

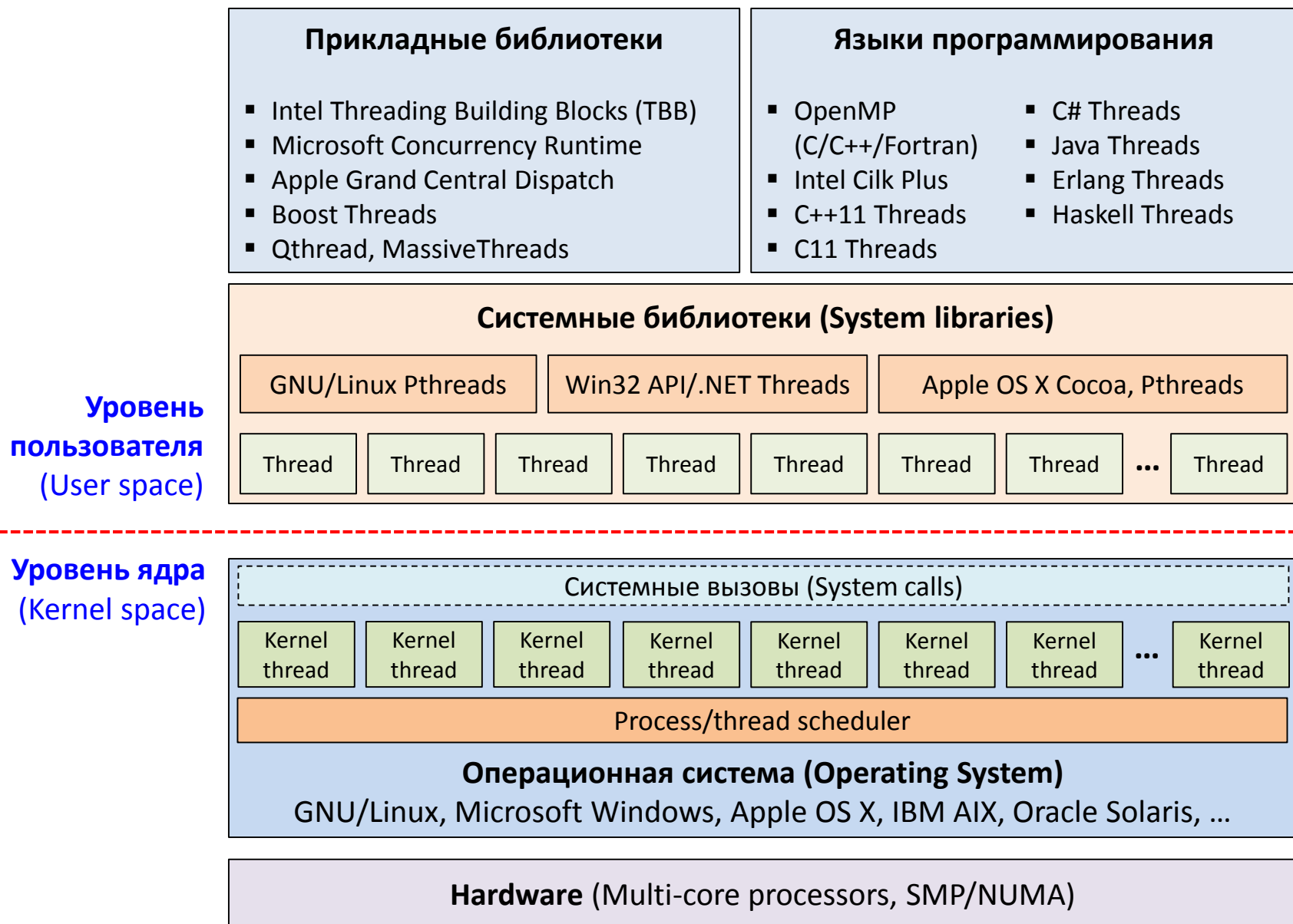
Лекция 7:
Многопоточное программирование
Часть 3
(Multithreading programming)

Курносов Михаил Георгиевич

**к.т.н. доцент Кафедры вычислительных систем
Сибирский государственный университет
телекоммуникаций и информатики**

<http://www.mkurnosov.net>

Программный инструментарий



Стандарт OpenMP

- **OpenMP (Open Multi-Processing)** – стандарт, определяющий набор директив компилятора, библиотечных процедур и переменных среды окружения для создания многопоточных программ
- Текущая версия стандарта – OpenMP 4.0
- Требуется поддержка со стороны компилятора



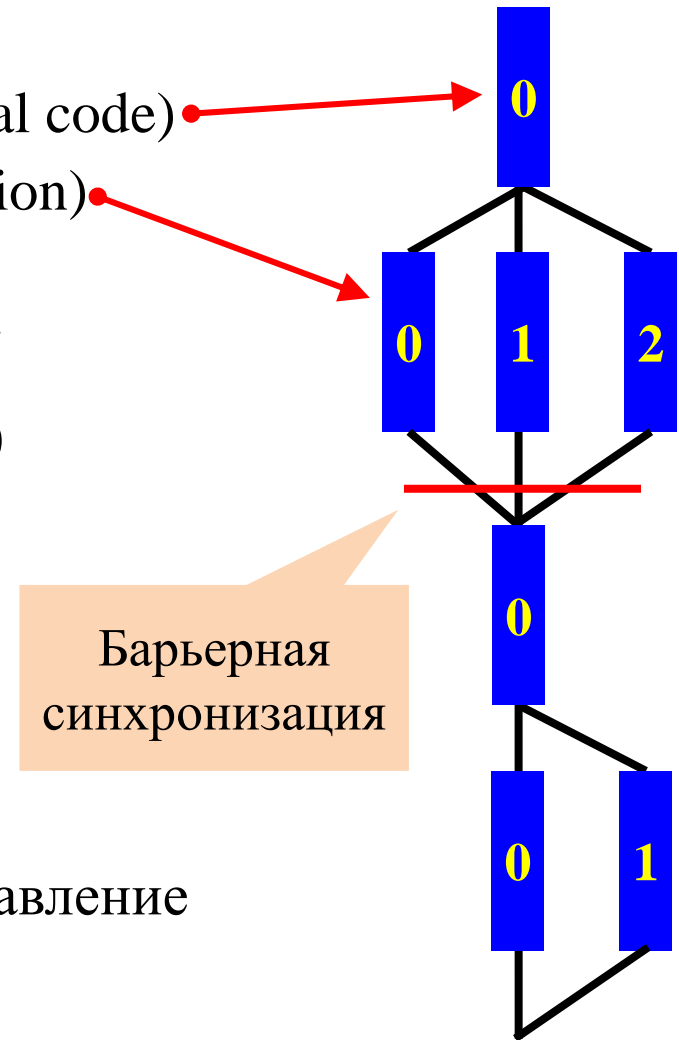
www.openmp.org

OpenMP

Compiler	Information
GNU GCC	Option: <code>-fopenmp</code> gcc 4.2 – OpenMP 2.5, gcc 4.4 – OpenMP 3.0, gcc 4.7 – OpenMP 3.1 gcc 4.9 – OpenMP 4.0
Clang (LLVM)	OpenMP 3.1 Clang + Intel OpenMP RTL http://clang-omp.github.io/
Intel C/C++, Fortran	OpenMP 3.1 Option: <code>-Qopenmp</code> , <code>-openmp</code>
Oracle Solaris Studio C/C++/Fortran	OpenMP 3.1 Option: <code>-xopenmp</code>
Microsoft Visual Studio 2012 C++	Option: <code>/openmp</code> OpenMP 2.0 only
Other compilers: IBM XL, PathScale, PGI, Absoft Pro, ...	

Структура OpenMP-программы

- Программа представляется в виде последовательных участков кода (serial code) и параллельных регионов (parallel region)
- Каждый поток имеет номер: 0, 1, 2, ...
- Главный поток (master) имеет номер 0
- Память процесса (heap) является общей для всех потоков
- OpenMP реализует динамическое управление потоками (task parallelism)
- OpenMP: data parallelism + task parallelism



Пример OpenMP-программы

```
#include <omp.h>

int main()
{
    #pragma omp parallel
    {
        printf("Thread %d\n", omp_get_thread_num());
    }

    return 0;
}
```

Компиляция OpenMP-программы

```
$ gcc -fopenmp -o prog ./prog.c
$ ./prog
Thread 0
Thread 1
Thread 3
Thread 2
```

```
$ export OMP_NUM_THREADS=2
$ ./prog
Thread 0
Thread 1
```

По умолчанию количество потоков = количеству
логических процессоров в системе

Пример OpenMP-программы

```
#include <omp.h>

int main()
{

#pragma omp parallel
{
#ifdef _OPENMP
    printf("Thread %d\n", omp_get_thread_num());
#endif
}

    return 0;
}
```


Директивы OpenMP

`#pragma omp <директива> [раздел [[,] раздел]...]`

- Создание потоков
- Распределение вычислений между потоками
- Управление пространством видимости переменных
- Механизмы синхронизации потоков
- ...

Создание потоков (parallel)

```
#pragma omp parallel
{
    /* Этот код выполняется всеми потоками */
}
```

```
#pragma omp parallel if (expr)
{
    /* Код выполняется потоками если expr = true */
}
```

```
#pragma omp parallel num_threads(n / 2)
{
    /* Создается n / 2 потоков */
}
```

На выходе из параллельного региона осуществляется барьерная синхронизация — все потоки ждут последнего

Создание потоков (sections)

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        /* Код потока 0 */
    }

    #pragma omp section
    {
        /* Код потока 1 */
    }
}
```

При любых условиях выполняется фиксированное количество потоков (по количеству секций)

Функции runtime-библиотеки

- `int omp_get_thread_num()` – возвращает номер текущего потока
- `int omp_get_num_threads()` – возвращает количество потоков в параллельном регионе
- `void omp_set_num_threads(int n)`
- `double omp_get_wtime()`

Директива master

```
#pragma omp parallel
{

    /* Этот код выполняется всеми потоками */

    #pragma omp master
    {
        /* Код выполняется только потоком 0 */
    }

    /* Этот код выполняется всеми потоками */

}
```

Директива `single`

```
#pragma omp parallel
{

    /* Этот код выполняется всеми потоками */

    #pragma omp single
    {
        /* Код выполняется только одним потоком */
    }

    /* Этот код выполняется всеми потоками */

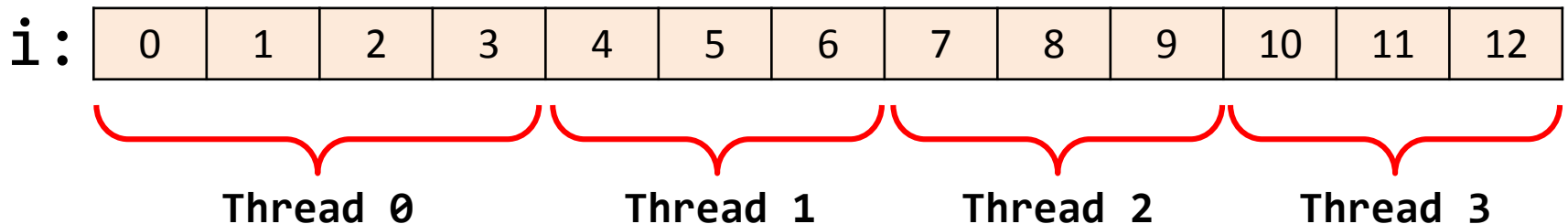
}
```

Директива for (data parallelism)

```
#define N 13

#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++) {
        printf("Thread %d i = %d\n",
            omp_get_thread_num(), i);
    }
}
```

Итерации цикла распределяются между потоками



Директива for

```
$ OMP_NUM_THREADS=4 ./prog
```

```
Thread 2 i = 7
```

```
Thread 2 i = 8
```

```
Thread 2 i = 9
```

```
Thread 0 i = 0
```

```
Thread 0 i = 1
```

```
Thread 0 i = 2
```

```
Thread 3 i = 10
```

```
Thread 3 i = 11
```

```
Thread 3 i = 12
```

```
Thread 0 i = 3
```

```
Thread 1 i = 4
```

```
Thread 1 i = 5
```

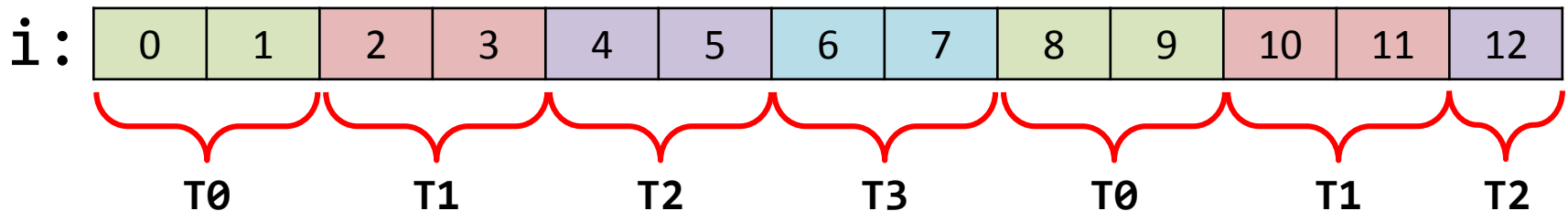
```
Thread 1 i = 6
```


Алгоритмы распределения итераций

```
#define N 13

#pragma omp parallel
{
    #pragma omp for schedule(static, 2)
    for (i = 0; i < N; i++) {
        printf("Thread %d i = %d\n",
            omp_get_thread_num(), i);
    }
}
```

Итерации цикла распределяются циклически (round-robin)
блоками по 2 итерации



Алгоритмы распределения итераций

```
$ OMP_NUM_THREADS=4 ./prog
```

```
Thread 0 i = 0
```

```
Thread 0 i = 1
```

```
Thread 0 i = 8
```

```
Thread 0 i = 9
```

```
Thread 1 i = 2
```

```
Thread 1 i = 3
```

```
Thread 1 i = 10
```

```
Thread 1 i = 11
```

```
Thread 3 i = 6
```

```
Thread 3 i = 7
```

```
Thread 2 i = 4
```

```
Thread 2 i = 5
```

```
Thread 2 i = 12
```

Алгоритмы распределения итераций

Алгоритм	Описание
static, m	Цикл делится на блоки по m итераций (до выполнения), которые распределяются по потокам
dynamic, m	Цикл делится на блоки по m итераций. При выполнении блока из m итераций поток выбирает следующий блок из общего пула
guided, m	Блоки выделяются динамически. При каждом запросе размер блока уменьшается экспоненциально до m
runtime	Алгоритм задается пользователем через переменную среды OMP_SCHEDULE

Директива for (ordered)

```
#define N 7

#pragma omp parallel
{
    #pragma omp for ordered
    for (i = 0; i < N; i++) {
        #pragma omp ordered
        printf("Thread %d i = %d\n",
            omp_get_thread_num(), i);
    }
}
```

- Директива ordered организует последовательное выполнение итераций ($i = 0, 1, \dots$) – **синхронизация**
- Поток с $i = k$ ожидает пока потоки с $i = k - 1, k - 2, \dots$ не выполнят свои итерации

Директива for (ordered)

```
$ OMP_NUM_THREADS=4 ./prog
```

```
Thread 0 i = 0
```

```
Thread 0 i = 1
```

```
Thread 1 i = 2
```

```
Thread 1 i = 3
```

```
Thread 2 i = 4
```

```
Thread 2 i = 5
```

```
Thread 3 i = 6
```

Директива for (nowait)

```
#define N 7

#pragma omp parallel
{
    #pragma omp for nowait
    for (i = 0; i < N; i++) {
        printf("Thread %d i = %d\n",
            omp_get_thread_num(), i);
    }
}
```

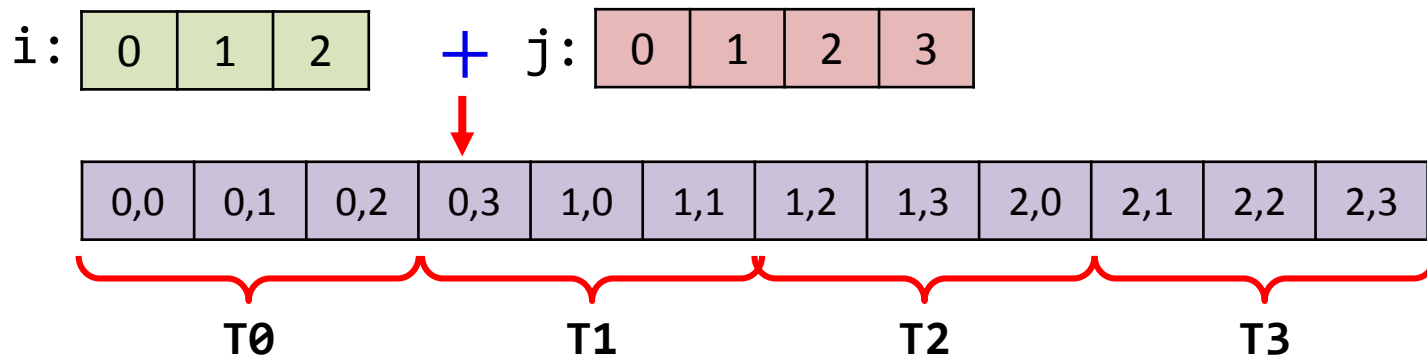
- По окончании цикла потоки не выполняют барьерную синхронизацию
- Конструкция **nowait** применима и к директиве sections

Директива for (collapse)

```
#define N 3
#define M 4

#pragma omp parallel
{
    #pragma omp for collapse(2)
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++)
            printf("Thread %d i = %d\n",
                omp_get_thread_num(), i);
    }
}
```

- **collapse(n)** объединяет пространство итераций *n* циклов в одно



Директива for (collapse)

```
$ OMP_NUM_THREADS=4 ./prog
```

```
Thread 2 i = 1
```

```
Thread 2 i = 1
```

```
Thread 2 i = 2
```

```
Thread 0 i = 0
```

```
Thread 0 i = 0
```

```
Thread 0 i = 0
```

```
Thread 3 i = 2
```

```
Thread 3 i = 2
```

```
Thread 3 i = 2
```

```
Thread 1 i = 0
```

```
Thread 1 i = 1
```

```
Thread 1 i = 1
```


Ошибки в многопоточных программах

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec(1000);

    std::fill(vec.begin(), vec.end(), 1);
    int counter = 0;

    #pragma omp parallel for
    for (std::vector<int>::size_type i = 0;
         i < vec.size(); i++)
    {
        if (vec[i] > 0) {
            counter++;
        }
    }
    std::cout << "Counter = " << counter << std::endl;
    return 0;
}
```

Ошибки в многопоточных программах

```
$ g++ -fopenmp -o ompprog ./ompprog.cpp
```

```
$ ./omprog  
Counter = 381
```

```
$ ./omprog  
Counter = 909
```

```
$ ./omprog  
Counter = 398
```

На каждом запуске итоговое значение Counter разное!

Правильный результат Counter = 1000

Ошибки в многопоточных программах

```
#include <iostream>
#include <vector>
```

```
int main()
{
```

```
    std::vector<
```

```
    std::fill(vec
```

```
    int counter
```

```
#pragma omp para
```

```
    for (std::ve
```

```
        i < vec.size())
```

```
    {
```

```
        if (vec[i] > 0) {
```

```
            counter++;
```

```
        }
```

```
    }
```

```
    std::cout << "Counter = " << counter << std::endl;
```

```
    return 0;
```

```
}
```

Потоки осуществляют конкурентный доступ к переменной counter – одновременно читают её и записывают

Состояние гонки (Race condition, data race)

```
#pragma omp parallel
{
    counter++;
}
```

C++
→

```
movl [counter], %eax
incl %eax
movl %eax, [counter]
```

Идеальная последовательность выполнения инструкций 2-х потоков

Thread 0	Thread 1		Memory (counter)
			0
movl [counter], %eax		←	0
incl %eax			0
movl %eax, [counter]		→	1
	movl [counter], %eax	←	1
	incl %eax		1
	movl %eax, [counter]	→	2

counter = 2

Состояние гонки (Race condition, data race)

```
#pragma omp parallel
{
    counter++;
}
```

C++
→

```
movl [counter], %eax
incl %eax
movl %eax, [counter]
```

Возможная последовательность выполнения инструкций 2-х потоков

Thread 0	Thread 1		Memory (counter)
			0
movl [counter], %eax		←	0
incl %eax	movl [counter], %eax	←	0
movl %eax, [counter]	incl %eax	→	1
	movl %eax, [counter]	→	1
			1

Error: Data race

counter = 1

Состояние гонки (Race condition, data race)

- **Состояние гонки (Race condition, data race)** – это состояние программы, в которой несколько потоков одновременно конкурируют за доступ к общей структуре данных (для чтения/записи)
- Порядок выполнения потоков заранее не известен – носит случайный характер
- Планировщик динамически распределяет процессорное время, учитывая текущую загрузженность процессорных ядер, а нагрузку (потоки, процессы) создают пользователи, поведение которых носит случайных характер
- Состояние гонки данных (Race condition, data race) трудно обнаруживается в программах и воспроизводится в тестах
- Состояние гонки данных (Race condition, data race) – это типичный пример **Гейзенбага (Heisenbug)**

Обнаружение состояния гонки (Data race)

Динамические анализаторы


- Valgrind Helgrind, DRD
- Intel Thread Checker
- Oracle Studio Thread Analyzer
- Java ThreadSanitizer
- Java Chord

Статические анализаторы кода

- PVS-Studio (viva64)
- ...

Valgrind Helgrind

```
$ g++ -fopenmp -o ompprog ./ompprog.cpp
$ valgrind --helgrind ./ompprog
```



```
==8238== Helgrind, a thread error detector
==8238== Copyright (C) 2007-2012, and GNU GPL'd, by OpenWorks LLP et al.
==8238== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==8238== Command: ./ompprog report
...
==8266== -----
==8266== Possible data race during write of size 4 at 0x7FEFFD358 by thread #3
==8266== Locks held: none
==8266==    at 0x400E6E: main._omp_fn.0 (ompprog.cpp:14)
==8266==    by 0x3F84A08389: ??? (in /usr/lib64/libgomp.so.1.0.0)
==8266==    by 0x4A0A245: ??? (in /usr/lib64/valgrind/vgpreload_helgrind-amd64-linux.so)
==8266==    by 0x34CFA07C52: start_thread (in /usr/lib64/libpthread-2.17.so)
==8266==    by 0x34CF2F5E1C: clone (in /usr/lib64/libc-2.17.so)
==8266==
==8266== This conflicts with a previous write of size 4 by thread #1
==8266== Locks held: none
==8266==    at 0x400E6E: main._omp_fn.0 (ompprog.cpp:14)
==8266==    by 0x400CE8: main (ompprog.cpp:11)...
```


Директивы синхронизации

- Директивы синхронизации позволяют управлять порядком выполнения заданных участков кода потоками
- **#pragma omp critical**
- **#pragma omp atomic**
- **#pragma omp ordered**
- **#pragma omp barrier**

Критические секции

```
#pragma omp parallel for private(v)
for (i = 0; i < n; i++) {
    v = fun(a[i]);
    #pragma omp critical
    {
        sum += v;
    }
}
```

Критические секции

```
#pragma omp parallel for private(v)
for (i = 0; i < n; i++) {
    v = fun(a[i]);
    #pragma omp critical
    {
        sum += v;
    }
}
```

- **Критическая секция (Critical section)** – участок кода в многопоточной программе, выполняемый всеми потоками последовательно
- Критические секции снижают степень параллелизма

Управление видимостью переменных

- **private(list)** – во всех потоках создаются локальные копии переменных (начальное значение)
- **firstprivate(list)** – во всех потоках создаются локальные копии переменных, которые инициализируются их значениями до входа в параллельный регион
- **lastprivate(list)** – во всех потоках создаются локальные копии переменных. По окончании работы всех потоков локальная переменная вне параллельного региона обновляется значением этой переменной одного из потоков
- **shared(list)** – переменные являются общими для всех потоков

Атомарные операции

```
#pragma omp parallel for private(v)
for (i = 0; i < n; i++) {
    v = fun(a[i]);
    #pragma omp atomic
    sum += v;
}
```

Атомарные операции

```
#pragma omp parallel for private(v)
for (i = 0; i < n; i++) {
    v = fun(a[i]);
    #pragma omp atomic
    sum += v;
}
```

- Атомарные операции “легче” критических секций (не используют блокировки)
- Lock-free algorithms & data structures

Параллельная редукция

```
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < n; i++) {
    sum = sum + fun(a[i]);
}
```

- Операции директивы reduction:

+, *, -, &, |, ^, &&, ||, max, min

- OpenMP 4.0 поддерживает пользовательские функции редукции

Директивы синхронизации

```
#pragma omp parallel
{
    /* Code */
    #pragma omp barrier
    /* Code */
}
```

- Директива **barrier** осуществляет ожидание достижения данной точки программы всеми потоками

#pragma omp flush

```
#pragma omp parallel
{
    /* Code */
    #pragma omp flush(a, b)
    /* Code */
}
```

- Принудительно обновляет в памяти значения переменных (Memory barrier)
- Например, в одном потоке выставляем флаг (сигнал к действию) для другого

Умножение матриц v1.0

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                c[i][j] = c[i][j] +
                    a[i][k] * b[k][j];
            }
        }
    }
}
```

Умножение матриц v1.0

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                c[i][j] = c[i][j] +
                    a[i][k] * b[k][j];
            }
        }
    }
}
```

Ошибка!

Переменные j, k – общие для всех потоков!

Умножение матриц v2.0

```
#pragma omp parallel
{
#pragma omp for shared(a, b, c) private(j, k)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                c[i][j] = c[i][j] +
                    a[i][k] * b[k][j];
            }
        }
    }
}
```

Директива task (OpenMP 3.0)

```
int fib(int n)
{
    if (n < 2)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

- Директива task создает задачу (легковесный поток)
- Задачи из пула динамически выполняются группой потоков
- Динамическое распределение задачи по потокам осуществляется алгоритмами планирования типа work stealing
- Задач может быть намного больше количества потоков

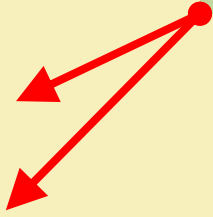
Директива task (OpenMP 3.0)

```
int fib(int n)
{
    int x, y;


    if (n < 2)
        return n;
    #pragma omp task shared(x, n)
        x = fib(n - 1);
    #pragma omp task shared(y, n)
        y = fib(n - 2);
    #pragma omp taskwait
        return x + y;
}

#pragma omp parallel
#pragma omp single
    val = fib(n);
```

Каждый
рекурсивный
вызов — это задача



Ожидаем
завершение
дочерних задач



Пример Primes (sequential code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```

- Программа подсчитывает количество простых чисел в интервале [start, end]

Пример Primes (serial code)

```
int is_prime_number(int num)
{
    int limit, factor = 3;

    limit = (int)(sqrtf((double)num) + 0.5f);
    while ((factor <= limit) && (num % factor))
        factor++;
    return (factor > limit);
}
```


Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```

Data race

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        #pragma omp critical
        nprimes++;
}
```

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        #pragma omp critical
        nprimes++;
}
```

Увеличение счетчика можно
реализовать без блокировки

(Lock-free algorithm)

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for reduction(+:nprimes)
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

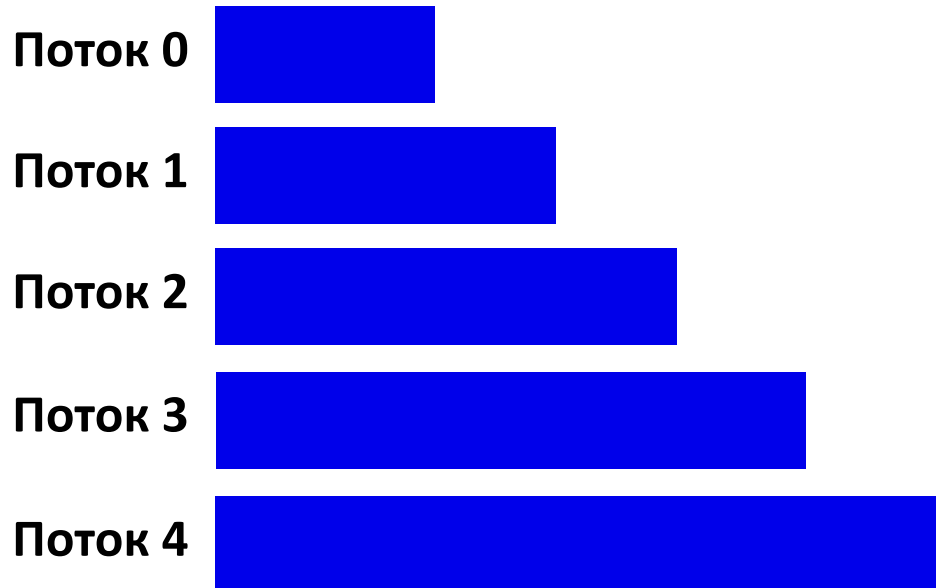
nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for reduction(+:nprimes)
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```

Время выполнения
`is_prime_number(i)`
зависит от значения *i*

Пример Primes (parallel code)

```
#pragma omp parallel for reduction(+:nprimes)
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```



Время выполнения
потоков различно!

Load Imbalance

Пример Primes (parallel code)

```
start = atoi(argv[1]);
end = atoi(argv[2]);

if ((start % 2) == 0 )
    start = start + 1;

nprimes = 0;
if (start <= 2)
    nprimes++;

#pragma omp parallel for schedule(static, 1)
                                reduction(+:nprimes)
for (i = start; i <= end; i += 2) {
    if (is_prime_number(i))
        nprimes++;
}
```


Блокировки (locks)

- **Блокировка, мьютекс (lock, mutex)** – это объект синхронизации, который позволяет ограничить одновременный доступ потоков к разделяемым ресурсам (реализует взаимное исключение)
- **OpenMP:** `omp_lock_set/omp_lock_unset`
- **POSIX Pthreads:** `pthread_mutex_lock/pthread_mutex_unlock`
- **C++11:** `std::mutex::lock/std::mutex::unlock`
- **C11:** `mtx_lock/mtx_unlock`
- **Блокировка (lock)** может быть рекурсивной (вложенной) – один поток может захватывать блокировку несколько раз

Spin locks (циклические блокировки)

- **Spin lock (блокировка в цикле)** – это вид блокировки, при котором процесс ожидающий освобождения блокировки не “засыпает”, а выполняет цикл ожидания (busy waiting)
- Рекомендуется использовать если время пребывания в критической секции меньше времени переключения контекстов
- Плюсы: spin lock позволяет быстро среагировать на освобождение блокировки
- Минусы: spin lock всегда занимает ресурсы процессорного ядра

```
pthread_spinlock_t lock;  
pthread_spin_init(&lock, PTHREAD_PROCESS_PRIVATE);  
  
pthread_spin_lock(&lock);  
counter++;  
pthread_spin_unlock(&lock);  
  
pthread_spin_destroy(&lock);
```

Блокировки чтения-записи (rwlocks)

- **Read-write lock** – это вид блокировки, которая позволяет разграничить доступ потоков на запись и чтение разделяемых структур данных
- Блокировка на запись не может быть получена, пока не освобождены все блокировки на чтение (rdlock)

```
int readData(int i, int j)
{
    pthread_rwlock_rdlock(&lock);
    int result = data[i] + data[j];
    pthread_rwlock_unlock(&lock);
    return result;
}

void writeData(int i, int j, int value)
{
    pthread_rwlock_wrlock(&lock);
    data[i] += value;
    data[j] -= value;
    pthread_rwlock_unlock(&lock);
}
```

Блокировки (locks)

```
#include <omp.h>

int main()
{
    std::vector<int> vec(1000);

    std::fill(vec.begin(), vec.end(), 1);
    int counter = 0;
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel for
    for (std::vector<int>::size_type i = 0; i < vec.size(); i++) {
        if (vec[i] > 0) {
            omp_set_lock(&lock);
            counter++;
            omp_unset_lock(&lock);
        }
    }
    omp_destroy_lock(&lock);
    std::cout << "Counter = " << counter << std::endl;
    return 0;
}
```

Взаимная блокировка (Deadlock)

- **Взаимная блокировка (deadlock, тупик)** – ситуация когда два и более потока находятся в состоянии бесконечного ожидания ресурсов, захваченных этими потоками
- **Самоблокировка (self deadlock)** – ситуация когда поток пытается повторно захватить блокировку, которую уже захватил (deadlock возникает если блокировка не является рекурсивной)

Взаимная блокировка (Deadlock)

```
void deadlock_example()  
{  
#pragma omp sections  
{  
    #pragma omp section  
    {  
        omp_lock_t lock1, lock2;  
        omp_set_lock(&lock1);  
        omp_set_lock(&lock2);  
        // Code  
        omp_unset_lock(&lock2);  
        omp_unset_lock(&lock1);  
    }  
    #pragma omp section  
    {  
        omp_lock_t lock1, lock2;  
        omp_set_lock(&lock2);  
        omp_set_lock(&lock1);  
        // Code  
        omp_unset_lock(&lock1);  
        omp_unset_lock(&lock2);  
    }  
}  
}
```

1. T0
захватывает
Lock1

2. T0 ожидает
Lock2

1. T1
захватывает
Lock2

2. T1 ожидает
Lock1

OpenMP 4.0: Поддержка ускорителей (GPU)

```
sum = 0;
#pragma omp target device(acc0) in(B,C)
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < N; i++)
    sum += B[i] * C[i]
```

- `omp_set_default_device()`
- `omp_get_default_device()`
- `omp_get_num_devices()`

OpenMP 4.0: SIMD-конструкции

- SIMD-конструкции для векторизации циклов (SSE, AVX2, AVX-512, AltiVec, ...)

```
void minex(float *a, float *b, float *c, float *d)
{
    #pragma omp parallel for simd
    for (i = 0; i < N; i++)
        d[i] = min(distsq(a[i], b[i]), c[i]);
}
```


OpenMP 4.0: Thread Affinity

- Thread affinity – привязка потоков к процессорным ядрам
- `#pragma omp parallel proc_bind(master | close | spread)`
- `omp_proc_bind_t omp_get_proc_bind(void)`
- Env. variable `OMP_PLACES`
- `export OMP_NUM_THREADS=16`
- `export OMP_PLACES=0,8,1,9,2,10,3,11,4,12,5,13,6,14,7,15`
- `export OMP_PROC_BIND=spread,close`

OpenMP 4.0: user defined reductions

```
#pragma omp declare reduction (reduction-identifier :  
                                typename-list : combiner) [identity(identity-expr)]
```

```
#pragma omp declare reduction (merge : std::vector<int> :  
                                omp_out.insert(omp_out.end(),  
                                                omp_in.begin(), omp_in.end()  
                                                ))  
  
void schedule(std::vector<int> &v, std::vector<int> &filtered)  
{  
    #pragma omp parallel for reduction (merge : filtered)  
    for (std::vector<int>::iterator it = v.begin();  
         it < v.end(); it++)  
    {  
        if (filter(*it))  
            filtered.push_back(*it);  
    }  
}
```

POSIX Threads

- **POSIX Threads** – это стандарт (POSIX.1c Threads extensions (IEEE Std 1003.1c-1995)), в котором определяется API для создания и управления потоками
- Библиотека **pthread** (pthread.h) ~ 100 функций
 - Thread management - creating, joining threads etc.
 - Mutexes
 - Condition variables
 - Synchronization between threads using read/write locks and barriers
- Семфаторы POSIX (префикс sem_) могут работать с потоками pthread, но не являются частью стандарта (определены в стандарте POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993))

POSIX pthreads API

- Все типы данных и функции начинаются с префикса `pthread_`

Prefix	Functional group
<code>pthread_</code>	Threads themselves and miscellaneous subroutines
<code>pthread_attr_</code>	Thread attributes objects
<code>pthread_mutex_</code>	Mutexes
<code>pthread_mutexattr_</code>	Mutex attributes objects.
<code>pthread_cond_</code>	Condition variables
<code>pthread_condattr_</code>	Condition attributes objects
<code>pthread_key_</code>	Thread-specific data keys
<code>pthread_rwlock_</code>	Read/write locks
<code>pthread_barrier_</code>	Synchronization barriers

<https://computing.llnl.gov/tutorials/pthreads>

POSIX pthreads API

- Компиляция программы с поддержкой POSIX pthreads API

Compiler / Platform	Compiler Command	Description
Intel GNU/Linux	<code>icc -pthread</code>	C
	<code>icpc -pthread</code>	C++
PGI GNU/Linux	<code>pgcc -lpthread</code>	C
	<code>pgCC -lpthread</code>	C++
GNU GCC GNU/Linux, Blue Gene	<code>gcc -pthread</code>	GNU C
	<code>g++ -pthread</code>	GNU C++
IBM Blue Gene	<code>bgx1c_r / bgcc_r</code>	C (ANSI / non-ANSI)
	<code>bgx1C_r, bgx1c++_r</code>	C++

Создание потоков

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

- Создает поток с заданными атрибутами attr и запускает в нем функцию start_routine, передавая ей аргумент arg
- Количество создаваемых в процессе потоков стандартом не ограничивается и зависит от реализации
- Размер стека потока можно задать через атрибут потока attr
- Размер стека по умолчанию: getrlimit(RLIMIT_STACK, &rlim);

```
$ ulimit -s          # The maximum stack size  
8192  
$ ulimit -u          # The maximum number of processes  
1024                # available to a single user
```

Завершение потоков

- Возврат (return) из стартовой функции (start_routine)
- Вызов pthread_exit()
- Вызов pthread_cancel() другим потоком
- Процесс (и его потоки) завершаются вызовом exit()

Создание и завершение потоков

```
#include <pthread.h>
#define NTHREADS 5

void *thread_fun(void *threadid) {
    long tid = (long)threadid;
    printf("Hello from %ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NTHREADS];
    int rc; long t;

    for (t = 0; t < NTHREADS; t++) {
        rc = pthread_create(&threads[t], NULL, thread_fun, (void *)t);
        if (rc) {
            printf("ERROR %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```


Создание и завершение потоков

```
$ gcc -pthread -o prog ./prog.c
```

```
$ ./prog
```

```
Hello from 1!
```

```
Hello from 4!
```

```
Hello from 0!
```

```
Hello from 2!
```

```
Hello from 3!
```

```
int rc; long t;

for (t = 0; t < NTHREADS; t++) {
    rc = pthread_create(&threads[t], NULL, thread_fun, (void *)t);
    if (rc) {
        printf("ERROR %d\n", rc);
        exit(-1);
    }
}
pthread_exit(NULL);
}
```

Ожидание потоков

- Функция **pthread_join()** – позволяет дождаться завершения заданного потока
- Поток может быть типа “detached” или “joinable” (default)
- К detached-потoku не применима функция pthread_join (поток создается и существует независимо от других)
- Joinable-поток требует хранения дополнительных данных
- Тип потока можно задать через его атрибуты или вызвав функцию **pthread_detach**

Ожидание потоков

```
#include <pthread.h>
#define NTHREADS 5

// ...

int main(int argc, char *argv[]) {
    pthread_t threads[NTHREADS];
    int rc; long t;
    void *status;

    for (t = 0; t < NTHREADS; t++) {
        rc = pthread_create(&threads[t], NULL, thread_fun,
                           (void *)t);
    }

    for (t = 0; t < NTHREADS; t++) {
        rc = pthread_join(threads[t], &status);
    }
    pthread_exit(NULL);
}
```

Синхронизация потоков

- Функция `pthread_self()` – возвращает идентификатор потока
- Функция `pthread_equal()` – позволяет сравнить идентификаторы двух потоков

Взаимные исключения (mutex)

- **Mutex** (mutual exclusion) – это объект синхронизации “взаимное исключение”
- Мьютексы используются для создания критических секций (critical sections) – областей кода, которые выполняются в любой момент времени только одним потоком
- В критических секциях, как правило, содержится код работы с разделяемыми переменными
- `pthread_mutex_init()` – инициализирует мьютекс
- `pthread_mutex_destroy()` – уничтожает мьютекс
- `pthread_mutex_lock()` – блокирует выполнение потока, пока он не захватит (acquire) мьютекс
- `pthread_mutex_trylock()` – осуществляет попытку захватить мьютекс
- `pthread_mutex_unlock()` – освобождает (release) мьютекс

Взаимные исключения (mutex)

```
node_t *l1ist_delete(int value)
{
    node_t *prev, *current;
    prev = &head;

    pthread_mutex_lock(&prev->lock);
    while ((current = prev->link) != NULL) {
        pthread_mutex_lock(&current->lock);
        if (current->value == value) {
            prev->link = current->link;
            pthread_mutex_unlock(&current->lock);
            pthread_mutex_unlock(&prev->lock);
            current->link = NULL;
            return current;
        }
        pthread_mutex_unlock(&prev->lock);
        prev = current;
    }
    pthread_mutex_unlock(&prev->lock);
    return NULL;
}
```

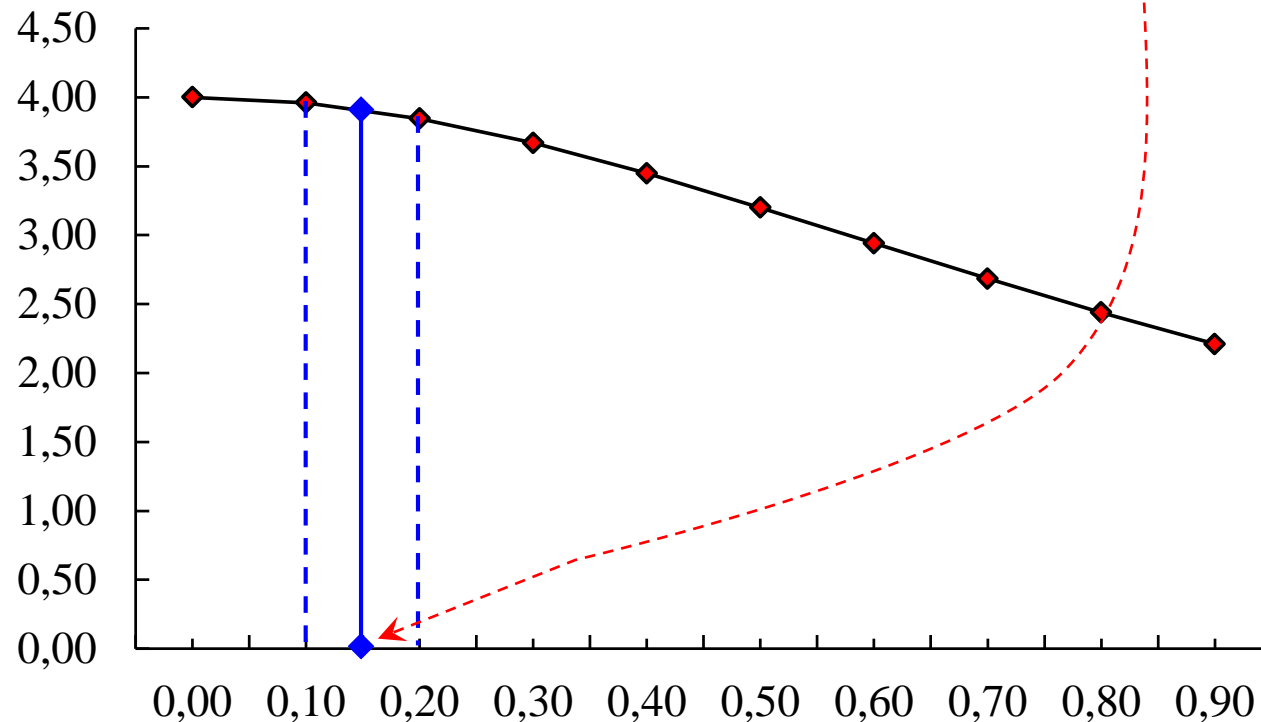
Вычисление числа π

- Приближенное вычисление числа π

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

$$\pi \approx h \sum_{i=1}^n \frac{4}{1 + \underbrace{(h(i - 0.5))^2}_{\text{green bracket}}}$$

$$h = \frac{1}{n}$$



pi.c

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

volatile long double pi = 0.0;
pthread_mutex_t piLock;
long double intervals;
int numThreads;

void *computePI(void *id)
{
    long double x, width, localSum = 0;
    int i, threadID = *((int*)id);    width = 1.0 / intervals;
    for (i = threadID ; i < intervals; i += numThreads) {
        x = (i + 0.5) * width;
        localSum += 4.0 / (1.0 + x * x);
    }
    localSum *= width;
    pthread_mutex_lock(&piLock);
    pi += localSum;
    pthread_mutex_unlock(&piLock);
    return NULL;
}
```


pi.c (продолжение)

```
int main(int argc, char **argv)
{
    pthread_t *threads;
    void *retval;
    int *threadID;
    int i;

    if (argc == 3) {
        intervals = atoi(argv[1]);
        numThreads = atoi(argv[2]);
        threads = malloc(numThreads * sizeof(pthread_t));
        threadID = malloc(numThreads * sizeof(int));
        pthread_mutex_init(&piLock, NULL);
        for (i = 0; i < numThreads; i++) {
            threadID[i] = i;
            pthread_create(&threads[i], NULL, computePI, threadID + i);
        }
        for (i = 0; i < numThreads; i++)
            pthread_join(threads[i], &retval);

        printf("Estimation of pi is %32.30Lf \n", pi);
    } else {
        printf("Usage: ./a.out <numIntervals> <numThreads>\n");
    }
    return 0;
}
```

Windows API для многопоточной обработки

- **Win32 API Threads `CreateThread`** – СИСТЕМНЫЙ ВЫЗОВ

```
HANDLE WINAPI CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

- **С run-time library `_beginthread`** (обертка вокруг `CreateThread`, корректно инициализирует libc)

```
uintptr_t _beginthread(void *start_address(void *),
                        unsigned int stack_size,
                        void *arglist);
```

- **MFC `AfxBeginThread`** – MFC-обертка вокруг `CreateThread`

```
CWinThread *AfxBeginThread(...);
```

Windows API для многопоточной обработки

- .NET System.Threading

```
public sealed class Thread : CriticalFinalizerObject, _Thread
```

C11 threads

```
#include <threads.h>

void threadfun()
{
    printf("Hello from thread\n");
}

int main()
{
    thrd_t tid;
    int rc;

    rc = thrd_create(&tid, threadfun, NULL)
    if (rc != thrd_success) {
        fprintf(stderr, "Error creating thread\n");
        exit(1);
    }
    thrd_join(tid, NULL);
    return 0;
}
```

Ссылки

- Эхтер Ш., Робертс Дж. **Многоядерное программирование.** – СПб.: Питер, 2010. – 316 с.
- Эндрюс Г.Р. **Основы многопоточного, параллельного и распределенного программирования.** – М.: Вильямс, 2003. – 512 с.
- Darryl Gove. **Multicore Application Programming: for Windows, Linux, and Oracle Solaris.** – Addison-Wesley, 2010. – 480 p.
- Maurice Herlihy, Nir Shavit. **The Art of Multiprocessor Programming.** – Morgan Kaufmann, 2008. – 528 p.
- Richard H. Carver, Kuo-Chung Tai. **Modern Multithreading : Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs.** – Wiley-Interscience, 2005. – 480 p.
- Anthony Williams. **C++ Concurrency in Action: Practical Multithreading.** – Manning Publications, 2012. – 528 p.
- Träff J.L. **Introduction to Parallel Computing //**
<http://www.par.tuwien.ac.at/teach/WS12/ParComp.html>