)%bufSize ; //критическая секция
        sem_post(&full) ; //занятых ячеек увеличилось на 1
        pthread_mutex_unlock(&mutexD) ;
```

Задача о кольцевом буфере. Семафоры

```
        Printf(
            "Producer %d: Writes value = %d to cell [%d]\n",
                        pNum, data, rear) ;
           sleep(2);
       }
       return nullptr;
}
```

Задача о кольцевом буфере. Семафоры

```
//стартовая функция потоков – потребителей (читателей)
void *Consumer(void *param) {
    int cNum = *((int*)param);
    int result ;
    while (1) {
        //извлечь элемент из буфера
        pthread_mutex_lock(&mutexF) ; //защита чтения
        //количество занятых ячеек уменьшить на единицу
        sem_wait(&full) ;
        result = buf[front] ;
        front = (front+1)%bufSize ; //критическая секция
        //количество свободных ячеек увеличилось на 1
        sem_post(&empty) ;
        pthread_mutex_unlock(&mutexF) ;
```

Задача о кольцевом буфере. Семафоры

```
        //обработать полученный элемент
        Printf(
            "Consumer %d: Reads value = %d from cell [%d]\n",
                        cNum, result, front) ;
            sleep(5);
    }
    return nullptr;
}
```

Задача о кольцевом буфере. Семафоры

```
int main() {
    srand(seed);
    int i ;
    //инициализация мутексов и семафоров
    pthread_mutex_init(&mutexD, nullptr) ;
    pthread_mutex_init(&mutexF, nullptr) ;
    //количество свободных ячеек равно bufSize
    sem_init(&empty, 0, bufSize) ;
    //количество занятых ячеек равно 0
    sem_init(&full, 0, 0) ;
```

Задача о кольцевом буфере. Семафоры

```
//запуск производителей
pthread_t threadP[3] ;
int producers[3];
for (i=0 ; i<3 ; i++) {
    producers[i] = i + 1;
    pthread_create(&threadP[i],nullptr,Producer,
                          (void*)(producers+i)) ;
}
```

Задача о кольцевом буфере. Семафоры

```
    //запуск потребителей
    pthread_t threadC[4] ;
    int consumers[4];
    for (i=0 ; i < 4 ; i++) {
        consumers[i] = i + 1;
        pthread_create(&threadC[i],nullptr,Consumer,
                                        (void*)(consumers+i)) ;
    }
    //пусть главный поток тоже будет потребителем
    int mNum = 0;
    Consumer((void*)&mNum) ;
    return 0;
}
```

Условные переменные

При условной синхронизации потоков вместо **семафоров** естественно использовать **условные переменные**, которые, как и семафоры, осуществляют блокировку потоков и их пробуждение.

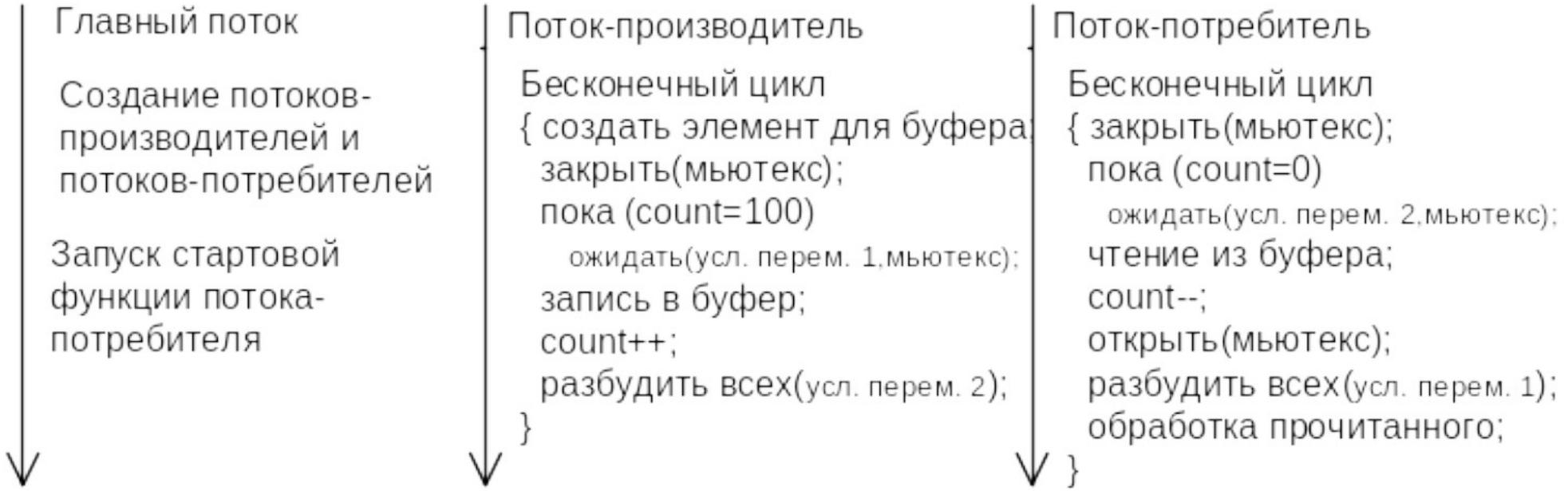
Условные переменные, могут приостанавливать работу потока, заставляя его ожидать сигнал к пробуждению.

Можно разбудить один или все потоки, ожидающие сигнала от данной условной переменной.

Условные переменные, в отличие от семафоров, **не обладают внутренними состояниями**.

Описание условий блокирования или пробуждения потока возлагается на программиста.

Условные переменные. Схема потоков



Семафоры **full** и **empty**, отображающие насколько пуст или полон буфер, могут быть заменены условными переменными.
Для описания условий к вызовам функций условных переменных воспользуемся глобальной переменной count, показывающей, сколько ячеек буфера заняты.

Задача о кольцевом буфере. Условные переменные

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

const int bufSize = 10;
int buf[bufSize] ; //буфер
int front = 0 ; //индекс для чтения из буфера
int rear = 0 ; //индекс для записи в буфер
int count = 0 ; //количество занятых ячеек буфера
unsigned int seed = 101; // инициализатор ГСЧ
```

Задача о кольцевом буфере. Условные переменные

```
pthread_mutex_t mutex; // мьютекс для условных переменных

// поток-писатель блокируется этой условной переменной,
// когда количество занятых ячеек становится = bufSize
pthread_cond_t not_full ;

// поток-читатель блокируется этой условной переменной,
// когда количество занятых ячеек становится равно 0
pthread_cond_t not_empty ;
```

Задача о кольцевом буфере. Условные переменные

```
//стартовая функция потоков – производителей (писателей)
void *Producer(void *param) {
    int pNum = *((int*)param);
    int data, i ;
    while (1) {
        //создать элемент для буфера
        data = rand() % 11 - 5 ;
        //поместить элемент в буфер
        pthread_mutex_lock(&mutex) ; //защита записи

        //заснуть, если количество занятых ячеек = N
        while (count == bufSize ) {
            pthread_cond_wait(&not_full, &mutex) ;
        }
```

Задача о кольцевом буфере. Условные переменные

```
        //запись в общий буфер
        buf[rear] = data ;
        rear = (rear+1)%bufSize ;
        count++ ; //появилась занятая ячейка
        //конец критической секции
        pthread_mutex_unlock(&mutex) ;
        //разбудить читателей после добавления в буфер
        pthread_cond_broadcast(&not_empty) ;
        Printf(
        "Producer %d: Writes value = %d to cell [%d]\n",
                  pNum, data, rear) ;
        sleep(2);
    }
    return NULL;
}
```

Задача о кольцевом буфере. Условные переменные

```
//стартовая функция потоков – потребителей (читателей)
void *Consumer(void *param) {
    int cNum = *((int*)param);
    int result ;
    while (1) {
        //извлечь элемент из буфера
        pthread_mutex_lock(&mutex) ; //защита чтения
        //заснуть, если количество занятых ячеек = нулю
        while (count == 0) {
            pthread_cond_wait(&not_empty, &mutex) ;
        }
        //изъятие из буфера – начало критической секции
        result = buf[front] ;
        front = (front+1)%bufSize ; //критическая секция
        count-- ; //занятая ячейка стала свободной
```

Задача о кольцевом буфере. Условные переменные

```
        // конец критической секции
        pthread_mutex_unlock(&mutex) ;
        // разбудить потоки-писатели
        // после получения элемента из буфера
        pthread_cond_broadcast(&not_full) ;

        //обработать полученный элемент
        Printf(
            "Consumer %d: Reads value = %d from cell [%d]\n",
            cNum, result, front) ;
        sleep(5);
    }
    return NULL;
}
```

Задача о кольцевом буфере. Условные переменные

```
int main() {
    srand(seed);
    int i ;
    //инициализация мьютексов и семафоров
    pthread_mutex_init(&mutex, NULL) ;
    pthread_cond_init(&not_full, NULL) ;
    pthread_cond_init(&not_empty, NULL) ;
    //запуск производителей
    pthread_t threadP[3] ;
    int producers[3];
    for (i=0 ; i<3 ; i++) {
        producers[i] = i + 1;
        pthread_create(&threadP[i],NULL,Producer,
                                    (void*)(producers+i)) ;
    }
```

Задача о кольцевом буфере. Условные переменные

```
    //запуск потребителей
    pthread_t threadC[4] ;
    int consumers[4];
    for (i=0 ; i < 4 ; i++) {
        consumers[i] = i + 1;
        pthread_create(&threadC[i],NULL,
                       Consumer, (void*)(consumers+i)) ;
    }

    //пусть главный поток тоже будет потребителем
    int mNum = 0;
    Consumer((void*)&mNum) ;
    return 0;
}
```

Блокировки чтения-записи

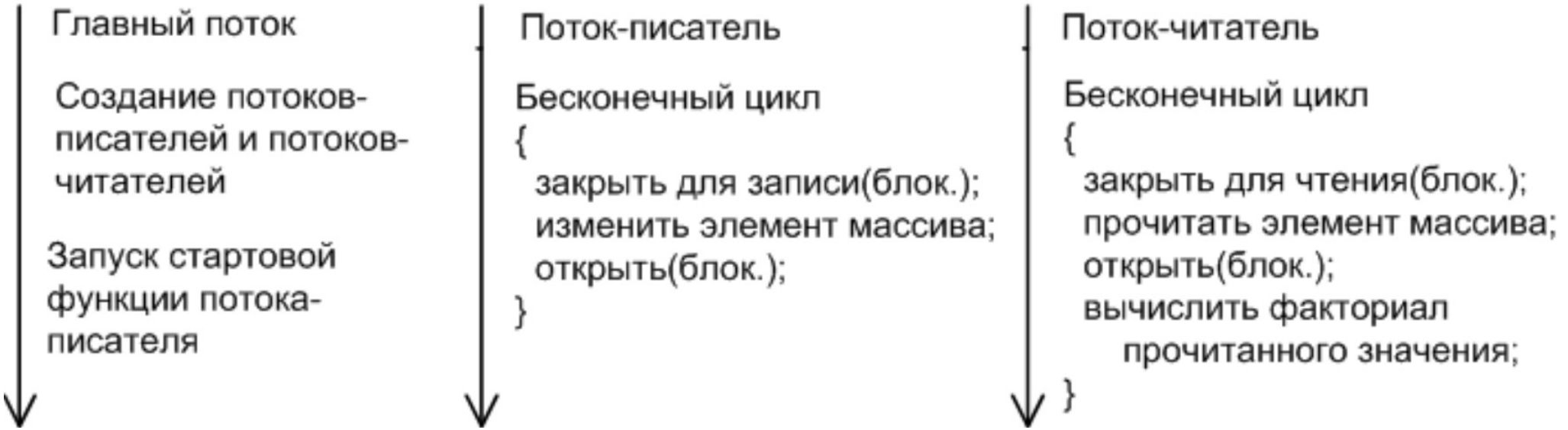
Блокировка чтения-записи, как и мьютекс, может находиться или в открытом или в закрытом состоянии.
Отличие: блокировка может быть закрыта для записи или закрыта для чтения.
Если блокировка закрыта для записи, то никакой другой поток не может закрыть блокировку.
Если блокировка закрыта для чтения, то никакой поток не может закрыть блокировку для записи.
Если закрытие блокировки в данный момент невозможно, поток останавливается до того момента, когда блокировка станет открытой.

Блокировки чтения-записи

Блокировки чтения-записи предоставляют возможность читать общие данные, сразу многим потокам, или изменять общие данные только одному потоку, при этом операции чтения и операция записи не могут быть выполнены одновременно.

*Использование блокировок чтения-записи актуально, когда одновременно работает несколько **потоков – читателей**. В противном случае, правильнее пользоваться двоичными семафорами, так как операции закрытия и открытия двоичного семафора происходят быстрее, чем аналогичные операции блокировки чтения-записи.*

Читатели-писатели с общим одномерным массивом



Многопоточное приложение с потоками-писателями и потоками-читателями, работающими с общим одномерным массивом. Писатели изменяют произвольный элемент массива. Читатели получают значение произвольного элемента массива и находят факториал прочитанного значения. Для защиты операций с общими данными использованы блокировки чтения-записи.

Читатели-писатели с общим одномерным массивом

Для защиты критических секций – операций с общим массивом введем блокировку чтения-записи.

Потоки-писатели перед изменением элемента массива будут ***закрывать блокировку для записи***.

Потоки-читатели перед получением значения из массива будут ***закрывать блокировку для чтения***.

Благодаря этому, потоки-читатели смогут работать параллельно, потоки-писатели смогут изменять данные только последовательно, и операции чтения и операции записи не будут происходить одновременно.

Читатели-писатели с общим одномерным массивом

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

const int arrSize = 10;
int A[arrSize] ;  //общий массив
pthread_rwlock_t rwlock ;  //блокировка чтения-записи
```

Читатели-писатели с общим одномерным массивом

```
//стартовая функция потоков-читателей
void *funcRead(void *param) {
    int rNum = *((int*)param);
    while (1) {
        int number = random() % arrSize; //получить случайный индекс
        //закрыть блокировку для чтения
        pthread_rwlock_rdlock(&rwlock) ;
        //прочитать данные из общего массива – критическая секция
        int a = A[number] ;
        //открыть блокировку
        pthread_rwlock_unlock(&rwlock) ;
        unsigned int F = 1 ;  //вычислить факториал
        for (int i = 2 ; i <= a ; i++) F=F*i ;
        printf("Читатель %d: Элемент[%d] -> %d! = %u\n",
              rNum, number , a, F) ;
        sleep(3);
    }
    return nullptr;
}
```

Читатели-писатели с общим одномерным массивом

```
//стартовая функция потоков-писателей
void *funcWrite(void *param) {
    int wNum = *((int*)param);
    while (1) {
        //получить случайный индекс
        int number = random() % arrSize;
        //закрыть блокировку для записи
        pthread_rwlock_wrlock(&rwlock) ;
        //изменить элемент общего массива – критическая секция
        A[number] = random() % 10 + 1;

        //открыть блокировку
        pthread_rwlock_unlock(&rwlock) ;
        fprintf(stdout,"Писатель %d: Элемент[%d] = % d\n",
                                        wNum, number, A[number]) ;
        sleep(4);
    }
    return nullptr;
}
```

Читатели-писатели с общим одномерным массивом

```
int main() {
    // Начальная инициализация массива
    for(int i = 0; i < arrSize; i++) {
        A[i] = 1;
    }

    //инициализация блокировки чтения-записи
    pthread_rwlock_init(&rwlock, NULL) ;
    //заполнение общего массива
    for (int i=0 ; i<arrSize ; i++)  {
        A[i] = random()/(RAND_MAX/10) ;
    }
```

Читатели-писатели с общим одномерным массивом

```
//создание четырех потоков-читателей
pthread_t threadR[4] ;
int readers[4];
for (int i=0 ; i < 4 ; i++) {
    readers[i] = i+1;
    pthread_create(&threadR[i], NULL, funcRead,
                                    (void*)(readers+i)) ;
}

//создание трех потоков-писателей
pthread_t threadW[3] ;
int writers[3];
for (int i=0 ; i < 3 ; i++) {
    writers[i] = i+1;
    pthread_create(&threadW[i], NULL, funcWrite,
                                    (void*)(writers+i)) ;
}
```

Читатели-писатели с общим одномерным массивом

```
    //пусть главный поток будет потоком-писателем
    int mNum = 0;
    funcWrite((void*)&mNum) ;
    return 0;
}
```

Барьеры

Барьер – механизм синхронизации, позволяющий останавливать выполнение группы потоков, пока все потоки из этой группы не дойдут до определенной точки выполнения, помечаемой вызовом функции ожидания барьера.
При создании барьер получает **целое положительное число**, которое показывает количество потоков в этой группе.
Поток, ожидающий на барьере, останавливается, пока общее количество потоков, ожидающих на этом же барьере, ***не станет равно числу***, указанному ***при инициализации барьера***.

Как только, число потоков, ожидающих на барьере, становиться достаточным, все эти потоки возвращаются к работе, и барьер может быть использован снова.

Барьеры. Применение

Создать многопоточное приложение, в котором работают два потока, обладающие собственными массивами одинаковой размерности. Каждый поток должен заменить нулями, те элементы своего массива, которые равны соответственным элементам массива другого потока.

При решении задачи не пользоваться общими массивами.

Барьеры. Применение

В описанной схеме барьер служит как для синхронизации, так и для предотвращения логических ошибок. Синхронизация потоков осуществляется ожиданием на барьере на каждом шаге цикла. Это позволяет потокам сравнивать элементы массивов с соответственными индексами. Для предотвращения появления логических ошибок необходимо двойное ожидание на барьере на каждом шаге цикла. Рассмотрим, к чему может привести снятие одного из ожиданий.

Барьеры. Применение

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

int const arrSize = 10;

//для обмена информацией между потоками
//первый поток записывает результат в M0, второй в M1
int M0;
int M1;

pthread_barrier_t barr ; //барьер
pthread_t mythread ;     //идентификатор для дочернего потока
```

Барьеры. Применение

```
// Генератор массива
void RndArray (int *A, int size) {
    for(int i = 0; i < size; i++) {
        A[i] = rand() % 3 + 1;
    }
}

// Вывод массива
void OutArray (int *A, int size) {
    for(int i = 0; i < size; i++) {
        std::cout << A[i] << "   ";
    }
    std:: cout << "\n";
}
```

Барьеры. Применение

```
//стартовая функция
void* func(void *param) {
    int p = *((int*)param);
    int Array[arrSize] ; //собственный массив потока
    RndArray(Array, arrSize);   // инициализация от 1 до 3

    // Согласование вывода данных
    pthread_mutex_lock(&mutex) ; //защита операции записи
    // Вывод результатов
    std::cout << "Source array. Thread " << p << ": ";
    OutArray(Array, arrSize);
    //конец критической секции
    pthread_mutex_unlock(&mutex);
```

Барьеры. Применение

```
    for (int i=0 ; i<arrSize ; i++) {
        //первый поток записывает элемент в M0, второй в M1
        if (p == 0 ) { M0=Array[i] ; }
        if (p == 1 ) { M1=Array[i] ; }
        pthread_barrier_wait(&barr) ; //ожидание на барьере
        //если элементы массивов ==, заменим их нулями
        if (M0 == M1) { Array[i]=0 ; }
        pthread_barrier_wait(&barr) ; //ожидание на барьере
    }
    // Согласование вывода данных
    pthread_mutex_lock(&mutex) ; //защита операции записи
    // Вывод результатов
    std::cout << "Result array. Thread " << p << ": ";
    OutArray(Array, arrSize);
    //конец критической секции
    pthread_mutex_unlock(&mutex);
    return nullptr;
}
```

Барьеры. Применение

```
int main() {
    int zero = 0;
    int one  = 1;
    auto seed = clock();
    srand(seed);

    pthread_mutex_init(&mutex, nullptr) ;

    //инициализация барьера со значением 2
    pthread_barrier_init(&barr, nullptr, 2) ;
    //создание дочернего потока
    pthread_create(&mythread, nullptr, func, (void *)&zero) ;
    func((void *)&one) ;

    pthread_join(mythread, nullptr);
    return 0;
}
```

Open Multi-Processing (OpenMP)

Открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран.

Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Первая версия появилась в 1997 году, предназначалась для языка Fortran.
Для C/C++ версия разработана в 1998 году.
В 2008 году вышла версия OpenMP 3.0. В июле 2014-го вышла версия 4.0

Open Multi-Processing (OpenMP)

OpenMP реализует параллельные вычисления с помощью многопоточности. Главный (master) поток создает набор подчиненных (slave) потоков и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами. Количество процессоров не обязательно должно быть больше или равно количеству потоков).

Задачи, выполняемые потоками параллельно, так же, как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка — прагм. Количество создаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне, при помощи переменных окружения.

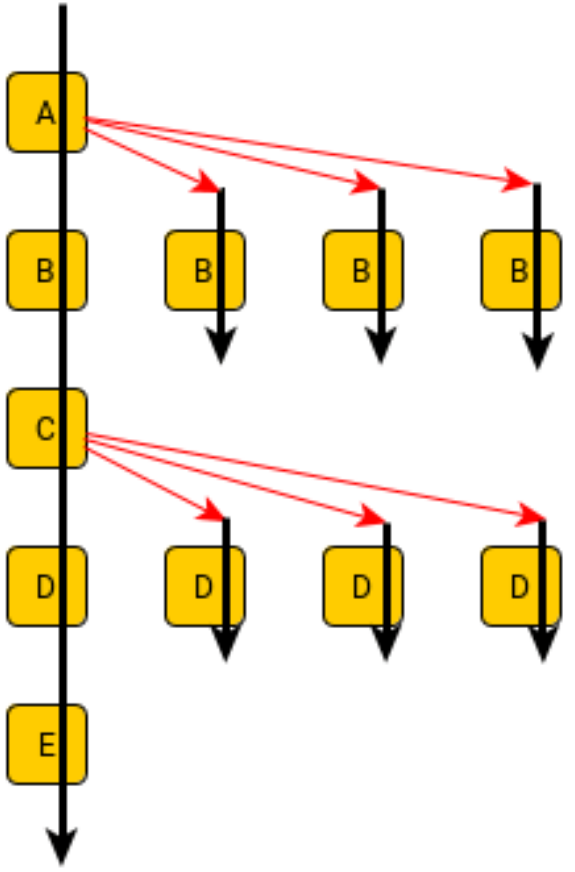
Open Multi-Processing (OpenMP)

Ключевые элементы:

- конструкции для создания потоков (parallel);
- конструкции распределения работы между потоками (DO/for и section);
- конструкции для управления данными (shared, private...);
- конструкции для синхронизации потоков (critical, atomic, barrier...);
- процедуры библиотеки времени выполнения (omp_get_thread_num...),
  переменные окружения

Open Multi-Processing (OpenMP)

```
#include "omp.h"

int main() {
  // A - single thread
#pragma omp parallel
  {
    // B - many threads
  }
  // C - single thread
#pragma omp parallel
  {
    // D - many threads
  }
  // E - single thread
}
```



Hello, OpenMP!

```
// c++ omp01.cpp -o ompcpp -fopenmp
#include <iostream>
#include <omp.h>

int main(int argc, char** argv)
{
    #pragma omp parallel
    {
        auto count = omp_get_thread_num();
        auto ItsMe = omp_get_num_threads();
        std::cout << "Hello, OpenMP! I am "
                  << count << " of " << ItsMe << "\n";
    }
    return 0;
}
```

1) value++;

```
// c++ main.cpp -fopenmp
#include <omp.h>
#include <iostream>

int main() {
  int value = 10;
  #pragma omp parallel
  {
    auto num = omp_get_thread_num();
    value++;
    #pragma omp critical
    {
        std::cout << " thread " << num
                  << ":  value = " << value << std::endl;
    }
  }
}
```

2) value++;

```
// c++ main.cpp -fopenmp
#include <omp.h>
#include <iostream>

int main() {
  int value = 10;
  #pragma omp parallel
  {
    auto num = omp_get_thread_num();
    #pragma omp critical
    {
        value++;
    }
    std::cout << " thread " << num
              << ":  value = " << value << std::endl;
  }
}
```

3) value++;

```
// c++ main.cpp -fopenmp
#include <omp.h>
#include <iostream>

int main() {
  int value = 10;
  #pragma omp parallel
  {
    auto num = omp_get_thread_num();
    #pragma omp critical
    {
        value++;
        std::cout << " thread " << num << ":  value = " <<
value << std::endl;
    }
  }
}
```

Интеграл. Метод прямоугольников

```
#include <cmath>
#include <functional>
#include <iostream>

double fTest(double x) {
    return 0.5 * x;
}

int main() {
    auto integralValie = rectIntegral(fTest, 0, 5, 10000000);
    std::cout << "Integral value = " << integralValie << "\n";
    return 0;
}
```

Интеграл. Метод прямоугольников

```
double rectIntegral(
    const std::function<double(double)> &fun,
    const double a, const double b, const int n)
{
    double h = fabs((b - a) / n);
    double sum = 0;

    #pragma omp parallel reduction (+: sum)
    {
        #pragma omp for
        for (int i = 0; i < n; ++i) {
            sum += fun(a + i * h) * h;
        }
    }
    return sum;
}
```

Используемые источники

1. Википедия. Многопоточность - https://ru.wikipedia.org/Многопоточность

2. Википедия. Поток выполнения - https://ru.wikipedia.org/wiki/Поток_выполнения

3. Википедия. Процесс (информатика) - https://ru.wikipedia.org/Процесс_(информатика)

4. Википедия. POSIX Threads https://ru.wikipedia.org/wiki/POSIX_Threads

5. Википедия. OpenMP - https://ru.wikipedia.org/wiki/OpenMP

6. Блог программиста - https://pro-prof.com/

Используемые источники

5.Уильямс Энтони. C++. Практика многопоточного программирования. — СПб.: Питер, 2020. — 640 с.
6. Грегори Р. Эндрюс. Основы многопоточного, параллельного и распределенного программирования. - М.: Издательский дом "Вильямс", 2003.
https://l.wzm.me/_coder/custom/parallel.programming/main.htm
7. Богачёв К. Ю. Основы параллельного программирования : учебное пособие / К. Ю. Богачёв. — 4-е изд., электрон. — М. : Лаборатория знаний, 2020. — 345 с.