

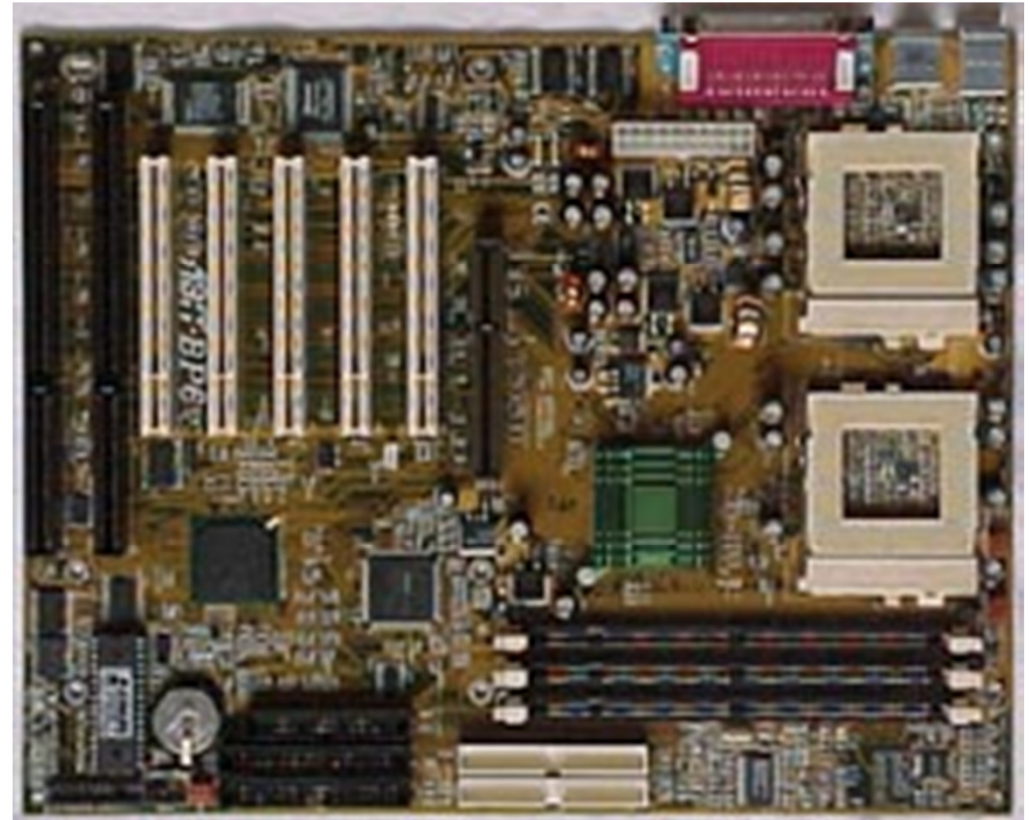
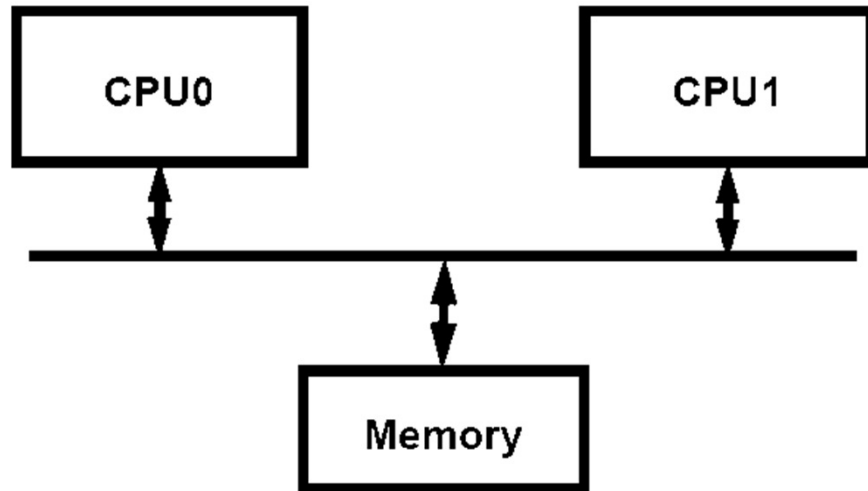
Лекция 5:
Многопоточное программирование.
Стандарт OpenMP
(Multithreaded programming)

Курносов Михаил Георгиевич

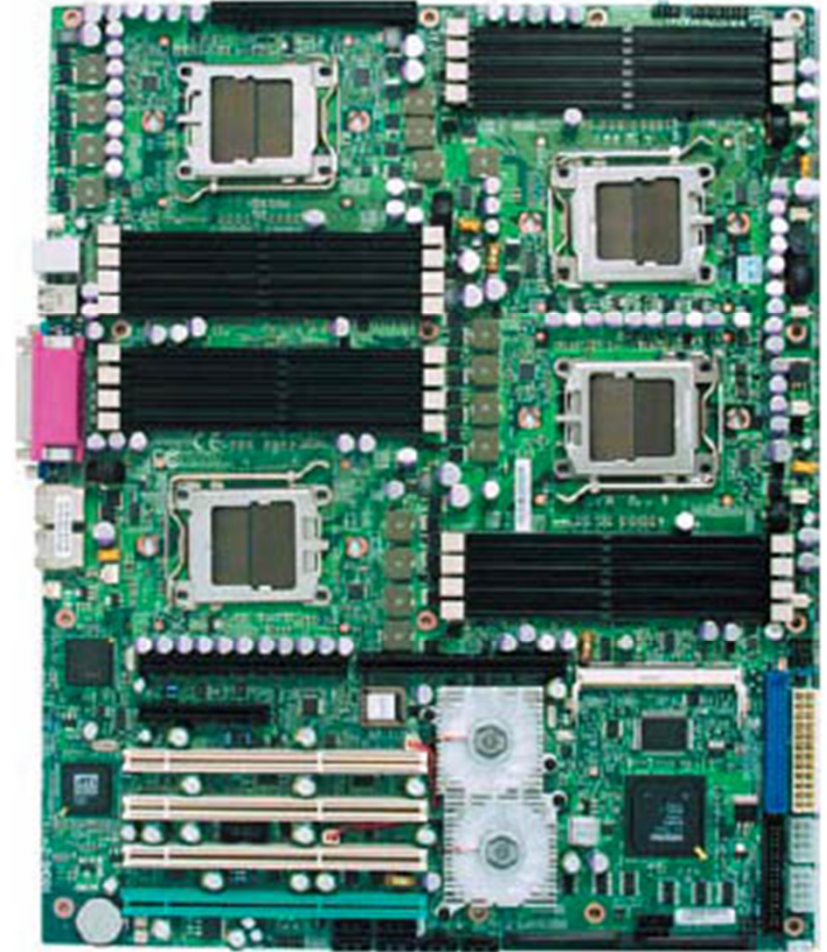
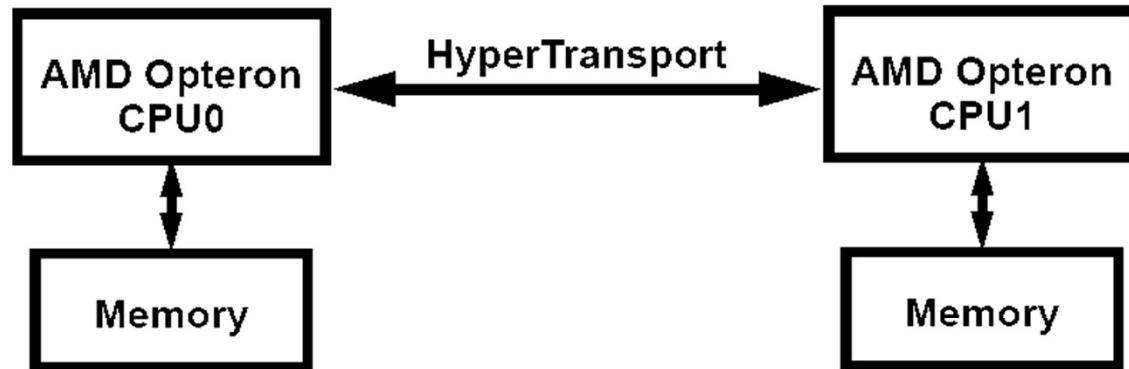
к.т.н. доцент Кафедры вычислительных систем
Сибирский государственный университет
телекоммуникаций и информатики

<http://www.mkurnosov.net>

Symmetric multiprocessing (SMP)



Non-Uniform Memory Architecture (NUMA)



Многопроцессорные ВС с общей памятью

- Операционные системы оперируют понятием логический процессор (Logical processor).
- Логический процессор может быть сопоставлен как ядру процессора, так как и ядру процессора с HyperThreading.
- Используется режим разделения времени – процессы пользователей разделяют во времени ресурсы процессорных ядер. Эта задача решается планировщиком операционной системы.

GNU/Linux

```
$ cat /proc/cpuinfo
```

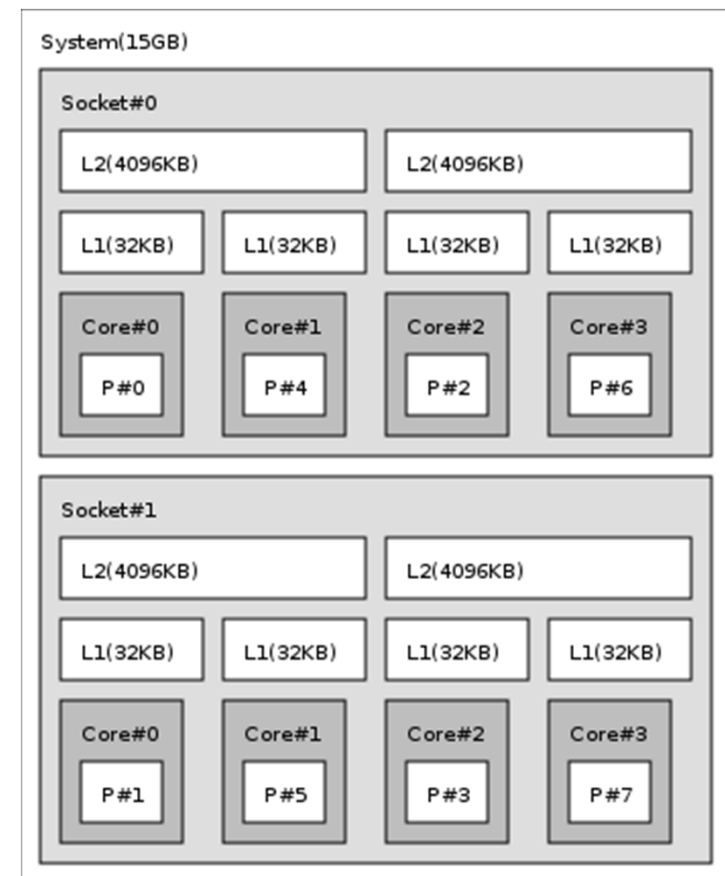
```
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 42
model name    : Intel(R) Core(TM) i5-2520M
stepping      : 7
microcode     : 0x28
cpu MHz       : 800.000
cache size    : 3072 KB
physical id   : 0
siblings      : 4
core id       : 0
cpu cores     : 2
...
```

GNU/Linux NUMA nodes

- Информация о NUMA-узлах расположена в каталоге:

`/sys/devices/system/node`

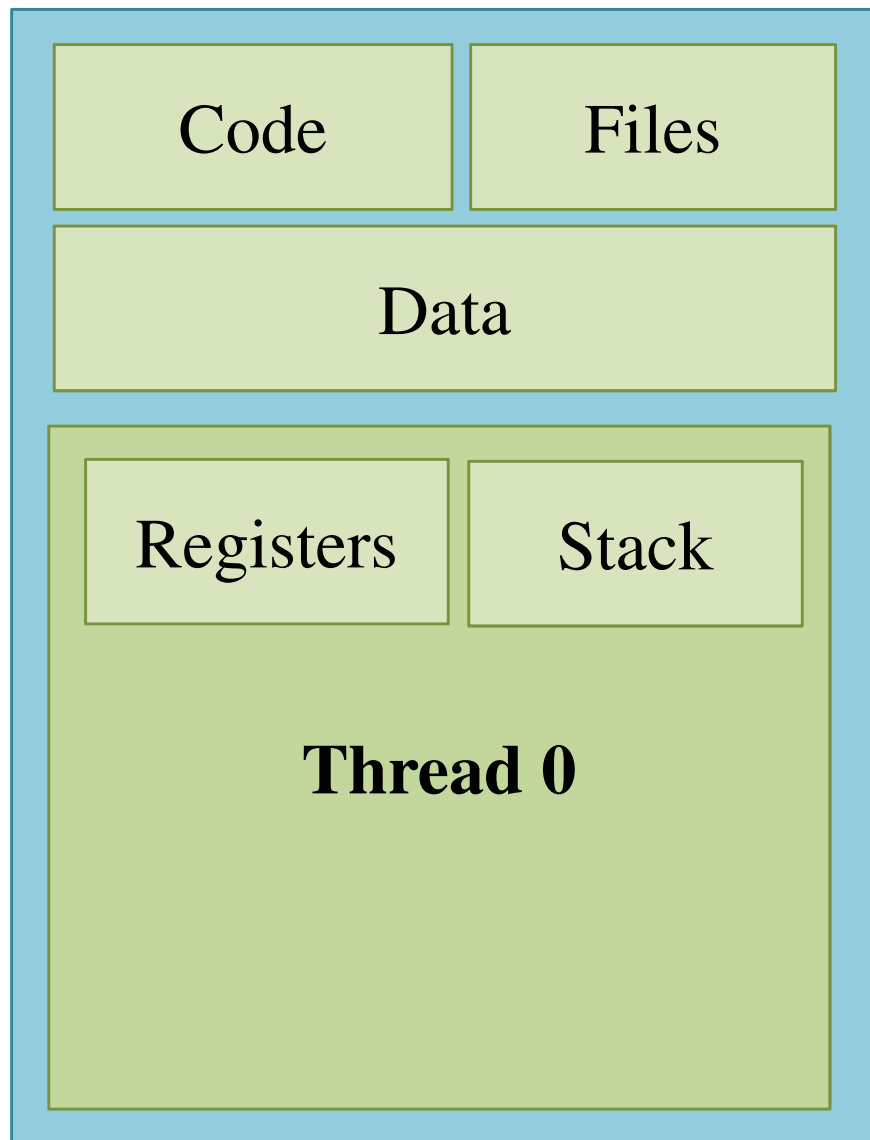
- Пакет **hwloc** – библиотека для доступа к информации о ядрах и их соединениях



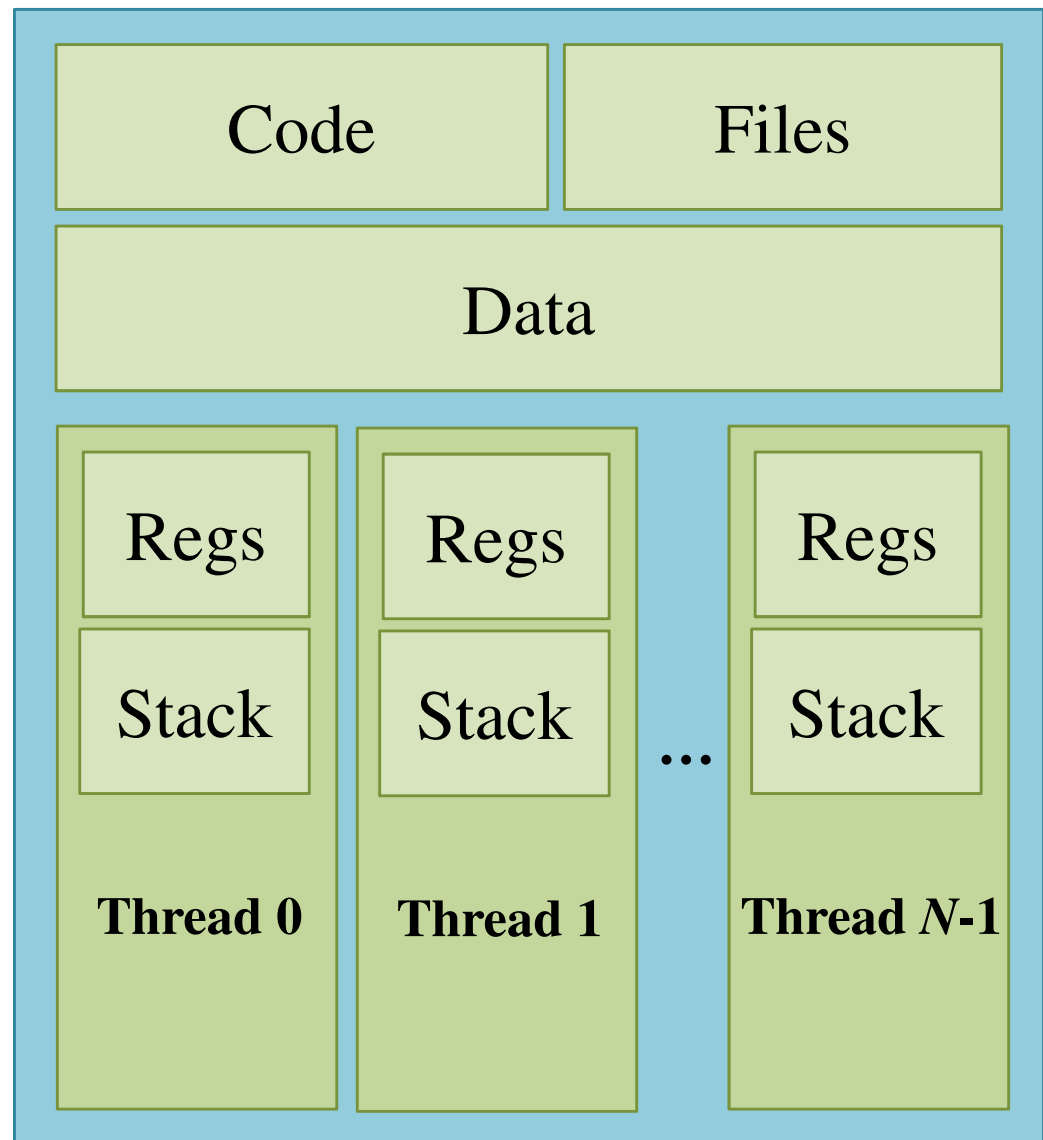
Процессы и потоки

- **Процесс (process)** – это выполняющийся экземпляр программы, владеющий системными ресурсами (памятью, открытыми файлами, атрибутами безопасности).
- Процесс состоит из одного или нескольких потоков выполнения (threads).
- **Поток выполнения (execution thread)** – это исполняемый код, который имеет собственный стек и часть контекста процесса (регистры).
- Потоки разделяют ресурсы процесса (код, память – heap, дескрипторы)
- Переключение потоков (context switching) выполняется быстрее переключения процессов.

Процессы и потоки



Single-threaded program



Multi-threaded program

Многопоточное программирование

Microsoft Windows

- C run-time library

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

- Win32 API Threads

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

Многопоточное программирование

Apple OS X/iOS

- Cocoa threads (NSThread class)
- POSIX threads (pthreads)
- Multiprocessing Services

Android

- Java Threads

GNU/Linux

- POSIX threads (pthreads)

Многопоточное программирование

**Средства создания многопоточных программ,
обеспечивающие их переносимость на уровне
ИСХОДНЫХ КОДОВ**

- C11 threads (<threads.h>)
- OpenMP
- Intel Cilk Plus (C/C++ language extension, open source)
- Intel Threading Building Blocks (C++ template library, open source)
- Intel Array Building Blocks (C++ library)

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); /* Wait */
    return 0;
}
```

POSIX Threads (pthreads)

```
#include <pthread.h>

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

int main()
{
    pthread_t tid;

    pthread_create(&tid, NULL, threadfun, NULL);
    pthread_join(tid, NULL);

    return 0;
}
```

OpenMP

- **OpenMP (Open Multi-Processing)** – стандарт, определяющий набор директив компилятора, библиотечных процедур и переменных окружения, предназначенных для создания многопоточных программ.
- Текущая версия – OpenMP 3.1 (www.openmp.org).
- Для использования требуется поддержка со стороны компилятора.

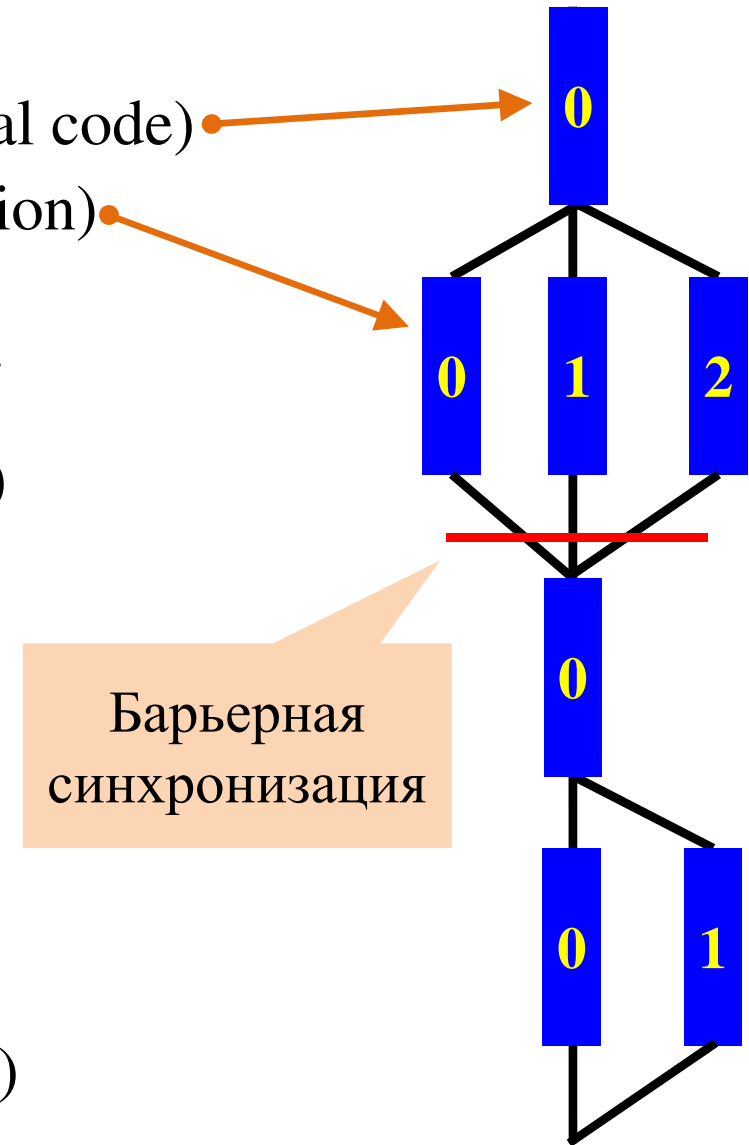


OpenMP

| 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 |
| Intel C/C++, Fortran | Option: <code>-Qopenmp</code> , <code>-openmp</code> |
| Oracle Solaris Studio C/C++/Fortran | Option: <code>-xopenmp</code> |
| Microsoft Visual Studio 2010 C++ | Option: <code>/openmp</code> OpenMP 2.0 only |
| Other compilers: IBM XL, PathScale, PGI, Absoft Pro, ... | |

OpenMP

- 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
- Динамическое управление количеством потоков (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
- Функции runtime-библиотеки
- Переменные среды окружения

Создание потоков (parallel)

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

```
#pragma omp parallel if (expr)
{
    /* Потоки создаются 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 */
}

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

При любых условиях выполняется фиксированное количество потоков – по количеству секций

Создание потоков (sections)

```
#pragma omp parallel sections num_threads(100)
{
#pragma omp section
{
    /* Код потока */
}

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

Создается 2 потока



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
{
#pragma omp master
{
    /* Код выполняется только потоком 0 */
}
    /* Этот код выполняется всеми потоками */
}
```

Директива single

```
#pragma omp parallel
{
#pragma omp single
{
    /* Код выполняется только одним потоком */
}
    /* Этот код выполняется всеми потоками */
}
```

Директива for

```
#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);
    }
}
```

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, которые до выполнения 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
- 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*)** сворачивает *n* ЦИКЛОВ В ОДИН

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

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

Переменная `sum` одновременно перезаписывается и считывается несколькими потоками – **Data Race!**

Ошибки в многопоточных программах

```
#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

Ошибки в многопоточных программах

```
#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

Ошибки в многопоточных программах

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

- Операции директивы reduction:
+, *, -, &, |, ^, &&, ||, max, min

Директивы синхронизации

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

- `#pragma omp critical`
- `#pragma omp atomic`
- `#pragma omp ordered`
- `#pragma omp barrier`
- `#pragma omp flush`

#pragma omp flush

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

- Принудительно обновляет в памяти значения переменных
- 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];
            }
        }
    }
}
```

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, которая добавляется в пул задач (task pool)
- Задачи из пула динамически выполняются потоками
- 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 (serial 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++;
}
```

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;

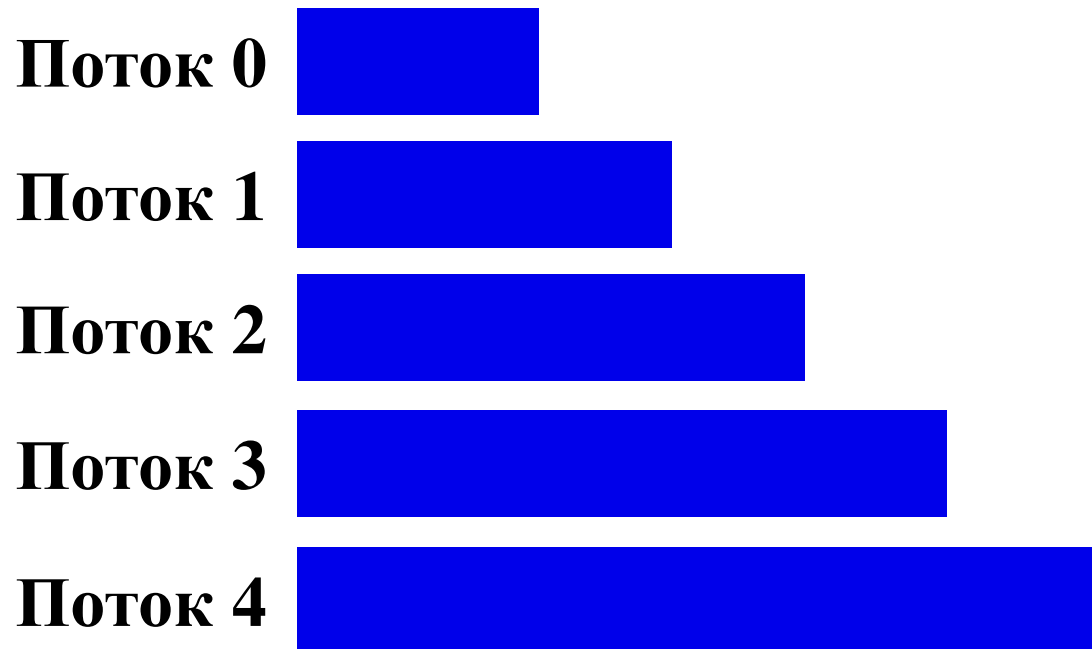
nprimes = 0;
if (start <= 2)
    nprimes++;

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

Время выполнения
зависит от 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++;
}
```

Deadlock & Livelock

- **Взаимная блокировка (Deadlock)** – состояние в которой несколько потоков находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими потоками.

| | |
|--|--|
| 1. T1 захватывает A 2. T1 ожидает B | 1. T2 захватывает B 2. T2 ожидает A |
|--|--|

- **Livelock** – потоки не блокируются (как во взаимной блокировке), а занимается бесполезной работой, их состояние постоянно меняется – но, тем не менее, они не производят никакой полезной работы.
- **Пример**
Двое встречаются лицом к лицу. Каждый из них пытается посторониться, но они не расходятся, а несколько секунд сдвигаются в одну и ту же сторону.

Data race

- **Состояние гонки (Race condition)** – состояние при котором несколько потоков одновременно читают/пишут в разделяемую область памяти

```
/* Thread 0 */  
while (!stop) {  
    x++;  
}
```

```
/* Thread 1 */  
while (!stop) {  
    if (x % 2 == 0)  
        printf("x = %d\n", x);  
}
```

Последовательность выполнения (пусть $x = 0$)

- $\text{if } (x \% 2 = 0)$ выполняется
- $x++$ ($x = 1$)
- `printf` выводит 1,
(должен был вывести 0 –
на четность проверили)

numactl

```
$ numactl -hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3
node 0 size: 3983 MB
node 0 free: 592 MB
node distances:
node    0
  0:    10
```

```
$ numactl --physcpubind=0,1,2,3 ./prog
```