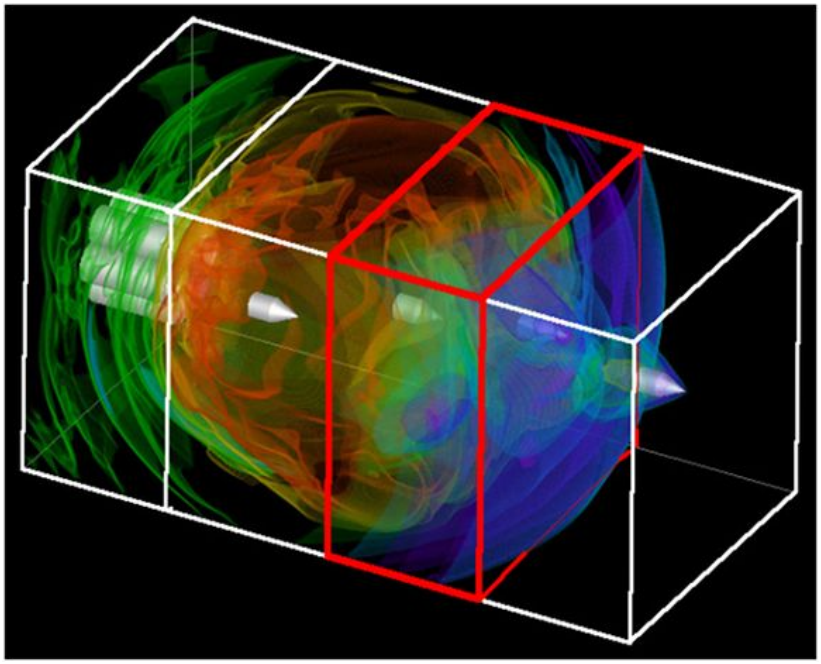
Лекция 7

Технологии создания параллельных программ

Формы параллелизма

Параллелизм по задачам

Параллелизм по данным



Формы параллелизма

Задача:
1. Найти число нулей.
2. Найти число единиц.
3. Определить чего больше.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

1. Найти число нулей.

2. Найти число единиц.

3. Определить чего больше.

1. Найти число нулей.
2. Найти число единиц.
3. Определить чего больше.

$$A = \left(\begin{array}{cc|cc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{array}\right)$$

1. Найти число нулей.
2. Найти число единиц.
3. Определить чего больше.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

1. Найти число нулей.
2. Найти число единиц.

1. Найти число нулей.
2. Найти число единиц.

агрегирование

3. Определить чего больше.

Средства разработки параллельных программ

- Программирование на стандартных и широко распространённых языках программирования с использованием высокоуровневых **коммуникационных библиотек и интерфейсов (API)** для организации межпроцессного взаимодействия.

ACE, ARCH, BIP, BLACS, BSPlib, CVM, Counterpoint, FM, Gala, GA, HPVM, ICC, JIAJIA, KELP, LPARX,
MPI, MPL, OOMPI, **OpenMP**, P4, Para++, Phosphorus, PVM, Quarks, ROMIO, ShMem, SVMlib,
TOOPS, Treadmarks

- MPI (Message Passing Interface) - хорошо стандартизованный механизм для построения программ по модели обмена сообщениями. Существуют стандартные "привязки" MPI к языкам C, C++, Fortran 77, Fortran 90. Существуют бесплатные и коммерческие реализации почти для всех суперкомпьютерных платформ, а также для сетей рабочих станций UNIX и Windows NT. В настоящее время MPI - наиболее широко используемый и динамично развивающийся интерфейс из своего класса.

- OpenMP - программный интерфейс (API) для программирования компьютеров с разделяемой памятью (SMP/NUMA). OpenMP можно использовать для программирования на языках Fortran и C/C++.

Средства разработки параллельных программ

- Введение специальных "распараллеливающих" конструкций в язык программирования. При этом могут создаваться оригинальные **параллельные языки или параллельные расширения** существующих (с сохранением преемственности).
  - Параллельные расширения и диалекты языка Fortran:
    Fortran-DVM, Cray MPP Fortran, F--, Fortran 90/95, Fortran D95, Fortran M, Fx, HPF, Opus, Vienna Fortran,
  - Параллельные расширения и диалекты языков C/C++:
    DVM, A++/P++, CC++, Charm/Charm++, Cilk, HPC, HPC++, Maisie, Mentat, mpC, MPC++, Parsec, pC++, sC++, uC++,
  - Другие параллельные языки и расширения:
    HOPMA, ABCL, Adl, Ada, Concurrent Clean, MC#, Erlang, Linda, Modula-3, NESL, Occam, Orca, Parallaxis, Phantom, Sisal, SR, ZPL

https://parallel.ru/tech/tech_dev/

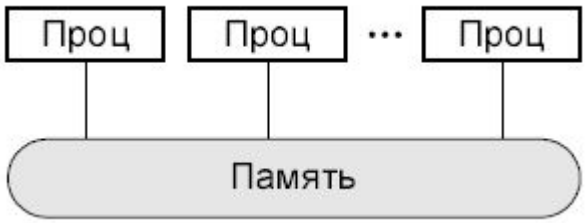
Средства разработки параллельных программ

- Использование средств **автоматического распараллеливания** последовательных программ.
  BERT 77, FORGE, KAP, PIPS, VAST, V-Ray.
- Программирование на стандартных языках. Использование в качестве конструктивных элементов заранее распараллеленных процедур из **специализированных библиотек**.
  ATLAS, Aztec, BlockSolve95, DOUG, GALOPPS, JOSTLE, NAMD, P-Sparslib, Distributed Parallelization at CWP, PIM, ParMETIS, PARPACK, PBLAS, PETSc, PGAPack,PLAPACK, ScaLAPACK, SPRNG.
- Использование инструментальных систем, облегчающих **создание и проектирование** параллельных программ.
  CODE, HeNCE, GRADE, TRAPPER, EDPEPPS, Reactor, DEEP, Converse.
- Использование **специализированных прикладных пакетов**.
  - Задачи инженерного анализа, прочности, теплофизики, деформации, упругости, пластичности, электромагнетизма (ANSYS, MSC.NASTRAN, ABAQUS, LS-DYNA).
  - Задачи аэро- и гидродинамики, механики жидкостей и газов, горения и детонации (CFX, FLUENT, STAR-CD, FLOWVISION, FLOW-3D, GDT).
  - Задачи акустического анализа (LMS Virtual Lab. Acoustic, COMET/Acoustics).

MIMD

Параллельные компьютеры MIMD

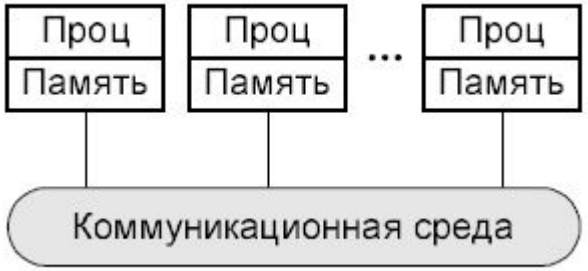
С общей



Пример:

1) Symmetric Multi Processors (SMP);
2) Parallel Vector Processor (PVP) (Cray T90);

С распределенной па



massive parallel processing (**MPP)**

Кластеры

Кластер – группа компьютеров, объединенных в локальную вычислительную сеть (ЛВС) и способных работать в качестве единого вычислительного ресурса.

Кластеры и ВС:

- Кластеры ⊂ распределенные ВС;
- Кластер для users – одна система;
- Кластер – быстрая связь между узлами;
- Кластер – узкая специализация

MPI (message passing interface)

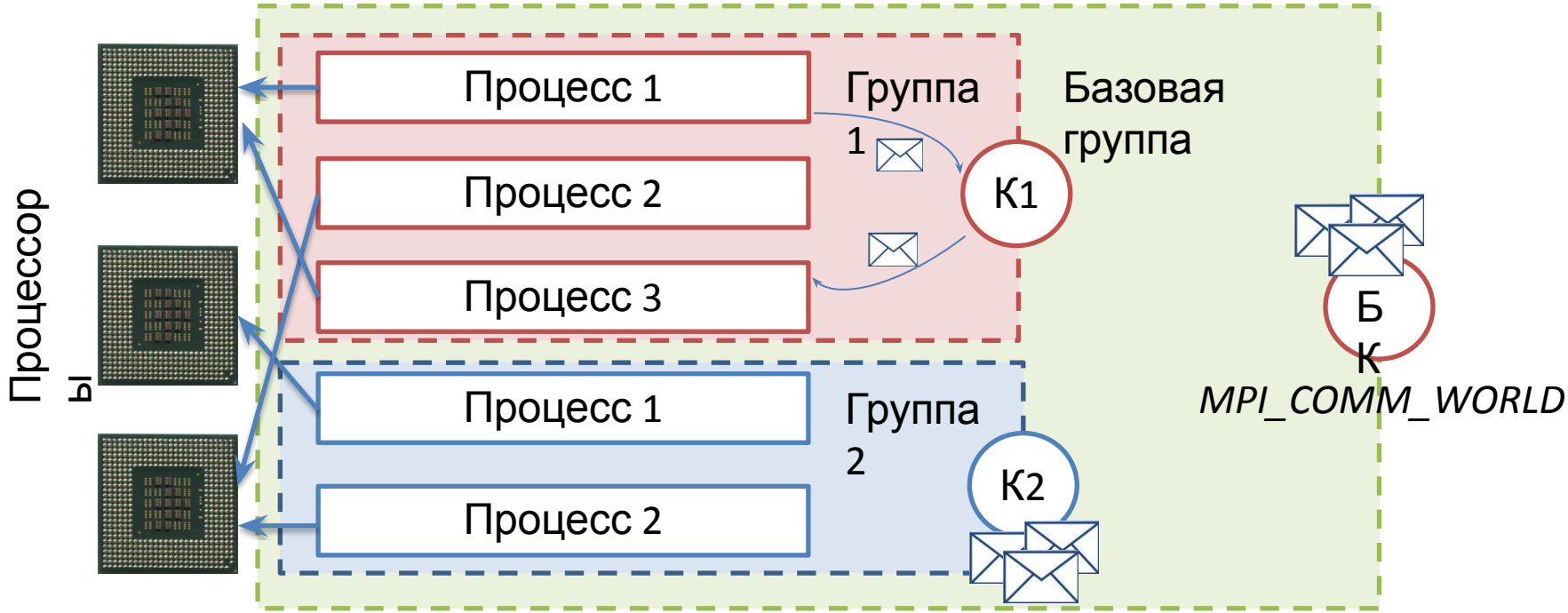
Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу.

Первая версия MPI разрабатывалась в 1993—1994 году и вышла в 1994 (MPI 1).

Уильям Гроупп — профессор информатики в Иллинойсском университете в Урбана-Шампейн. Гроупп участвовал в создании интерфейса Message Passing Interface, также известного как MPI, а также Portable Extensible Toolkit for Scientific Computation,

MPI. Терминология и обозначения

MPI - message passing interface



MPI. Терминология и обозначения

Процессор - интегральная схема, исполняющая машинные инструкции.

Процесс - совокупность команд, выполняемых на одном вычислительном узле.

Группа – это упорядоченное множество процессов.

- вложенные группы;
- базовая группа;
- последовательная нумерация процессов в группе;
- имена групп.

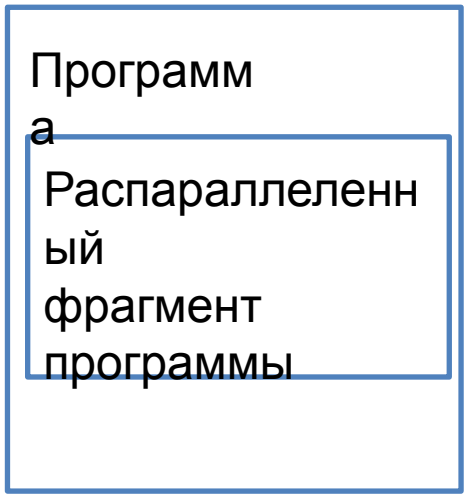
Сообщение - данные, передаваемые между процессами.

- отправитель — ранг (номер в группе) отправителя сообщения;
- получатель — ранг получателя;
- идентификатор — имя сообщения;
- коммуникатор — имя группы процессов.

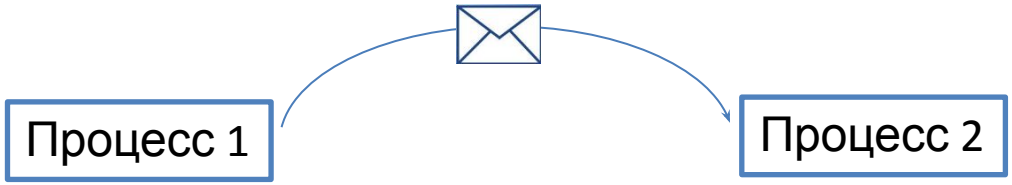
Коммуникатор - специальный объект, отвечающий за связь в группе.

MPI. 130 функций

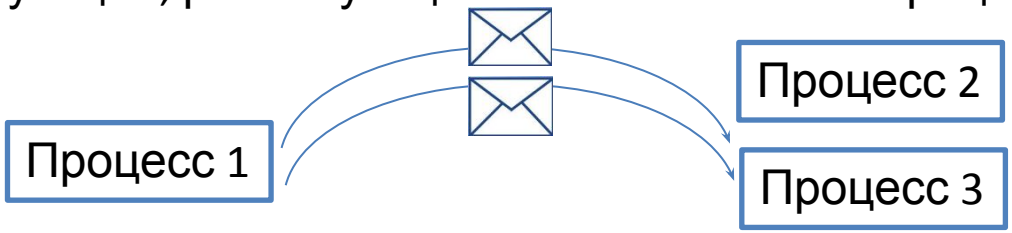
- функции инициализации и закрытия MPI процессов;



- функции, реализующие коммуникационные операции типа точка-точка;



- функции, реализующие коллективные операции;

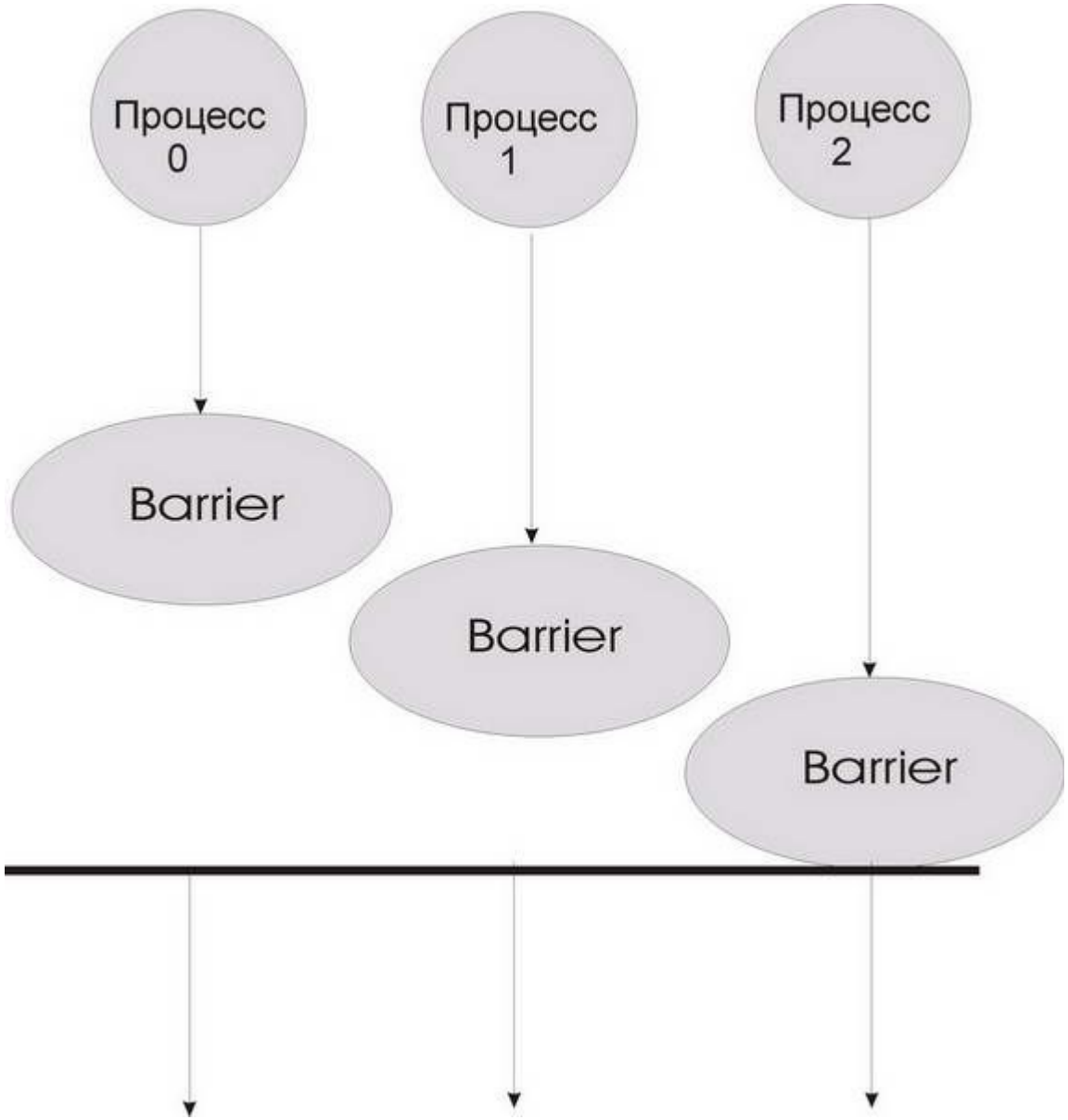


- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

MPI: Hello, World!

```
#include <mpi.h>

int main(int argc, char *argv[])
{
    int rank, commsize;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello, World: process %d of %d\n",
           rank, commsize);

    MPI_Finalize();
    return 0;
}
```

Точки синхронизации



Точки синхронизации

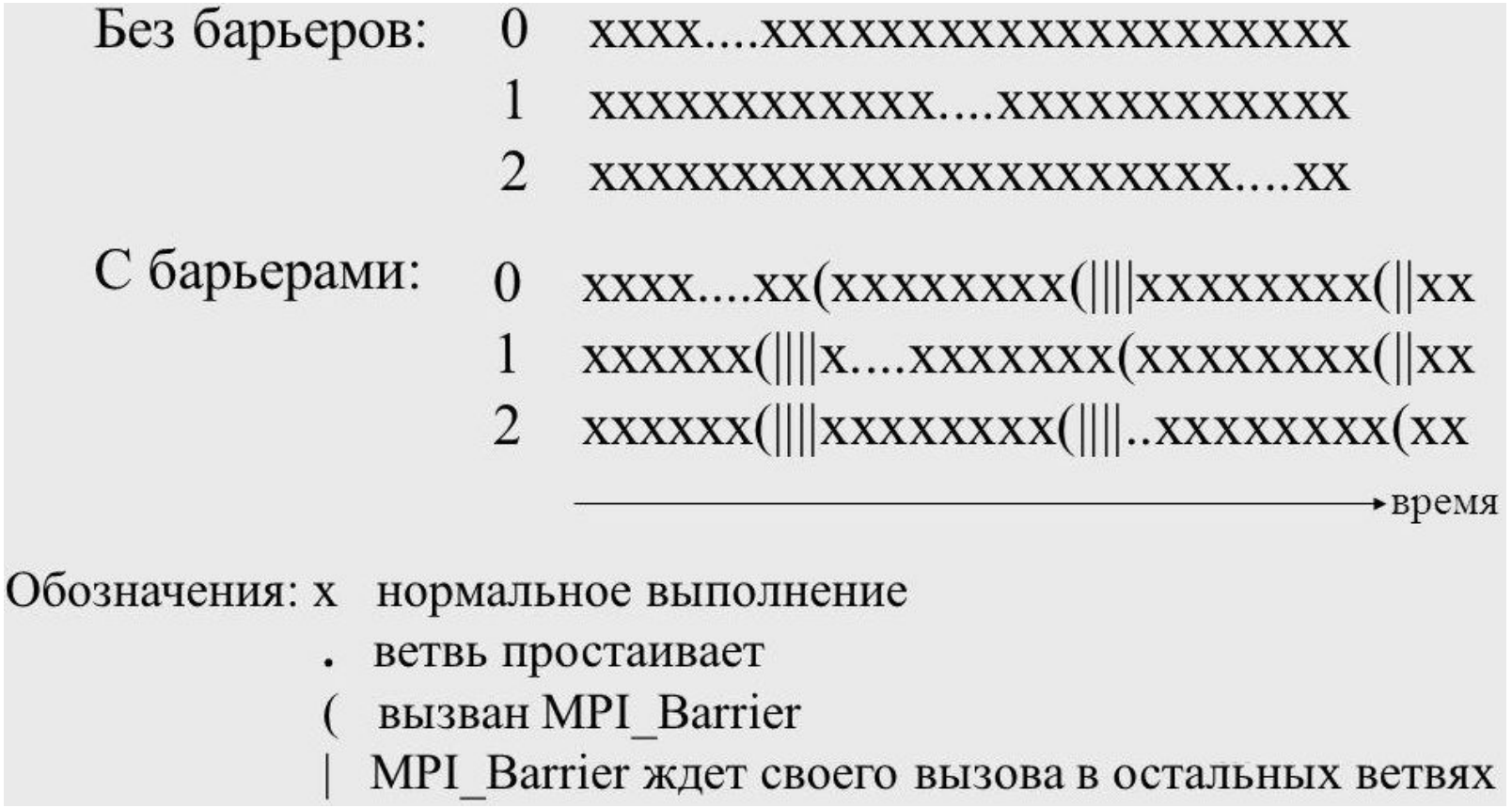
```
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
if (rank == 0)
{
        MPI_Bsend(buf1,20,MPI_INT,1,25,MPI_COMM_WORLD);
        MPI_Ssend(buf2,20,MPI_INT,1,26,MPI_COMM_WORLD);
        printf("Отправка данных окончена \n");
}
else if (rank==1)
{
        MPI_Recv(source1, 20, MPI_INT, 0, 26, MPI_COMM_WORLD, &status);
        MPI_Recv(source2, 20, MPI_INT, 0, 25, MPI_COMM_WORLD, &status);
        printf("Прием данных окончен \n");
}
MPI_Barrier(MPI_COMM_WORLD);
printf("Завершение работы процесса %d \n",rank);
```

Точки синхронизации

MPI_BARRIER (COMM)

Внимание!

Функция MPI_Barrier определяет **коллективную** операцию, и, тем самым, при использовании она должна вызываться всеми *процессами* используемого коммуникатора

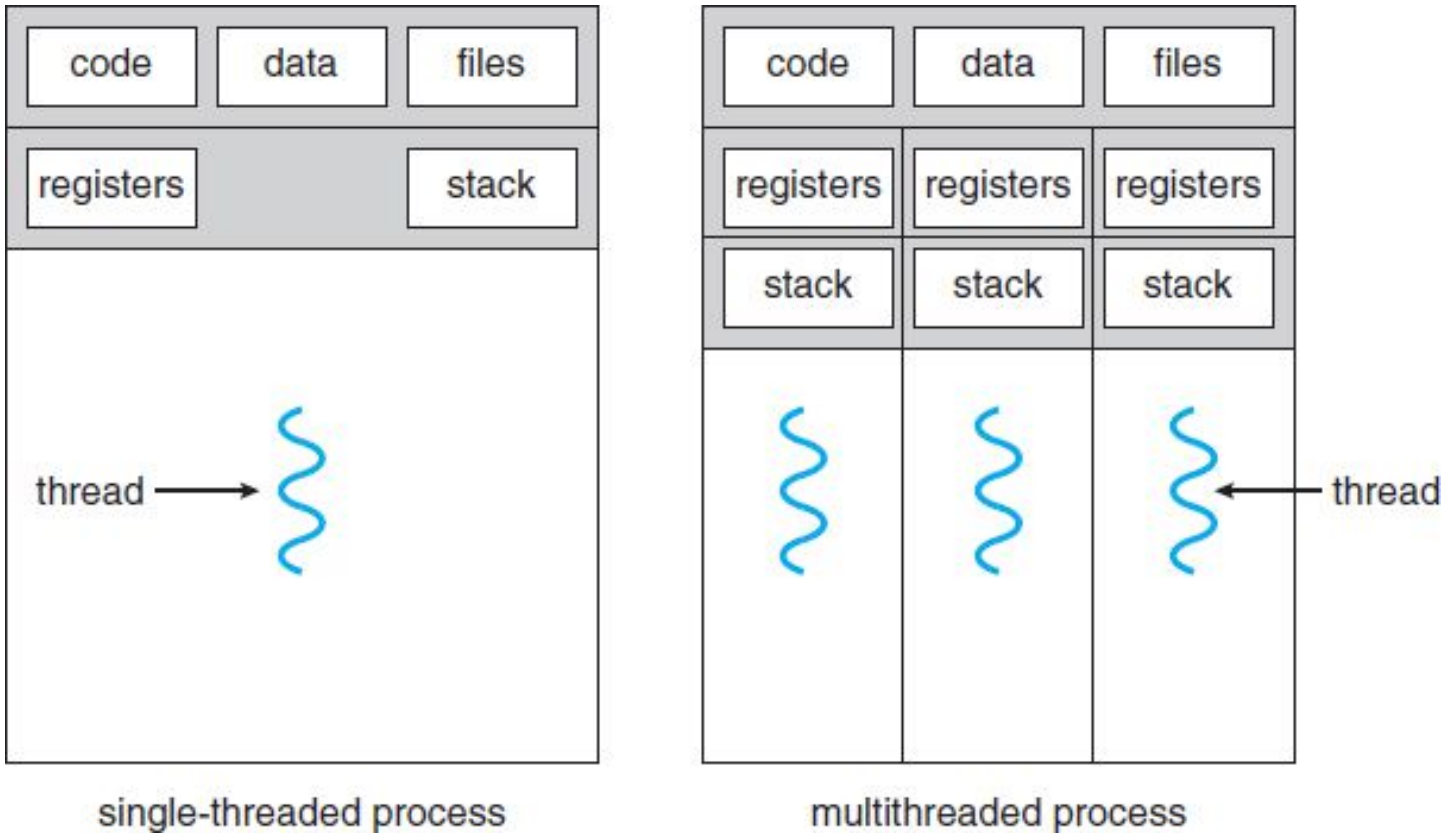


POSIX Threads

POSIX - Portable Operating System Interface for UNIX

POSIX - это стандарт, описывающий интерфейс между операционной системой и прикладной программой.

Потоки

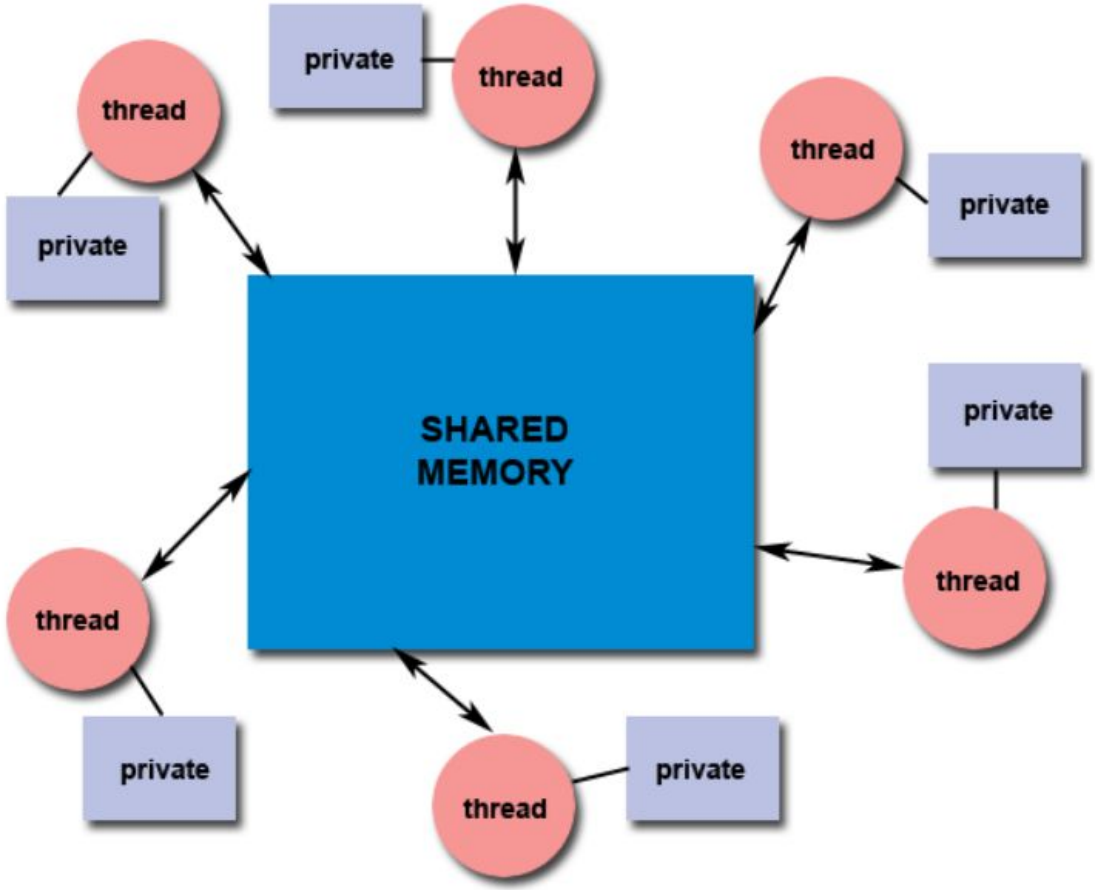


single-threaded process

multithreaded process

Модель разделяемой памяти

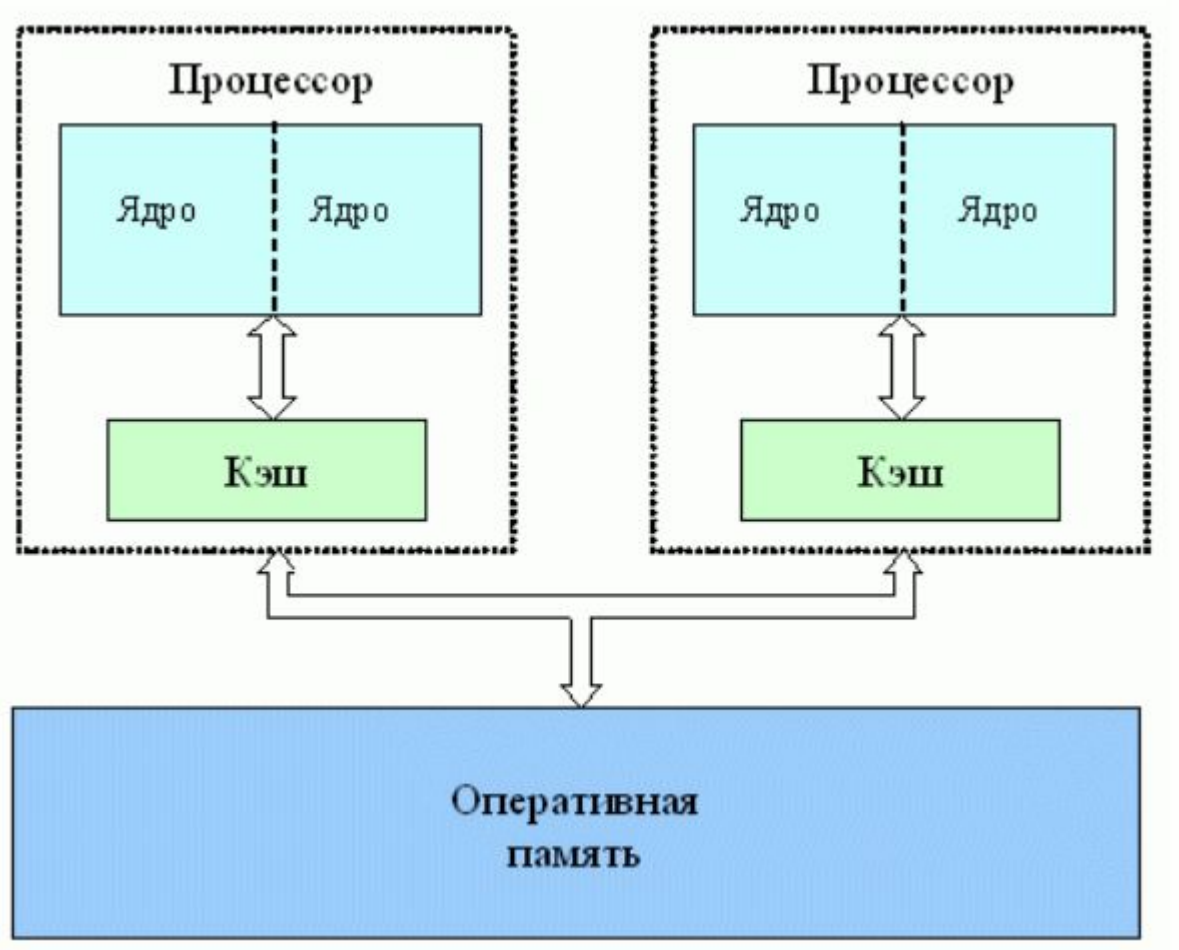
- Все потоки имеют доступ к разделяемой глобальной памяти
- Данные могут быть как приватными так и общими
- Общие данные доступны всем потокам
- Приветные – только одному
- Требуется синхронизация для доступа к общим данным



Симметричные мультипроцессорные системы (SMP)

Архитектура	Система состоит из нескольких однородных процессоров и массива общей памяти (обычно из нескольких независимых блоков). Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью. Процессоры подключены к памяти либо с помощью общей шины (базовые 2-4 процессорные SMP-сервера), либо с помощью коммутатора (HP 9000). Аппаратно поддерживается когерентность кэшей.
Примеры	HP 9000 V-class, N-class; SMP-сервера и рабочие станции на базе процессоров Intel (IBM, HP, Compaq, Dell, ALR, Unisys, DG, Fujitsu и др.).
Масштабируемость	Наличие общей памяти сильно упрощает взаимодействие процессоров между собой, однако накладывает сильные ограничения на их число - не более 32 в реальных системах. Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры.
Операционная система	Вся система работает под управлением единой ОС (обычно UNIX-подобной, но для Intel-платформ поддерживается Windows NT). ОС автоматически (в процессе работы) распределяет процессы/нити по процессорам (scheduling), но иногда возможна и явная привязка.
Модель программирования	Программирование в модели общей памяти. (POSIX threads, OpenMP). Для SMP-систем существуют сравнительно эффективные средства автоматического распараллеливания.

Архитектура многопроцессорных систем с общей (разделяемой) с однородным доступом памятью



POSIX threads

POSIX threads или **Pthreads** определяет набор типов и функций для программирования потоков.

- ***Типы данных:***
  - pthread_t: дескриптор потока
  - pthread_attr_t: перечень атрибутов потока

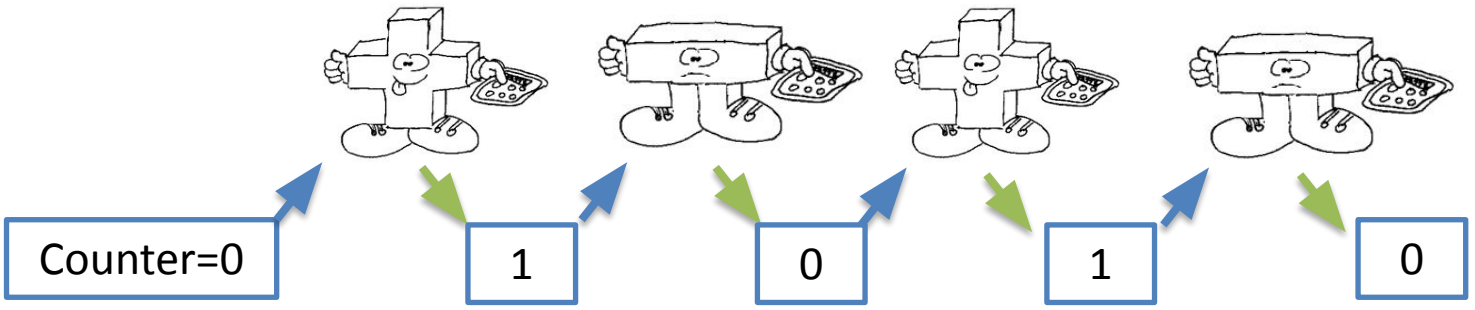
- ***Функции управления потоками:***
  - pthread_create(): создание потока
  - pthread_exit(): завершение потока (должна вызываться функцией потока при за вершении)
  - pthread_cancel(): отмена потока
  - pthread_join(): подключиться к другому потоку и ожидать его завершения. pthread_detach(): отключиться от потока, сделав его при этом отдельным

- ***Функции синхронизации потоков:***
  - pthread_mutex_init(), pthread_mutex_destroy(), pthread_mutex_lock(), pthread_mute x_trylock(), pthread_mutex_unlock(): с помощью мьютексов
  - pthread_cond_init(), pthread_cond_signal(), pthread_cond_wait(): с помощью услов ных переменных

POSIX threads. Пример 1

Пример: несколько потоков обращаются к одной общей переменной.
Часть потоков эту переменную увеличивают на единицу (plus потоки);
Часть потоков уменьшают эту переменную на единицу (minus потоки);
Число plus и minus потоков равно.

Ожидаемый результат: к концу работы программы значение исходной переменной будет прежним.



POSIX threads. Пример 1

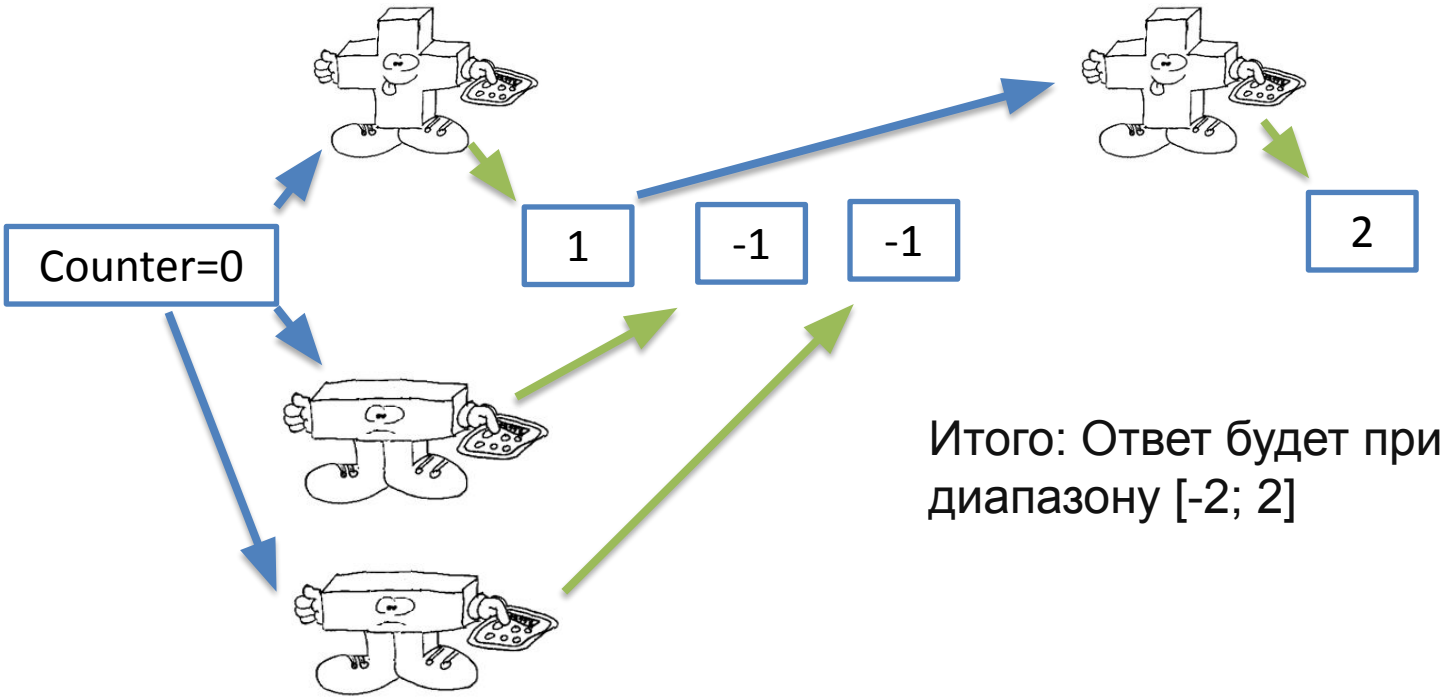
```c
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <pthread.h>

static int counter = 0;

void* minus(void *args) {
    int local;

    local = counter;
    printf("min %d\n", counter);
    local = local - 1;
    counter = local;
    return NULL;
}

void* plus(void *args) {
    int local;

    local = counter;
    printf("pls %d\n", counter);
    local = local + 1;
    counter = local;
    return NULL;
}

```

```c
#define NUM_OF_THREADS 100

int main() {
    pthread_t threads[NUM_OF_THREADS];
    size_t i;

    printf("counter = %d\n", counter);
    for (i = 0; i < NUM_OF_THREADS/2; i++) {
        pthread_create(&threads[i], NULL, minus, NULL);
    }
    for (; i < NUM_OF_THREADS; i++) {
        pthread_create(&threads[i], NULL, plus, NULL);
    }
    for (i = 0; i < NUM_OF_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("counter = %d", counter);
    _getch();
    return 0;
}
```

1 запуск: Ответ: 0
2 запуск: Ответ: 1
3 запуск: Ответ: 4
4 запуск: Ответ: 0
5 запуск: Ответ: -2
6 запуск: Ответ: 0

POSIX threads. Пример 1

Причины:
Наличие локальной переменной local.
Использование тяжёлой и медленной функция printf.
Отсутствие синхронизации потоков.



Итого: Ответ будет принадлежать диапазону [-2; 2]

POSIX threads. Мьютекс

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

static int counter = 0;
pthread_mutex_t mutex;

void* minus(void *args) {
    int local;
    //Блокировка: теперь к ресурсам имеет доступ только один
    //поток, который владеет мьютексом. Он же единственный,
    //кто может его разблокировать
    pthread_mutex_lock(&mutex);
        local = counter;
        printf("min %d\n", counter);
        local = local - 1;
        counter = local;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

void* plus(void *args) {
    int local;
    pthread_mutex_lock(&mutex);
        local = counter;
        printf("pls %d\n", counter);
        local = local + 1;
        counter = local;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

Объявление

Захват

Освобождение

Захват

Освобождение

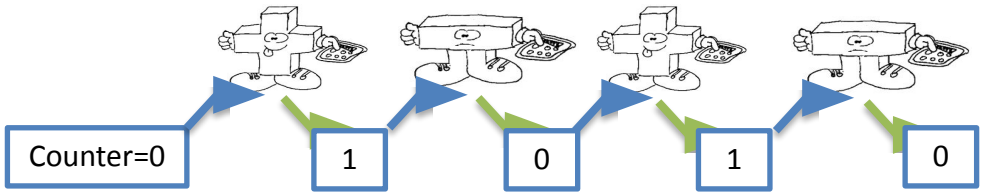
POSIX threads. Мьютекс

```
34  #define NUM_OF_THREADS 100
35
36  int main() {
37      pthread_t threads[NUM_OF_THREADS];
38      size_t i;
39
40      printf("counter = %d\n", counter);
41      //Инициализация мьютекса
42      pthread_mutex_init(&mutex, NULL);
43      for (i = 0; i < NUM_OF_THREADS/2; i++) {
44          pthread_create(&threads[i], NULL, minus, NULL);
45      }
46      for (; i < NUM_OF_THREADS; i++) {
47          pthread_create(&threads[i], NULL, plus, NULL);
48      }
49      for (i = 0; i < NUM_OF_THREADS; i++) {
50          pthread_join(threads[i], NULL);
51      }
52      //Уничтожение мьютекса
53      pthread_mutex_destroy(&mutex);
54      printf("counter = %d", counter);
55      _getch();
56      return 0;
57  }
```

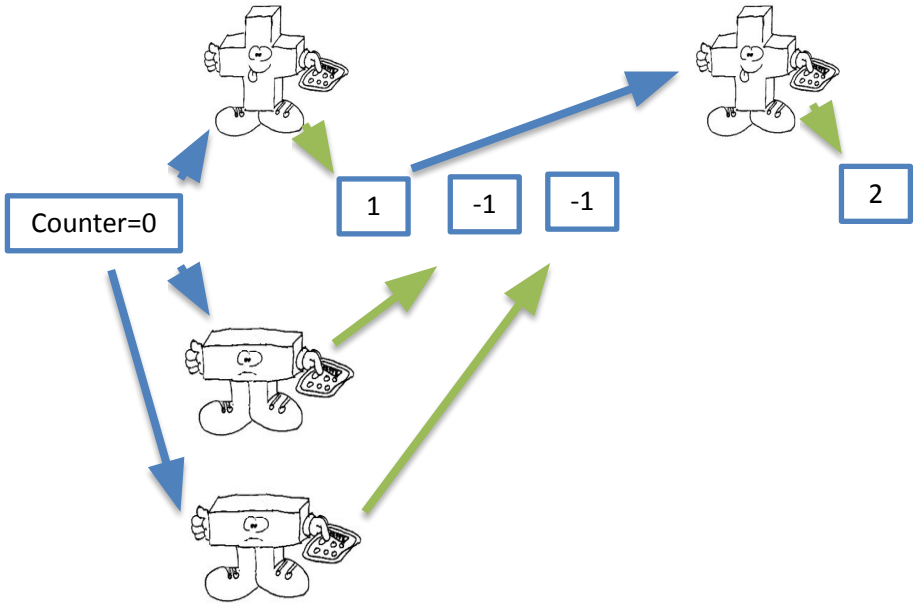
Инициализация

Уничтожение

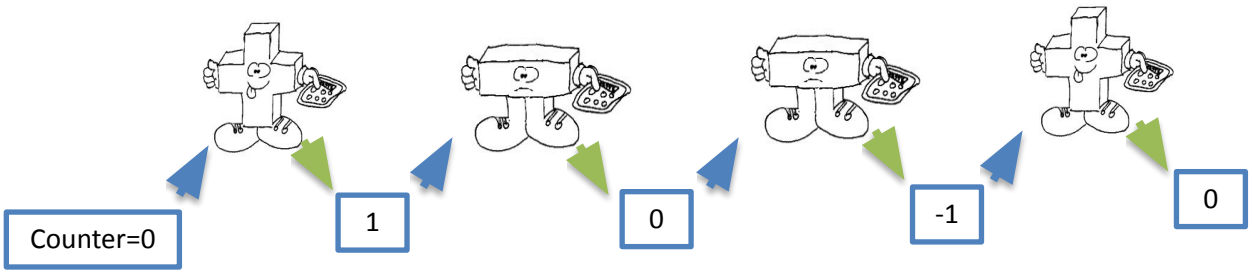
POSIX threads. Мьютекс



Ожидаемый сценарий



Плохой сценарий



Хороший сценарий

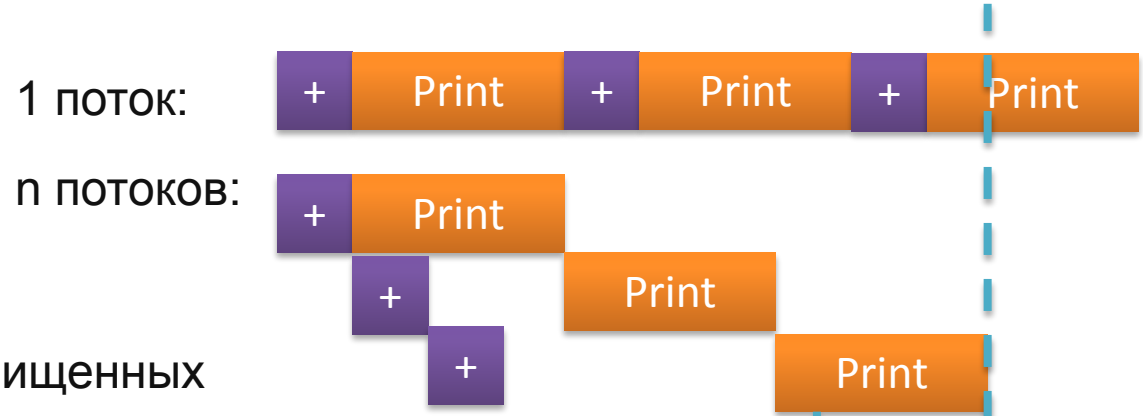
POSIX threads. Мьютекс

При использовании мьютекса:

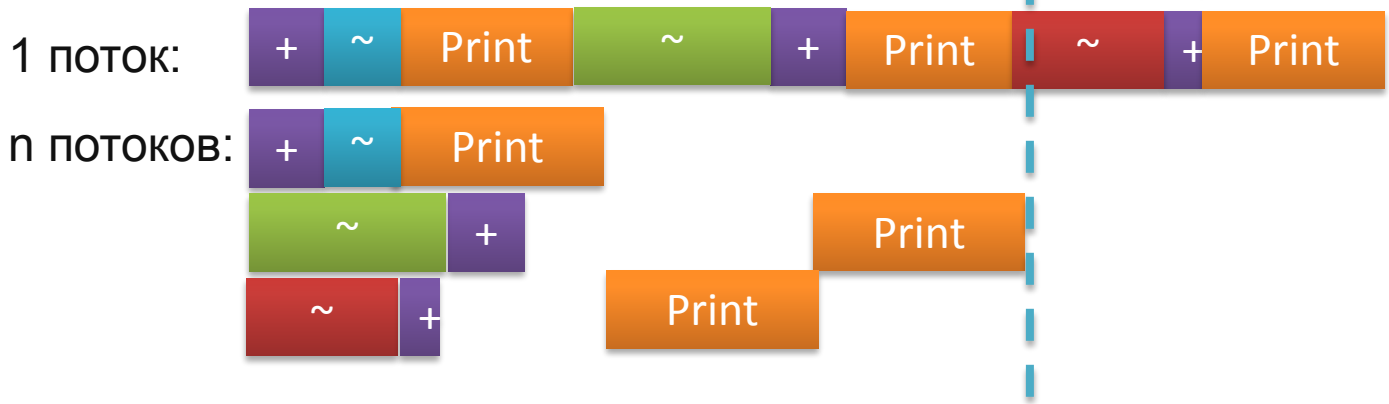
- исполнение защищённого участка кода происходит последовательно всеми потоками, а не параллельно;
- порядок доступа отдельных потоков не определён.

В чем ускорение?

- использование освободившихся ресурсов;



- Наличие у потоков не защищенных участков

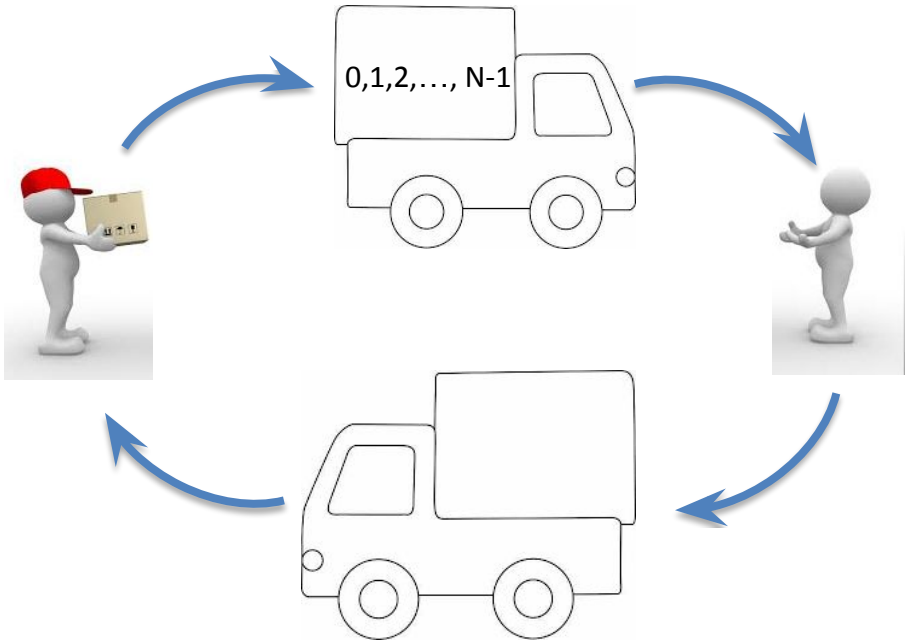


POSIX threads. Условные переменные (conditional variables)

- `pthread_cond_init()` – создание *условной переменной*;
- `pthread_cond_signal()` – разблокировка *условной переменной*;
- `pthread_cond_wait()` – ожидание по *условной переменной*.

Сценарий производитель-потребитель

- 2 процесса — **производитель** и **потребитель** — работают с общим ресурсом (буфером);
- буфер имеет максимальный размер N;
- **производитель** записывает в буфер данные последовательно в ячейки 0,1,2,..., пока он не заполниться;
- **потребитель** читает данные из буфера в обратном порядке, пока он не опустеет;
- запись и считывание не могут происходить одновременно.



Сценарий производитель-потребитель

Наивное решение

```
int buf[N];
int count = 0;
void producer()
{       while (1)
        {
                int item = produce_item();
                while (count == N - 1)
                        /* do nothing */ ;
                buf[count] = item;
                count++;
}}
void consumer()
{       while (1)
        {
                while (count == 0)
                        /* do nothing */ ;
                int item = buf[count - 1];
                count--; consume_item(item);
} }
int main()
{ make_thread(&producer);
make_thread(&consumer); }
```

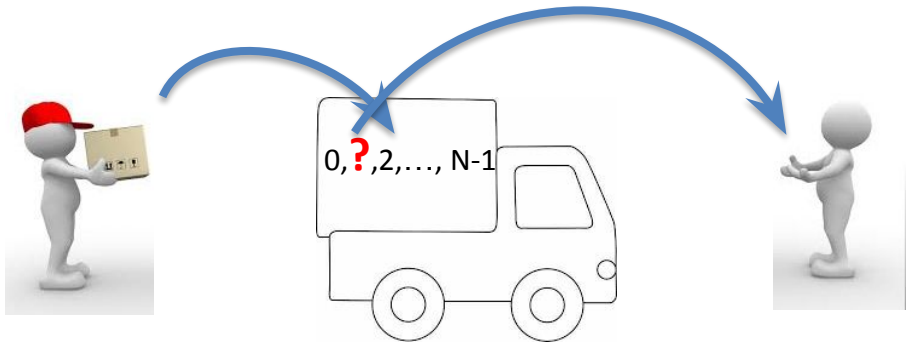
Проблемы «наивного» решения

```
void producer()
{       while (1)
        {
                int item = produce_item();
                while (count == N - 1)
                 /* do nothing */ ;
                 buf[count] = item;
                 count++;
}}
```

(2)

(4)

(1), (5)




```
void consumer()
{ while (1)
    {      while (count == 0)
       /* do nothing */ ;
           int item = buf[count - 1];
           count--; consume_item(item);
    } }
```

(6)

(3)

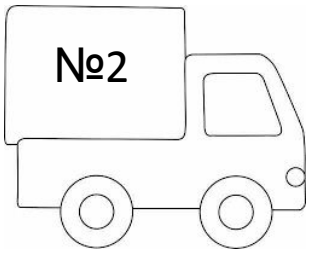
1. **Возможное образование «дырки»:**
   - (1) Пусть count=2
   - (2) Создан элемент
   - (3) Потребитель пересчитает count=1
   - (4) Поставщик запишет в count=2
   - и т.д., например, (5) и count=3
   - (6) count=2 и значение затрется новым
2. **Количество прозв. и потреб. может быть >1**
3. **Бессмысленная трата вычислительных ресурсов**

Сценарий производитель-потребитель

- Основная процедура создает три потока.
- Два потока выполняют работу и обновляют переменную count.
- 2-й и 3-й потоки могут сработать только 10 раз
- Первый поток ожидает, пока переменная count не достигнет указанного значения = 12.

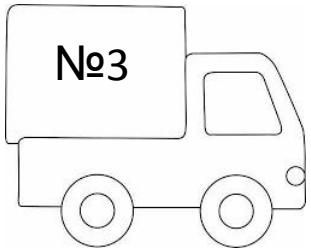
1, 4, 6, 8, 10, 12, 139, 141, 143, 145

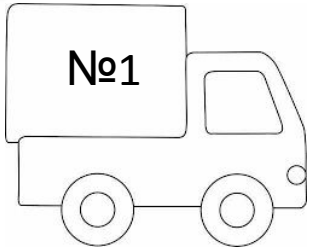
TCOUNT = 10



2, 3, 5, 7, 9, 11, 138, 140, 142, 144

TCOUNT = 10





Count+=125
Print (count)

Сценарий производитель-потребитель

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define NUM_THREADS  3
6  #define TCOUNT 10
7  #define COUNT_LIMIT 12
8
9  int     count = 0;
10 int     thread_ids[3] = {0,1,2};
11 pthread_mutex_t count_mutex;
12 pthread_cond_t count_threshold_cv;
```

Число потоков

Число срабатываний 2-го и 3-го потока

Момент срабатывания 1-го потока

Создаваемые числа

https://computing.llnl.gov/tutorials/pthreads/#ConditionVariables

Сценарий производитель-потребитель

```
14  void *inc_count(void *t)
15  {
16    int i;
17    long my_id = (long)t;
18
19    for (i=0; i<TCOUNT; i++) {
20      pthread_mutex_lock(&count_mutex);
21      count++;
22
23      /*
24      Check the value of count and signal waiting thread when condition is
25      reached.  Note that this occurs while mutex is locked.
26      */
27      if (count == COUNT_LIMIT) {
28        pthread_cond_signal(&count_threshold_cv);
29        printf("inc_count(): thread %ld, count = %d  Threshold reached.\n",
30               my_id, count);
31        }
32      printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
33             my_id, count);
34      pthread_mutex_unlock(&count_mutex);
35
36      /* Do some "work" so threads can alternate on mutex lock */
37      sleep(1);
38      }
39    pthread_exit(NULL);
40  }
```

Поставщик

Сценарий производитель-потребитель

```
42  void *watch_count(void *t)
43  {
44    long my_id = (long)t;
45
46    printf("Starting watch_count(): thread %ld\n", my_id);
47
48    /*
49    Lock mutex and wait for signal.  Note that the pthread_cond_wait
50    routine will automatically and atomically unlock mutex while it waits.
51    Also, note that if COUNT_LIMIT is reached before this routine is run by
52    the waiting thread, the loop will be skipped to prevent pthread_cond_wait
53    from never returning.
54    */
55    pthread_mutex_lock(&count_mutex);
56    while (count<COUNT_LIMIT) {
57      pthread_cond_wait(&count_threshold_cv, &count_mutex);
58      printf("watch_count(): thread %ld Condition signal received.\n", my_id);
59      }
60      count += 125;
61      printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
62    pthread_mutex_unlock(&count_mutex);
63    pthread_exit(NULL);
64  }
```

Сценарий производитель-потребитель

```
66  int main (int argc, char *argv[])
67  {
68    int i, rc;
69    long t1=1, t2=2, t3=3;
70    pthread_t threads[3];
71    pthread_attr_t attr;
72
73    /* Initialize mutex and condition variable objects */
74    pthread_mutex_init(&count_mutex, NULL);
75    pthread_cond_init (&count_threshold_cv, NULL);
76
77    /* For portability, explicitly create threads in a joinable state */
78    pthread_attr_init(&attr);
79    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
80    pthread_create(&threads[0], &attr, watch_count, (void *)t1);
81    pthread_create(&threads[1], &attr, inc_count, (void *)t2);
82    pthread_create(&threads[2], &attr, inc_count, (void *)t3);
83
84    /* Wait for all threads to complete */
85    for (i=0; i<NUM_THREADS; i++) {
86      pthread_join(threads[i], NULL);
87    }
88    printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);
89
90    /* Clean up and exit */
91    pthread_attr_destroy(&attr);
92    pthread_mutex_destroy(&count_mutex);
93    pthread_cond_destroy(&count_threshold_cv);
94    pthread_exit(NULL);
95
96  }
```

Сценарий производитель-потребитель

```
Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
inc_count(): thread 3, count = 2, unlocking mutex
watch_count(): thread 1 going into wait...
inc_count(): thread 3, count = 3, unlocking mutex
inc_count(): thread 2, count = 4, unlocking mutex
inc_count(): thread 3, count = 5, unlocking mutex
inc_count(): thread 2, count = 6, unlocking mutex
inc_count(): thread 3, count = 7, unlocking mutex
inc_count(): thread 2, count = 8, unlocking mutex
inc_count(): thread 3, count = 9, unlocking mutex
inc_count(): thread 2, count = 10, unlocking mutex
inc_count(): thread 3, count = 11, unlocking mutex
inc_count(): thread 2, count = 12  Threshold reached. Just sent signal.
inc_count(): thread 2, count = 12, unlocking mutex
watch_count(): thread 1 Condition signal received.
watch_count(): thread 1 count now = 137.
inc_count(): thread 3, count = 138, unlocking mutex
inc_count(): thread 2, count = 139, unlocking mutex
inc_count(): thread 3, count = 140, unlocking mutex
inc_count(): thread 2, count = 141, unlocking mutex
inc_count(): thread 3, count = 142, unlocking mutex
inc_count(): thread 2, count = 143, unlocking mutex
inc_count(): thread 3, count = 144, unlocking mutex
inc_count(): thread 2, count = 145, unlocking mutex
Main(): Waited on 3 threads. Final value of count = 145. Done.
```

Задача производитель-потребитель

- Если решать только на мутексах, нужно 3 мутекса и холостой цикл на входе
- Решается с одним мутексом и одной условной переменной (т.е. условная переменная ~ эквивалентна 2 мутексам)

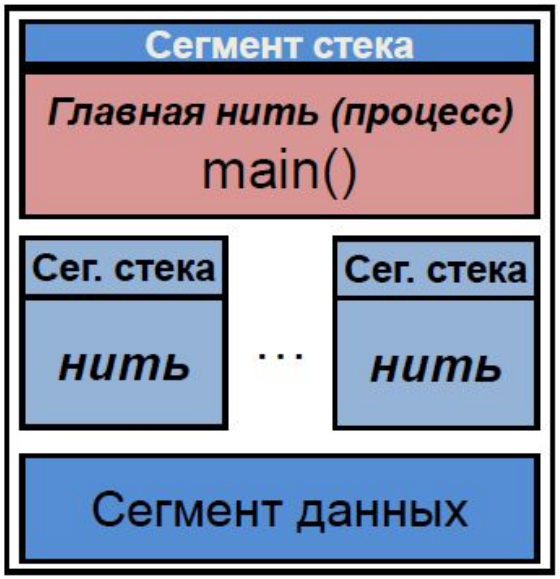
Классические задачи синхронизации

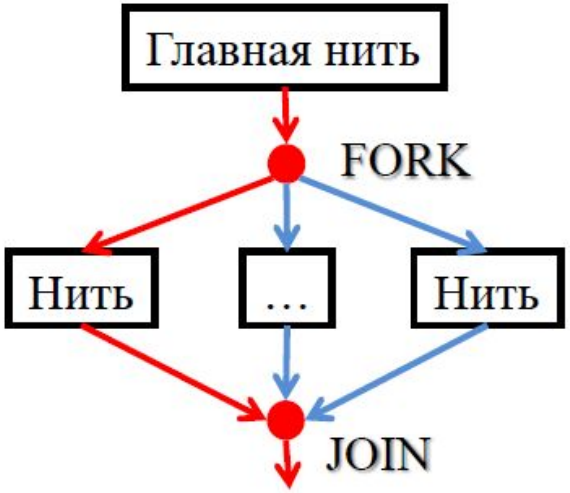
Классические задачи синхронизации — это модельные задачи, на которых исследуются различные ситуации, которые могут возникать в системах с разделяемым доступом и конкуренцией за общие ресурсы.

К ним относятся задачи:

- Производитель-потребитель,
- Читатели-писатели,
- Обедающие философы,
- Спящий парикмахер,
- Курильщики сигарет,
- Проблема Санта-Клауса и др.

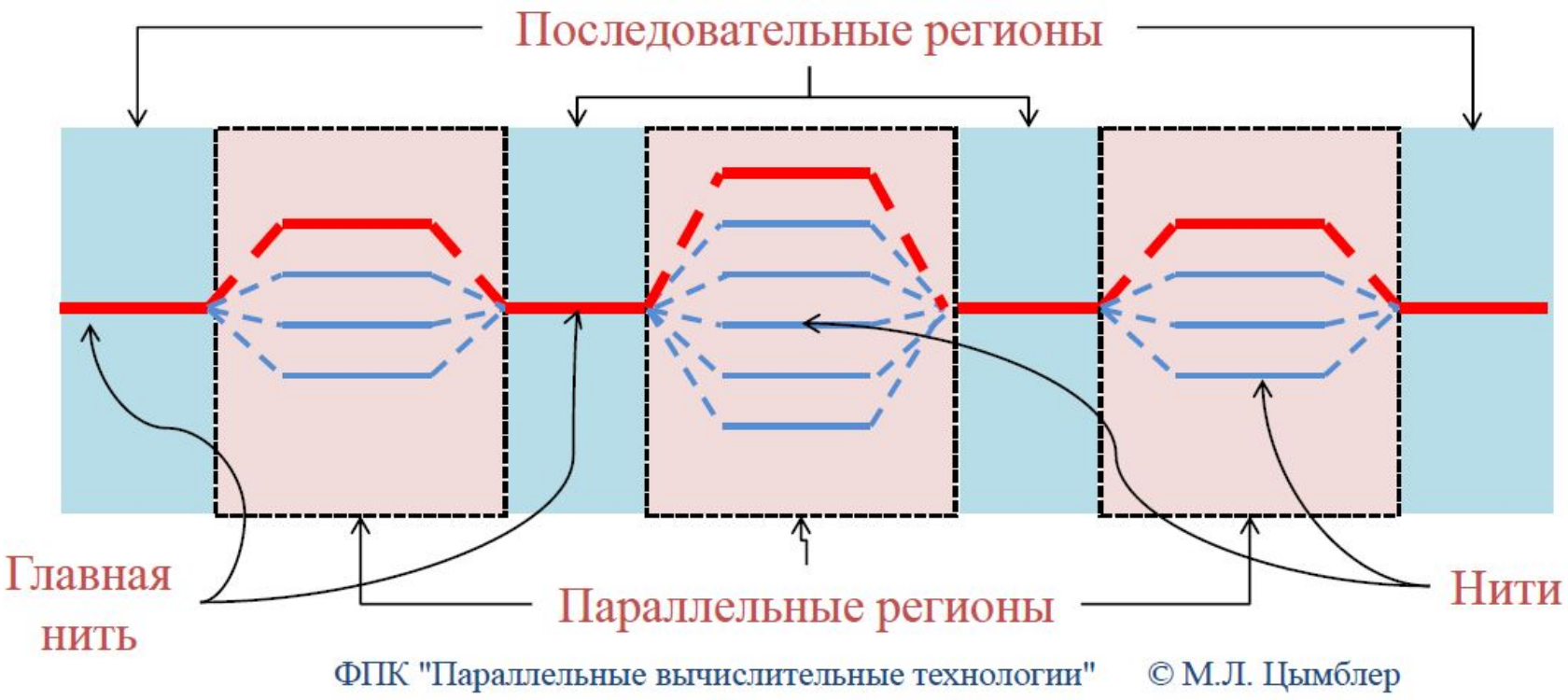
Модель "пульсирующего" параллелизма FORK-JOIN





- Программа–полновесный *процесс.*
- Процесс может запускать *легковесные процессы (нити),* выполняющиеся в фоновом режиме.
- Процесс приложения –*главная нить*.
- Нить может запускать другие нити в рамках процесса. Каждая нить имеет собственный сегмент стека.
- Нити разделяют общую память.
- Обмены между нитями осуществляются посредством чтения/записи данных в общей памяти.
- Нити выполняются на различных ядрах одного процессора.
- Все нити процесса разделяют сегмент данных процесса.

Модель "пульсирующего" параллелизма FORK-JOIN



OpenMP

OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads (или аналогичными библиотеками нитей)

Достоинства

- Отсутствие межпроцессорных передач сообщений.
- Распараллеливание сравнительно простых последовательных программ, как правило, не требует больших усилий (порою достаточно включить в последовательную программу всего лишь несколько директив OpenMP)
- Возможность поэтапной разработки параллельных программы. Директивы OpenMP могут добавляться в последовательную программу.
- Высокая переносимость параллельных программ между разными компьютерными системами. Параллельная программа, разработанная на алгоритмическом языке C или Fortran с использованием технологии OpenMP, как правило, будет работать для разных вычислительных систем с общей памятью.

https://www.intuit.ru/studies/courses/542/398/lecture/9179

Структура OpenMP. Директивы.

Конструктивно в составе технологии *OpenMP* можно выделить:

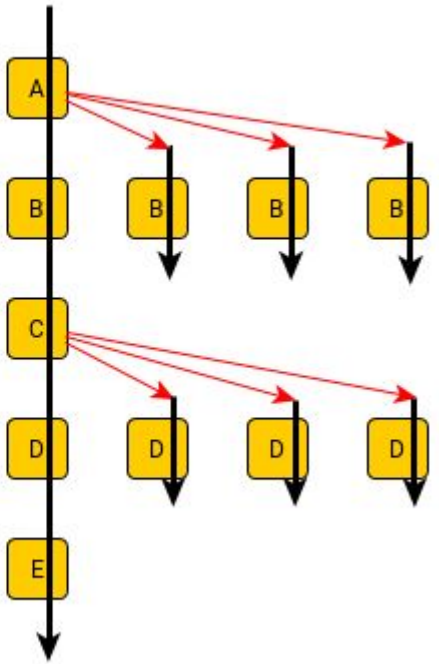
- Директивы,
- Библиотеку функций,
- Набор переменных окружения.

В общем виде формат директив *OpenMP* :

#pragma omp <имя_директивы> [<параметр>[[,] <параметр>]…]

Пример директивы:

```
#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
}
```



https://pro-prof.com/archives/4335

Директива parallel для определения параллельных фрагментов

Синтаксис:

#pragma omp parallel [<параметр> ...] <блок_программы>

Пример параллельной программы

```
#include <omp.h>
main () {
 /* Выделение параллельного фрагмента*/
#pragma omp parallel
 {
   printf("Hello World !\n");
 }/* Завершение параллельного фрагмента */
}
```

Пример простой программы

Последовательный код

```
void main()
{

printf("Hello!\n");
}
```

Результат

```
Hello!
```

Параллельный код

```
void main()
{
#pragma omp parallel
{
    printf("Hello!\n");
}
}
```

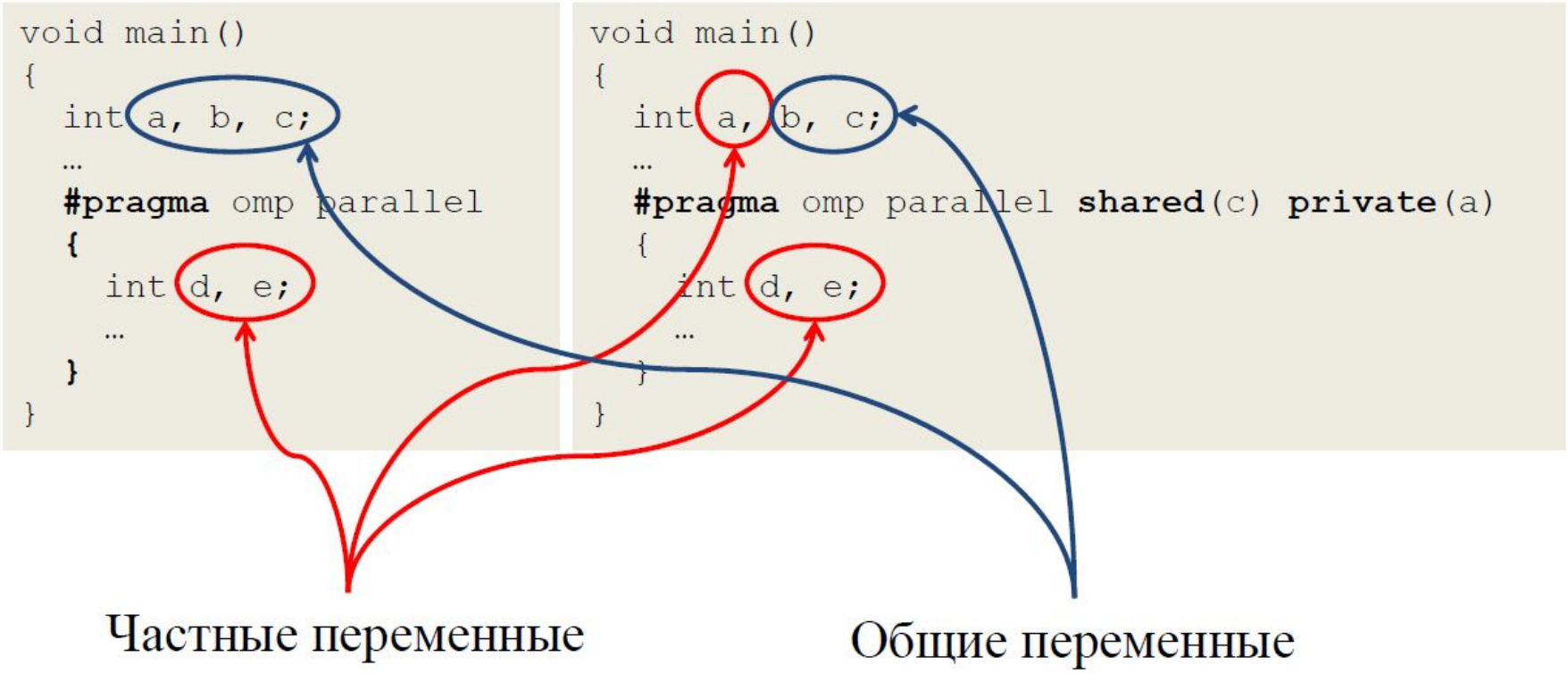
Результат (для 2-х нитей)

```
Hello!
Hello!
```

Частные и общие переменные

```
void main()
{
  int a, b, c;
  …
  #pragma omp parallel
  {
    int d, e;
    …
  }
}
```

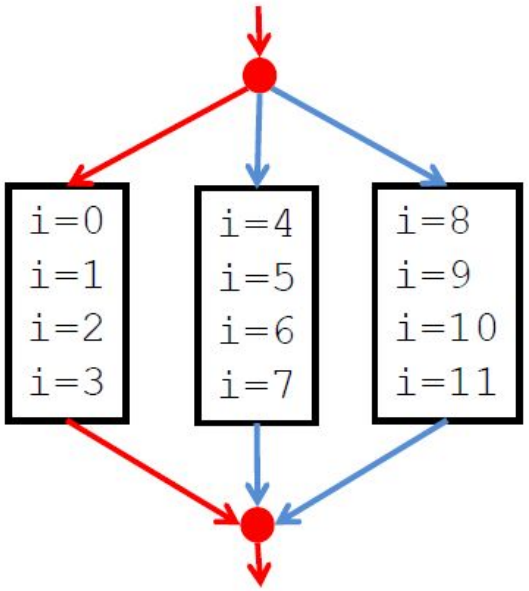
```
void main()
{
  int a, b, c;
  …
  #pragma omp parallel shared(c) private(a)
  {
    int d, e;
    …
  }
}
```



Конструкции OpenMP для распределения работ

- параллельный цикл for/DO
- параллельные секции (sections)
- конструкция single
- конструкция master

Распараллеливание по данным для циклов




```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<N; i++) {
    res[i] = big_calc();
  }
}
```

```
#pragma omp parallel for
for (i=0; i<N; i++) {
  res[i] = big_calc();
}
```

#pragma omp for [<параметр> ...]
<цикл_for>

- Счетчик цикла по умолчанию является частной переменной.
- По умолчанию вычисления распределяются равномерно между нитями.
- Используя условие nowait для цикла можно разрешить основной нити не дожидаться завершения дочерних нитей.
- По умолчанию барьером для потоков является конец цикла. Все потоки достигнув конца цикла дожидаются тех, кто еще не завершился, после чего основная нить продолжает выполняться дальше.

Параллельные секции

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        printf("T%d: foo\n", omp_get_thread_num());
    }
    #pragma omp section
    {
        printf("T%d: bar\n", omp_get_thread_num());
    }
}
```

- Каждая секция выполняется в отдельном потоке, что позволяет производить декомпозицию по коду.
- В случае, когда необходимо чтобы основной поток не ждал завершения остальных потоков следует использовать условие nowait.

Конструкция singie

- Определяет код, который выполняется только одной (первой пришедшей в данную точку) нитью.
  - Остальные нити пропускают соответствующий код и ожидают окончания его выполнения.
  - Если ожидание других нитей необязательно, может быть добавлен параметр `nowait`.

```
#pragma omp parallel
{
#pragma omp single
  printf("Start Work #1.\n");
  Work1();
#pragma omp single
  printf("Stop Work #1.\n");
#pragma omp single nowait
  printf("Stop Work #1 and start Work #2.\n");
  Work2();
}
```

Конструкция master

- Определяет код, который выполняется только одной главной нитью.
- Остальные нити пропускают соответствующий код, не ожидая окончания его выполнения.

```
#pragma omp parallel
{
#pragma omp master
  printf("Beginning Work1.\n");
  Work1();
#pragma omp master
  printf("Finishing Work1.\n");
#pragma omp master
  printf("Finished Work1 and beginning Work2.\n");
  Work2();
}
```

Условия выполнения

Пример. Цикл должен быть распараллелен при условии, что итераций цикла больше, чем 2000

```
#pragma omp parallel
{
#pragma omp for if(n>2000)
  {
    for(i=0; i<n; i++)
      a[i] = work(i);
  }
}
```

Синхронизация вычислений

В OpenMP предусмотрены следующие конструкции синхронизации:

- **critical** – критическая секция
- **atomic** – атомарность операции
- **barrier** – точка синхронизации
- **master** – блок, который будет выполнен только основным потоком. Все остальные потоки пропустят этот блок. В конце блока неявной синхронизации нет.
- **ordered** – выполнять блок в заданной последовательности
- **flush** – немедленный сброс значений разделяемых переменных в память.

Синхронизация вычислений. Директива critical

- Определяет *критическую секцию* –участок кода, выполняемый одновременно не более чем одной нитью.
- Наличие критической секции в параллельном блоке гарантирует, что она в каждый конкретный момент времени будет выполняться только одним потоком.
- Критические секции могут снабжаться именами.
- Критические секции считаются независимыми, только если они используют разные имена.
- По умолчанию, все неименованные критические секции имеют одно имя

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
#pragma omp critical (Xaxis_critical_section)
  x_next = Queue_Remove(x);
  Process(x_next);
#pragma omp critical (Yaxis_critical_section)
  y_next = Queue_Remove(y);
  Process(y_next);
}
```

Пример (некорректное использование)

```
#pragma omp parallel for private(i) shared(a,xmax)
for(i=0; i<N; i++){
    if(a[i]>xmax)
#pragma omp critical
       xmax = a[i];
}// for
```

Пример (корректное использование, но не эффективное)

```
#pragma omp parallel for private(i) shared(a,xmax)
for(i=0; i<N; i++){
#pragma omp critical
    if(a[i]>xmax)
       xmax = a[i];
}// for
```

Синхронизация вычислений. Директива barrier

Определяет *барьер* –точку в программе, которую должна достигнуть каждая нить, чтобы все нити продолжили вычисления.

```
#pragma omp parallel shared (A, TmpRes, FinalRes)
{
  DoSomeWork(A, TmpRes);
  printf("Processed A into TmpRes\n");
#pragma omp barrier
  DoSomeWork(TmpRes, FinalRes);
  printf("Processed B into C\n");
}
```

Синхронизация вычислений. Директива atomic

- Определяет переменную в левой части оператора присваивания, которая должна корректно обновляться несколькими нитями.
- В этом случае происходит предотвращение прерывания доступа, чтения и записи данных, находящихся в общей памяти, со стороны других потоков.
- Применяется эта синхронизация только для операторов, следующих непосредственно за определяющей ее директивой.
- Синхронизация atomic - очень дорогая операция с точки зрения трудоемкости выполнения программы.

```
1.  #include "omp.h"
2.  #include <iostream>
3.
4.  int main() {
5.    int value = 123;
6.    #pragma omp parallel
7.    {
8.      #pragma omp atomic
9.      value++;
10.     #pragma omp critical (cout)
11.     {
12.       std::cout << value << std::endl;
13.     }
14.   }
15. }
```

Синхронизация вычислений. Директива ordered

Синхронизация типа ordered используется для определения потоков в параллельной области программы, которые выполняются в порядке, соответствующем последовательной версии программы.

Пример:

```
#pragma omp parallel private(myid)
{
    myid = omp_get_thred_num();
#pragma omp for private(i)
    for(i=0; i<8; i++)
#pragma omp ordered
        printf("T%d: %d\n", myid, i);
}
```

Результат:

```
T0: 0
T0: 1
T0: 2
T0: 3
T1: 4
T1: 5
T1: 6
T1: 7
```

Синхронизация вычислений. Директива flush

- Эта конструкция осуществляет немедленный сброс значений разделяемых переменных в память.
- Таким образом гарантируется, что во всех потоках значение переменной будет одинаковое.
- Неявно flush присутствует в следующих директивах: barrier, начале и конце критических секций, параллельных циклов, параллельных областей, single секций..
- С ее помощью можно посылать сигналы потоком используя переменную как семафор. Когда поток видит, что значение разделяемой переменной изменилось, то это говорит, что произошло событие и следовательно можно продолжить выполнение программы далее

#pragma omp flush [(список переменных)]

Сравнение стандартов

MPI

- Плюсы
  - Переносимость для систем с общей и распределенной памятью
  - Масштабируемость при увеличении узлов
  - Отсутствие проблемы размещения данных
- Минусы
  - Сложность разработки и отладки
  - Высокая латентность, низкая пропускная способность
  - Явные коммуникации
  - Сложность балансировки загрузки

OpenMP

- Плюсы
  - Простая реализация параллелизма
  - Низкая латентность, высокая пропускная способность
  - Неявные коммуникации
  - Динамическая балансировка загрузки
- Минусы
  - Переносимость только для систем с общей памятью
  - Масштабируемость в пределах одного узла
  - Возможная проблема размещения данных
  - Отсутствие возможности задать порядок нитей

Архитектура MPI+OpenMP: плюсы и минусы

Плюсы

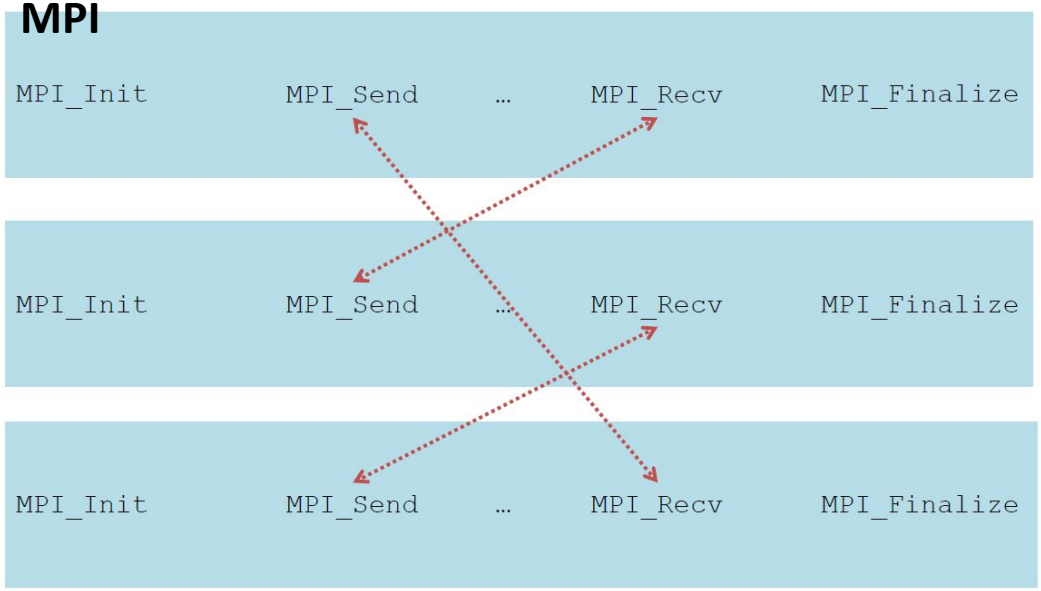
- Удобное применение для кластеров с SMP-узлами:
  - MPI –между узлами
  - Избегаем накладных расходов на MPI-коммуникации внутри узла
  - OpenMP –внутри узла
  - Получаем передачу сообщений большего размера за меньшее время и динамическую балансировку загрузки.
- Потенциальная возможность получить большее ускорение, чем "чистый" MPI или "чистый" OpenMP.

Минусы

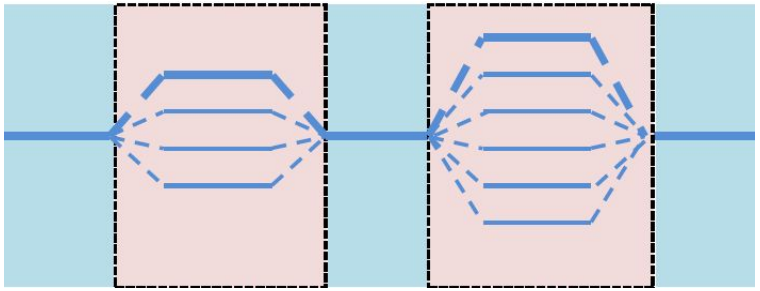
- Меньшая масштабируемость OpenMP.
- Возможность тупиков в MPI.
- Накладные расходы на обработку нитей:
- Во время MPI-обмена все нити, кроме одной, бездействуют
- Необходимость пересечения вычислений и коммуникаций для лучшей производительности

MPI, OpenMP, MPI+OpenMP

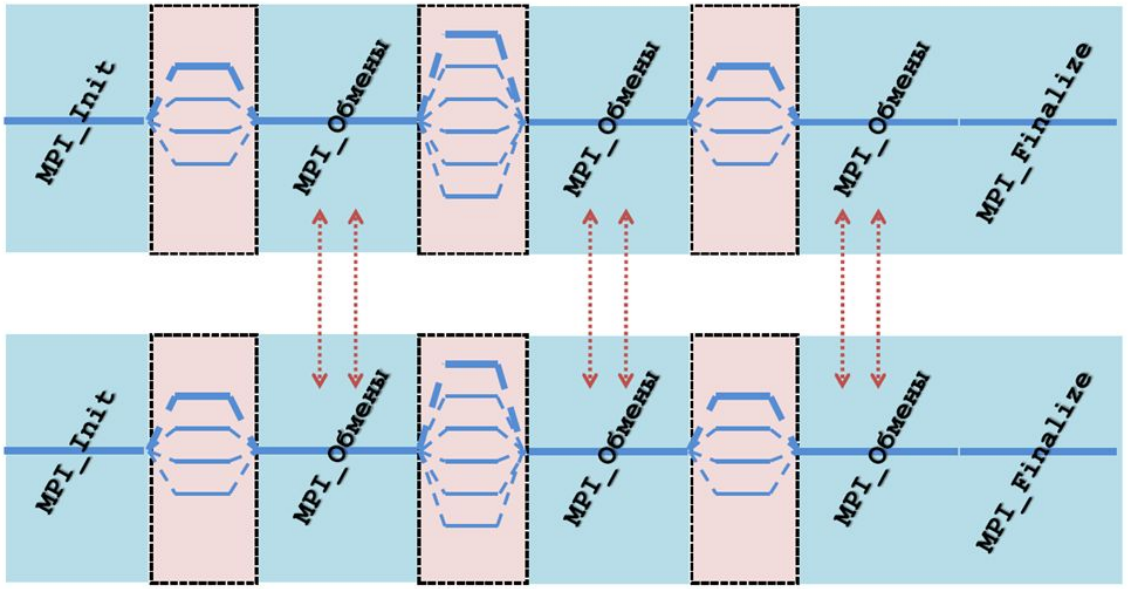
MPI



OpenMP



MPI+OpenMP



MPI+OpenMP программа

```
#include "mpi.h"
#include "omp.h"
int main(int argc, char *argv[])
{
  int rank, numtasks;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  ... /* Вычисления, вызовы функций MPI-обменов */

  omp_set_num_threads(необходимое_количество_нитей);
  #pragma omp parallel for private(…) shared(…) …
  for (…) {
     …
  }

  ... /* Вычисления, вызовы функций MPI-обменов */
  MPI_Finalize();
  return 0;
}
```

Уровни поддержки нитей в MPI

- MPI_THREAD_SINGLE
  - MPI-процесс исполняет единственную нить
- **MPI_THREAD_FUNNELED**
  - MPI-процесс может запустить несколько нитей
  - MPI-вызовы разрешены только в той нити, которая выполнила инициализацию MPI
- MPI_THREAD_SERIALIZED
  - MPI-процесс может запустить несколько нитей
  - MPI-вызовы разрешены в любой нити, но не более чем одной нитью одновременно
- MPI_THREAD_MULTIPLE
  - MPI-процесс может запустить несколько нитей
  - MPI-вызовы разрешены в любой нити без ограничения на количество одновременных вызовов

MPI-программа с поддержкой нитей

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int required = MPI_THREAD_FUNNELED;
  int rank, provided;

  MPI_Init_thread(&argc, &argv, required, &provided);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (mpi_rank==0) {
    switch (provided) {
    case MPI_THREAD_SINGLE: /* */ break;
    case MPI_THREAD_FUNNELED: /* */ break;
    case MPI_THREAD_SERIALIZED: /* */ break;
    case MPI_THREAD_MULTIPLE: /* */ break;
    default: /* */;
  }
  MPI_Finalize();
  return 0;
}
```

MPI+OpenMP программа

```
// Для MPI_THREAD_FUNNELED
#pragma omp barrier
#pragma omp master
 MPI_xxx(…)
```

```
// Для MPI_THREAD_SERIALIZED
#pragma omp barrier
#pragma omp single
 MPI_xxx(…)
```

```
// Для MPI_THREAD_FUNNELED
#pragma omp parallel
{
 if (my_thread_number==0)
   MPI_xxx(…); // Обмен
 else
   // Вычисления
}
```